Proceedings of the Ninth IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 14-16, 2005, Phoenix, AZ, USA

A COMPARATIVE STUDY OF JAVA OBFUSCATORS

Jeffrey MacBride, Christopher Mascioli, Scott Marks, Ying Tang, Linda M. Head, and Ravi, P. Ramachandran
Electrical & Computer Engineering
Rowan University
201 Mullica Hill Road
Glassboro, NJ 08028
USA
{macbri78, mascio88, markss02}@students.rowan.edu; {tang, head, ravi}@rowan.edu

ABSTRACT
The use of Java is growing due to its platform independence and ability to be transferred easily across the internet. Although these features are advantageous, software becomes more susceptible to theft and misuse since the original source code is preserved in bytecode format. One viable protection technique that has gained increasing attention is code obfuscation, which unintelligibly transforms the source code making it more difficult to reverse engineer, while preserving functionality. With a plethora of commercial obfuscation tools available, to analyze and evaluate the strength of these programs is paramount. Although the methods employed for obfuscation are not as plentiful as the number of programs available, the importance of evaluating these methods is commensurate. This paper focuses on the performance of two commercial obfuscators, DashO-Pro and KlassMaster, solely employing the method of control flow obfuscation. Qualitative analysis is conducted by obfuscating three different sorting algorithms with increasing complexity. The relationship between the performance of each program and the complexity of the source code is then established.

KEY WORDS
Measurement, Performance, Security, Java obfuscation, Performance analysis, Complexity

1 Introduction

With the advent of computer networks and mobile technologies, it has become a trend to distribute software in platform-independent formats over the Internet. Such an example would be the *Java bytecode*. While this trend offers important advantages with respect to cost, configurability, and portability, software becomes more susceptible to theft and misuse since the original source code is preserved in bytecode. More nefarious is the fact that attackers can easily decompile bytecode and extract proprietary algorithms and data structures from it.

Software obfuscation is an attractive and practical approach being explored to rectify this problem. The basic premise of obfuscation is to transform a program into a functionally identical one that is much more difficult to understand [1, 2, 3]. It not only hides the program information from prying eyes, but also tries to make the logic and data structure of the code as meaningless as possible even after the attackers reverse engineer it. Reverse engineering becomes futile when the cost of understanding and/or stealing intellectual property of a code becomes prohibitively high, especially if the task is more arduous than that of rewriting the program. Although there is tremendous scientific and commercial interest in developing obfuscation tools, the lack of analysis techniques and metrics for evaluating the strength of various obfuscation tools is still one of the greatest challenges in code protection research [4].

The intention of this paper is to analyze two commercially available obfuscation tools: DashO-Pro from Pre-Emptive Solutions [5] and KlassMaster from Zelix [6], and provide qualitative measurement of their performance due to control flow obfuscation. A relationship between their performance and the complexity of the code structure is established. The rest of the paper is organized as follows. Section 2 provides a definition of obfuscation and gives an overview of its different forms. The methodology is outlined in the third section along with an explanation of the different algorithms used for testing. Section 4 presents the results of the tests, followed by the conclusion and future research in section 5.

2 Obfuscation

Code obfuscation is essentially a transformation of source code from legible to unreadable, so that the transformed code cannot be easily reverse engineered [7]. Obfuscation can be performed in a variety of ways to render source code undecipherable to users. A few of the methods include renaming classes and functions, restructuring control flow, string encryption, class splitting, and class coalescing [8, 9]. The general obfuscation procedure is presented in Subsection 2.1. Subsection 2.2 focuses on control flow obfuscation.

2.1 General Obfuscation Process

The process of compilation and reverse engineering are illustrated in Fig. 1A. Compilation refers to the translation of source code into machine code. For a Java application or applet, source code is compiled into class files. As stated earlier, malicious reverse engineering can decompile files in bytecode format back to original source code, which is detrimental to intellectual property.

Obfuscation, on the other hand, transforms the class files to make their logic and data structure difficult to understand and the cost of reverse engineering prohibitively high (see Fig. 1B for the generic framework).

Figure 1A. The process of compilation and reverse engineering

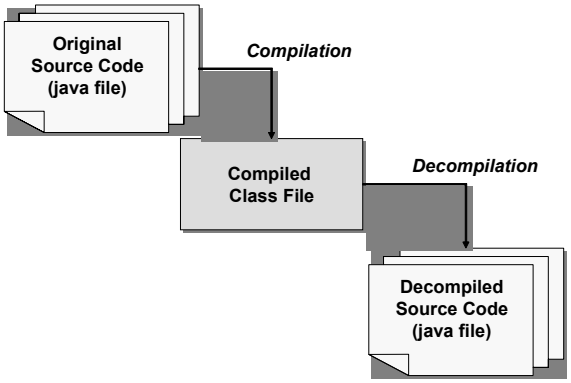
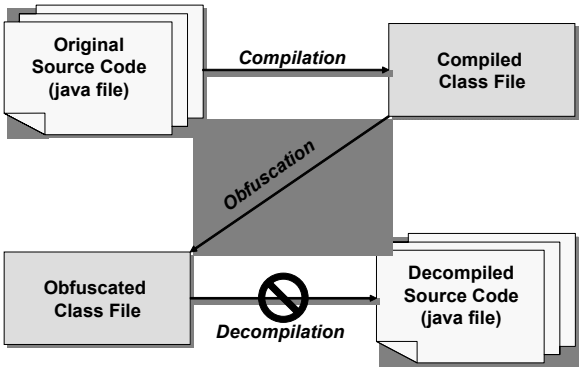


Figure 1B. Framework of obfuscation



2.2 Control Flow Obfuscation

Currently, commercial obfuscators allow the user to specify which obfuscation technique to use on the applications. The majority of commercial obfuscators have included control flow as a primary form of obfuscation available in their programs. Control flow obfuscation restructures algorithms which changes the flow of the original program. For example, a looping algorithm, using keywords such as *for* and *do-while*, may be obfuscated into a branching algorithm, utilizing the keywords *if, goto, break, continue,* and *label*. A typical example of a control flow transformation is given in Figures 2 and 3.

Figure 2. Example of original source code

```
Class count
{
       public static void main(String[] args)
      {
       for(int x=0; x<10;x++)
              System.out.println(x);
       }
}
```

3 Evaluation

The purpose of control flow obfuscation is to restructure the branching and looping statements in an application. However, altering the control flow can be a hindrance to users of the application. Specifically, runtime may increase to such a drastic level that it in effect nullifies the efficacy of obfuscation as a form of security.

Figure 3. Example of obfuscated code

```
Class count
{
   count()
    {
    }
       public static void main(String[]
args)
       {

       int x=0;
_L1:
      goto _L4;
_L2:
       System.out.println(x);
       x++;
       goto _L1;

_L3;
       break;
_L4;
       if (x < 10) goto _L2; else goto _L3;

       }
}
```

Currently, three criteria are considered in evaluating the quality of obfuscation methods; including potency, resilience, and cost [8]. The potency refers to how much obscurity is added to the program, which can be measured by analyzing certain parameters of the obfuscated code. These parameters include the number of classes added, variable dependencies, and the level of inheritance. The resilience of the program is a measure of how effectively the obfuscated program withstands attacks from either the programmer or a de-obfuscator. The measure of resilience can be based on the amount of time it takes to convert the obfuscated code back into readable source code. The cost of the obfuscation refers to how much computational overhead is added to the obfuscated application.

This paper focuses only on measures of execution cost. Three sorting algorithms, bubblesort, quicksort, and radixsort [10], are used to test the performance of DashO-Pro and KlassMaster due to control flow obfuscation. DashO-Pro costs $1,500 per seat [5] and KlassMaster costs $400 per seat [6]. The sorting algorithms are selected because of their different complexity with regard to O-notation [11]. Detailed information pertaining to these sorting algorithms is presented in the following subsections.

3.1 Bubblesort Algorithm

Bubblesort takes an array of values and divides it into two partitions: one of sorted values and one of unsorted values. In each step, the largest element found so far in the unsorted partition is moved down and appended to the

end of the sorted partition. The sorting proceeds until the elements in the unsorted partition are exhausted. The O-notation for bubblesort is $O(n^2)$ where n refers to the number of elements in the array [12]. The implementation of the sorting routine is rather simple. It only takes four lines of code containing two loops and a single conditional statement.

3.2 Quicksort Algorithm

Quicksort first selects an element that is used as a split-point from the list of given elements. All the numbers smaller than the split-point are moved to one side of the list and the rest are brought to the other side. Then each list is subdivided into two smaller lists; one containing the elements less than the split-point element and the other containing the elements greater than the split-point element. These two lists are again individually sorted using the recursive function quicksort. The implementation used for the testing is stack-based instead of recursive due to limitations within Java. The O-notation for quicksort is $O(n \cdot \log n)$ [13].

3.3 Radixsort Algorithm

Radixsort first sorts the elements by the least-significant digit. It then sorts within each radix and resorts each of the radices. A comparable example would be arranging a deck of cards first by suit. Then within each suit, the cards would be sorted in order. The O-notation for radixsort is $O(n)$ [14].

4 Performance Testing

4.1 Metrics

A dynamic array is created for the testing procedure where the size of the array, denoted by *N*, can be changed at runtime from 500 to 2,000,000. These tests are conducted one-hundred times for each size of the array. The mean of the one-hundred tests is then computed and represented as a data point. The amalgamation of the means of the array sizes constitutes the performance curve for the sorting algorithm. This procedure runs for the original sorting algorithms as well as the obfuscated versions using DashO-Pro and KlassMaster. The performance curves for the original as well as the obfuscated ones from DashO-Pro and KlassMaster are placed on the same set of axis in order to show their relationship. A least-squares-fit is also introduced to approximate an equation to each one of the curves in order to achieve a concrete differentiation between the three curves. The equations used in the experiments are $k \cdot n^2$ for bubblesort, $k \cdot n \cdot \log n$ for quicksort, and $k \cdot n$ for radixsort, where k is a constant. Any difference between the curves will be represented in the constant *k*.

4.2 Experimentation Results

4.2.1 Bubblesort

As shown in Figure 4A, the three performance curves of bubblesort are shaped like a second order polynomial which corresponds directly to the O-notation of the algorithm. Note that, the first 1000 data points are re-plotted with a smaller scale in Fig. 4B for clarity. The top curve represents the performance of the algorithm being obfuscated through KlassMaster. It is evident that there is a significant performance decrease from the original source code approximately 41.18%. While a performance loss exists between the original algorithm and KlassMaster's version, there exists a slight performance increase of 2.29% from the original to the transformed version using DashO-Pro. Such an increase is small in magnitude compared to the performance decrease of the original to KlassMaster's version. While this may seem counter-intuitive, the performance increase in DashO-Pro's version is a result of the complex structure of the bytecode which is automatically optimized through the program.

Figure 4A. Performance curves for bubblesort algorithm

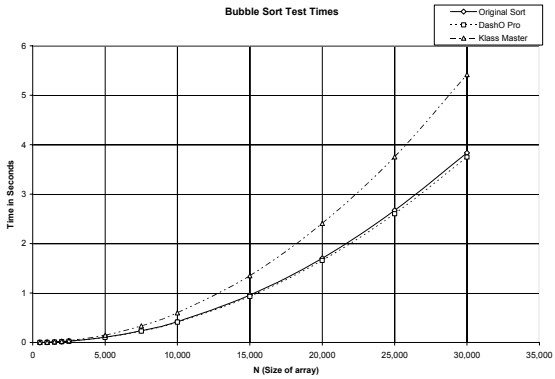


Figure 4B. Close-up of the first 1000 data points of bubblesort algorithm

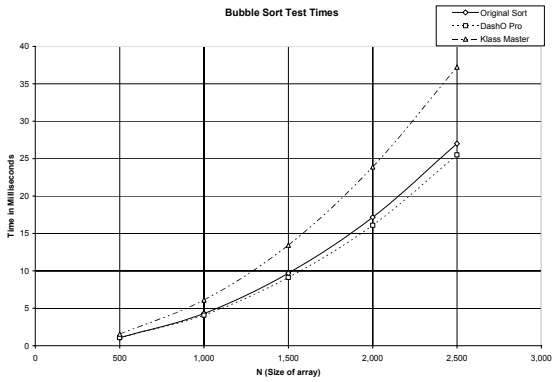


Table 1 shows the three versions of the Bubblesort algorithm and their structures. As shown in the second column, the original sorting possesses only two *for* loops and a single *if* statement. This simple coding structure is converted to eleven *goto* and *label* statements after being obfuscated with KlassMaster. Because the original code is simplistic, essentially the obfuscator has to work harder to rearrange the algorithm to make its structures appear more convoluted. Because the code is altered to such a high degree, the performance loss occurred using KlassMaster is significant. The same is true for the version obfuscated