Code Obfuscation Techniques for Software Protection

Lixi Chen

Department of Computer Science

University of Auckland

Auckland, New Zealand

lche059@ec.auckland.ac.nz

Abstract

An important security problem is to protect software against malicious host attacks. Since the malicious hosts are responsible for the program's execution, there seems little the program can do to protect itself from disclosure, tampering and incorrect execution [7]. This paper will review some existing code obfuscation techniques for protecting software against those attacks. We will focus our attention on fending off reverse engineering attacks based on static analysis. The computational complexity of some of those techniques will also be briefly discussed.

1. Introduction

Protecting the intellectual property contained in software against unauthorized access has been a central issue in computer security. The malicious host problem is a special case of this general problem. In a malicious host problem, client code is sent to and executed in a malicious host. So such a host can attack the client codes every way it can and there is nothing the owner of client code can do to stop such an attack. Possible attacks include software piracy attack, malicious reverse engineering attack and tampering attack [3]. In a software piracy attack, an attacker makes illegal copies of client code and then resells them. An attacker can also reverse engineer the client code to obtain some secret information contained in the code which belongs to the owner of code only. For example, the client code may contain an algorithm that will give its owner edge over his competitors and thus is a trade secret. His competitors can reveal this secret via reverse engineering attack. Tampering attack refers to an attack in which an attacker extracts, modifies, or otherwise tamper with the secret information contained in the client code [3].

In this paper, we will concentrate on how to protect client code against the second type of attacks, namely the reverse engineering attack. As mentioned above, client code is dispatched to and executed in malicious hosts. So the owner of client code does not have any control over what the destination host can do with its code. Fortunately, although we are not able to stop those attacks, there are ways to make it difficult enough for those attacks to achieve the desired success. Code obfuscation has been the major protection mechanism against reverse engineering attacks. To see how code obfuscation can be useful, we must first understand how reverse engineering works. A reverse engineering attack is basically a process that reverses a piece of software in its executable binary form back into the source code that it is written in [1]. It inevitably involves studying the control flow and data flow of the program. This information can be acquired through static analysis of program. “Static analysis refers to techniques designed to extract information from a static image of a computer program [6].” There are other ways to acquire program information, through dynamic analysis for example. In this paper, we will restrict ourselves to deal with reverse engineering attacks based on static analysis. Now we explain the basic idea behind code obfuscation. Code obfuscation involves applying a set of obfuscating transformations

to a program, thereby rendering the transformed program unintelligible for a computer automated static analyzer but functionally equivalent to the original program. Obfuscating transformations can be categorized as lexical transformation, control transformation and data transformation [3]. Lexical transformation merely modifies the lexical structure of program, scrambling the identifiers for example, and cannot withstand any serious determined attacks. Thus we consider only two other types of transformations.

In this paper, we will review some existing code obfuscation techniques. The remainder of this paper is structured as follows: In section 2, we present Java control obfuscation using opaque predicates. In section 3, we present the other set of transformations which includes flattening the control graph, introducing aliases and data-dependent branches. In section 4, we will look at the evaluation of these obfuscating transformations, and the computational complexity of them in particular. Section 5 draws a conclusion for this paper.

2. Java Control Obfuscation Using Opaque Predicates

In this section, we discuss some Java control obfuscation techniques introduced by [2]. These techniques can be applied equally well to other languages such as C which is very similar to Java in program syntax and high level language constructs.

All of these techniques make use of opaque predicates to obfuscate the control flow of program. “A predicate P is opaque if it has some property q which is known a priori to the obfuscator, but which is difficult for the deobfuscator to deduce [2].” We will use the notation used in the original paper. For a predicate P, if it always evaluates to false, we write P^F. If it always evaluates to true, we write P^T. If P may evaluate to either true or false, we write $P^?$.

2.1 Some Transformations Based On Opaque Predicate

We first give some simple examples of opaque predicate.

```
void opaque(){
        int x = 2, y = 5;
        if (y > 10) /* P^F */
                …
        if(x * y % 2 == 0) /* P^T */
```

```
        …
    if(Math.random()*10 > y) /* P?  random is the java math library function*/

        …

}
```

We now present some transformations to illustrate the basic usage of opaque predicate.
The first transformation is to insert irrelevant opaque predicates to hide the real control flow of program. Those predicates are irrelevant since they do not contribute to the computation of program at all. For example, assume there is a block B in a program. B can be partitioned into n sub-blocks and all sub-blocks are executed in sequence. We can add an opaque predicate that always evaluates to true in the middle of those blocks. Since it always evaluates to true, so the sub-blocks after the predicate will be always executed as the original program does. But an automatic deobfuscator may not be able deduce this fact because it does not know that the predicate will always evaluate to true. An example of such predicates would be $7y^2 – 1 != x^2$ for any integers x and y [2].

The second transformation is to extend loop conditions. The idea is to add an opaque predicate P^F or P^T which does not affect the termination of the loop. Again this transformation works under the assumption that a deobfuscator cannot deduce that the predicate always evaluates to true or false.

The third transformation takes the advantage of the fact that there does not exist a one-to-one correspondence between the Java virtual machine code and the Java language. We can introduce virtual instruction sequences to which there is no correspondent construct in Java language so that a Java decompiler would have problems producing Java source code.

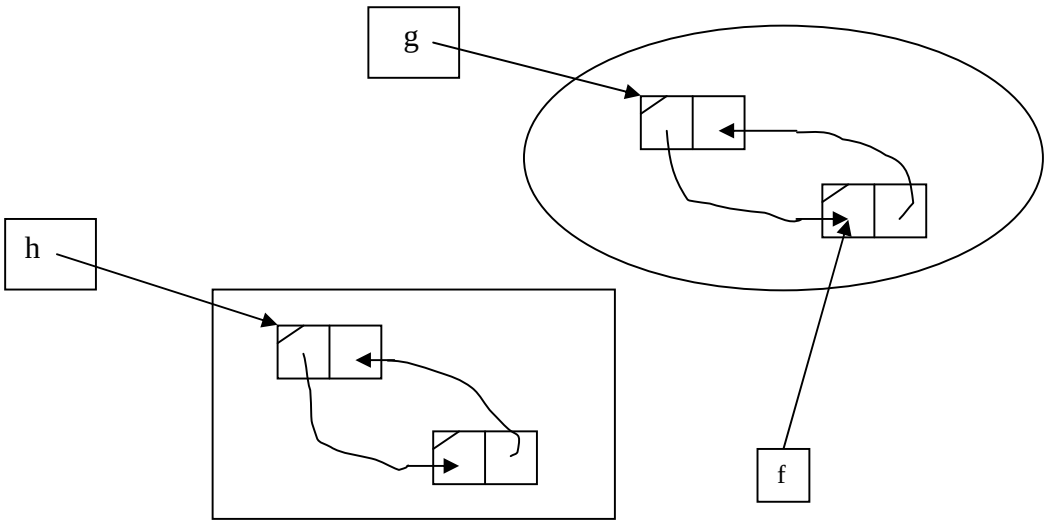
2.2 Complex Opaque Predicates

So far we have seen that the quality of our transformations is solely dependent on the quality of opaque predicates. We have introduced some simple examples of opaque predicates in the last section. However, they are far from being good. They could be broken using simple global static analysis. In this section, we will introduce some complex opaque predicates with a much higher resistance to attack.

Recall that static analysis is commonly employed by reverse engineering attacks employ to acquire program semantic information. We also know that inter-procedural static analysis is significantly complicated with the existence of alias. In fact, May alias problem is undecidable [4]. Alias refers to the situation in which two or more variables in a program point to the same memory location. The opaque predicates discussed in this section exploit this difficulty.

We first introduce a dynamic Java structure Node. A node consists of a Boolean field and two pointers fields which can point to other nodes. Then we can construct opaque predicates as follows:

1. We construct two sets of nodes. Each node (its two pointer fields) can point to other nodes in the same set. A pointer may point to the other pointer field in the same node as well. So there is no connection between the two sets. There are three global pointers into the two sets. We also provide a function "move" that updates the pointers randomly within the same set. The following example, which is the simpler version of that introduced in the original paper, illustrates the construction. Three global pointers are g, h and f. Each set has two nodes.



2. Then we can construct opaque predicates by manipulating pointers. The following code segment shows how to do it with reference to the above simple construction. Assuming the Boolean field of node is called bool. The first predicate always evaluates to false since there is no connection between the two sets and so g and h cannot possibly point to the

same node. The second predicate could evaluate to either true or false since g and f could point to the same node after applying “move”. The last two predicates are constructed using the same property.

```
Node g, h;
if (g == h) /* this is a P^F */
        …
if (g == f) /* this is a P^? */
        …
g.bool = true;
f.bool = false;
if (g.bool) /* this is a P^?*/
        …
g.bool = true;
h.bool = false;
if (g.bool) /* this is a P^T*/
        …
```

These opaque predicates pose more difficulty to a deobfuscator because they are constructed using pointers, which means there is possibility of aliases. That is, it is difficult for a static analyzer to figure out whether g, h and f point to the same node.

2.3 Discussion

We have discussed how to construct opaque predicates of high quality. The construction relies on the fact inter-procedural static analysis is very difficult with the presence of alias. Hence they can provide good protection against reverse engineering attacks that rely on static analysis. The original paper also describes how to construct opaque predicates using concurrency. The idea is to take advantage of the fact the parallel programs are more difficult to analyze statically.

3. Obfuscation By Flattening the Control Graph

In this section, we will discuss some code obfuscation techniques introduced by [6]. These techniques are designed to defeat static analysis as those discussed in the previous section.

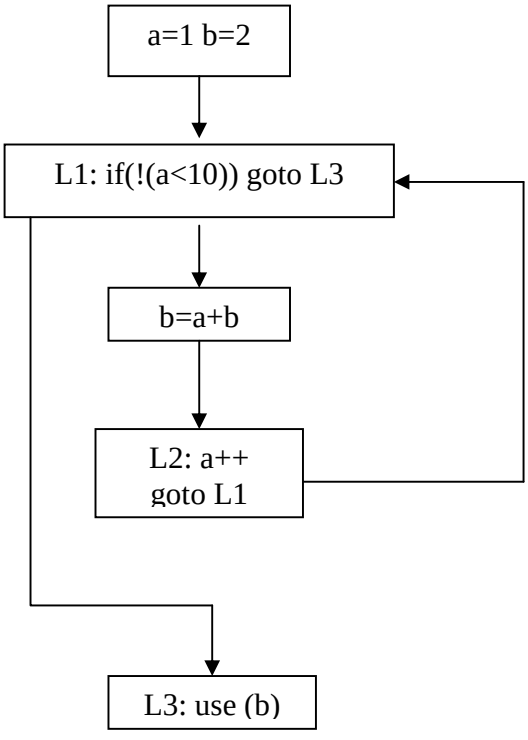
Static analysis can be either flow-sensitive or flow-insensitive. These techniques work against flow-sensitive static analysis only. Static analysis can be broken down into control flow analysis and data flow analysis. The technical basis of these transformations is to make control flow and data-flow analysis co-dependent [6].

3.1 Control Flow Transformations

In the control flow analysis phase of static analysis, a program control graph is built. The transformations we will introduce will make building such a graph difficult by "obstruct[ing] static detection of branch targets and call [6]". We illustrate these transformations through an example. Again we adapt the example in the original paper. Consider the code segment below.

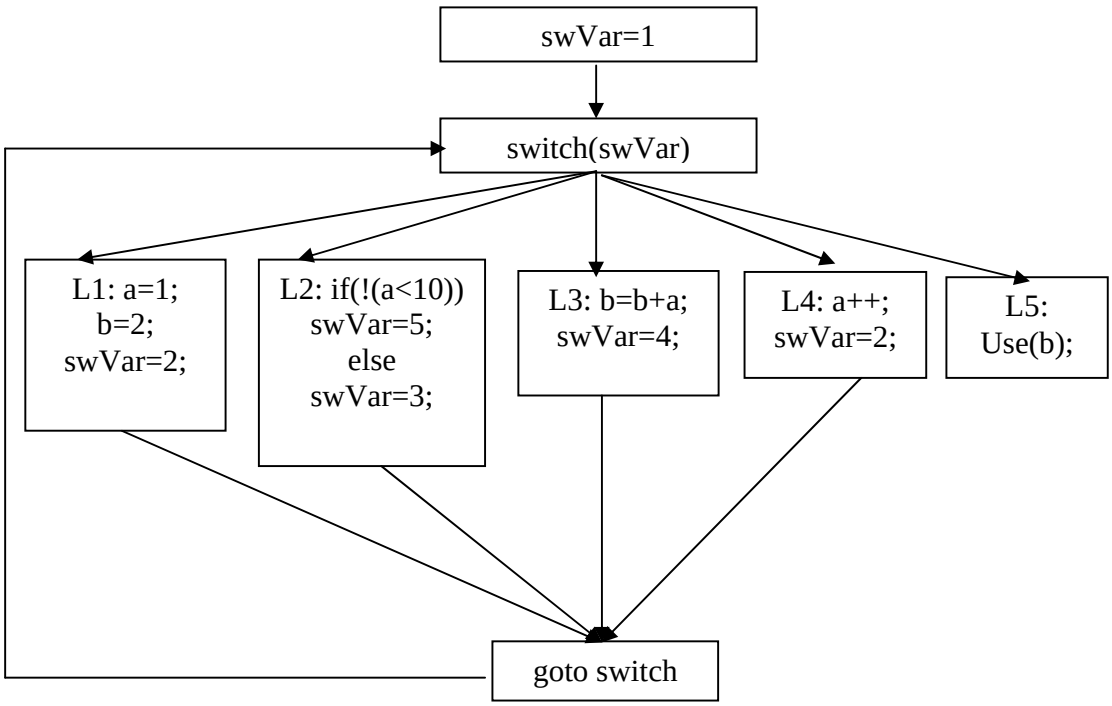
```
int a = 1, b =2;
while (a < 10) {
        b = a + b;
        a++;
}
use(b);
```

We first transform the high level code into the graph below using if-then-goto constructs.



It is obvious that a static analyzer will not have difficulty deducing branch target address hence the control flow of code since the code contains a direct jump address.

Next, we introduce the switch statement to replace the direct goto's so that the target addresses of goto's are dependent on the program data. The transformation is illustrated in the graph below. Obviously the transformed code is functionally equivalent to the original code. But now a static control flow analysis will not be able to figure out the control flow inside the switch block. Because on the entry to the switch block, identifying which sub-block to execute next requires the knowledge of swVar, which cannot be acquired statically. What a static control flow analysis can know at best is that there are 5 sub-blocks inside the switch block and "every block is potentially the immediate predecessor of every other block [6]." The new flow graph is flattened compared with the original one, thus the name "flattening the control graph".



3.2 Data Flow Transformations

To further determine the program flow inside the switch block, a static analyzer must perform the data flow analysis since the branch target is dependent on how easy to learn the definition of swVar. In this section, we introduce some transformations to further hinder the data flow analysis. Again, these transformations exploit the fundamental difficulty that static analysis must deal with, the presence of aliases. The idea is to introduce aliases that contribute to the computation of swVar thereby influencing the computation of branch targets.

Firstly, we introduce a global array whose elements may be initialized statically (known at compile time). Then we replace the constant assignments to swVar in the example above with the array elements. So a static analyzer must figure out the array values before it can decide the definition of swVar.

Secondly, we introduce a sufficient number of pointer variables. Then we assign those pointers to data variables including array elements. So we can replace computations that access data variables directly with equivalent computations that access dada variables through pointers.

To see how these transformations can confuse a static analyzer, we give an example below. Suppose in a program without flattening transformation, the following instructions will be executed in sequence. Assume they are inside a loop and therefore can be flattened later.

```
int global_array[100]; /* will be outside the switch block */
int *p,  a = 1, b = 2;     /* will be outside the switch block*/
p = &global_array[30];
a = a + b;
p = &a;
*p++;
```

It is clear to a static analyzer that the first definition of p is overwritten by the second assignment to p.

Then after applying the flattening, these instructions are split into blocks inside the switch block. Here is the possible outcome of flattening.

In block A

```
p = &global_array[30];
a = a + b;
swVar = global_array[f()]; /*where f is a function that computes the array subs*/
```

In block B

```
P = &a;
*p++;
……
swVar = global_array(f());
```

Recall that at this stage a static analyzer does not know which block will be executed first. Hence it will not be able to tell the first definition of p will be overwritten. So it will report that the elements of the global array will be updated dynamically. And, that is the result we are looking for. That is, a static analyzer cannot deduce the definition of swVar efficiently. In fact, it is an NP-hard problem, which we will discuss in the next section.

4. Evaluation of Obfuscation Transformations

So far we have discussed some code obfuscation transformations to protect client code against reverse engineering attacks based on static analysis. But we have not formally addressed a fundamental question “how well will these transformations fare against those attacks?” We mentioned that both sets of transformations exploit the difficulty posed by aliases to static analysis. We will look at some details in this section.

Two papers presents different ways to evaluate their transformations.

In [2], the authors classify four attributes of obfuscation. They are potency, resilience, stealth and cost. By potency, we mean “how much obscurity it adds to the program [2]”. By resilience, we mean “how difficult it is to break for an automatic deobfuscator [2]”. By stealth, we mean “how well the obfuscated code blends in with the rest of the program [2]”.

By cost, we mean “how much computational overhead it adds to the obfuscated application [2]”. The authors claim that opaque predicates using aliases are:

1. stealthy because the added code is very similar to normal Java code thus not easily distinguished from the rest of program.
2. resilient because the construction of opaque predicates is based on the difficult static analysis problem when there exist aliases. However, the authors did not provide formal complexity evaluation. We do not know exactly whether the static analysis problem with the presence of opaque predicates is undecidable or intractable.
3. cheap because opaque predicates are easy to construct.

In [6], Wang et al. take a mathematical and empirical approach to the evaluation of their techniques. They claim that for static analysis to defeat transformed code using their techniques is NP-hard. Formally, they present a theorem: “in the presence of general pointers, the problem of determining precise indirect branch target addresses is NP-hard [6]”. They proved this theorem. The proof idea [6] is as follows:

Firstly, we present the reduction from 3SAT problem to the above problem. Note this reduction has no correlation to previous examples.

Consider the 3SAT problem for $AND^{n}_{i=1}$ (V_{i1} OR V_{i2} OR V_{i3}) where V belongs to {v_1,…, v_m}, and v_1, …, v_m are Boolean literals that can be either true or false. In the reduction shown below, the branch target address is stored in the array element A[*true]. F1() and f2() are two functions that program flow can branch to. The if conditions are not specified since we assume that all paths are possible to be taken. V prime means the complement of v.

L1: int *true, *false, **v_1, **v_2, …, **v_m, *A[];

L2: A[*true] = &f1();

L3: if(-) {v_1 = &true; v_1’ = &false} else{v_1 = &true; v_1’ = &false}

if(-) {v_2 = &true; v_2’ = &false} else{v_2 = &true; v_2’ = &false}

…

if(-) {v_n = &true; v_n' = &false} else{v_n = &true; v_n' = &false}

L4: if(-) A[** v_{11}'] = &f2() else if(-) A[**v_{12}'] = &f2() else A[**v_{13}'] = &f2()

if(-) A[** v_{21}'] = &f2() else if(-) A[**v_{22}'] = &f2() else A[**v_{23}'] = &f2()

…

if(-) A[** v_{n1}'] = &f2() else if(-) A[**v_{n2}'] = &f2() else A[**v_{n3}'] = &f2()

L5:

Secondly, we explain how this reduction works.

We observe that at point L2, the branch target address is the address of function f1(). Now we rephrase the problem as a decision problem: does the branch target address remain the same at point L5. We feel it is easier to understand the proof this way since the 3SAT problem is a decision problem. The original problem may be viewed as an optimization or a search problem. But we know that the search and optimization problems can be reduced to their decision versions with polynomial time overhead [5]. We will show that this construction above works by mapping the yes instance of 3SAT to the yes instance of our problem and the no instance of 3SAT to the no instance of our problem. In other words, the branch target address remains the address of function f1() if and only if there is a satisfying assignment for the 3SAT.

Code segment L1 declares all the variables and an array A[].

Code segment L2 assigns the address of function f1() to the branch target address.

Code segment L3 assigns the true or false to Boolean literals. For example, v_1 = &true would assign the true to v_1 .

A satisfying truth assignment would imply that there must exist one path between L4 and L5 on which the value of A[*true] is never updated. We can find the path as follows: take the first line of code that corresponding to the first clause as an example, one of v_{11}, v_{12} and v_{13} must be true; Say v_{11} is true, then we take A[**v_{11}'] = &f2() , in which case v_{11}' is false; So A[*true] is not updated. We can carry on this procedure for every line of code until we reach L5 where A[*true] remains &f1(). Hence yes maps to yes.

An unsatisfying truth assignment would imply that there exists at least one clause in which every literal is false. Again we take the first line of code that corresponding to the first clause as an example. All of v_{11}, v_{12} and v_{13} must be false. Then it is obvious that A[*true] will be updated to &f2(). Hence a no instance maps to a no instance.

Finally, it is obvious that this reduction can be carried out in polynomial time.
Also, we observe that the decision version of the original problem is in the class NP since given a certificate (an assignment of Boolean variables), it can be decided in polynomial time whether A[*true] will be updated. So we claim that the decision version the original problem is NP-complete.

Since we know that "any optimization problem, the decision version of which is NP-complete, is itself NP-hard [5]", the proof is completed.

The authors also conducted some experiments to evaluate the performance of the transformed programs. The results show that the transformations can be costly.

Compared to the evaluation scheme used in [2], what is missing is the stealth of transformations. But we think the stealth of these transformations is probably not a cause for serious concern considering the nature of these transformations. Even if an attacker can spot the flattened blocks, the effort required to defeat the transformations will not be reduced. While for an opaque predicate, if an attacker can spot its location, a simple attempt to remove it may be rewarding if it happens to be a P^T.

5. Conclusion

We have discussed two sets of code obfuscation transformations in this paper. The first set of transformations introduces opaque predicates to obfuscate the control flow of Java programs. The second set of transformations includes flattening the control flow graph, and introducing aliases and data dependent branches. The technical basis of both sets of transformations is that precise static analysis is very difficult with the presence of aliases. In particular, statically analyzing programs transformed by the second set of

transformations has proven NP-hard. However, one should not be over optimistic about this claim. We all know SAT is NP complete. But if we restrict the number of times each literal can appear to two, then this particular instance of SAT is polynomial time solvable. Although the above general problem is NP-hard, it may require significantly less effort to statically analyze some instances of transformed programs that have some special properties. Recently, there has been some progress being made in this field. Cloakware has claimed that their code obfuscation products (cloaked programs) pose PSPACE complete challenge to attackers. This is an impressive claim. Unfortunately, we do not have enough information about the work they have done. Nevertheless, we believe the transformation of that quality is possible to achieve.

References:

1. http://whatis.techtarget.com/definition/0,289893,sid9_gci507015,00.html
2. C. Collberg, C. Thomborson, Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.
3. C. Collberg, C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection.
4. W. Landi. Undecidability of Static analysis.
5. B. Moret. The Theory of Computation.
6. C. Wang et al. Software Tamper Resistance: Obstructing Static Analysis of Programs.
7. C. Wang et al. A Security Architecture for Survivability Mechanisms.