Semantic Obfuscation and Software Intention

Sheryl Duggins, Frank Tsui, Orlando Karam, and Zoltan Kubanyi
Department of Computer Science and Software Engineering, Southern Polytechnic State University, Marietta, Georgia, 30060 USA

Abstract - Software protection (SP) research on intellectual property (IP) protection has primarily focused on entertainment media such as games, music and videos, using Digital Rights Management (DRM) systems [1]. Today SP research has broader goals, and includes all types of software with the aim of preventing tampering, reverse engineering and illegal redistribution. In this paper we propose an approach to protect software IP by increasing its complexity to prevent reverse engineering. We introduce four Conjectures for SP through obfuscation and provide rationale for why these four Conjectures make logical sense. We also discuss the results of an experiment verifying our conjectures.

Keywords: Obfuscation, Security, Software Protection, Software Complexity, Intellectual Property

1 Introduction

While many software solutions are provided through open source, the question of proprietary interest still remains. Much of the key intellectual property is often still hidden and protected. In this paper we explore one potential approach to protect software IP based on complexity. Software complexity [2, 3, 4, 5] has traditionally been studied with the goal of reducing complexity such that software quality and software development productivity may be improved. In this paper we propose the reverse, namely, that increasing complexity may also have some positive effects for the software industry.

With the explosion of outsourcing software development, there came the potential of one’s outsourcing partners becoming competitors; if we give them access to certain components, even with source code for some of them, what would stop them from eventually reverse-engineering those components and incorporating them into a competing product? However, the scope of potential adversaries who may try to attack the intellectual property of our software is far greater than one’s outsourcing partners. As Colberg and Thomborson [6] identified, there are three types of attack: software piracy, malicious reverse engineering, and tampering.

Software piracy is a $58.8 billion dollar per year industry [7] and is simply the illegal copying and resale of software. Reverse engineering is increasingly being utilized due to the prevalence of tools to aid in this task and the fact that software is frequently distributed in forms like Java bytecode that are easy to decompile and reverse engineer. Tampering deals with the extraction or modification of software that contains encryption keys like e-commerce applications. Therefore, the question of how can we protect the IP of our software component is becoming an issue of much greater importance [8, 1, 9, 10, 11, 12] and can be handled either legally through patents and copyright or technically through techniques like syntactic obfuscation and encryption. We show that protection through increasing the complexity via semantic obfuscation may be another avenue of protection.

This paper will begin with a brief introduction to obfuscation for security, and explore the types of obfuscation used for defenses against malicious attacks. We then present an example to illustrate our approach in the Simple Scenario section. Then we pose a number of Conjectures and demonstrate the merits of these Conjectures. Conjectures (a) and (b) are discussed in the Syntactical and Enumeration Analysis section and Conjectures (c) and (d) are discussed in the Semantic Analysis section. We then discuss the results of an experiment designed to verify our conjectures and follow with conclusions.

2 Obfuscation Overview

The term obfuscation means attempting “to transform a program into an equivalent one that is harder to reverse engineer” [6, p.737]. The earliest work on this was done two decades ago by Cohen [13], who suggested increasing the complexity of a system to a level such that the difficulty of attack is too high to be worth the effort. He called this “security through obscurity” and advocated program evolution as the technique to increase complexity. This was the first of many techniques for syntactic, or code obfuscation. Code obfuscation involves a number of transformations that change a given program into an equivalent program such that obscurity is maximized and execution time is minimized. Researchers today are still developing techniques for syntactic obfuscation, including recent work done on instruction embedding [14] and cloud protection [15]. In addition to software developers using code obfuscation to protect their products from reverse engineering, writers of viruses and malware are also using code obfuscation to hide their work from virus scanners [16]. Regardless of the technique being utilized, all researchers in SP realize that the attacker is human and therefore has
creativity and motivation, and thus will eventually be able to circumvent any defense in some period of time [1].

Our work is unique in that we consider transformations based on both the syntax and the semantics of the code. Semantic obfuscation utilizes the fact that the attacker is human, and therefore has a predefined set of background knowledge. This background will determine whether or not the true intention of a program will be discovered and the attack successful. We posit that semantic transformations will take longer to decipher and could therefore be a new direction for SP research.

3 Simple Scenario

We first describe a very simple scenario to demonstrate the approach. Suppose we are asked to write a program that will provide the sum of 1 to n consecutive integers, where n is a non-zero, positive integer. A likely pseudo-code solution, Solution A, may be the following.

1. initialize sum, n as integers of value zero
2. initialize counter as integer of value 1
3. ask for and read in the value of n
4. validity check the input value of n
5. if the input n is valid then repeatedly perform { (sum = sum + counter) and increment counter by 1} while the counter is still less than n; then print out sum and terminate
6. if the input n is invalid then issue a message and return to step 3

An alternative approach, and perhaps a less likely one, is to use a mathematical formula to calculate the sum of the numbers 1 through n as follows. Call this Solution B.

1. initialize sum, n as integers of zero value
2. ask for and read in the value of n
3. validity check the input value of n
4. if the input n is valid then compute (sum = (n * (n+1))/2); print out sum and terminate
5. if the input n is invalid then issue a message and return to step 2

Even for such a simple problem, the two solutions look very different. Solution A has 6 statements, and Solution B has 5 statements. Let the cardinality of Solution X, |X|, represent the number of statements in Solution X. Then we not only have |B| ≤ |A|, but also |A|/|B| = 6/5, showing that |A| is a 20% increase over |B|. However, this expansion may be worse than it seems. We will see later that if we consider the permutations of 5 statements versus the permutations of 6 statements, the increase is significantly more.

Now consider a third approach, using recursion, to add the numbers 1 through n. Call this Solution C.

1. ask for and read in the value of n
2. validity check the input value of n
3. if the input n is zero then return 0; print out sum and terminate
4. if the input n is not equal to zero return n + C(n-1)
5. if the input n is invalid then issue a message and return to step 1

Furthermore, note that the “crux” of the solution is in statement 5 of Solution A and is in statement 4 of Solution B. Statement 4 in Solution B is a straightforward arithmetic computational assignment followed by print and termination. On the other hand, statement 5 in Solution A involves a loop which iteratively performs an arithmetic computational assignment while a certain condition is true, then prints and terminates when that condition is no longer true. Let us look at the different types of activities involved in these two statements. Both statements have an arithmetic computation, a print, and a termination. But statement 5 of Solution A includes a loop-structural statement with a terminating condition. Thus statement 5 of Solution A includes one more computational task type, a loop structure. If we compare both of these two solutions with Solution C, we see that statement 4 of Solution C involves recursion, which is quite a bit more complicated than a simple loop structure, even though the cardinality of Solution A is greater than that of Solution C.

In terms of these syntactical measurements, cardinality of solution and number of different computational task types, Solutions A and C’s source statements, as represented with the pseudo-code, will be considered more complex. If more complex implies its intention is less likely to be discovered, then one might believe that Solution A or C is the better solution. Thus Solution A or C may be thought, by some, as more protective of the intent of the solution than Solution B. Now, let’s review what a perpetrator may do to ascertain the intent of these solutions. One obvious and commonly used procedure is to observe the behavior of the three solutions by feeding different inputs to the solutions, similar to black-box testing approach. For illustrative purpose, we will ignore the small potential problem of integer division by 2 in statement 4 of Solution B. Then all 3 solutions will behave alike in terms of what they output. Since all 3 solutions will behave alike with the same inputs, one may conclude that the intent of all solutions will either never be cracked or cracked simultaneously. Under such black-box analysis, the effort required to discover the intent behind the three solutions will essentially be equal. If the pseudo-code or the actual source code were not available, black-box approach may be the only available approach.

Consider the scenario where the source code or the pseudo-code became available (as in by reverse engineering or from a partner). Then would there be a difference in the effort expended for the discovery? According to the earlier syntactical analysis, Solution A should be more difficult, or take more effort, to analyze. On the other hand, if one is not familiar with the term (n*(n+1))/2, then statement 4 in Solution B may present quite a challenge. Similarly, for those who are unfamiliar with recursion, Solution C may be even
more challenging. We also note that the variable name, “sum”, may be a give-away. Thus we replace the variable name “sum” with some arbitrary, possibly misleading name such as “product.” The two respective statements, 5 in Solution A and 4 in Solution B, would look as follows.

5. if the input n is valid then repeatedly perform \{ (product = product + counter) and increment counter by 1\} while the counter is still less than n; then print out product and terminate

4. if the input n is valid then compute \(\text{product} = (n \times (n+1))/2\); print out product and terminate

This purposeful renaming of a variable further from its contextual intent may make statement 4 in Solution B even more difficult to analyze. The confusion introduced by this purposeful renaming in statement 5 of Solution A may not be as much as that in Solution B. But it opens a window into what semantics behind the syntax may bring, and the topic will be discussed in the section on semantic analysis.

We have illustrated several Conjectures with this simple example, and they are presented below.

(a) Cardinality of solution, in terms of number of statements should make a difference in the effort required to discover the real intention.

(b) Syntactical complexity, measured by some volume of syntax whether it is cardinality of solution or number of different operational types, should contribute to the effort needed for discovery of the solution intent or purpose.

(c) Unfamiliar semantics behind a simple syntactical term such as \((n \times (n+1))/2\) may be a deterrent to discovery of solution intent.

(d) Picking syntactical terms, such as “product” as opposed to “sum,” which have less affinity to the real, semantic intent also contributes as a deterrent to discovery of the real intention.

4 Syntactical and Enumeration Analysis

First, let us explore Conjecture (a) and the impact due to the increase in cardinality of solution or the increase in the number of programming or pseudo-code statements. The simple example above of 5 statements versus 6 statements, without considering the content of each statement, showed an increase of 20% in cardinality of solution. Now, let us consider the permutations of 5 statements. There are 5! different ways the statements may be ordered, of which, there may be several that would suffice as the solution. For example, the order of the last two statements of Solution A may be interchanged, and the solution would still be the same. So there is more than one permutation of sequence of statements that would suffice. We, as author of the statements, may keep and hide the real order and then re-order the statements to protect the solution and confuse the perpetrator.

If we employ the reordering of the sequence of statements as one of the methods of obfuscation, then even a small increase in cardinality of the solution may make a large difference. The example of \(|A|/|B| = 6/5\) with a 20% increase is minor compared to \(6! / 5!\). Note that the number of permutations for Solution A is 6 times more than the number of permutations for Solution B. Thus by increasing the cardinality of solution, one can disproportionately increase the permutations of the statements. For example, we can take Solution B and expand it by splitting statement 4 into three statements as follows:

- if the input n is valid, then compute \(y = n \times (n+1)\)
- \(\text{sum} = y/2\)
- print out \text{sum} and terminate

We can further add one more initialization statement for variable \(y\) into Solution B and further increase its cardinality. Obviously, there are many more ways to increase the number of statements, resequence and confuse the potential perpetrator.

If the cardinality of solution, \(|Z|\), is \(x\), then adding one more statement to the solution to get \(|Z| = x+1\) would increase the permutations by a factor of \(x+1\). Adding two more statements would increase the permutations by a factor of \((x+1) \times (x+2)\) or an order of \(x^2\). Thus the following general proposition may be stated.

- If \(k\) more statements are added to a list of source code statements that has cardinality of solution of \(x\), then the number of permutations of those statements increases by an order of \(x^k\).

This is a tremendous increase in the possibilities of confusing the potential code-pirate. Thus we believe Conjecture (a) has a high potential in protecting the intellectual property.

Introducing confusion with additional syntactical terms and unfamiliar terms is one of the primary types of code obfuscation. We have introduced one more variable, y, in the above discussion when we split statement 4 of Solution B into three statements. In software engineering, many studies of different complexity metrics [3, 17, 18, 5] exist. A classical one that counts distinct syntactical terms and occurrences of the terms as a contributor to complexity is Halstead’s metric. If \(n\) is the number of distinct operators and operands and \(N\) is the sum of the occurrences of distinct operators and operands, then Halstead’s volume is defined as \(V=N \times \text{Log}_2 n\). Thus an increase of \(k\) new terms increases the Halstead volume to at least \((N+k) \times \text{Log}_2 \quad n+k\). We say “at least” because \(k\) distinct terms may have more than \(k\) occurrences. While this increase is not as dramatic as the increase in permutations of statements, it is still greater than a linear increase in number.
Thus introducing more syntactical terms into the solution also provides more opportunity for confusion. We, thus, believe Conjecture (b) also has a high potential for software protection.

Next we examine Conjectures (c) and (d) via the relation of syntax to the intended semantics.

5 Semantic Analysis

In this section, we explore the relationship of syntax to the intended semantics and the resulting possibility of obfuscation. Hitherto we have focused on the syntax and the enumeration of the syntactical statements except when the term “sum” was purposely changed to “product.” Although it was a simple syntactical change and the semantics of the statement in terms of syntactical arithmetic assignment rule was preserved, it was meant to purposely elicit some confusion in semantics.

While Conjectures (a) and (b) dealt with syntactical obfuscation, Conjectures (c) and (d) are related to semantics and intentions. We know from complexity studies and previous work on syntactical obfuscation [1, 2, 4, 5, 6, 13, 14, 17, 18] that adding more terms or introducing misleading terms increases complexity, thus more obfuscation. However, the value of semantic obfuscation is virtually unexplored. We believe that discovering the intention of a piece of code requires not only semantic knowledge, but specific semantic knowledge in the correct background. Therefore, the more difficult statement to analyze in our code solutions is the intention of statement 4 of Solution B, if one is not familiar with the mathematics. Thus the interpretation function requires both the semantics of the syntactical terms and the correct background to properly map to the intention as shown below.

Interpretation (semantics, background) \( \rightarrow \) Intention

The semantics of the term “\((\text{sum} = (n * (n+1))/2);\)” in statement 4 of Solution B can be analyzed by following the syntactical rules which parse the variables, n and sum, and the constant 2 along with the operators of \(+, *, /, (, ), \) and \(=\). Combining the specific meanings of each token and following the syntactical rules allows us to develop the semantics of the expression. However, even with clear syntactical rules, for those who are unfamiliar with the mathematical equation, it may still not be obvious that the intention here is to add the sequence of integers from 1 through n. So, while the semantics of the syntactical term may be clear and the computational result is correct, the intention behind the semantics may continue to be a mystery. That is, the term \((n * (n+1))/2\) may still not be clear to a human reader. Thus we further need the concept of interpreting the semantics to intention. In order to determine the intention, we need the "correct" background for the Interpretation function to realize the intention of the semantics as shown in the Figure 1.

![Figure 1: Interpretation and Picking Correct Background](image)

We now turn our attention to statement 5 in Solution A above. We realize that, loosely speaking, it has the same intention as statement 4 in Solution B, though the semantics are different. It includes the following: “repeatedly perform \(\{(\text{sum} = \text{sum} + \text{counter})\}\) while the counter is still less than n.” If we used Halstead’s measure of volume to count the number of operators and operands according to \(V = N*(\log n)\), then this segment of statement 5 from Solution A will certainly have more volume and thus be considered more complex than the “compute \((\text{sum} = (n * (n+1))/2);\)” segment in statement 4 of Solution B. However, to computer programmers, the intent of this seemingly more complex looping statement is quite clear. That is because when we read pseudo-code, we are already using the background knowledge of computer programming. Within the programming background, the loop statement’s intent becomes very clear. Thus to bring confusion (and protect the software), it would be more powerful to move as far away from the real intention as possible. Although for this solution, the background of mathematics and the background of programming would both satisfy the intention; many programmers are less familiar with the background of mathematics. Changing the background and causing confusion on intention is a viable way to bring obfuscation. Conjectures (c) and (d) both address this notion of obscuring the real intent and thus are also high potentials for IP protection.

6 Experimental Results

We developed an experiment to assess the validity of our assumptions and gave the experiment to students of our Software Engineering (SWE) Capstone course, which is taken by both undergraduate and graduate SWE students. These students had taken all of the core software engineering courses as well as all of the required math courses (e.g., Calculus and Discrete Math) required for the BSSWE or the MSSWE. It should be noted that our BSSWE is an ABET accredited program and our MSSWE is based on the Model Curriculum for Graduate Programs in Software Engineering. Thus, all Capstone students had also completed programming courses through Data Structures. While our sample size was small, we believe it was representative of typical practicing
software engineers, since about half of the participants were already practicing software engineers and the rest were in the process of looking for a SWE position. Capstone is taken during the final term, so all of the students were graduating at the end of the term. Our sample size was 14, of which 4 were graduate students, and 10 were undergraduate students.

We created two versions of the test, each containing three versions of the code samples; each version contained either the formula-based or the recursive-based solution, followed by the semantically challenged version using “product” instead of “sum”, and ending with the straightforward loop version of sum. We built a program to administer the tests and collect the data. The students were presented with the code while a timer was going, and were asked to click “Got it!” when he/she understood what the code was doing. At that point the timer was stopped, and an input box was displayed asking for a brief explanation of what the code was doing. This was repeated for the three code samples, and there was also an option to “Give Up” for each code version. Our findings were very encouraging and supported our intuition on the benefits of semantic obfuscation.

6.1 Findings

Most students took significantly longer trying to decipher the formula-based and the recursive-based solutions, indicating that they did not appear to have the mathematical background needed to understand the semantics of those methods. For the formula-based solution, most students described the intent by simply writing out the code – doing a literal translation; for example: “a (recursive?) method taking in a number, multiplying it by one greater than it and then dividing it by 2.” Recall this solution wasn’t recursive and due to the lack of semantic knowledge in math, the students missed the true intent, which was successfully obfuscated. In the recursive solution, the common response was similar, again the students typically gave a literal translation of the code; for example: “if n equals to zero then return zero, else return n plus class name n minus 1”. The whole idea of recursion was missed, seemingly indicating a lack of semantic knowledge of recursion, and an obfuscation of the true intent of the code. Again it should be mentioned that all students had passed Data Structures, Calculus, and Discrete Mathematics, and should have had the requisite semantic knowledge to understand all of the code samples. So why didn’t they figure it out?

We believe that the true intent of the code was obfuscated both by the semantics as well as the students’ inability to select the correct background knowledge for the Interpretation function to realize the intention of the semantics. While each student in the experiment had learned the requisite knowledge to be able to decipher the true intention of each code sample, that correct background knowledge was not successfully utilized, and the intention was hidden.

As shown in the table below, the discovery of the true intention in these samples was very low – no one deciphered the formula-based solution and only one student (who was a professional software engineer) figured out the recursive solution. The “product” term also confused the students as the semantics interfered with their understanding, and consequently this code sample took longer than the “sum” version to process and several students came up with the wrong explanation of the intent. For example, a fairly common response was “determines if the product value is equal to the int”, which means the students were unable to figure out the true intention due to the semantic interference caused by the term “product”. This was precisely what we were expecting to happen, and it illustrates the potential of semantic obfuscation for IP protection. Finally, the students were familiar with the straightforward loop structure that had no obfuscation since the term “sum” was used. Consequently this code sample yielded the shortest time for comprehension as well as the greatest accuracy of explanation and intention.

While this was an informal experiment, it was certainly encouraging enough that we are planning on a more formal experiment in our Usability Lab with a larger pool of students and a more thorough set of tools. It need also be mentioned that to utilize this style of obfuscation in a real-world setting, one would need to use automated tools to facilitate the semantic obfuscation: tools capable of inserting semantic obfuscation as well as reverting back to the original code.

<table>
<thead>
<tr>
<th>Code Sample</th>
<th>% Correct</th>
<th>Time Range in Sec.s</th>
<th>Mean in Sec.s</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>20</td>
<td>7-33</td>
<td>9</td>
<td>Lacking math semantics background</td>
</tr>
<tr>
<td>Prod Loop</td>
<td>40</td>
<td>6-63</td>
<td>25.8</td>
<td>Semantic obfuscation of “product”</td>
</tr>
<tr>
<td>Sum Loop</td>
<td>100</td>
<td>8-18</td>
<td>11.8</td>
<td>Known syntax &amp; correct background</td>
</tr>
</tbody>
</table>

| **Version 2** |           |                     |               |           |
| Formula      | 0         | 8-138               | 50            | Lacking math semantics background |
| Prod Loop    | 40        | 9-61                | 33            | Semantic obfuscation of “product” |
| Sum Loop     | 80        | 5-36                | 9.6           | Known syntax & correct background |
While this was an informal experiment, it was certainly encouraging enough that we are planning on a more formal experiment in our Usability Lab with a larger pool of students and a more thorough set of tools. It need also be mentioned that to utilize this style of obfuscation in a real-world setting, one would need to use automated tools to facilitate the semantic obfuscation: tools capable of inserting semantic obfuscation as well as reverting back to the original code.

7 Summary and Conclusions

Software complexity study and the desire to simplify software originated from the needs of reducing development effort and reducing error and defect rates in software. In this paper we explored the reverse; we looked at introducing complexity and the potential leveraging of complexity to protect our intellectual property. Four Conjectures for protecting our software through obfuscation were introduced. We explored and provided rationale of why these four Conjectures make logical sense and should be considered for further formal experiments. We believe that complexity, especially used with semantic obfuscation, may be considered a positive tool besides the legal channels for protecting our software intellectual property. Our Conjectures were also demonstrated by the results of our student experiment.

8 References


