Code Obfuscation
Final Report

Boaz Goldstein
Guy Hizkiau
Or Kadouri

Advisor: Eitan Koch

14/3/2011
Abstract
Code obfuscation is the art of hiding an application's algorithms from a malicious party while still allowing end users to use the application. These techniques are therefore somewhat like hiding a pink elephant in central park, while everyone is out in central park on a pink elephant hunt. The literature offers many methods of obfuscation, though most are either not very resilient against anyone with knowledge of the method used, or very resilient but at a high cost and in an un-stealthy manner. We sought to explore methods for code obfuscation that attempt to be both highly resilient and very stealthy. Also we were aiming to introduce a grading scale for code obfuscating methods based on the four main criteria which are recommended in the literature: potency, resilience, stealth and cost.
Literature Survey
We started from the article that appear in the project syllabus page, and by the guidance of our advisor Eitan, continued to look through articles about opaque predicates. There seemed to be several implementations to opaque predicates concept. We chose the one that we thought was most suitable, which introduced questions about data structures as part of the opaque predicates. Along the way we came across an interesting approach, which suggested using exceptions handling as part of the opaque predicates handling mechanism, we will refer to this point later on.
We also read articles about the subject of finding metrics to evaluate obfuscated code in general, and our method to obfuscate code in particular. We came to understand during this survey that this niche in computer science not yet reached its full potential and will require more study in order to get better results. A list of the articles we used can be found in the bibliography section of this report.

Original Objectives of the Project
The chosen methods of obfuscation we decided to use revolve mostly around opaque predicates. Opaque predicates are predicates, which their outcome is known by the programmer a priori, but are evaluated at run time. We sought to introduce exception handling to opaque predicates, which will make the control flow much harder to deduce and spread over different levels of the calling stack.
In the second part of the project we focused on creating a grading scale for measuring an obfuscator's "quality". This scale was to consist of four parts, which were to be given different weights in the final score. Those four criteria are:
- Potency - the degree of which a human observer is unable to understand the original algorithms.
- Resilience - the obfuscated code's ability to withstand de-obfuscation attempts by automatic obfuscators.
- Cost - how much complexity, both time and space, is added to the program.
- Stealth - the degree of with the obfuscated code blends into the original code. Initial research showed that measuring stealth is equivalent to natural language processing, and while specific instances of un-stealthiness can be detected, it is NP-hard to obtain a measure of general stealth.

What We Accomplished and How
Our goal was to build an eclipse plugin that will use the AST (Abstract Syntax Tree) of a given project to obfuscate it automatically. We used eclipse API to get this data for a given project. We continued the work by building a binary graph data structure that will suit our goals.
The graph has functionality that includes: inserting a new node, linking two existing nodes, merging two graphs, splitting a graph into two, and more.
In order to accomplish the obfuscation we introduced a unique concept which we called "contracts". A contract is a quartet of Boolean arguments giving us information about the current graph structure. Given pointers to two nodes in the graph (let's refer to them by g and h), the contract will state whether they refer to the same node, is there a path from g to h, is there a path from h to g, and whether or not g has a right son. After implementation of this concept was
accomplished, we divided the obfuscation into three sequential parts we needed to perform on the original code in order to obfuscate it:

1. **Contract Assigner.** In order to let us know what is the graph’s structure in variant locations in the code we chose random contracts for each block of the input code and made a map of blocks and their contracts. Also we needed to decide when we were to introduce new pointers to the graph and when to use such pointers that already exist. This was done by figuring out where were the possible entry points of the program. Those where the places where we needed to introduce new graph pointers, that’s because this may be the first code that runs in the project.

2. **Contract Implementer.** The implementer’s objective was to link between the contracts. For example, two consequential blocks, A and B, were assigned with contracts X and Y respectively. The implementer needs to link the exit point of block A to the entry point of block B so that the graph’s structure when entering block B will be as expected. It will do this by putting between them a set of randomly generated graph operations which a known outcome. This required the implementer to break all the expressions in the code consisting of more than a single method invocation into several expressions (further increasing the obfuscation’s potency). Another job the implementer needed to do was to add the node.java file to the given project in order to allow us to "import" it in the obfuscated files, and use the Node class.

3. **Obfuscator.** After all the contracts were assigned and implemented we can finally use them to introduce to the project opaque predicates that rely on them. The obfuscator works in one of two ways (randomly, per block):
   - The obfuscator will wrap the block with an if statement, while using the predicates in the expression. Than it will add live or dead code where need. For example, if the predicates evaluates as true, than the original code will be inserted into the than section, and dead code will be inserted into the else section. The predicates don’t have to evaluate to true or false, but can rather be evaluated as “maybe”, when asked about situations we cannot predict (such has the size of the graph). That’s because all the operations are random, and the graph can look in infinite ways
   - The obfuscator also uses the contracts to introduce new catch-try-finally blocks. By that we will introduce live and dead code in a much more sophistication, because it is much harder to determine when will an exception be thrown (we used null pointer exceptions because they are the most popular and can be thrown at almost any point of the code).

Live code is essentially the same as the original code, only it was padded with some irrelevant operations and declarations, occasionally turn a for loop into a while loop, and so on...

Also, it is far harder for the attacker to keep track of every possible line of code that might throw an exception rather than keeping track only of if statements.

In the second part of the project we found implementation to 3 of the 4 criteria mentioned before:
1. **Potency.** We used an existent statistics Eclipse plugin to measure the average nesting complexity of the code, the average cyclomatic complexity, and the code’s length. We ran all three measurements before and after the obfuscation was performed. The formula to determine the final grade of each criterion was:

$$S_{\text{new}} - S_{\text{before}}$$

$$S_{\text{before}}$$

We used this formula to evaluate our obfuscator in 4 different scenarios. See figure 1 for the results.

![Figure 1](image)

2. **Resilience.** We tested the obfuscated code against 2 de-obfuscators (JMD and JDO), they both failed to resolve our obfuscation. We believe that to improve results we can always use more methods of obfuscation to withstand more de-obfuscators.

3. **Cost.** In most of our tests the speed difference was insignificant compared to Java's load times. We did however construct a specially designed edge-case where there was a X2000 drop in speed.

**Complying With the Original Demands and Deadlines**

- We finished both the obfuscator and the code metrics in time and even added more randomly factors to the opaque predicates and code generators in order to thwart possible attacks. Such attacks might include running the code many times on a profiler machine and see which parts of code ran and which didn't (an attack Professor Eli Biham suggested during the mid-semester presentation and was discussed with our advisor).

- The original time demands as planned in the opening report were:
  
  o 06/11/2010
    
    Getting used to eclipse and development of the nesting complexity metric.
    
    *Accomplished successfully.*
  
  o 03/12/2010
    
    Development of obfuscator, creating an opaque predicate
generator and a naive method of insertion of predicates.  
*Accomplished almost successfully.*  
*There was a 2 weeks delay in this part.*

10/01/2011  
Code-complete, including all metrics and opaque predicates with exceptions.  
*Due to the unpredicted volume of cases in the java language we failed to think of in advance, it took much more time than was originally planned and led us to finish mush of the coding in the last week.*

**Ideas for Future Development**

Obfuscating code is never bullet proof, and with enough time and resources it can always be broken, but the question is price.  
If we have an algorithm we want to obfuscate, and the money and resources the attacker needs to invest in order to break those obfuscations exceeds the amount of money he would have used to hire people to create a similar algorithm, then we succeeded.  
Therefore, to further straighten our obfuscator, we could add more features to it, such as breaking and joining classes, learning the original code “common language” and using when generating dead code, and making the exception predicates span over more levels of the calling stack. If we’ll use several obfuscation methods we’ll surely make the code much more difficult to understand, and therefore more costly to de-obfuscate.
Bibliography


