Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution

Hui Xu*†, Yangfan Zhou‡§, Yu Kang‡, Fengzhi Tu*, Michael R. Lyu*†
* Shenzhen Research Institute, The Chinese University of Hong Kong
† Dept. of Computer Science and Engineering, The Chinese University of Hong Kong
‡ School of Computer Science, Fudan University
§ Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education

***Abstract*—Control-flow obfuscation increases program complexity by semantic-preserving transformation. Opaque predicates are essential gadgets to achieve such transformation. However, we observe that real-world opaque predicates are generally very simple and engage little security consideration. Recently, such insecure opaque predicates have been severely attacked by symbolic execution-based adversaries and jeopardize the security of control-flow obfuscation. This paper, therefore, proposes *symbolic opaque predicates* which can be resilient to symbolic execution-based adversaries. We design a general framework to compose such opaque predicates, which requires introducing challenging symbolic analysis problems (*e.g.,* symbolic memory) in each opaque predicate. In this way, we may mislead symbolic execution engines into reaching false conclusions. We observe a novel *bi-opaque property* about symbolic opaque predicates, which can incur not only false negative issues but also false positive issues to attackers. To evaluate the efficacy of our idea, we have implemented a prototype obfuscation tool based on Obfuscator-LLVM and conduct experiments with real-world programs. Our evaluation results show that symbolic opaque predicates demonstrate excellent resilience to prevalent symbolic execution engines, such as BAP, Triton, and Angr. Moreover, although the costs of symbolic opaque predicates may vary for different problem settings, some predicates can be very efficient. Therefore, our framework is both secure and usable. Users can follow the framework to introduce symbolic opaque predicates into their obfuscation tools and made them more powerful.**

I. Introduction

Obfuscation is a widely employed technique which protects software from reverse engineering. It transforms programs into unintelligible versions while preserving their original functionalities. Obfuscation can be achieved via lexical transformation, control-flow transformation, data-flow transformation, *etc* [1]. Such obfuscation transformation techniques are orthogonal to each other and can be employed simultaneously.

This paper focuses on *control-flow obfuscation*, which increases software complexity (*e.g.,* by adding bogus control flows) against reverse control-flow analysis. *Opaque predicates* are essential gadgets to achieve such obfuscation transformation. An opaque predicate is a predicate whose value is known before obfuscation time but difficult to be deduced by reverse analysis. Because it holds some deterministic properties, we can employ opaque predicates to transform a program without changing its semantics. For example, we can add a bogus code block after a constantly false opaque predicate and guarantee the code block would never be executed. In practice, *opaque constant* (*e.g.,* $x^2 \neq -1$) is the most prevalent type of opaque predicates adopted by obfuscation tools, such as Obfuscator-LLVM [2]. Although other approaches (*e.g.,* unsolved conjectures [3]) may demonstrate better security, they are not widely adopted due to either implementation or performance issues [4].

Recently, the security of opaque predicates has been greatly challenged due to the development of symbolic execution techniques. Notably, Ming *et al.* have proposed an opaque predicate detection approach based on symbolic execution [5]; Yadegari *et al.* have demonstrated the effectiveness of deobfuscation attacks based on symbolic execution [6]. Symbolic execution is a program analysis approach that models the conditions for executing alternative control flows. It attempts to find test cases that can satisfy such conditions. If a condition cannot be satisfied, it may indicate a bogus control flow or an opaque predicate. Symbolic execution-based attacks may not be new to the research community. But due to the development of symbolic execution techniques, such attacks become practical recently and jeopardize the robustness of obfuscated software.

In this work, we propose a novel framework to manufacture *symbolic opaque predicates* which are resistant to symbolic execution-based adversaries. A key procedure in our framework is to introduce challenging problems for symbolic execution to analyze, such as employing symbolic memory and parallel programming [7]. Moreover, we observe a *bi-opaque property* of such opaque predicates, *i.e.,* it may either mislead an attacker into falsely recognizing an opaque predicate as a normal predicate, or to falsely recognizing a normal predicate as an opaque predicate.

We have implemented a prototype tool based on Obfuscator-LLVM [2]. Our tool[1] automatically replaces the opaque predicates generated by Obfuscator-LLVM with symbolic opaque predicates in IR (intermediate representative) level. It employs a repository-based mechanism to manage different templates of symbolic opaque predicates. Currently, we have implemented several templates in the repository, which attack symbolic execution with symbolic memory, floating-point numbers, covert propagation, and parallel programming. The

[1]Our project url is https://github.com/hxuhack/symobfuscator

tool is flexible such that users to extend the repository with their own templates.

We have evaluated the resilience of our idea against three prevalent symbolic execution engines, including BAP [8], Triton [9], and Angr [10]. The results demonstrate that symbolic opaque predicates have excellent resilience against symbolic execution-based attacks. Then we evaluate the *stealth* of symbolic opaque predicates against human adversaries when obfuscating real programs, including both general Linux programs and encryption programs. Our experimental results show that the implemented opaque predicates do not incur obvious abnormal instruction patterns. We also evaluate the *cost* of the implemented predicates. Experimental results show that some symbolic opaque predicates incur almost no overhead in comparison with the default opaque predicates adopted in Obfuscator-LLVM, such as those employing symbolic memory and floating-point numbers. Other opaque predicates may incur obvious execution overhead, such as those employing covert propagation and parallel programming. However, this does not degrade the usability of our framework as long as there are some efficient symbolic opaque predicates. The cost issue can be mitigated in practice by allowing users to filter inefficient predicates or to prioritize the predicates according to their preferences. Our approach is thus promising to be adopted by real-world obfuscation tools.

We conclude our primary contributions as follows.

- This paper proposes *symbolic opaque predicates* and demonstrates a framework to manufacture such predicates. Our experimental results show that symbolic opaque predicates are secure against symbolic execution-based attacks and they are usable.
- We observe the novel *bi-opaque property* of such opaque predicates, which extends the classic understanding about opaque predicates.

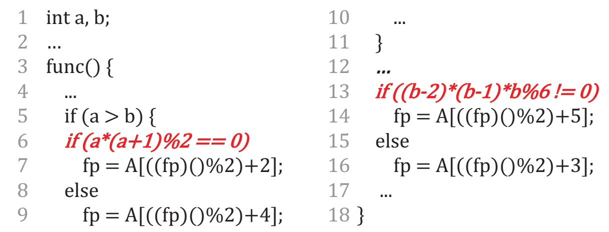
The rest of the paper is organized as follows. Section II-A discusses our motivating examples and defines the adversary model of this paper. Section III introduces our framework for composing symbolic opaque predicates. Section IV evaluates the security and cost of our approach. Section V discusses the related work. Finally, Section VI concludes the paper.

II. Backgrounds

A. Motivation

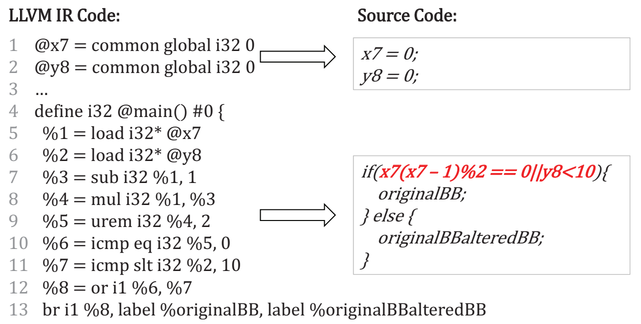
Our investigation is mainly motivated by the vulnerability of real-world opaque predicates. Opaque predicates are essential gadgets for control-flow obfuscation. As stated by Collberg *et al.* [1], the security of opaque predicates largely determines the security of control-flow obfuscation. However, we notice that many real-world opaque predicates are not very strong. Below, we use two examples to demonstrate the issue.

The first example is from a highly cited paper [11], which proposes an approach to obfuscate programs with NP-hard security. To compose NP-hard problems, the authors introduce pointer analysis problems and control pointer alignments with opaque predicates. In this way, they can compose 3-SAT



```
1  int a, b;
2  ...
3  func() {
4     ...
5     if (a > b) {
6     if(a*(a+1)%2 == 0)
7        fp = A[((fp)()%2)+2];
8     else
9        fp = A[((fp)()%2)+4];
10       ...
11    }
12    ...
13    if((b-2)*(b-1)*b%6 != 0)
14       fp = A[((fp)()%2)+5];
15    else
16       fp = A[((fp)()%2)+3];
17    ...
18 }
```

(a) The opaque predicate example in [11]



LLVM IR Code:

```
@x7 = common global i32 0
@y8 = common global i32 0
...
define i32 @main() #0 {
  %1 = load i32* @x7
  %2 = load i32* @y8
  %3 = sub i32 %1, 1
  %4 = mul i32 %1, %3
  %5 = urem i32 %4, 2
  %6 = icmp eq i32 %5, 0
  %7 = icmp slt i32 %2, 10
  %8 = or i1 %6, %7
  br i1 %8, label %originalBB, label %originalBBalteredBB
```

Source Code:

```
x7 = 0;
y8 = 0;
```

```
if(x7(x7 – 1)%2 == 0||y8<10){
   originalBB;
} else {
   originalBBalteredBB;
}
```

(b) An opaque predicate generated by Obfuscator-LLVM. For easy reading, we translate the LLVM IR code to source code

Fig. 1: Real-world opaque predicate examples.

problems in the constraint models. However, the underlying opaque predicates in the paper are not strong enough. We demonstrate this in Figure 1(a), which includes two opaque predicates: the first one $a * (a + 1)\%2 \; == \; 0$ (line 6) is constantly true for any integer a; the second one $(b - 2) * (b - 1) * b\%6 \neq 0$ (line 13) is constantly false for any integer b. When such predicates are processed by a symbolic execution engine, the engine would detect that the constraints $a * (a + 1)\%2 \neq 0$ and $(b - 2) * (b - 1) * b\%6 \neq 0$ cannot be satisfied. Such predicates would be reported as opaque predicates by symbolic execution-based attackers. As a result, the NP-hard problem can be simplified to a polynomial-time problem.

Figure 1(b) demonstrates another opaque predicate example generated by Obfuscator-LLVM [2]. Obfuscator-LLVM is an opensource obfuscation tool for C programs and has been commercialized recently. In this example, the opaque predicate $x7 * (x7 - 1)\%2 \; == \; 0 || x8 < 10$ is always true, which can be easily detected by symbolic execution techniques. We have reviewed the source code of Obfuscator-LLVM and found that the opaque predicate is the only supported one. The authors indeed have left comments in the code and stated that the opaque predicate should be improved.

Besides, there are many other investigations relying on such insecure opaque predicates, *e.g.,* [12, 13]. These examples demonstrate a severe vulnerability of current opaque predicates in practice. More resilient opaque predicates are therefore necessary to improve the security of control-flow obfuscation techniques.

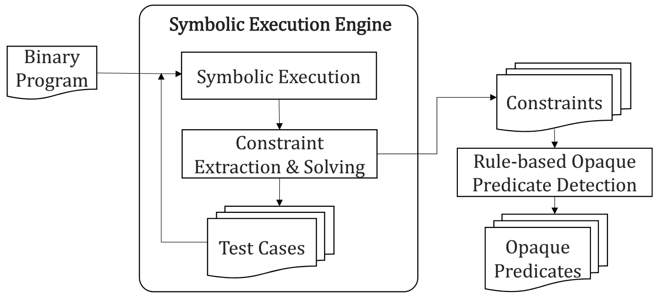

Symbolic Execution Engine
Binary Program
Symbolic Execution
Constraint Extraction & Solving
Test Cases
Constraints
Rule-based Opaque Predicate Detection
Opaque Predicates

Fig. 2: A conceptual framework of opaque predicate detection based on symbolic execution techniques.

B. Adversary Model

This work considers an adversary model as follows. Suppose an obfuscated binary program is obtained by an attacker, she can employ symbolic execution techniques to detect opaque predicates from the obfuscated program and further deobfuscate the program. We demonstrate a framework for such opaque predicate detection attacks in Figure 2. Overall, a symbolic execution engine is employed to extract the conditions along control paths as constraint models; then a rule-based detection module is employed to detect opaque predicates from the constraint models.

In general, a symbolic execution engine for binaries includes a core symbolic execution module, and a constraint solving module. The symbolic execution module can be implemented in two ways: dynamic or static. Dynamic symbolic execution is also known as concolic (concret and symbolic) execution. BAP [8] and Triton [9] are two typical concolic execution engines. They first execute a program with concrete values, and then perform symbolic analysis on the generated instruction traces. Comparatively, a static symbolic execution engine firstly lifts a binary program to high-level intermediate codes and then perform symbolic execution on the codes with static analysis approaches. Angr [10] adopts the second approach. Both the two approaches can generate constraint models for opaque predicate detection.

To better demonstrate the principle of symbolic execution, we discuss more details about the concolic execution technique, which has been adopted by Ming *et al.* [5] for opaque predicate detection. Concolic execution includes several key steps: instruction tracing and lifting, trace slicing, and constraint extraction and solving.

Instruction Tracing and Lifting: In each round of concolic execution, we trace the executed instructions along a control flow. The instructions are assembly codes by default. To model the semantics of each instruction, an instruction lifter is required. The lifter translates assembly codes to a high-level intermediate language (IL), which models the memory and register operations with variables. In practice, not all instructions are useful, and sometimes a taint analysis engine is employed to filter out the instructions irrelevant to any symbolic variables. This step outputs a sequence of instructions modeled with IL.

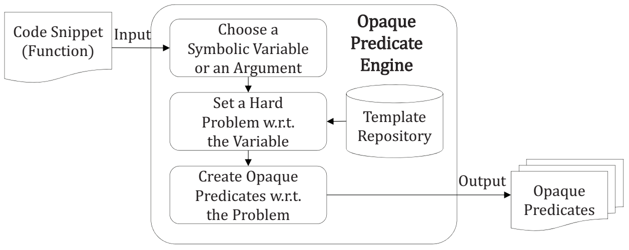

Code Snippet (Function)
Input
Choose a Symbolic Variable or an Argument
Opaque Predicate Engine
Set a Hard Problem w.r.t. the Variable
Template Repository
Create Opaque Predicates w.r.t. the Problem
Output
Opaque Predicates

Fig. 3: A framework to compose symbolic opaque predicates.

Trace Slicing: A control flow may contain several conditional branches, and each branch requires a constraint model to be satisfied for triggering the branch. We may get the negations of each constraint model and solve them to generate test cases that can trigger alternative control flows. This step outputs several sliced sequences of instructions, each indicating a new control flow possibility.

Constraint Extraction and Solving: The symbolic execution engine extracts a condition from each sliced trace, and models the condition with a contraint modeling language, such as CVC [14] or SMT-Lib [15]. Then the engine employs a constraint solver (*e.g.,* STP [16] or Z3 [17]) to solve the models, and the results are new test cases that can trigger corresponding control paths. For opaque predicate detection, the constraint models should be passed to a rule-based opaque predicate detection engine.

Rule-based Opaque Predicate Detection: The constraint model generated by a symbolic execution engine is generally in conjunctive normal form (CNF), *i.e.,* $\lambda_1 \wedge \lambda_2 \wedge ... \wedge \lambda_n$. Each clause λ_i represents a predicate. Then the CNF is processed according to opaque predicate detection rules, such as the rules to detect opaque constants, or contextual opaque predicates [5]. Since the rules are upper-level applications, we do not discuss their details. Instead, we focus on attacking the underlying symbolic execution engines. If the generated CNF is incorrect, it is likely that such attackers would reach false conclusions.

III. Approach

In this section, we first introduce our idea to compose *symbolic opaque predicates* by attacking the challenges up against symbolic execution; then we discuss the *bi-opaque property* of such opaque predicates; finally, we demonstrate how the predicates work in practice.

A. Idea in a Nutshell

Intuitively, we can employ the weakness of symbolic execution to compose opaque predicates such that they can evade detection from symbolic execution-based adversaries. This is feasible because symbolic execution faces some challenges, and real-world symbolic execution tools have to adopt heuristic methods to handle them. Introducing such challenging problems into a program may incur error for symbolic execution.

```
int func(int symvar){
   int j = symvar;
   if(j == 7){
      Foo();
   }
}
```

(a) Original toy program.

```
int func(int symvar){
   int j = symvar;
   int l1_ary[] = {1,2,3,4,5,6,7};
   int l2_ary[] = {j,1,2,3,4,5,6,7};
   int i = l2_ary[l1_ary[j%7]];
   if(i == j)
      Bogus();
   if(i == 1 && j == 7)
      Foo();
}
```

(b) Symbolic memory.

```
int func(int symvar){
   int j = symvar;
   float f = j/1000000.0;
   if(f==0.1){
      Bogus();
   }
   if(1024+f == 1024 && f>0 && j==7){
      Foo();
   }
}
```

(c) Floating-point number.

```
int func(int symvar){
   int i, j = symvar;
   FILE *fp = fopen("covp.tmp", "w");
   if(fp == NULL) {
      exit(1);
   }
   fprintf(fp,"%d",j);
   fclose(fp);
   fp = fopen("covp.tmp", "r");
   fscanf(fp,"%d",&i);
   fclose(fp);
   if(i == 7){
      Foo();
   }
   if(i != j){
      Bogus();
   }
}
```

(d) Covert symbolic propagation.

```
int64_t ThreadProp(int64_t in){
      int64_t out = in;
      pthread_t tid;
      int rc1 = pthread_create(&tid,
              NULL, Inc, (void *) &out);
      int rc2 = pthread_create(&tid,
              NULL, Dec, (void *) &out);
      pthread_join(tid, NULL);
      return out;
}
int func(int symvar){
   int i, j = symvar;
   i = ThreadProp(j);
   if(i == 6){
      Foo();
   }
   if(i == j){
      Bogus();
   }
}
```

(e) Parallel programming.

Fig. 4: Opaque predicate examples attacking the challenges of symbolic execution.

Figure 3 demonstrates a general framework to compose such opaque predicates. Suppose the input is a code snippet or a function which contains arguments. Then we can choose an argument as the symbolic variable and create a challenging problem related to the variable. The challenging problem is selected from a repository of predefined templates. We may create hundreds of such templates by attacking different challenges of symbolic execution or employ different problem settings. Finally, we can create opaque predicates based on the symbolic variable protected by the problem.

Note that at least one symbolic variable should get involved in a challenging problem. Because only such problems matter to symbolic execution. If a problem does not include any symbolic value, all the problem-related instructions would be pruned by the symbolic execution engine. This can be proved with Hore Logic [18] following the principle of symbolic execution [7]. Because involving symbolic variables is a prerequisite for composing such opaque predicates, we name our opaque predicates as *symbolic opaque predicates*. If a function has no argument, then we have to introduce fake arguments or employ global symbolic variables.

B. Bi-Opaque Property

Traditional opaque predicates aim to evade from detection, such that the obfuscated control-flow graph cannot be easily simplified. In other words, they try to mislead adversaries into falsely recognizing them as normal predicates. Failing to detect them would cause *false negative* issues for adversaries. With symbolic opaque predicates, an interesting observation is that we may also introduce *false positive* issues, *i.e.,* we may mislead adversaries into falsely recognizing normal predicates as opaque predicates.

In this way, a predicate can be opaque in either a way, which is the novel *bi-opaque property* of our approach. Specifically, we name the two types of opaque predicates: *type I opaque predicate* which intends to introduce false negatives and *type II opaque predicate* which intends to introduce false positives. Next, we use several examples to demonstrate how to compose symbolic opaque predicates with the bi-opaque property.

C. Demonstration

Suppose Figure 4(a) is a function to obfuscate, then Figure 4(b) demonstrates how to obfuscate it with symbolic opaque predicates. Specifically, the predicates employ the challenge of symbolic memory.

Symbolic memory is a difficult problem for program analysis because it involves pointer analysis issues, which can be NP-hard or even undecidable [19]. In this example, we compose two integer arrays. The symbolic value $j\%7$ points to an element within the first array, and the element serves as an offset of the second array. The selected element from the second array is assigned to a new variable i. In this way, i is a symbolic value protected by the challenging problem, and we can compose symbolic opaque predicates with i.

For example, we can compose a type I opaque predicate that cannot be satisfied, such as $i == j$. With the opaque predicate, we can insert a bogus code block (*i.e.,* `Bogus()`) which would never be executed. The security of the predicate depends on the capability of symbolic execution engines. If a symbolic execution engine employs no mechanism to handle symbolic memory, it would generate incorrect constraint models and falsely recognize the predicate as a normal predicate.

To compose a type II opaque predicate, we first select an ordinary predicate, $j == 7$. Then we modify the predicate by introducing a new condition related to i, such as $i ==$

$1\&\&j == 7$. The modification does not change the semantics of the original predicate because $i == 1$ is always true when j equals 7. Such condition can be easily generated because the value of i can be calculated from any j. In assembly codes, the new predicate will be dissembled into two predicates $i == 1$ and $j == 7$. The second predicate $j == 7$ will only be evaluated if the first predicate is true. If a symbolic execution engine does not support symbolic memory, it cannot solve the constraint of $i == 1$ and cannot reach the ordinary predicate $j == 7$.

D. Template Generalization

With the above example, we have demonstrated how our idea works in practice. Now we discuss how to implement the challenging problem as a template.

In general, a template is a code fragment in a compiler pass, which inserts, deletes, or modifies the program to be compiled. Algorithm 1 demonstrates such a template which implements the challenging symbolic memory problem in Figure 4(b). The algorithm inputs an `icmp` instruction and outputs symbolic opaque predicates. Suppose the `icmp` compares if a symbolic variable equals to an integer, the template first parses the instruction and get a symbolic variable $symVar$ and a constant $ciObj$. Then, we define the types of the two arrays and initialize them. Next, we can create an integer variable i and initialize it with the value $l2\_ary[l1\_ary[j\%7]]$.

Based on the protected symbolic variable i, we can directly create a type I opaque predicate with a comparison instruction $i == j$. To compose a type II opaque predicate, we have to introduce one more `icmp` instruction. The new instruction compares if i equals to a value, and it should be true if the original `icmp` (*i.e., inst*) is true. In this example, according to the array setting, when j equals to a constant value, the value of i can be determined as $j\%7 + 1$.

E. Template Enrichment

Employing only one template is vulnerable to pattern recognition. We have to create different opaque predicates to increase the security level. This can be achieved in two ways. Firstly, we may create more templates by employing different problem settings. Secondly, we may create new templates by employing new challenges.

1) Employing New Settings: For each challenge that symbolic execution is faced with, we may compose a great many templates. Take the symbolic memory as an example, one can create arrays with different elements, employ a different modular, use three arrays instead of two arrays, store the array with heap instead of stack. All such methods ensure that the resulting symbolic opaque predicates are different in binaries or assembly codes.

2) Employing new Challenges: Another orthogonal approach is to employ new challenges, such as *floating-point number*, *covert propagation*, and *concurrent program*.

Figure 4(c) is an example that composes opaque predicates based on the challenge of floating-point numbers. A floating-point number is an approximation of a real number with a fixed length of digits in the form of $significant \times b^e$. It enables the computer to handle very large numbers or very small numbers with only limited memory space. As a trade off, floating-point numbers sacrifice the precision. Floating-point numbers may incur troubles to symbolic execution because reasoning over rational numbers and real numbers may lead to inconsistencies [20, 21]. In this example, because the float type cannot represent 0.1 precisely, no matter which value we assign to `symvar`, $f == 0.1$ cannot be satisfied. To compose a type II predicate, we can change the predicate $j == 7$ to $(1024 + f == 1024)\&\&(f > 0)\&\&(j == 7)$. The new predicate aims to fool symbolic execution engines that the constraint $(1024+f == 1024)\&\&(f > 0)$ cannot be satisfied, which is true in the domain of real numbers. However, it can be satisfied in the domain of floating-point numbers. For example, $f = 0.000007$ is a solution. In this way, the type II opaque predicate can be satisfied when $j = 7$, which preserves the semantics. If a symbolic execution engine cannot handle such floating-point numbers, it may falsely regarded $f == 0.1$ as a normal predicate, and the type II predicate as an opaque predicate.

Figure 4(d) demonstrates how to compose opaque predicates by attacking the challenge of covert propagation. Symbolic execution requires precise tracking on the propagation of the symbolic values. However, symbolic values may be propagated in many ways via I/O (input/output) operations. In this example, the symbolic value j is propagated via a file on the disk and then assigned to i. We can compose a type I opaque constant $i! = j$, which will always be false. If a symbolic execution engine cannot track the propagation, it would treat i as a constant and regard the opaque predicate as a normal one. To compose a type II opaque predicate, we can change the predicate $j == 7$ to $i == 7$, where i equals to j. This modification keeps the original semantics of the program. However, a symbolic executor may consider i as a constant and reach false conclusions.

Figure 4(e) is another example that introduces a simple parallel computing problem. Parallel programs are difficult to handle for symbolic executions because the execution order is not only determined by the programs, but also by the host computer. Therefore, we cannot generate a static control-flow graph for the program, which is a basis for classic symbolic execution to work. In this example, we create two more threads that modify the value of a symbolic variable j: one thread increases in to $in + 1$, and another decreases in to $in - 1$. Due to parallel execution, the two threads compute on the same value of in simultaneously. The value of i is determined by the thread that terminates late, which should be the second thread in our example. Finally, the return value of the `ThreadProp` function should equal to $j - 1$. Based on the protected symbolic variable i, we can compose a type I opaque predicate as $i == j$, and a type II opaque predicate as $i == 6$.

Similar to Algorithm 1, we can extract templates based on such examples. Note that this work does not intend to enumerate all such templates to create symbolic opaque predicates. Rather, we would like to show a general framework

Algorithm 1: An LLVM template for creating symbolic opaque predicates based on the symbolic memory example in Figure 4(b).

```
/* Input an icmp instruction; output 2 opaque
   predicates                                        */
input : inst
output: type1Opq,type2Opq
/* Parse the icmp instruction                        */
Value* left ← inst−>getOperand(0) ;
Value* right ← inst−>getOperand(1) ;
Value* symVar ;
ConstantInt* ciObj ;
if isa<ConstantInt> (*left) then
    ciObj ← left ;
    symVar ← right ;
end
else if isa<ConstantInt> (*right) then
    symVar ← left ;
    ciObj ← right ;
end
if !symVar−>getType()−>isIntegerTy() then
    return;
end
/* Define the size of the two arrays.                */
ArrayType* ar1AT ← ArrayType::get(intType, 7) ;
ArrayType* ar2AT ← ArrayType::get(intType, 8) ;
/* Allocate storage for the arrays                   */
AllocaInst* ar1AI ← new AllocaInst(ar1AT, "", inst) ;
AllocaInst* ar2AI ← new AllocaInst(ar2AT, "", inst) ;
/* ...                                               */
/* Here we omit several lines of codes that
   initialize the elements of each array.            */
/* ...                                               */
/* Create a new variable j that equals to
   symVar, and then load j.                          */
AllocaInst* jAI ← new AllocaInst(varType, "", inst) ;
StoreInst* jSI ← new StoreInst(symVar, jAI, inst) ;
LoadInst* jLI ← new LoadInst(jAI, "", inst) ;
/* Compute j%7.                                      */
BinaryOperator* remBO ← BinaryOperator::Create(SRem, jLI,
   cInt7, "", inst);
/* Get an element from the array ar1AI with an
   index remBO; load its value to l1LI.              */
std::vector<Value*> l1Vec, l2Vec;
l1Vec.push_back(cInt0);
l1Vec.push_back(remBO) ;
ArrayRef<Value*> l1AR(l1Vec);
Instruction* l1EPI ← GetElementPtrInst::CreateInBounds(
   ar1AI, l1AR,"", inst);
LoadInst* l1LI ← new LoadInst(l1EPI,"", false, inst);
/* Get an element from the array ar2AI with an
   index l1LI; load its value to iLI.                */
l2Vec.push_back(cInt0);
l2Vec.push_back(l1LI);
ArrayRef<Value*> l2AR(l2Vec);
Instruction* l2EPI ← GetElementPtrInst::CreateInBounds(
   ar2AI, l2AR,"", inst);
LoadInst* iLI ← new LoadInst(l2EPI, "", false, inst);
/* Compose a type I opaque predicate, i == j.
   */
ICmpInst* type1Opq ← new ICmpInst(inst, ICMP_EQ, iLI,
   jLI, "");
/* Compose a type II opaque predicate,
   i == j%7 + 1&&inst .                              */
BinaryOperator* addBO ← BinaryOperator::Create(ADD,
   remBO, cInt1,"", inst);
ICmpInst* leftOpq ← new ICmpInst(inst, ICMP_EQ, iLI,
   cInt1, "");
BinaryOperator* andBO ← BinaryOperator::Create(AND,
   leftOpq, inst,"", inst);
ICmpInst* type2Opq ← new ICmpInst(inst, ICMP_EQ,
   cInt1,andBO,"");
```

and demonstrate how it works. This can shed light to more types of symbolic opaque predicates.

IV. Evaluation

A. Evaluation Criteria

According to Collberg *et al.* [22], the evaluation criteria for assessing software obfuscation quality include *potency*, *resilience*, *stealth*, and *cost*. However, not all of the criteria are applicable to our work. We will evaluate symbolic opaque predicates with resilience, stealth, and cost.

Resilience evaluates how the obfuscation technique can hold up against automatic attacks. In this work, we assume the attackers are symbolic execution-based adversaries, which are automatic attacks. We should evaluate the security of symbolic opaque predicates against symbolic execution.

Stealth assesses whether an obfuscation technique is suspicious to human attackers. A stealthy opaque predicate should not incur abnormal instruction patterns or obvious statistical difference with normal predicates.

Cost measures the overhead incurred by obfuscation. Opaque predicates may incur overhead in both program size and execution time. We should evaluate such overhead when obfuscating real programs with symbolic opaque predicates and compare the overhead with existing opaque predicates.

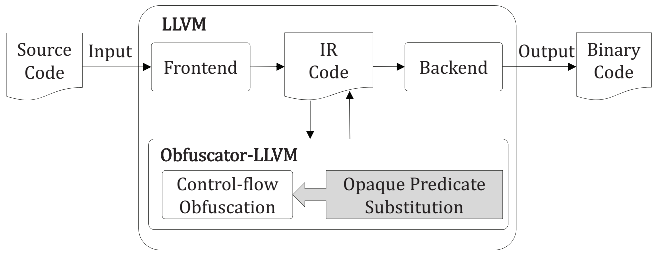


Fig. 5: Prototype implementation based on Obfuscator-LLVM.

We will not evaluate *potency* because it is not applicable to opaque predicates. *Potency* measures how much obscurity can be added to the program. This is the major objective of general obfuscation or control-flow obfuscation, rather than opaque predicates.

B. Prototype Implementation

We have implemented a prototype obfuscation tool based on Obfuscator-LLVM [2]. Obfuscator-LLVM is an obfuscation

TABLE I: Evaluation results about the resilience of our opaque predicates in Figure 4. Notation: $\times_{fn}$: a type I opaque predicate causes false negative issues to a symbolic execution engine; $\times_{fp}$: a type II opaque predicate causes false positive issues to a symbolic execution engine; $\surd_{\times}$: the predicate is insecure, but we can find corner cases to defeat the symbolic execution engine.

Templates of Symbolic Opaque Predicates		**Symbolic Execution Tools** BAP	Triton	Angr
Symbolic Memory	Type I	$\times_{fn}$	$\times_{fn}$	$\times_{fn}$
	Type II	$\times_{fp}$	$\times_{fp}$	$\times_{fp}$
Floating-point Numbers	Type I	$\times_{fn}$	$\times_{fn}$	$\surd_{\times}$
	Type II	$\times_{fp}$	$\times_{fp}$	$\surd_{\times}$
Covet Symbolic Propagation	Type I	$\times_{fn}$	$\times_{fn}$	$\times_{fn}$
	Type II	$\times_{fp}$	$\times_{fp}$	$\times_{fp}$
Parallel Programming	Type I	$\times_{fn}$	$\times_{fn}$	$\times_{fn}$
	Type II	$\times_{fp}$	$\times_{fp}$	$\times_{fp}$

tool for C programs based on LLVM compiler [23]. We adopt LLVM as our compiler basis because it is open-source released and has achieved wide usage in both research and industrial fields.

Figure 5 describes the framework of our prototype. The source code of a program is firstly processed by an LLVM frontend, which transforms the source code to intermediate representatives (IR). For C programs, the frontend is Clang. IR is the core object processed in LLVM. LLVM provides a basic framework for performing program analysis tasks based on IR. It allows users to customize their own compilation passes for specific program analysis tasks, such as optimization and obfuscation. Obfuscator-LLVM in nature applies several compilation passes to obfuscate programs in IR level. Finally, the IR will be compiled to binaries by a corresponding backend (*e.g.,* for X86_64 system).

Based on the framework of LLVM, we implement the feature of symbolic opaque predicates as a compiler pass. The pass can substitute the opaque predicates generated by Obfuscator-LLVM with resilient ones. We have implemented all the challenging problems discussed in Section III. Users can decide which opaque predicates will be employed during obfuscation.

Our prototype supports two methods to customize new templates of symbolic opaque predicates. The first one is to write a native LLVM pass which can insert IR (as shown in Algorithm 1) during compilation. To this end, users should be familiar with the IR syntax and LLVM APIs, which impose a steep learning curve. The second method requires only very little knowledge about LLVM development. Users can create new templates in source code level. Then they can compile the source code to object code and link it with the original program via static linkage. The second approach is somehow limited but it can facilitate the development process.

C. Resilience

Our adversary model assumes symbolic execution-based adversaries. Therefore, we mainly evaluate the resilience of symbolic opaque predicates with respect to the security against symbolic execution engines. If a symbolic execution engine fails in handling the proposed predicates, the corresponding adversaries should also suffer the same problems.

We choose three prevalent symbolic execution tools for resilience evaluation, including BAP [8], Triton [9], and Angr [10]. We consider several criteria when selecting them: 1) the tool should support binaries; 2) the tool should demonstrate good capabilities with high community impacts; 3) it should be free and open-source for public usage. To our best knowledge, these three tools are the only tools that meet our criteria. Other prevalent symbolic execution tools either do not support binaries, such as KLEE [24], or they are close-source, such as Mayhem [25]. By evaluating against these tools, we aim to show that the proposed opaque predicates can achieve good resilience to symbolic execution in practice.

Table I summarizes our evaluation results. The results are mainly based on the correctness of the generated constraint models. If a symbolic execution tool falsely models the constraint for a type I opaque predicate or reports a solution, we label the result as $\times_{fn}$; if it falsely models the constraint for a type II opaque predicate, or reports no solution, we label the result as $\times_{fp}$. The results show that all the tools suffer problems when handling our symbolic opaque predicates, except that Angr is capable of handling the floating-point example. We further analyze the details of each experimental result as follows.

First, we discuss the issues of symbolic execution tools when handling the predicates based on symbolic memory. For each tool, we demonstrate the issue of the tool with a figure. Figure 6(a) reports the issue of BAP. BAP only taints the first array value retrieving operation (line 18), and omits the second one (line 21). When modeling the constraint for the cmp instruction in line 24, the value of EAX is falsely retrieved from line 18, rather than line 21 or 23. As a result, BAP cannot model the constraint correctly. Triton also fails in tainting the array operation. As a result shown in Figure 6(b), Triton detects several branching points, but only one contains a constraint model. Finally, a test case 3 is reported a solution to trigger `Bogus()`, which is a false positive case. Moreover, Triton does not generate constraint models to reach the branch with `Foo()`, so it also suffers false negative issues. To run Angr, we have customized a python script which searches solutions that can trigger the addresses of `Foo()` and `Bogus()`. Our script is shown in Figure 6(c). We observe that the result of Angr depends on how we declare the size of the symbolic variable. If we declare the size as one byte, Angr can find a correct solution for reaching `Foo()`. However, if we declare the size as two bytes or four bytes, Angr finds two paths to `Foo()`, but it finds no solution. Moreover, it falsely reports a solution to reach `Bogus()`. We have further verified that if we change the 1-digit predicate (*i.e.,* $j == 7$) to a 2-digit predicate (*e.g.,* $j == 10$), Angr would suffer problems for all such size declarations. Therefore, the opaque predicate based on symbolic memory should be secure against Angr.

```
1   //stack initialization
2   //compute j%7
3    0x0000000000400d8b <+144>:  imul   %edx
4    0x0000000000400d8d <+146>:  lea    (%rdx,%rcx,1),%eax
5    0x0000000000400d90 <+149>:  sar    $0x2,%eax
6    0x0000000000400d93 <+152>:  mov    %eax,%edx
7    0x0000000000400d95 <+154>:  mov    %ecx,%eax
8    0x0000000000400d97 <+156>:  sar    $0x1f,%eax
9    0x0000000000400d9a <+159>:  sub    %eax,%edx
10   0x0000000000400d9c <+161>:  mov    %edx,%eax
11   0x0000000000400d9e <+163>:  shl    $0x3,%eax
12   0x0000000000400da1 <+166>:  sub    %edx,%eax
13   0x0000000000400da3 <+168>:  sub    %eax,%ecx
14   0x0000000000400da5 <+170>:  mov    %ecx,%edx
15   0x0000000000400da5 <+170>:  mov    %ecx,%edx
16   0x0000000000400da7 <+172>:  movslq %edx,%rax
17   //get an element from l1_ary
18   0x0000000000400daa <+175>:  mov    -0x40(%rbp,%rax,4),%eax
19   0x0000000000400dae <+179>:  cltq
20   //get an element from l2_ary (not tainted)
21   0x0000000000400db0 <+181>:  mov    -0x20(%rbp,%rax,4),%eax
22   0x0000000000400db4 <+185>:  mov    %eax,-0x44(%rbp)
23   0x0000000000400db7 <+188>:  mov    -0x44(%rbp),%eax
24   0x0000000000400dba <+191>:  cmp    -0x48(%rbp),%eax
25   0x0000000000400dbd <+194>:  jne    0x400dc9 <main+206>
26   0x0000000000400dbf <+196>:  mov    $0x0,%eax
27   0x0000000000400dc4 <+201>:  callq  0x400b8d <Bogus>
```

(a) BAP fails in tainting the second array retrieving operation in line 21. The tainted instructions are in dark black, and untainted instructions are in light black. The symbolic value j is initialized as 1.

```
[INFO]: Start triton...
[INFO]: entry: 400aa0, exit: 400dea
[INFO]: Take Snapshot
[INFO]: Find the main function
[INFO]: In main() we set :
[INFO]:       [0x7fffa485a39f] = 31 1
[INFO]: Exit point
[INFO]: Branching points: 2
[INFO]: Inputbound: 0
[DEBG]: Variable declaration@getModels()@solverEngine:(declare-fun SymVar_0 () (_ BitVec 8))
[INFO]: Model: 1
[INFO]: New testcase has been detected //for triggering bogus()
[INFO]: Inputbound: 1
[DEBG]: Variable declaration@getModels()@solverEngine:(declare-fun SymVar_0 () (_ BitVec 8))
[INFO]: Model: 0
[INFO]: Restore snapshot
[INFO]: Take Snapshot
[INFO]: Find the main function
[INFO]: In main() we set :
[INFO]:       [0x7fffa485a39f] = 33 3
[INFO]: Exit point
[INFO]: Branching points: 2
[INFO]: Inputbound: 1
[DEBG]: Variable declaration@getModels()@solverEngine:(declare-fun SymVar_0 () (_ BitVec 8))
[INFO]: Model: 0
[INFO]: Done !
```

(b) Triton also fails in tainting the second array operation. As a result, two branching points are detected, but no constraint models can be generated. The symbolic value j is initialized as 1.

```
#=============Angr Testing Script===============
proj = angr.Project(project_path, load_options={'auto_load_libs': True})
cfg = proj.analyses.CFG(fail_fast=True)
argv = [proj.filename]
sym_arg = claripy.BVS('sym_arg', 8*4)
argv.append(sym_arg)
state = proj.factory.entry_state(args=[argv])
path_group = proj.factory.path_group(state)

def check(p):
    #function body: find the address of a function; return true if it is reachable

path_group = path_group.explore(find=check)
#code to print the result
```

(c) Angr script for symbolic execution.

```
Results with different settings for argv[1]
------Set argv max length to 4 bytes------
Selecting a path to Foo! Find no solution.
Selecting a path to Foo! Find no solution.
Selecting a path to Bogus! Solution ('argv=', '-4')

------Set argv max length to 2 bytes------
Selecting a path to Foo! Find no solution.
Selecting a path to Foo! Find no solution.
Selecting a path to Bogus! Solution ('argv=', '-4')

------Set argv max length to 1 byte------
Selecting a path to Foo! Solution ('argv=', '7')
Selecting a path to Bogus! Find no solution.
```

(d) Result of Angr script.

Fig. 6: Failure report of symbolic execution tools in handling our opaque predicate example in Figure 4(b).

The floating-point example in Figure 4(c) has shown good resistance against BAP and Triton. The tools both report several unsupported floating-point instructions, including $divsd$, $unpcklpd$, $cvtpd2ps$, $movss$, *etc*. Consequently, they generate incorrect constraint models. Angr has achieved better performance, and it can handle our example correctly. However, we are still able to find other floating-point problems that Angr cannot handle. For example, if a constraint model requires a solution with decimal digits (*e.g.,* the solution is "0.000001"), Angr would report incorrect results. We have inquired the issue with one author from the Angr team. Their reply confirmed our result that Angr has good support with floating-point numbers, but not perfect. Therefore, Angr is not overwhelming for predicates with floating-point numbers. We may find better floating-point problems to compose symbolic opaque predicates in the future.

All the three tools fail in handling the covert propagation example in Figure 4(d). Our experimental results show that they haven't traced the instructions related to i. For example, we cannot find the comparison operation ($i == 7$) from the tainted instructions. This implies i is regarded as a constant. In this way, the condition ($i == 7$) may be falsely regarded as an opaque constant, while the second condition ($i != j$) will be falsely regarded as a normal predicate that can be satisfied. Using the same script in Figure 6(c), Angr falsely reports a solution for the type I opaque predicate and falsely reports no solutions for the type II opaque predicate.

The symbolic execution engines also fail in handling the parallel programming example in Figure 4(e). Angr reports an error "unable to concretize address for loading with the provided strategies". As a result, it falsely generates an incorrect result (*i.e.,* 0) for the type I opaque predicate, and it starves the computing resources when handling the type II opaque predicate and exits abnormally. BAP and Triton do not support parallel computing and they directly ignore all the instructions related to thread scheduling. Because for concolic execution, the instructions related to thread management are generally not tainted.

```
mov     edi, edx
sub     edi, 1
imul    edx, edi
and     edx, 1
cmp     edx, 0
setz    r8b
cmp     esi, 0Ah
setl    r9b
or      r8b, r9b
test    r8b, 1
jnz     loc_100002440
jmp     loc_1000045D1
```

(a) Obfuscator-LLVM.

```
cvtsi2sd  xmm0, dword ptr [rsi]
movsd     xmm1, [rbp+var_28]
divsd     xmm0, xmm1
cvtsd2ss  xmm0, xmm0
movss     dword ptr [rdx], xmm0
cvtss2sd  xmm0, dword ptr [rdx]
movsd     xmm2, [rbp+var_20]
ucomisd   xmm0, xmm2
mov       [rbp+var_51], r10b
jnz       loc_100001282
jp        loc_100001282
jmp       loc_1000014BF
```

(b) Floating-point numbers.

```
movsxd  rdx, dword ptr [rdx]
mov     rdi, rdx
mov     [rbp+var_41], r10b
mov     [rbp+var_50], rdx
call    _ThreadProp
mov     rcx, [rbp+var_50]
cmp     rcx, rax
jnz     loc_10000113C
mov     al, [rbp+var_41]
test    al, 1
jnz     loc_100000F98
jmp     loc_100001004
```

(c) Parallel programming.

Fig. 7: The assembly codes of symbolic opaque predicates.

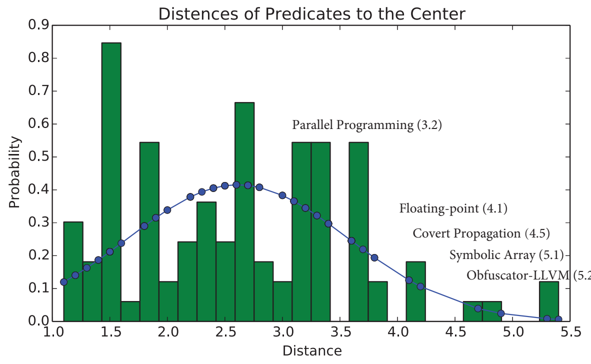


Fig. 8: A comparison of symbolic opaque predicates with ordinary predicates. The curve is the distance distribution of ordinary predicates and the histogram is the raw data.

Note that our evaluation results are from the view of symbolic execution engines rather than specific symbolic execution-based attackers. This is because all such attackers or specific detection rules would be effective only if the underlying symbolic execution engine performs correctly [5]. In this regard, our evaluation results remain valid for evaluating symbolic execution-based attackers.

D. Stealth

Currently, there is no standard evaluation method for stealth. Existing methods (*e.g.,* [26]) generally measure the statistical difference of instructions between obfuscated programs and ordinary programs. The less difference an obfuscation approach incurs, the stealthier it is.

To apply the idea on evaluating symbolic opaque predicates, we should measure the difference between a symbolic opaque predicate and ordinary predicates. In general, the difference depends on which challenging problem that a predicate employs. Different problems will generate different codes and corresponding assembly instructions. Figure 7 demonstrates the assembly codes of several opaque predicates. Figure 7(a) is the default opaque predicate generated by Obfuscator-LLVM, which is mainly composed of arithmetic operations. Figure 7(b) is the symbolic opaque predicate with floating-point numbers, which is mainly composed of floating-point operations. The two figures demonstrate obvious difference; however, all such instructions are widely used in ordinary programs.

TABLE II: Categorization of Instructions.

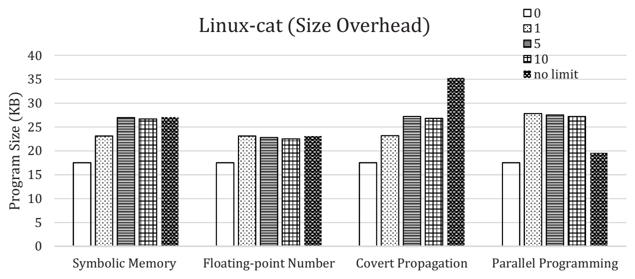
Category	Instructions
Arithmetic Instructions	imul, inc, sub, add, idiv, divsd, sbb
Logical Instructions	and, sar, xor, test, shr, shl, or, xorps
Instructions for Data Transfer	movaps, movsd, movabs, movzx, mov, movss, movsx, movsxd, stosd
Instructions Converting Data Dimension	cvtss2sd, cvtsi2sd, cvtsd2ss, cqo, cdq
Pointer Instructions	lea
Comparison Instructions	cmp, ucomisd
Jump Instructions	jle, jne, jge, jae, jl, je,jg, jp, ja, jbe, jno, jmp
Stack-related Instructions	pop, push, call, ret
Instructions Creating Boolean Variable	setge, setne, setg, seta, setb, setl, sete
Other Instructions	nop

In our experiment, we use a similarity-based approach to measure the difference between symbolic opaque predicates and ordinary predicates. To this end, we randomly select 100 ordinary predicates from the unobfuscated binaries. For each predicate, we arbitrarily select the 10 instructions before its conditional jump because such instructions would serve as essential information for reverse analysis. Then we categorize such instructions into several types with a categorization approach employed for malware detection [27]. Table II lists the categories and corresponding instructions in each category. Considering the space where each dimension is an instruction category, a predicate can be represented as a vector in that space. Then we can compute the center of the 100 ordinary opaque predicates, and compute the euclidean distance from each predicate to the center. Figure 8 shows the distribution of such distances. In our experiment, the average distance is 2.6, and the max distance is 5.4. For comparison, we also compute the distances from our symbolic opaque predicates to the center, which are between 3.2, 4.1, 4.5, and 5.1. The distances are smaller than the max distance of ordinary predicate. Moreover, they are slightly better than the distance of the default opaque predicate employed in Obfuscator-LLVM, which is 5.2.

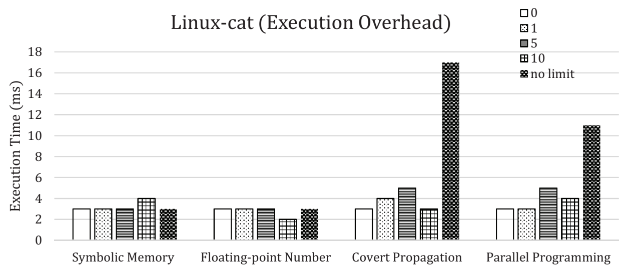
The opaque predicates based on parallel programming has the best performance in stealth. The main reason is that we have employed a call-based approach to implement the predicate. As shown in Figure 4(e), we implement the symbolic analysis problem in another function and only employ the return value in the main routine. In its binary code shown in Figure 7(c), only a `call` instruction is artificially added before the unconditional jump, and the rest instructions are mostly from the original program. By simply reading the instructions nearby a conditional jump, it would be difficult to discover the tricks of symbolic opaque predicates.

E. Cost

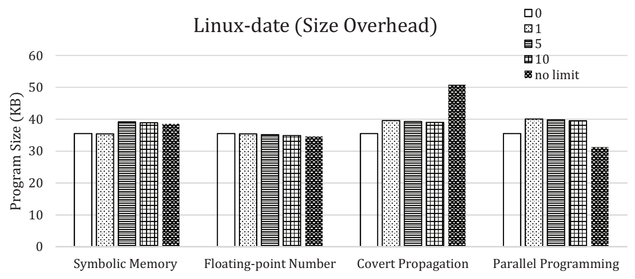
To evaluate the cost of symbolic opaque predicates, we obfuscate several general programs (*e.g.,* Linux commands such as `cat`, `ls`, `date`) and several encryption programs (*e.g.,* MD5 and AES). We choose encryption programs because



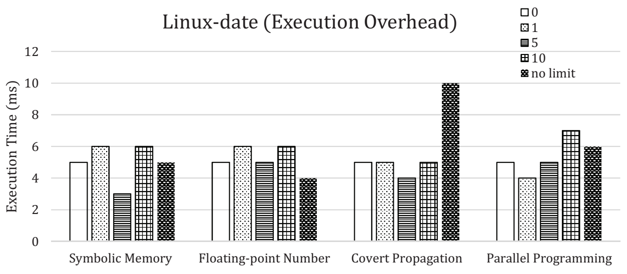
(a) Size overhead when obfuscating Linux command cat.



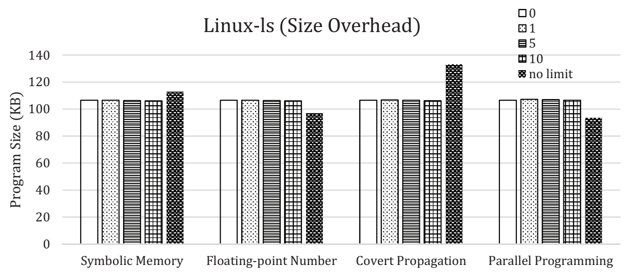
(b) Execution overhead when obfuscating Linux command cat.



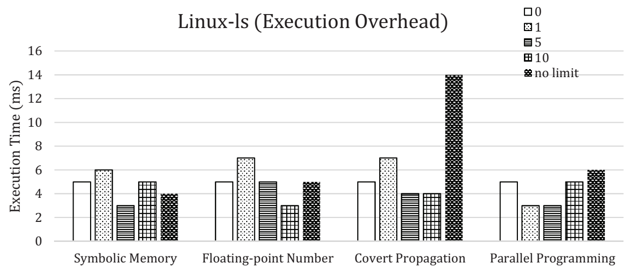
(c) Size overhead when obfuscating Linux command date.



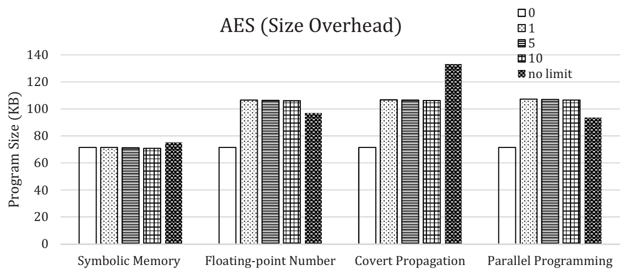
(d) Execution overhead when obfuscating Linux command date.



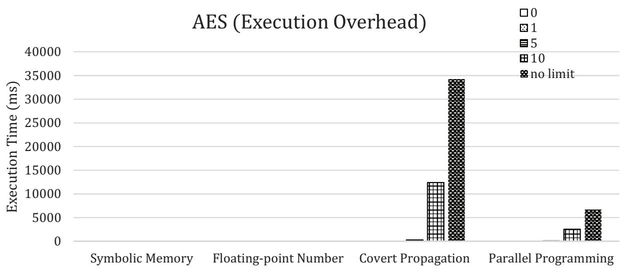
(e) Size overhead when obfuscating Linux command ls.



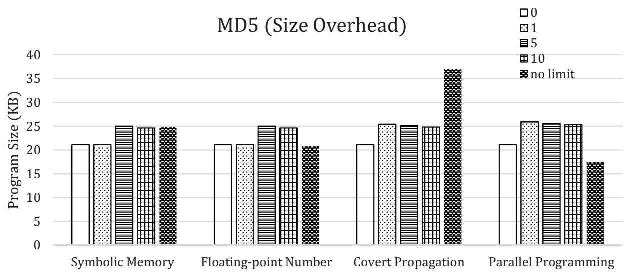
(f) Execution overhead when obfuscating Linux command ls.



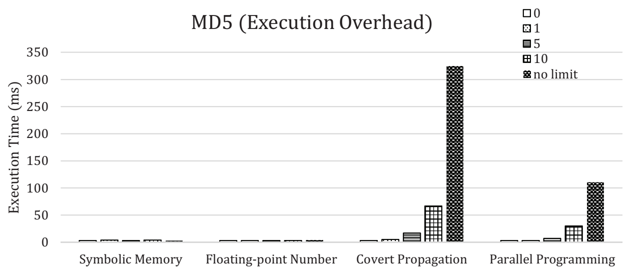
(g) Size overhead when obfuscating AES.



(h) Execution overhead when obfuscating AES.



(i) Size overhead when obfuscating MD5.



(j) Execution overhead when obfuscating MD5.

Fig. 9: Cost of symbolic opaque predicates.

they generally have higher security requirements, and therefore obfuscation is more needed. When obfuscating the programs, we employ 80% obfuscation rate (*i.e.,* a configuration of LLVM-Obfuscator) as the baseline. Then for each program, we replace a certain number (1, 5, 10, and no limit) of opaque predicates from the obfuscated software with the symbolic opaque predicates. We watch the performance variations with different numbers of symbolic opaque predicates.

Figure 9 shows our evaluation results. We measure the performance of obfuscation with both program size and execution time. From the result, we observe that the size overhead is not a big issue. The symbolic opaque predicates based on symbolic memory and floating-point both incur similar size overhead in comparison with the default opaque predicate employed by Obfuscator-LLVM. The covert propagation sample involves more instructions and therefore incurs more overhead. However, such cost can be mitigated by employing a call-based implementation. For example, although the parallel program sample also involves many instructions, the resulting obfuscated program is even smaller than the program obfuscated by the original Obfuscator-LLVM.

Some symbolic opaque predicates are also very efficient in execution time, such as those based on symbolic memory and floating-point numbers. Their costs are similar to the default opaque predicates employed in Obfuscator-LLVM. However, some symbolic opaque predicates incur much cost during execution. As shown in *e.g.,* Figure 9(h) and Figure 9(j), the execution overhead may be thousands of times when employing covert propagation and parallel programming to obfuscate encryption programs. Such predicates involve heavy operations (*e.g.,* file read/write, thread creation/execution) and incur nontrivial execution cost. The overhead seems acceptable for general Linux programs, but it can be amplified for encryption programs because the symbolic opaque predicates are nested in loops in such programs.

In a word, the cost of symbolic opaque predicates depends on the employed challenging problems and their implementation mechanisms. Some symbolic opaque predicates can be very promising with trivial costs. But we should be careful when employing other opaque predicates with heavy cost, especially when using them with loops. In practice, we may prioritize the cost of symbolic opaque predicates and preemptively employ more efficient ones. Note that there is still a large room to improve the usability issue, which is beyond the scope of this work.

V. Related Work

In this section, we first survey the recent achievement of software deobfuscation with symbolic execution techniques, which illustrates the importance of our research problem; then we elaborate the novelty of our research by comparing our work with existing opaque predicates which might also be resilient to symbolic execution.

A. Symbolic Execution for Deobfuscation

Recently, the development of symbolic execution techniques has bred several important attempts to deobfuscation(*e.g.,* [5, 6, 28, 29]). Ming *et al.* [5] proposed LOOP, which is a logic-oriented tool for opaque predicate detection. LOOP is made up of a symbolic execution engine and a rule-based predicate analyzer. The rule can detect three types of opaque predicates, including invariant opaque predicates, contextual opaque predicates, and dynamic opaque predicates. Another work [29] from the same group employs symbolic execution techniques to detect malware camouflage from obfuscated binaries. Yadegari *et al.* [28] proposed a generic framework to deobfuscate binaries based on symbolic execution. Their framework collects traces generated by a symbolic execution engine and then employs the traces to simplify the obfuscated control-flow graph. Their work is based on an enhanced symbolic execution engine (*i.e.,* ConcoLynx [6]). However, the tool is not available for public evaluation.

Besides, there are several other investigations that attack obfuscated software with symbolic execution techniques, such as [30]–[32]. Because the underlying techniques are similar, we do not discuss each of them in detail.

B. Comparison with Existing Opaque Predicates

Before this work, Wang *et al.* [3] have conducted another investigation that has a similar purpose with us. They propose to compose resilient opaque predicates by attacking the weakness of symbolic execution in handling loops. Specifically, they create opaque predicates with *unsolved conjectures*, which is a form of looped codes. A common characteristic of such *unsolved conjectures* is that they would eventually exit the loops with some convergence properties. For example, the Collatz conjecture takes an input $x \in \mathbb{N}^+$, and iteratively calculates $x = x/2$ if x is even, otherwise calculates $x = 3x + 1$. No matter what value x has bee initialized with, the loop always terminates with x equals to 1. Besides, there are other predicates that maybe secure against symbolic execution, such as the opaque predicate with *one-way function* [33], and the predicate involving dynamic updated objects [22]. Note that all such opaque predicates are secure because they attack some weakness of symbolic execution. Such approaches also comply with our framework, and we may extend our template repository with them.

In a word, our work is different from previous work in that our framework is more general. We emphasize the importance of employing symbolic variables rather than leveraging specific tricks. In other words, we highlight the common properties for an opaque predicate to be secure against symbolic execution.

VI. Conclusion

To conclude, this work studies the security issue of opaque predicates with respect to symbolic execution-based attacks. We have proposed a novel idea of symbolic opaque predicates and demonstrated a general framework to compose such predicates. A novel characteristic of symbolic opaque predicates is the bi-opaque property, which can incur either false negative or false positive issues to symbolic execution-based attackers. To demonstrate the usability of our approach, we have implemented a prototype obfuscation tool based on Obfuscator-LLVM and conducted real-world experiments. We have evaluated the resilience, stealth, and cost of some symbolic opaque predicates. Evaluation results show that symbolic opaque predicates exhibit good resistance against prevalent symbolic execution engines. Some opaque predicate examples are also stealthy and efficient. Therefore, symbolic opaque predicates should serves as a promising idea for practical obfuscation tools to improve their resistance against symbolic execution-based attacks.

ACKNOWLEDGMENTS

This work was substantially supported by the the National Basic Research Program of China (973 Project No. 2014CB347701), National Natural Science Foundation of China (Project Nos. 61672164 and 61332010), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund), and Microsoft Research Asia via 2018 MSRA Collaborative Research Award. Yu Kang is the corresponding author.

REFERENCES

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

[2] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm: Software protection for the masses," 2015.

[3] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *ESORICS*. Springer, 2011.

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, 2016.

[5] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[6] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[7] X. Hui, Z. Yangfan, K. Yu, and R. L. Michael, "Concolic execution on small-size binaries: Challenges and empirical study," in *Proc. of the 47th IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2017.

[8] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proc. of the International Conference on Computer Aided Verification*. Springer, 2011.

[9] F. Saudel and J. Salwan, "Triton: a dynamic symbolic execution framework," in *SSTIC*, 2015.

[10] Y. Shoshitaishvili and *et al.*, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[11] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, 2003.

[12] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[13] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[14] "CVC4," http://cvc4.cs.stanford.edu/web/, 2017.

[15] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proc. of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[16] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. of the International Conference on Computer Aided Verification*. Springer, 2007.

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[18] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, 1969.

[19] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proc. of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1991.

[20] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[21] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *Software Testing, Verification and Reliability*, vol. 16, no. 2, pp. 97–121, 2006.

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

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proc. of the IEEE International Symposium on Code Generation and Optimization*, 2004.

[24] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[26] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, "Translingual obfuscation," in *Proc. of the IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[27] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati, "Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms," in *Proc. of the 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*. IEEE, 2015.

[28] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[29] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Proc. of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[30] Y. Guillot and A. Gazet, "Automatic binary deobfuscation," *Journal in computer virology*, vol. 6, no. 3, pp. 261–276, 2010.

[31] F. Biondi, S. Josse, A. Legay, and T. Sirvent, "Effectiveness of synthesis in concolic deobfuscation," 2015.

[32] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *Proc. of the 1st IEEE/ACM International Workshop on Software Protection*, 2015.

[33] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.