Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method

Dongpeng Xu, Jiang Ming, and Dinghao Wu

College of Information Sciences and Technology
The Pennsylvania State University
{dux103,jum310,dwu}@ist.psu.edu

Abstract. Opaque predicate obfuscation, a low-cost and stealthy control flow obfuscation method to introduce superfluous branches, has been demonstrated to be effective to impede reverse engineering efforts and broadly used in various areas of software security. Conventional opaque predicates typically rely on the invariant property of well-known number theoretic theorems, making them easy to be detected by the dynamic testing and formal semantics techniques. To address this limitation, previous work has introduced the idea of dynamic opaque predicates, whose values may vary in different runs. However, the systematical design and evaluation of dynamic opaque predicates are far from mature. In this paper, we generalize the concept and systematically develop a new control flow obfuscation scheme called *generalized dynamic opaque predicates*. Compared to the previous work, our approach has two distinct advantages: 1) We extend the application scope by automatically transforming more common program structures (e.g., straight-line code, branch, and loop) into dynamic opaque predicates; 2) Our system design does not require that dynamic opaque predicates to be strictly adjacent, which is more resilient to the deobfuscation techniques. We have developed a prototype tool based on LLVM IR and evaluated it by obfuscating the GNU core utilities. Our experimental results show the efficacy and generality of our method. In addition, the comparative evaluation demonstrates that our method is resilient to the latest formal program semantics-based opaque predicate detection method.

Keywords: Software protection, obfuscation, opaque predicate, control flow obfuscation

1 Introduction

Predicates are conditional expressions that evaluate to true or false. An opaque predicate means its value are known to the obfuscator at obfuscation time, but it is difficult for an attacker to figure it out afterwards. Used together with junk code, the effect of opaque predicates results in a heavily cluttered control flow graph with redundant infeasible paths. Therefore, any further analysis based on the control flow graph will turn into arduous work. Compared with other control flow graph obfuscation methods such as control flow flattening [26] and

call stack tampering [24], opaque predicates are more stealthy because it is difficult to differentiate opaque predicates from original path conditions in binary code [4,5]. Also, another benefit of opaque predicates is they have a small impact on the runtime performance and code size. First proposed by Collberg et al. [6], opaque predicates have been applied widely in various ways, such as software diversification [10,15], metamorphic malware mutation [2,3], software watermarking [1,21], and Android Apps obfuscation [14]. Due to the low-cost and stealthy properties, most real-world obfuscation toolkits have supported inserting opaque predicates into a program, through link-time program rewriting or binary rewriting [8,13,18].

On the other hand, opaque predicate detection has attracted many security researchers' attention. Plenty of approaches have been proposed to identify opaque predicates inside programs. For instance, Preda et al. [23], Madou [17] and Udupa et al. [25] did research on opaque predicate detection based on the fact that the value of opaque predicate doesn't change during multiple executions. The invariant property of those "static" opaque predicates leads to the fact that they are likely to be detected by program analysis tools. Furthermore, recent research work [19] shows that even dynamic opaque predicate, which is more complicated and advanced than traditional static opaque predicates, can also be detected by their deobfuscation tool. Dynamic opaque predicates overcome the invariant weakness of static opaque predicates by using a set of correlated predicates. The authors claims that they can detect static and dynamic opaque predicates inside an execution binary trace.

Essentially, existing opaque predicates detection techniques utilize several weaknesses of opaque predicates. First, as mentioned above, the invariant property of traditional algebraic based opaque predicates reveals their existence. Second, the design of dynamic opaque predicate is far from mature. Existing technique can only insert dynamic opaque predicates into a piece of straight-line code. It cannot spread dynamic opaque predicates across branch conditions. This limitation leads to the consequence that all predicates constituting a dynamic opaque predicate are adjacent, which is utilized by advanced opaque predicates detection tools such as LOOP [19].

In order to overcome the limitations of current opaque predicates, we present a systematic design of a novel control flow obfuscation method, *Generalized Dynamic Opaque Predicates*, which is able to inject diversified dynamic opaque predicates into complicated program structures such as branch and loop. Being compared with the previous technique which can only insert dynamic opaque predicates into straight-line program, our new method is more resilient to program analysis tools. We have implemented a prototype tool based on the LLVM compiler infrastructure [16]. The tool first performs fine-grained data flow analysis to search possible insertion locations. After that it automatically transforms common program structures to construct dynamic opaque predicates. We have tested and evaluated the tool by obfuscating several hot functions of GNU core utilities with different obfuscation levels. The experimental results show that our method is effective and general in control flow obfuscation. Besides, we demon-

strate that our obfuscation can defeat the commercial binary difference analysis tools and the state-of-the-art formal program semantics-based deobfuscation methods. The performance data indicate that our proposed obfuscation only introduces negligible overhead.

In summary, we make the following contributions.

- First, we propose an effective and generalized opaque predicate obfuscation method. Our method outperforms existing work by automatically inserting opaque predicates into more general program structures like branches and loops, whereas previous work can only work on straight-line code.
- Second, we demonstrate our obfuscation is very resilient to the state-of-art opaque predicate detection tool.
- Third, we have implemented our method on top of LLVM and the source code is available.

The rest of the paper is organized as follows. Section 2 introduces the related work on opaque predicates and state-of-the-art opaque predicate detection methods. Section 3 presents our new obfuscation method, generalized dynamic opaque predicates in detail. Section 4 presents our implementation details. We evaluate our method in Section 5 and conclude the paper in Section 6.

2 Related Work

In this section, we first introduce the related work on static and dynamic opaque predicates. Then we discuss the drawbacks of current opaque predicate detection methods, which also inspires us to propose the generalized dynamic opaque predicates.

2.1 Static Opaque Predicates

Static opaque predicates indicates the opaque predicates whose value is fixed during runtime. Basically, there are two categories of static opaque predicates: invariant opaque predicates and contextual opaque predicates. According to previous research work [19], invariant opaque predicates refer to those predicates whose value always evaluates to true or false for all possible inputs. The predicate is opaque since it is difficult to know the value in advance except the obfuscator. Usually invariant opaque predicates are constructed by utilizing some algebraic theorems [21] or quadratic residues [1] as follows.

$$\forall x \in Z.\ (4x^2 + 4) \bmod 19 \not\equiv 0$$

As a result of its simplicity, there are large numbers of invariant opaque predicates candidates. On the other hand, the invariant feature also leads to the shortage of this category of opaque predicates. One possible way to detect invariant opaque predicates is to observe the branches that never change at run time with fuzzing testing [17].

The other kind of static opaque predicates is contextual opaque predicate. It is proposed by Drape [11] to avoid an opaque predicate always produces the fixed value for all inputs. Contextual opaque predicate only evaluates to always true or false under a given precondition. Typically this kind of opaque predicate is an implication relation between two predicates, which is elaborately constructed in a particular program context. An example of contextual opaque predicates is presented as follows.

$$\forall x \in Z.\ (7x-5) \bmod 3 \equiv 0 \Rightarrow (28x^2 - 13x - 5) \bmod 9 \equiv 0$$

In this example, the predicate $(28x^2 - 13x - 5) \bmod 9 \equiv 0$ is always true given $(7x - 5) \bmod 3 \equiv 0$ and x is an integer. In addition, the constant value in contextual opaque predicates can be further obfuscated so as to hide the implication relation [20].

2.2 Conventional Dynamic Opaque Predicates

Palsberg et al. [22] first introduce the concept of dynamic opaque predicates, which consist of a family of *correlated* predicates that all present the same value in one given execution, but the value may be changed in another execution. Thus the values of the dynamic opaque predicates switch dynamically at run time. Here we use the term "conventional dynamic opaque predicates" to distinguish it from the generalized dynamic opaque predicates we proposed in this paper. Particularly, since its design is still immature although the concept is noval, conventional dynamic opaque predicate can only be injected into straight-line programs, which results in that all predicates are set adjacently. We provide a conventional dynamic opaque predicate example as follows.

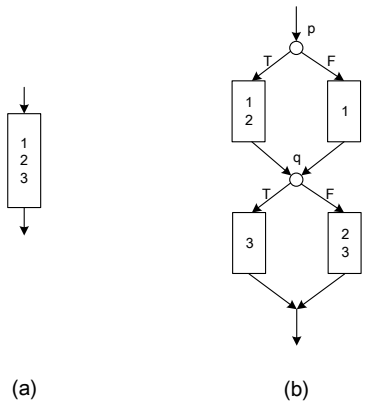


Fig. 1: An example of conventional dynamic opaque predicates.

Fig. 1(a) shows the original straight-line code and Fig. 1(b) shows the obfuscated version using conventional dynamic opaque predicates. In this paper,

we use a rectangle to represent a basic block and the numbers inside to indicate instructions. The small circles represent predicates. Section 3.2 provides more detailed description of those symbols. In Fig. 1(b), p and q are two correlated predicates. They are evaluated to both true or false in any given run. In the original program as shown in Fig. 1(a), three instructions are executed one by one: `[1 2 3]`. In the obfuscated version, each execution either follows all left branches ($p \wedge q$ holds) or all right branches ($\neg p \wedge \neg q$ holds). The same instruction sequence is executed in both cases: `[1 2]->[3]` when taking the left branches and `[1]->[2 3]` vice versa. Since the predicate q split the two paths into different segments, p and q have to be adjacent to maintain the semantic equivalence.

2.3 Opaque Predicate Detection

Collberg et al. [6] first propose the idea of opaque predicates to prevent malicious reverse engineering attempts. In addition, the authors also provide some ad-hoc detection methods, such as "statistical analysis". This approach utilize the assumption that, if a predicate that always produces the same result over a larger number of test cases, it is likely to be an opaque predicate. Due to the low coverage of inputs, statistical analysis could lead to high false positive rates.

Preda et al. [23] propose to detect opaque predicates by another method called abstract interpretation. However, their approach can only handle a specific type of known invariant opaque predicates. Madou [17] first identifies candidate branches that never changes at run time, and then verifies such predicates by fuzz testing with a considerably high error rate. Furthermore, Udupa et al. [25] utilize static path feasibility analysis to determine whether an execution path is feasible. Note that their approaches are still based on detection of invariant features such as infeasible branches, so they cannot detect the dynamic opaque predicates.

Currently, the state-of-the-art work on opaque predicate detection is LOOP [19], a logic oriented opaque predicate detection tool for obfuscated binary code. The authors propose an approach based on symbolic execution and theorem proving techniques to automatically detect static and dynamic opaque predicates. When detecting invariant opaque predicates, LOOP perform symbolic execution on an execution trace and check whether one branch condition is always true or false. Furthermore, it runs an logic implication check to decide whether one predicate is a contextual opaque predicate.

Particularly, LOOP is also able to detect the conventional dynamic opaque predicate. The detection is based on the fact that each predicate in a dynamic opaque predicates is semantically equivalent, or in another word, they implies each other logically, such as p and q in Fig. 1(b). Therefore, LOOP performs two implication check on two adjacent predicates. One is on the execution trace and the other is on the execution with the inverted path condition. Taking the example in Fig. 1, LOOP decides it is a dynamic opaque predicate when $p \Rightarrow q$ and $\neg p \Rightarrow \neg q$ both hold. By this approach, LOOP is able to check the conventional dynamic opaque predicate.

However, to our knowledge, LOOP still utilizes the limitation of conventional dynamic opaque predicates that all predicates should be adjacently injected into a straight-line code. When testing $\neg p \Rightarrow \neg q$, LOOP first generates a new trace by negating the path condition p. Then it tests whether the next predicate in the new trace is equivalent to $\neg q$. If so, LOOP further checks whether the new trace is semantically equivalent to the original trace. This procedure is very time consuming, which limits LOOP's searching capacity. Therefore, LOOP's heuristic is only checking two adjacent predicates such as p and q in Fig. 1. In the following sections, we present that our method overcome the limitation of existing conventional dynamic opaque predicate and lead to LOOP's poor detection ratio on our generalized dynamic opaque predicates.

3 Generalized Dynamic Opaque Predicates

In this section, we present the details of the generalized dynamic opaque predicates method. First, we introduce the concept of correlated predicate. After that, we explain how to insert generalized dynamic opaque predicates into straight-line programs, branches and loops.

3.1 Correlated Predicates

Correlated predicate, as briefly discussed in Section 2.2, is a basic concept in dynamic opaque predicate. In this section, we present the formal definition of correlated predicate. First we need to define *correlated variables*. *Correlated variables* is a set of variables that are always evaluated to the same value in any program execution. One common example of correlated variables is the aliases of the same variable, like the pointers in C or the references in C++ or Java.

Correlated predicates are a set of predicates that are composed of correlated variables and have a fixed relation of their true value. The fixed relation means that, given a set of correlated predicates, if the true value of one of them is given, all other predicates' true value are known. Usually, it is intuitive to construct correlated predicates using correlated variables. Table 1 shows some examples of correlated predicates. The integer variables `x`, `y` and `z` in the first column is the correlated variables (CV). The CP_1, CP_2 and CP_3 columns show three sets of different correlated predicates.

Here we take the CP_2 column as an example to show how correlated predicates work. First, since `x`, `y` and `z` are correlated integer variables, they are always equivalent. There are three predicates in CP_2, `x%2 == 1`, `y%2 == 0` and `z%2 == 1`. Note that `x`, `y` and `z` are integer variables, so they are either even or odd. Therefore, given the true value of any one of these predicates, we can immediately get the others' true values. Furthermore, it is not necessary that correlated predicates have similar syntax form. We can use semantically equivalent operations to create correlated predicates. CP_3 shows such an example. Although the syntax of each predicate is different from others, they still meet the definition of correlated predicates.

Table 1: Examples of correlates predicates.

CV	CP_1	CP_2	CP_3
x	`x > 0`	`x%2 == 1`	`x+x > 0`
y	`y > 0`	`y%2 == 0`	`2*y <= 0`
z	`z <= 0`	`z%2 == 1`	`z<<1 > 0`

One problem we need to pay attention to is that the value of the correlated variables should not be changed during the dynamic opaque predicates, which ensures that every correlated variable are evaluated to the same value in all dynamic opaque predicates in one execution. Therefore, we compute the def-use chain inside a function and choose the section between two definitions of a variable as the candidate to be obfuscated. Note that pointer access operations could still cause the variable's value changes. Our solution is performing a simple alias analysis to decide whether the pointer is an alias of the variable. If not, we can include the pointer access instructions inside the dynamic opaque predicates; otherwise not. Since alias analysis is complicated and difficult, we only run a light-weighted address-taken algorithm [12] in our implementation. It is flow-insensitive and context-insensitive. If the analysis cannot tell whether the pointer is an alias of the correlated variable, we will conservatively consider that it could point to the variable and exclude it from the dynamic opaque predicates candidates.

3.2 Straight-line Code

In this section, we present how to insert dynamic opaque predicate into a straight-line code. Before digging into the details, we first explain the symbols in the figures as follows.

1. A *rectangle* is a basic block.
2. A *number* in a rectangle represents one instruction.
3. A *circle* indicates a correlated predicate.
4. An *arrow* between two basic blocks indicates the control flow transfer. Typically, it is a conditional or unconditional jump. If there is only one arrow between two blocks, it is an unconditional jump; otherwise, it is a conditional jump.

Given the definition above, Fig. 2(a) shows a straight-line code which contains only one basic block, in which there are five sequential instructions. If a straight-line code comprises multiple basic blocks which are connected by unconditional jumps, it can be merged into one basic block. So for the ease of understanding, we use the single basic block example to present straight-line code.

When inserting dynamic opaque predicates into straight-line code, we have two strategies, depth-first and breadth-first, whose obfuscation result is shown in Fig. 2(c) and Fig. 2(e). Here we introduce the depth-first style first and briefly discuss the breadth-first later since they are similar. When inserting dynamic

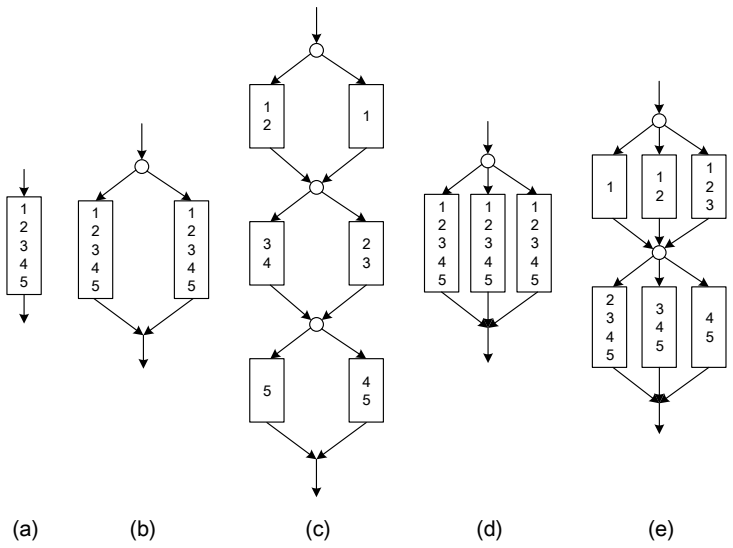


Fig. 2: Dynamic opaque predicate insertion in straight-line code.

opaque predicates in depth-first style, we select the first correlated predicate and then make a copy of the original basic block, as shown in Fig. 2(b). After that, the two basic blocks are split at different locations so as to create two chains of basic blocks in which each basic block are different with each other. At last, we insert other correlated predicates to ensure that the control flow takes either all left branches or all right branches.

Furthermore, when inserting depth-first dynamic opaque predicates, we could insert as many correlated predicates as we can by splitting the basic blocks at different locations. As shown in Fig. 2(c), those basic blocks constitute two chains, in which the execution flow will either take every left branch or right branch. We call the basic block sequence that consists of all left or right branches an *opaque trace*. In this paper, the multiple execution traces caused by the effect of opaque predicates are called *opaque trace*. As shown in Fig. 2(c), if the execution flow takes all the left branches, the opaque trace is `[1 2]->[3 4]->[5]`. Similarly, when taking all right branches, the opaque trace is `[1]->[2 3]->[4 5]`. Therefore, Fig. 2(c) contains two opaque traces.

Generally speaking, the steps to insert depth-first dynamic opaque predicates to a single basic block BB are described as follows.

1. Select a correlated variable and creating the first correlated predicate accordingly.
2. Clone a new basic block BB' from BB.

3. Split BB and BB' at different locations to create two sequences of basic blocks, or say, two opaque traces T_1 and T_2:

$$T_1 = BB_1 \rightarrow BB_2 \rightarrow \cdots \rightarrow BB_n$$
$$T_2 = BB'_1 \rightarrow BB'_2 \rightarrow \cdots \rightarrow BB'_n$$

4. Create and insert the remaining $n-1$ correlated predicates.
5. Insert conditional or unconditional jumps into the end of each basic block to create the correct control flow.

The other strategy is breadth-first inserting dynamic opaque predicates. It create more opaque traces via correlated predicates that have multiple branches. The inserting process is similar as depth-first. Assuming each predicate has three branches, the first step is to select and insert the first correlated predicate and create two copies of the original basic block as shown in Fig. 2(d). Then split the three basic blocks at different offsets so as to create three opaque traces. At last, insert the other correlated predicates and other jump instructions to adjust the CFG. The result is shown in Fig. 2(e).

Furthermore, we can easily create more complicated generalized dynamic opaque predicates by iteratively applying depth first and breadth first injection. For example, the basic block `[1,2,3]` can also be split to create a depth first generalized dynamic opaque predicate. Note that it naturally breaks the adjacency of the two predicates in Fig. 2(e). Being compared with the conventional dynamic opaque predicate shown in Section 2.2 which only has two adjacent predicates p and q, our method can insert more generalized and non-adjacent dynamic opaque predicates in straight-line code.

3.3 Branches

In the previous section, we present the approach to inserting dynamic opaque predicates into straight-line code. However, real world programs also consist of other structures such as branches and loops. When considering inserting dynamic opaque predicates into branches or loops, one straight forward idea is only inserting dynamic opaque predicates into basic blocks independently by treating them as straight-line code. However, this idea has one obvious problem: it doesn't spread the dynamic opaque predicates across the branch or loop condition, so essentially it is still the same as what we have done in Section 3.2.

In this section, we describe the process to insert dynamic opaque predicates into a branch program, which improves the program obfuscation level. For the ease of presenting our approach, we consider the branch program which contains three basic blocks as shown in Fig. 3(a). Our solution can also be applied to more complicated cases such as each branch contains multiple basic blocks. As shown in Fig. 3, `Cond` is the branch condition. BB_1 is located before the branch condition. BB_2 is the true branch and BB_3 is the false branch.

As the first step of inserting branch dynamic opaque predicate, we backwards search for an instruction that is independent from all instructions until

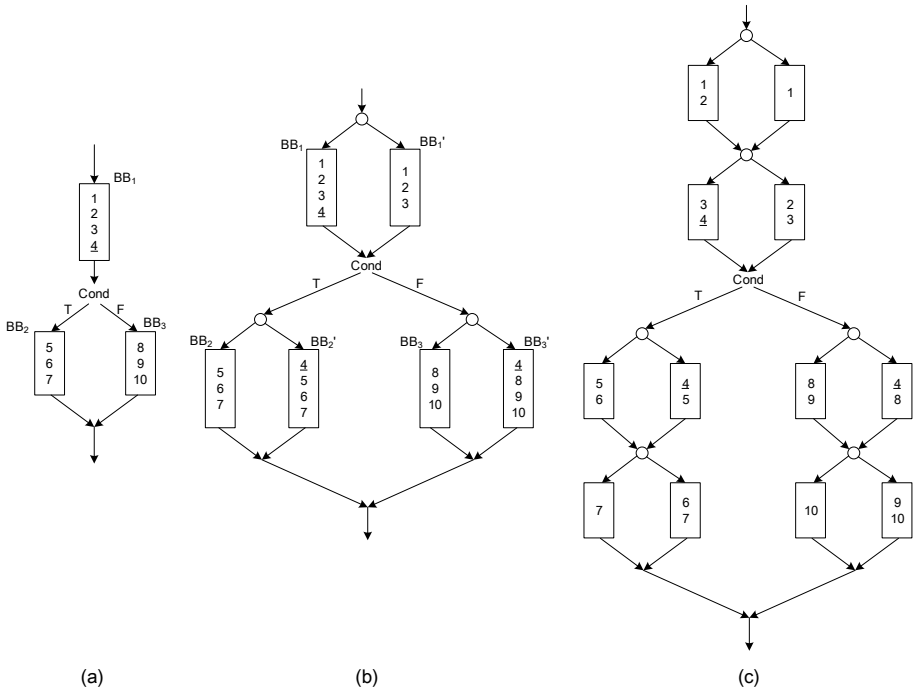


Fig. 3: Dynamic opaque predicate insertion in a branch program.

the branch condition, and also independent from the branch instruction. In this paper, this instruction is called a *branch independent instruction*. Essentially, it can be moved across the branch condition so as to create the offset in different opaque traces. In Fig. 3(a), the underlined instruction 4 is a branch independent instruction. Based on our observation, there are plenty of branch independent instructions. For example, the Coreutils program `ls` contains 289 branch conditions, in each of which we find at least one branch independent instruction. Typically, these instructions prepare data which are used both in the true and false branch.

After identifying the branch independent instruction, we select and insert the correlated variables, then make a copy of each basic blocks. Moreover, we move the instruction 4 along the right opaque trace across the branch condition and Fig. 3(b) shows the result. Note that due to instruction 4 is branch independent, so moving it to the head of basic blocks in the branches will not change the original program's semantics. At last, we create straight-line code dynamic opaque predicates for BB_1, BB_2 and BB_3. The final result of the obfuscated CFG is shown in Fig. 3(c). We briefly summarize the steps of inserting dynamic opaque predicates into a branch program as follows.

1. Find the branch independent instruction in BB_1.

2. Select and insert the correlated predicates.
3. Clone BB_1, BB_2 and BB_3 as BB'_1, BB'_2 and BB'_3.
4. Move the branch independent instruction from BB'_1 to BB'_2 and BB'_3.
5. Split basic blocks and create dynamic opaque predicates as in straight-line code.

3.4 Loops

Previous sections present the details about how to insert dynamic opaque predicates into straight-line code and branch programs. In this section, we consider inserting dynamic opaque predicates into a loop. In this paper, a loop refers to a program which contains a backward control flow, such as Fig. 4(a). BB_1 is the first basic block of the loop body and BB_2 is the last one. The dashed line indicates other instructions in the basic block. The dashed arrow means other instructions in the loop body, which could be a basic block, branch or even another loop. Particularly, if there is only one basic block in the loop body, BB_1 and BB_2 refer to the same basic block.

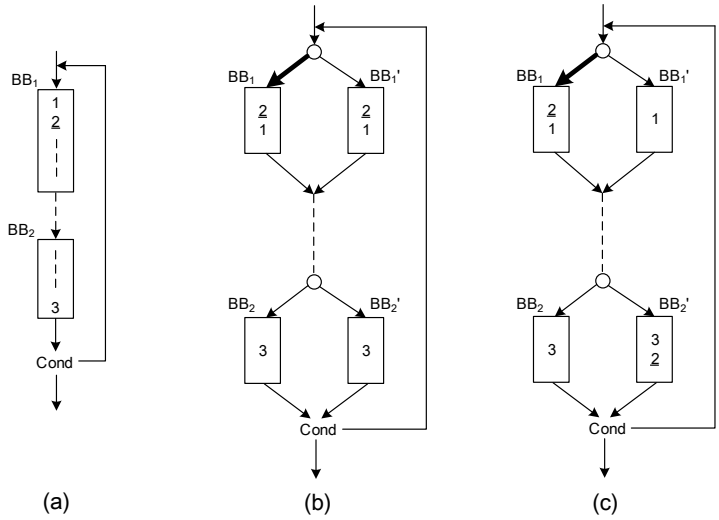


Fig. 4: Dynamic opaque predicate insertion in a loop.

The key idea in inserting dynamic opaque predicates to a loop program is finding a *loop independent instruction* and moving it across the loop condition in the same opaque trace. We define *loop independent instruction* as an instruction in a loop whose operands are all loop invariants. Loop invariant is a classical concept in compiler optimization. A variable is called loop invariant if its value never changes no matter how many times the loop is executed. For instance,

Fig. 5 shows a loop invariant. The variable `m` is defined outside the loop and is never changed inside the loop. Each iteration of the loop accesses the same array element `A[m]` and assigns it to the variable `x`. Therefore, `m`, `A[m]` and `x` are loop invariant. Note that here we use the C source code to present the idea. Actually we are working on the compiler IR level, where every instruction is close to a machine instruction. As a result, in the IR level, all instructions that only operate the loop invariants are loop independent instructions. For instance, the instruction that load the value of `A[m]` from memory to `x` is an loop independent instruction. Based on our observation, there are plenty of loop independent instructions inside a loop body, such as the instructions to compute a variable's offset address. In the experiment, we find at least loop independent instruction for each of the 61 loops in the Coreutils program `ls`.

```
1 for (i = 0; i < 10; i++) {
2     x = A[m];    /* loop invariant */
3     B[i] = x * i;
4 }
```

Fig. 5: An example of loop invariants.

In traditional compiler optimization, the loop independent instructions are extracted out of the loop body so as to reduce the loop body size and further improve the runtime performance. All compiler frameworks implement a data flow analysis to analyze and identify the loop invariants. In this paper, we take advantage of the loop independent instructions to create the offset between opaque traces. Consider the example shown in Fig. 4(a). First, we search and identify that instruction 2 is a loop independent instruction. Second, we lift the instruction 2 to the beginning of the loop body, since other instructions might need the output of instruction 2. Then we make copies of BB_1 and BB_2 as BB'_1 and BB'_2. After that we select the correlated predicates and initialize the first one to ensure that it takes the left branch. The bold arrow in Fig. 4(b) indicates the initialized predicates. We will soon discuss the reason. At last, the loop independent instruction 2 is moved from BB'_1 to BB'_2 and the final result is shown in Fig. 4(c). We summarize the steps of creating loop dynamic opaque predicates as follows.

1. Find the loop independent instruction I_i.
2. Lift I_i to the beginning of the loop body in BB_1.
3. Select the correlated predicates and initialize the first one correctly.
4. Clone BB_1 and BB_2 as BB'_1 and BB'_2.
5. Remove I'_i from BB'_1 and add it to the end of BB'_2.
6. Add dynamic opaque predicates as separate basic blocks and according jumps to build correct control flow.

Note that at the third step, we initialize the correlated variables so as to ensure the control flow goes to the left branch at the first iteration. The reason is that we have to make the loop invariant instructions executed at least once at the first iteration of the loop in order to assure all loop invariants loaded, computed and stored correctly. The value of correlated variables may change during the dashed part of the loop body so as to divert the execution flow to each opaque trace. Particularly, when the execution reaching the last iteration of the loop, there is a redundant instruction 2 if the execution follows the right branch. Since instruction 2 is loop independent, it doesn't affect the semantic of the program execution.

4 Implementation

Our implementation is based on Obfuscator-LLVM [13], an open source fork of the LLVM compilation suite that aims to improve the software security via code obfuscation and tamper-proofing. The architecture of our system is shown in Fig. 6. The generalized dynamic opaque predicate obfuscator (GDOP obfuscator) is surrounded with dashed lines. Basically, our automatic GDOP obfuscator works as a pass in LLVM framework. The workflow contains three steps. First, the LLVM frontend Clang read the source code and translate it into LLVM IR. Second, GDOP obfuscator reads the IR and inserts generalized dynamic opaque predicates to the appropriate location. At last, the LLVM backend outputs the executable program based on the obfuscated IR files.

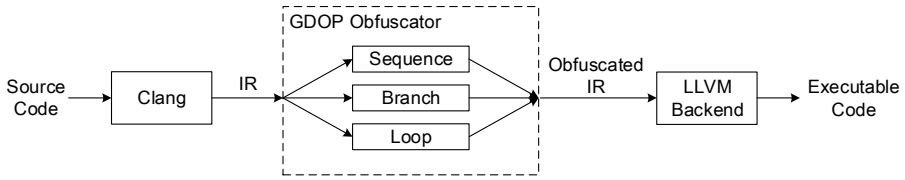


Fig. 6: The architecture of dynamic opaque predicate obfuscator.

Particularly, we implement the procedure of inserting generalized dynamic opaque predicates to a straight-line, branch and loop program as three separate passes, which includes 1251 lines of C++ code in total. We also write a driver program to invoke the three passes so as to insert all kinds of generalized dynamic opaque predicates. In addition, we implement a junk code generator to insert useless code into functions, such as redundancy branches and extra dependencies. Moreover, we provide a compiler option for users to configure the probability for inserting generalized dynamic opaque predicates. For each basic block, our obfuscator generates a random number between zero and one. If the number is smaller than the given probability, it tries to insert generalized dynamic opaque predicates into the basic block; otherwise it skips the basic block.

5 Evaluation

We conduct our experiments with several objectives. First, we want to evaluate whether our approach is effective to obfuscate control flow graph. To this end, we measure control flow complexity of GNU Coreutils with three metrics. We also test our tool with a commercial binary diffing tool which is based on control flow graph comparison. Last but not least, we want to prove our approach can defeat the state-of-the-art deobfuscation tool. Our testbed consists of an Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory, running Ubuntu Linux 12.04 LTS. We turn off other compiler optimization options by using `-g` option.

5.1 Obfuscation Metrics with Coreutils

This section shows our evaluation result of inserting generalized dynamic opaque predicates into the GNU Coreutils 8.23. Since the generalized dynamic opaque predicate is an intra-procedural obfuscation [22], we evaluate it by comparing the control flow complexity of the modified function before and after the generalized dynamic opaque predicate obfuscation. In this experiments, we choose five hot functions in the Coreutils program set by profiling. At the same time, we make sure all the functions containing at least ten basic blocks[1]. After profiling, the five hot functions we select are as follows.

1. `get_next`: This function is defined in `tr.c`. It returns the next single character of the expansion of a list.
2. `make_format`: This function is defined in `stat.c`. It removes unportable flags as needed for particular specifiers.
3. `length_of_file_name_and_frills`: This function is defined in `ls.c` for counting the length of file names.
4. `print_file_name_and_frills`: This function is also defined in `ls.c`. It prints the file name with appropriate quoting with file size and some other information as requested by switches.
5. `eval6`: This function is defined in `eval6.c` to handle sub-string, index, quoting and so on.

The metrics that we choose to show the CFG complexity are the number of CFG edges, the number of basic blocks and the cyclomatic number. The cyclomatic number is calculated as $e - n + 2$ where e is the number of CFG edges and n is the number of basic blocks. The cyclomatic number is considered as the amount of decision points in a program [9] and has been used as the metrics for evaluating obfuscation effects [7]. We first insert generalized dynamic opaque predicates into the hot functions with two different probability level: 50% and 100%. After that, we perform functionality testing to make sure our obfuscation is semantics-preserving. Table 2 shows the obfuscation metrics of the original clean version and the obfuscated version. The data shows that our dynamic opaque predicate obfuscation can significantly increase the program complexity.

[1] We do not consider dynamic link library functions because our approach takes the target program source code as input.

Table 2: Obfuscation metrics and BinDiff scores of hot functions in Coreutils.

Function	# of Basic Blocks			# of CFG Edges			Cyclomatic Number			Bindiff Score	
	Orig.	50%	100%	Orig.	50%	100%	Orig.	50%	100%	50%	100%
1	43	171	229	62	258	338	21	89	111	0.05	0.02
2	20	75	105	30	114	158	12	41	55	0.02	0.01
3	30	94	120	49	141	177	21	49	59	0.02	0.02
4	46	138	208	80	220	320	36	84	114	0.04	0.01
5	76	272	376	117	425	573	43	155	199	0.05	0.02

To test the control flow graph after our obfuscation is heavily cluttered, we also evaluate our approach with BinDiff[2], which is a commercial binary diffing tool by measuring the similarity of two control flow graphs. We run BinDiff to compare the 50% and 100% obfuscated versions with the original five programs and the similarity score is presented in the fifth column in Table 2. The low scores indicate that the obfuscated program is very different from the original version.

5.2 Resilience

In this experiment, we evaluate the resilience to deobfuscation by applying LOOP [19], the latest formal program semantics-based opaque predicate detection tool. The authors present a program logic-based and obfuscation resilient approach to the opaque predicate detection in binary code. Their approach represents the characteristics of various opaque predicates with logical formulas and verifies them with a constraint solver. According to the authors, LOOP is able to detect various opaque predicates, including not only simple invariant opaque predicates, but also advanced contextual and dynamic opaque predicates.

In our evaluation, we run two round of 100% obfuscation on the five Coreutils functions and use LOOP to check them. The results are presented in Table 3.

Table 3: The result of LOOP detection.

Function	Straight Line DOP			Branch DOP			Loop DOP		
	Total	Detected	Ratio	Total	Detected	Ratio	Total	Detected	Ratio
1	52	3	5.77%	21	0	0.00%	8	0	0.00%
2	28	2	7.14%	15	0	0.00%	6	0	0.00%
3	27	2	7.41%	23	0	0.00%	6	0	0.00%
4	54	5	9.26%	26	0	0.00%	8	0	0.00%
5	82	8	9.76%	52	0	0.00%	14	0	0.00%

As shown in Table 3, LOOP can detect very few number of the generalized dynamic opaque predicates inserted in straight-line code but fails to detect all

[2] http://www.zynamics.com/bindiff.html

those in branches and loops. We look into every generalized dynamic opaque predicate that is detected by LOOP and find that they are all conventional adjacent dynamic opaque predicates. We also check verify that LOOP fails to detect the remaining generalized dynamic opaque predicates.

We carefully analyze LOOP's report and find several reasons that lead to LOOP's poor detection ratio on generalized dynamic opaque predicates. First, iterative injection causes LOOP fails to detect majority of the generalized dynamic opaque predicates in straight-line code. Our obfuscation method can be iteratively executed on a candidate function, which means we are able to insert generalized dynamic opaque predicates into the same function several times. Note that each time we choose different correlated variables and different correlated predicates. Therefore, the generalized dynamic opaque predicates that are inserted by the later pass will break the adjacency of those inserted by the previous pass. In addition, junk code injection is another reason that prevents LOOP's detection.

Second, generalized dynamic opaque predicates spread across the branch or loop structure so they naturally break the adjacency property, which causes LOOP detects none of the generalized dynamic opaque predicates in branches and loops. For example, when we execute the loop shown in Fig. 4, there are two correlated but not adjacent predicates. They are separated by the instructions in the dashed line and the loop condition. Therefore, the detection method in the LOOP paper fails to detect the generalized dynamic opaque predicates.

5.3 Cost

This section presents the cost evaluation of our generalized dynamic opaque predicate obfuscation. We evaluate the cost from two aspects: binary code size and execution time. For binary code size, we measure and compare the number of bytes of the compiled programs that contain the five hot functions. For instance, we compare the size of `tr`'s binary code when inserting generalized dynamic opaque predicates to function `get_next` with different probabilities such as 50% and 100%. For the evaluation of execution time, we record and compare the execution time of clean version and the obfuscated program. We configure the switches and input files so as to ensure the control flow touches the obfuscated function.

Table 4: Cost evaluation of the dynamic opaque predicate obfuscation.

Function	Program	Binary Size (Bytes)				Execution Time (ms)		
		Orig.	50%	100%	Ratio	Orig.	50%	100%
1	`tr`	132,084	132,826	133,491	0.53%	2.2	2.2	2.4
2	`stat`	210,864	211,355	211,710	0.20%	4.0	4.0	4.1
3	`ls`	350,076	350,916	351,527	0.21%	23.2	23.4	23.7
4	`ls`	350,076	351,083	351,742	0.24%	23.2	23.3	23.8
5	`expr`	129,696	130,836	131,409	0.66%	0.6	0.6	0.6

Table 4 shows the evaluation result. We can observe that our approach slightly increases the binary code size, which is less than 0.7%. Moreover, according to our experiments, the generalized dynamic opaque predicates have a small impact on program performance. The execution time of most programs stays the same when inserting generalized dynamic opaque predicates with 50% probability and increases a little when inserting with 100% probability.

5.4 Case Study

As mentioned in Section 5.2, our generalized dynamic opaque predicates can be iteratively apply to a candidate program so as to create more obfuscated result. In this section, we provide a case study to show the result of generalized dynamic opaque predicate obfuscation iteration.

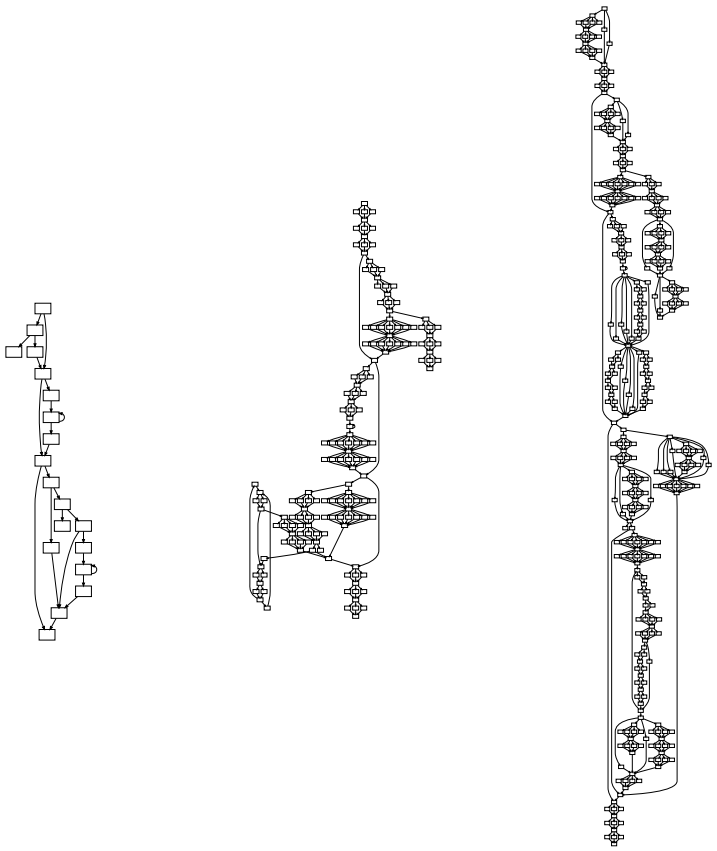
Table 5: Obfuscation metrics of `sort_files`.

Function	# of Basic Blocks			# of CFG Edges			Cyclomatic Number		
	Orig.	Round 1	Round 2	Orig.	Round 1	Round 2	Orig.	Round 1	Round 2
`sort_files`	19	160	405	25	255	539	8	97	136

The target function is the `sort_files` function in the `ls` program. We choose this function since its CFG size is appropriate and it contains straight-line codes, branches and loops, which are suited for inserting all three categories of generalized dynamic opaque predicates. We perform two rounds of generalized dynamic opaque predicate obfuscation with 100% probability. Table 5 presents the same measures as shown in the last section and Fig. 7 shows the result CFG. Fig. 7(a) shows the original CFG of `sort_files` Fig. 7(b) presents the CFG after the first round of dynamic opaque predicate obfuscation. Next, we perform another round of dynamic opaque predicate obfuscation on (b) and the result is shown in Fig. 7(c). The comparison of the three CFGs clearly indicates that our generalized dynamic opaque predicate obfuscation can significantly modify the intra-procedural control flow graph.

6 Conclusion

Opaque predicate obfuscation is a prevalent control flow obfuscation method and has been widely applied both in malware and benign software protection. Dynamic opaque predicate obfuscation is regarded as a promising method since the predicate values may vary in different executions and thus make them more resilient to detection. However, little work discusses the systematical design of dynamic opaque predicates in detail. Also, some recent advanced deobfuscation tools utilize certain specific properties as ad hoc heuristics to detect dynamic opaque predicates. In this paper, we present generalized dynamic opaque predicates to address these limitations. Our method automatically inserts dynamic



(a) The original CFG. (b) The CFG after one round of obfuscation. (c) The CFG after two rounds of obfuscation.

Fig. 7: Comparison between CFGs after different rounds of dynamic opaque predicate obfuscation.

opaque predicates into common program structures and is hard to be detected by the state-of-the-art formal program semantics-based deobfuscation tools. The experimental results show the efficacy and resilience of our method with negligible performance overhead.

Availability

To better facilitate future research, we have released the source code of our dynamic opaque predicate obfuscation tool at `https://github.com/s3team/gdop`.

Acknowledgements

We thank the anonymous reviewers 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) grants N00014-13-1-0175 and N00014-16-1-2265.

References

1. Arboit, G.: A method for watermarking Java programs via opaque predicates. In: Proceedings of 5th International Conference on Electronic Commerce Research (ICECR-5) (2002)
2. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06) (2006)
3. Bruschi, D., Martignoni, L., Monga, M.: Code normalization for self-mutating malware. IEEE Security and Privacy 5(2) (2007)
4. Cappaert, J., Preneel, B.: A general model for hiding control flow. In: Proceedings of the 10th Annual ACM Workshop on Digital Rights Management (DRM'10) (2010)
5. Chen, H., Yuan, L., Wu, X., Zang, B., Huang, B., Yew, P.c.: Control flow obfuscation with information flow tracking. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42) (2009)
6. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. rep., The University of Auckland (1997)
7. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'98) (1998)
8. Collberg, C., Myles, G., Huntwork, A.: Sandmark–a tool for software protection research. IEEE Security and Privacy 1(4), 40–49 (July 2003)
9. Conte, S.D., Dunsmore, H.E., Shen, V.Y.: Software engineering metrics and models. Benjamin-Cummings Publishing Co., Inc. (1986)
10. Coppens, B., De Sutter, B., Maebe, J.: Feedback-driven binary code diversification. ACM Transactions on Architecture and Code Optimization (TACO) 9(4) (Jan 2013)
11. Drape, S.: Intellectual property protection using obfuscation. Tech. Rep. RR-10-02, Oxford University Computing Laboratory (2010)
12. Hind, M., Pioli, A.: Which pointer analysis should i use? In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00). pp. 113–123. ACM (2000)
13. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM - software protection for the masses. In: Proceedings of the 1st International Workshop on Software PROtection (SPRO'15) (2015)
14. Kovacheva, A.: Efficient Code Obfuscation for Android. Master's thesis, University of Luxembourg (2013)
15. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated software diversity. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14) (2014)

16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04) (2004)
17. Madou, M.: Application Security through Program Obfuscation. Ph.D. thesis, Ghent University (2007)
18. Madou, M., Van Put, L., De Bosschere, K.: 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)
19. Ming, J., Xu, D., Wang, L., Wu, D.: LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15) (2015)
20. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07) (December 2007)
21. Myles, G., Collberg, C.: Software watermarking via opaque predicates: Implementation, analysis, and attacks. Electronic Commerce Research 6(2), 155 – 171 (April 2006)
22. Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., Zhang, Y.: Experience with software watermarking. In: Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00) (2000)
23. Preda, M.D., Madou, M., Bosschere, K.D., Giacobazzi, R.: Opaque predicate detection by abstract interpretation. In: Proceedings of 11th International Conference on Algebriac Methodology and Software Technology (AMAST'06) (2006)
24. Roundy, K.A., Miller, B.P.: Binary-code obfuscations in prevalent packer tools. ACM Journal Name 1, 21 (2012)
25. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: Reverse engineering obfuscated code. In: Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05) (2005)
26. Wang, C., Hill, J., Knight, J.C., Davidson, J.W.: Protection of software-based survivability mechanisms. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01) (2001)