Proceedings of the 10th IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 13–15, 2006, Dallas, TX, USA
ISBN Hardcopy: 0-88986-642-2 / CD: 0-88986-599-X

A QUALITATIVE ANALYSIS OF JAVA OBFUSCATION

Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, Ravi Ramachandran
Electrical & Computer Engineering, Rowan University
201 Mullica Hill Road, Glassboro, NJ 08028

ABSTRACT

Code obfuscation is a promising defense technology that secures software in a way that makes the cost of reverse engineering prohibitively high. While there are a number of commercial obfuscation tools on the market, there is no standard measurement to analyze and evaluate their strength. This paper addresses this challenge. An analytical metrics is developed to quantify the performance of obfuscation in terms of potency, resilience, and cost. Four commercial obfuscators are then evaluated using the proposed method.

KEY WORDS

Java obfuscation, Performance analysis, Evaluation metric

1. Introduction

Java programming, being platform-independent, offers important advantages with respect to cost, configurability, and portability [1]. However, software written in Java, then, becomes susceptible to theft and misuse since the original source code is preserved in bytecode. Given enough time, effort and determination, attackers can always reverse engineer the bytecode and extract proprietary algorithms and data structures [2]. To this end, commercial obfuscation tools, called Java obfuscators, have been released to prevent this type of theft in the way that transform a program into a functionally identical one that is much harder to understand [4]. Each obfuscator uses different techniques to transform code, which, in turn, alters the performance of the application [5]. With no standard to quantify performance of these tools, one cannot fully distinguish the quality of such a transformation.

The evaluation of a transformation has been studied in theory but nothing has ever implemented [3, 6]. While our previous work conducted a comparative study of two commercially available Java obfuscators [1, 6], the lack of analysis metrics for assessing the strength of various obfuscation tools is still one of the greatest challenges in code protection research. This paper focuses on designing and implementing a qualitative method to evaluate how well obfuscation is performed. The rest of paper is organized as follows. Section 2 discusses how the overall quality of an obfuscation transformation is measured. The three major criteria, potency, resilience and cost of obfuscation, are then discussed in detail in sections 3, 4, and 5, respectively. A pilot study is conducted in section 6, followed by the conclusion in Section 7.

2. Quality of Obfuscation

The overall performance for an obfuscator is denoted as $S_{quality}$, the quality of obfuscation. The factors that weigh into the quality include potency, resilience, and cost [2]. The potency, S_{pot}, refers to how much obscurity is added to the code that prevents human beings from understanding it. The resilience, S_{res}, is a measure of how strong the program can resist an attack against a de-obfuscator. Such attack can be defined as an attempt to transform the code back to its original structure by a computer application. The potency and resilience have positive impact on the overall quality of obfuscation since they illustrate how well the transformation protects the code by making the logic and data structure as meaningless as possible. Considering that the cognitive ability of a computer program is far inferior to that of humans, the resilience of a transformation is given a higher weight than potency. The cost of obfuscation, S_{cst}, refers to how much computational overhead is added to a transformed program. Compared to potency and resilience, cost presents negative impact on the overall quality of obfuscation. It is usually measured by comparing the resources (e.g., time, memory etc.) needed to execute the transformed program with respect to that for the original one. Measuring these factors with different weights will yield a normalized score for the quality of obfuscation as illustrated in Equation 1. Please note that 0.4 and 0.6 are chosen as the weights associated with potency and resilience in this project, respectively. However, the qualitative trend remains the same regardless the weight values as longer as the resilience score carries a higher weight than the potency score.

$$S_{quality} = 0.4 \cdot S_{pot} + 0.6 \cdot S_{res} - S_{cst} \quad (1)$$

3. Measurement of Potency

Trying to qualitatively measure potency becomes difficult since the analysis is based on human cognitive ability. Breaking up the potency into smaller measurements allows for an easier analysis. This project uses this approach and breaks potency into the following four complexity areas: nesting complexity, control flow complexity, variable complexity, and program length.

3.1. Nesting Complexity

Nesting complexity measures the number of iterative loops at different hierarchical levels in an application. In this project, we define the hierarchical levels of loops as

follows. A *level-2* loop would be the inner loop within the body of an outer one, where the outer one is called a *level-1* loop. In the same fashion, *a level-n* loop is the loop within the body of a *level-n-1* loop. Eventually, the nesting complexity is calculated using Equation 2, where $count_i$ is the number of iterative loops in the nesting level *i*, and N is the deepest level of nesting in the application. For example, the code snippet of Figure 1 illustrates one level-1, two level-2 and two level-3 loops, which results in $c_{nesting} = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2 = 11$.

$$c_{nesting} = \sum_{i=1}^{N} level - i \cdot count_i \quad (2)$$

```
Loop1{
  Loop2{
    Loop3{
      Loop4{}
    }
  }
}
```

Figure 1 Looping Construct

A ratio is then taken to represent a change in the transformation which yields $s_{nesting}$, the nesting score in Equation 3, where $c'_{nesting}$ and $c_{nesting}$ denote the nesting complexity of the transformed code and the original one, respectively. This score illustrates how well a transformation change the nesting complexity. A score of .25 would be interpreted as there being a 25% increase of nesting complexity in the transformation. Any negative score (a decrease) is considered zero.

$$s_{nesting} = \frac{c'_{nesting} - c_{nesting}}{c_{nesting}} \quad (3)$$

3.2. Control Flow Complexity

Control flow obfuscation restructures algorithm that changes the flow of the original program. Two types of flow changes are consider in this project: sequential (top-down) and non-sequential (spaghetti code). When code executes from the top of the application towards the bottom, it is considered to be top-down, or sequential. Non-sequential, as shown in Fig. 2, is when an application contains labels and *goto* statements where the code can jump around; also known as spaghetti code. The top-down approach is measured in the nesting complexity while non-sequential in the control flow complexity. Eventually, the control flow complexity score is further broken into label score, goto score, and unreadable score. Although each score is calculated in a similar fashion, it measures different perspectives of the flow structure.

The label score is the percentage of duplicate labels in a transformed application. A duplicate label would be the labels appearing in two different methods with the same name. The score increases with the number of duplicate labels, making it harder to comprehend the code. The calculation is illustrated in Equation 4 below, where $l'_{duplicate}$ is a number of duplicate labels and l'_{total} is the total number of labels in the transformed application.

$$s_{label} = \frac{l'_{duplicate}}{l'_{total}} \quad (4)$$

```
Label1:
   statement
   goto Label3
Label2:
   statement
   goto Label1
Label3:
   Goto Label2
```

Figure 2 Non-Sequential Structure

The *goto* score is the percentage of non-sequential *goto* labels in a transformed application. A non-sequential *goto* is when a *goto* statement references a line of code that comes before the statement, thus, making the flow non-sequential. The *goto* score is computed as follows, where $g'_{non-seq}$ and g_{total} are the number of non-sequential *goto* statements and total *goto* statements, respectively

$$s_{goto} = \frac{g'_{non-seq}}{g'_{total}} \quad (5)$$

The third factor involves the lines of code that a decompiler could not translate back to the source code. In this case, the decompiler leaves a line of java instructions that is readable by the virtual machine but not for human interpretation. Though the ability of a decompiler will be discussed later, a decompiler's inability to read code hinders a human's ability as well. The unreadable score is the percentage of unreadable lines of code in a transformed application. This score is proportional to the number of uncompiled lines in the code. Equation 6 illustrates this score where $c'_{undecompiled}$ is the number of unreadable lines of code and c'_{LOC} is the total lines of code in the transformed application.

$$s_{unreadable} = \frac{c'_{undecompiled}}{c'_{LOC}} \quad (6)$$

3.3. Variable Complexity

Variable complexity has four factors that yield its measurement. These factors, including a duplicate variable score, extra variable score, description variable score, and string encryption, affect the cost of the transformation as well as the readability of the code. Extra and duplicate variables may cause confusion, while non-descriptive variables and string encryption affect the initial review of code.

The duplicate variable score, $s_{duplicate}$, defined in Equation 7, measures the percentage of variables with the same name but different meanings in transformed code, where $v'_{duplicate}$ and v'_{total} are the number of duplicate variables and total variables in the transformed application, respectively. An example would include the variable *a* having two separate meanings depending on what method the variable is declared in.

$$s_{duplicate} = \frac{v'_{duplicate}}{v'_{total}} \quad (7)$$

The extra variable score, s_{extra}, measures variables with different names but the same meaning in transformed code. If two variables b and c, are both used in a method and have the same values and meaning in the method, then there is an extra variable. This score is calculated similar to the duplicate variable score as seen in Equation 8, where v'_{extra} is the number of extra variables.

$$s_{extra} = \frac{v'_{extra}}{v'_{total}} \quad (8)$$

The next factor, the descriptive variable score, $s_{variable}$, is defined as a Boolean expression describing if the transformation renames descriptive variables in original code to non-descriptive ones or not. There are three possible scenarios that could take place. The first two are that the obfuscation does not conduct variable renaming regardless original code with or without descriptive variables. Both of these scenarios yield a score of zero. The last scenario is that the original code does have descriptive variables while the obfuscation transformation changes them into non-descriptive ones. This scenario yields a score of one.

The final factor in the measurement of variable complexity is the string encryption score, s_{string}. This score has an integer range from zero to three with four possible scenarios. The first is no string encryption taking place, with a score of zero. The second is the occurrence of string encryption with a decryption method detected in the same class file, yielding a score of one. The third is the occurrence of string encryption with a decryption method detected in the application. This yields a score of two. The last scenario yields a score of three, which is the occurrence of string encryption with no decryption method detected. The score is directly related to the level of string encryption performed during obfuscation.

3.4. Program Length

The final sub-metric in the measurement of potency is a ratio of how many lines of code (LOC) are added or removed in comparison to the original program length. A positive number exhibits an increase in count and a negative number exhibits a decrease. Equation 9 demonstrates the score calculation, where c'_{LOC} and c_{LOC} are the count of the transformed code and of the original one, respectively.

$$s_{LOC} = \frac{c'_{LOC} - c_{LOC}}{c_{LOC}} \quad (9)$$

3.5. Overall Measurement of Potency

The potency score, S_{pot}, is measured on a 100-point scale, where 100 means extremely potent and zero extremely weak. There are nine total variables, as discussed above, that factor into how potent a transformation is. Each of the nine factors is weighted, according to how much they influence the potency. A low weight is assigned to the factors that add minor overhead into the original code, resulting in slightly increased time for deciphering the code. A medium weight implies that it has not changed the top-down structure of the code, but adds more time to decipher the code. A high weight means that the transformation removes the sequential methodology of structure, along with adding much overhead and deciphering time. In this project, we assign a low weight to the unreadable, duplicate variable, extra variable and descriptive variable scores. The nesting, string encryption, and LOC score obtains a median weight. The high weight is given to the label and *goto* scores. Taking the three weight classes and how many scores are in each, it is determined that the low, medium, and high weights are 6.25, 12.50, and 18.75, respectively. Using this information, Equation 10 exhibits the final equation to quantify the potency measurement of an obfuscation transformation, where x=12.50, y = 18.75, and z = 6.25.

$$S_{pot} = x \cdot s_{nesting} + y \cdot s_{label} + y \cdot s_{goto} + z \cdot s_{unreadable} + z \cdot s_{duplicate} + z \cdot s_{extra} + z \cdot s_{descriptive} + x \cdot s_{string} + x \cdot s_{LOC} \quad (10)$$

4. Measurement of Resilience

Resilience is how a transformation holds up an attack against an automated de-obfuscator or de-compiler. De-obfuscation is only making an obfuscated code more readable but for deobfuscation to happen the code must be decompiled first.

Due to the lack of commercial de-obfuscators in the market, this analysis is solely based on decompilation of code. Three de-compilers utilized in this study include *JAD*, *DJ Java*, and *Cavaj*. *JAD* is a console-based freeware application. *DJ Java* is a stand-alone Window-based application independent of the presence of a virtual machine. This means that the application can decompile code without a JVM being installed on a system. DJ Java displays the decompiled code along with a parse tree. *Cavaj* is another program that works similar to DJ Java displaying the de-complied code and parse tree as well.

The grading scale designed for resilience is based on the results of these de-compilations. The scale has three possible scenarios, with scores from zero to two. A score of zero means that no errors result from the decompilation. A score of one represents that errors occur during tree parsing. A score of two signifies a failure of decompilation. The resilience score for an obfuscation transformation on a Java program comprised of multiple class files is calculated by averaging the individual scores of the transformation on classes.

5. Measurement of Cost

Calculating cost is a straight-forward process in comparison to potency and resilience. It typically

measures extra resources that an obfuscated application consumes during runtime. Three types of resources, memory, storage space, and runtime, are considered in this study.

5.1. Memory

The memory score is a ratio of additional memory consumption of an obfuscated program against that of the original one. The memory score partitions into two types: heap memory, which is the memory consumed by the application during runtime, and object memory, which is the memory allocated by various objects used in the program, such as integers and strings, etc.

In several cases, heap memory is an important factor, and the effect of memory consumption on an application depends on how the code is structured. For example, recursive functions tend to allocate large amounts of memory in the execution stack due to the continuous process of calling themselves. Programs structured recursively run the risk of being heavily affected by obfuscation transformation. This is due to all the added overhead being placed on the execution stack as the recursive statement repeatedly calls itself.

Memory allocated for creating objects is a potential hazard for a given application because obfuscators tend to create duplicate and extra variables. Therefore, applications encompassing numerous variables in the original code are subject to allocating a large amount of object memory subsequent to obfuscation. The more objects that exist in the original program pose better chance to have more duplicate variables in the transformed one. Thus, more memory is consumed after obfuscation.

Equations 11 and 12 describe how these two factors are calculated, where $p'_{heapmem}$ and $p'_{\mathrm{var}iablemem}$ are the amount of heap memory and object memory consumed by the obfuscated code during the runtime; and $p_{heapmem}$ and $p_{\mathrm{var}iablemem}$ are the amount of heap memory and object memory consumed by the original code during the runtime

$$s_{m-heap} = \frac{p'_{heapmem} - p_{heapmem}}{p_{heapmem}} \quad (11)$$

$$s_{m-\mathrm{var}} = \frac{p'_{\mathrm{var}iablemem} - p_{\mathrm{var}iablemem}}{p_{\mathrm{var}iablemem}} \quad (12)$$

How these factors contribute to the total memory score is then illustrated in Equation 13 where the constant a = 0.375 and b = 0.625. These constants are chosen based on our pilot studies to properly weight the heap memory as being more crucial than the object creation memory.

$$s_{memory} = a \cdot s_{m-\mathrm{var}} + b \cdot s_{m-heap} \quad (13)$$

5.2. Storage

Although the economical cost of file storage decreases as technology progresses, there are situations where the increase of file size is inconvenient or critical. For example, a user may write a software package with a total file size of smaller than the capacity of the media where the software is stored. After being obfuscated, the software package's file size can increase to a value greater than the capacity of the media, which is a problem for developers. Having considered this, the storage rating of an obfuscator is then calculated in Equation 14, where $p'_{filesize}$ and $p_{filesize}$ denote the file size of the obfuscated program and the original one, respectively.

$$s_{storage} = \frac{p'_{storage} - p_{storage}}{p_{storage}} \quad (14)$$

5.3. Runtime

The last contribution to the cost score is an application's runtime. Runtime can be extremely important if an application's purpose is to perform a set of operations quickly and efficiently. For example, if an extensive algorithm is more efficient than others, adding overhead to this algorithm counteracts its efficiency.

Although much application's success is not stringently related to the runtime, efficiency is still proper etiquette for the final version of an application. In this project, the runtime score, $s_{runtime}$, is computed in Equation 15 where $p'_{runtime}$ is the obfuscated program's runtime and $p_{runtime}$ is that of the original one.

$$s_{runtime} = \frac{p'_{runtime} - p_{runtime}}{p_{runtime}} \quad (15)$$

The overall cost calculation is presented in Equation 16 where the variables x_2, y_2, and z_2 are 0.40, 0.15, and 0.45, respectively. These constants are chosen based on our pilot runs with the justification that the memory and runtime are far more "expensive" resources than the size of the obfuscated application. As technology progresses, data storage space is less and less of an issue. However, it is still a factor to consider because in some situations, storage space is strictly limited. Therefore, the file size is a factor towards the cost but weighted less than the other factors.

$$S_{cst} = x_2 \cdot s_{memory} + y_2 \cdot s_{storage} + z_2 \cdot s_{runtime} \quad (16)$$

Negative values represent positive impact on the overall quality of a transformation. This can occur if there is a significant amount increase in performance, such as the obfuscated application consuming less memory, requiring less runtime, and/or consuming less disk space.

6. Pilot Study

The metrics developed above are tested on four commercial obfuscators: *DashO-Pro*, *KlassMaster*, *SmokeScreen*, and *Allatori*, through five different Java applications with an increasing complexity: *pi calculator*, *bubblesort*, *quicksort*, *radixsort*, and *Mandelbrot* algorithms. All tests are running on the same computer with the Windows XP Professional operating system and

a Java runtime environment present. The five Java applications are installed along with four obfuscator tools and three de-compilers. This set-up guarantees accurate results because everything uses the same resources. The potency and resilience are manually measured and calculations are performed in a spreadsheet format. The cost analysis is performed using *NetBeans* performance profiler. The *NetBeans* profiler runs on a separate machine and all profiling results are obtained over the network. This prevents the actual *NetBeans* profiler from interfering with the performance of the application being analyzed.

6.1. Potency Score

Figure 3 display the potency results of each obfuscator's transformation on five Java applications. As stated earlier, the score is on a 100-point scale, where a zero indicates no obscurity added during the transformation and a 100 a high level of obscurity. It is clear to see in our results that SmokeScreen and *KlassMaster* add a high level of obscurity to the code after transformation. Compared to them, the performance of Allatori in terms of potency is mediocre. DashO-Pro has the lowest score for each application, indicating that a minimum level of obscurity is added. The variance in the scores between applications is minimal indicating that there is no direct relation between the level of added obscurity and code complexity. More analysis needs to be performed to verify the relationship between the potency and complexity.

6.2. Resilience Score

Based on the grading scaled designed earlier, the resilience scores are calculated that reflect how the obfuscated sorting algorithms, *Mandelbrot*, and *pi calculator* through different obfuscators resist the attack against three chosen de-compilers. These scores for the same obfuscator against different de-compilations are then averaged in response to a particular obfuscator's resilience performance as shown in Fig 4. Please note that the scores are eventually normalized to the 0-100 scale, although not shown in the paper, in order to be worked into the overall obfuscation metric.

Each Java program obfuscated via *DashO-Pro* is completely attainable without any decompilation and parsing errors, granting it a zero resilience score. Allatori and *KlassMaster* have very close results, each of which illustrates good resilient transformations. Smokescreen is the most resilient obfuscator tested. Both *JAD* and *DJ Java* crash while decompiling most of the files obfuscated through Smokescreen. *Cavaj* can decompile the code obfuscated via Smokescreen but not parse a tree.

6.3. Cost Score

After compiling all the data obtained from test runs, the following chart, shown in Figure 5, is generated to illustrate the overall cost analysis. The first observation is the inconsistent behavior of DashO-Pro. The BubbleSort and QuickSort algorithms result in a noticeable cost decrease. While there is a cost increase in the RadixSort obfuscated via DashO (unlike the other two sorting algorithms), it is smaller than the scores of the other three obfuscators. The cost results concerning the pi calculator and Mandelbrot application also draw our attention. Although low runtimes, low memory consumption, and smaller file sizes are relatively positive aspects of an application, the performance increase is strictly related to a lack of aggression during the obfuscation process.

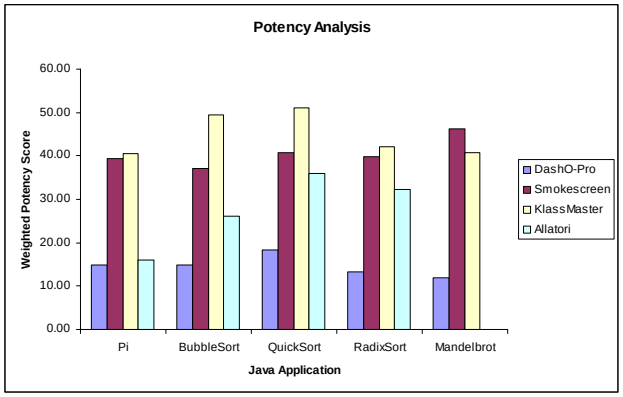

Figure 3 Potency Analysis

	Klassmaster	Allatori	Dash - O	SmokeScreen
Mandlebrot	0.5	0.458333333	0	0.875
Pi Calculator	0.666666667	0.666666667	0	1.666666667
BubbleSort	0.666666667	0.666666667	0	1.666666667
QuickSort	0.666666667	0.666666667	0	1.666666667
RadixSort	0.666666667	0.666666667	0	1.666666667

Figure 4 Resilience Test Array

KlassMaster has consistent scores, where cost increases among each program are very similar regardless of the nature of the applications (sorting algorithms or computation algorithms applications). Therefore, we conclude that *KlassMaster* will yield predictable results when used on one's application.

Allatori follows the trend similar to *KlassMaster* which consequently increases the cost per application. Allatori shows slightly larger cost increases for the more complex sorting algorithms and fewer increases for the basic sorting algorithms. While *KlassMaster*'s trend is consistent like Allatori's, it is indirectly related to the code complexity. Cost increases remain similar with calculation-based applications such as the Pi calculator and Mandelbrot.

SmokeScreen's cost trend is similar to that of Allatori where cost increases are noted with the two most complicated sorting algorithms. Virtually identical results occur when performing analysis on the pi calculator. The major factor that sets SmokeScreen apart from the other obfuscators is the cost rating for the Mandelbrot program. There is a drastic increase in the average runtime of Mandelbrot obfuscated by SmokeScreen, resulting that the score is three to four times larger than the original one.

Each obfuscator has its own cost score calculated with the consideration of data from all five applications. Some results are expected because during analysis, some obfuscators seem to have similar trends. *KlassMaster* and Allatori have similar responses to certain types of applications while SmokeScreen suffers due to its negative performance pertaining to the Mandlebrot program. The main concern during our analysis is the results generated from DashO-Pro transformations. As stated previously, the low cost score corresponds to a low potency and resilience score. One may draw the conclusion that DashO Pro is more of an optimizer than an obfuscator.

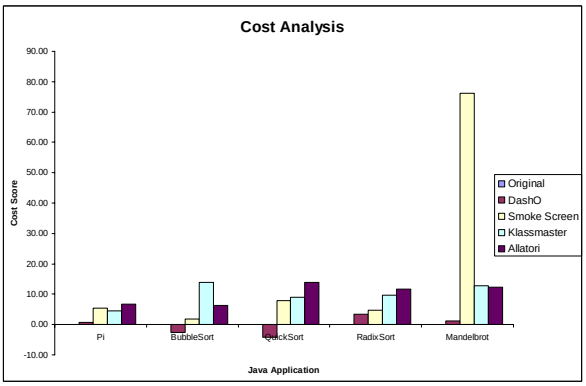

Figure 5 Cost Analysis

6.4. Overall Quality of Obfuscation

With the data obtained above, the final scores of the quality of obfuscation are then calculated for each obfuscator using Equation 1.

Careful inspection of Figure 6 indicates that *SmokeScreen* obfuscates Java code with the highest level of obscurity (potency and resilience) and the least amount of memory increase in comparison to other obfuscators. The exception is the Mandelbrot transformation. This score is low since the increase of overall cost is very high, drastically decreasing the overall quality score. *KlassMaster* transforms code better than Allatori, whose scores are at the average level on the pi and sorting applications. *DashO-Pro* has the lowest score for each application indicating that its overall quality of obfuscation is very low.

7. Conclusion

Obfuscation uses clever techniques to deter reverse engineering of Java code by making the cost of reverse engineering prohibitively high. While there are a number of obfuscators on the market, to develop a qualitative method in evaluating the performance of such tools becomes critical. This project studies the evaluation methods in theory and develops an analytical metric in quantifying the strength of various obfuscation transformations in terms of potency, resilience and cost. The increasing complexity of code is indirectly related to the decreasing level of obfuscation performed on a given application which in turn is related to the increase in cost [1]. Therefore, there is a tradeoff between such properties that a developer must consider while choosing a proper obfuscation tool. Though some obfuscators are scored lower than others, developers might utilize such tools if a low level of obfuscation and performance loss is needed. Our metric will assist developers to make such decisions. According to our analysis of four commercially available obfuscators using the developed metric, we conclude that obfuscation is still juvenile in the field of software protection, especially for Java programming. One of the contributions of this analysis is to help developers design an obfuscator that utilizes the most prominent features of each application.

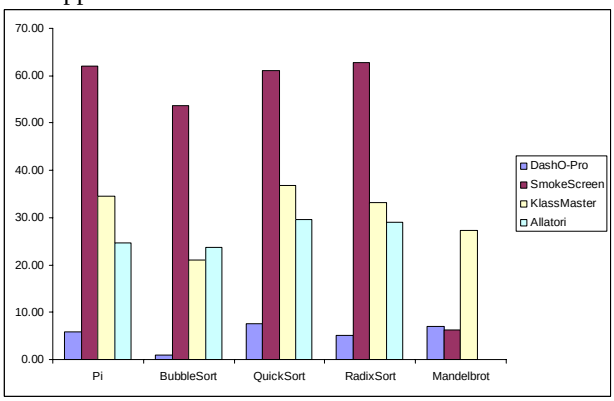

Figure 6 Overall Quality Analysis

This research can be continued in several directions. For instance, it is worthwhile to benchmark our findings in this project and to consolidate our analytical metrics. Based on our observations in this project, one complementary technique for software protection could be to obfuscate the program interpretation (e.g., JVM) instead of obfuscating the program itself [6].

References

[1] MacBride, J., Mascioli, C., Marks, S., Tang, Y., Head, M. L., Ramachandran, R. P., "A Comparative Study of Java Obfuscators," in Proceedings of the IASTED International Conference on Software Engineering and Applications (SEA 2005), Phoenix, AZ, Nov. 14-16, 2005, pp.82-86.

[2] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," The University of Auckland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[3] D. Low, "Protecting Java Code via Java Obfuscation," ACM Crossroads, Spring 1998 issue.

[4] S. Drape, "Obfuscation of Abstract Data Types," Ph.D. dissertation, University of Oxford, Oxford, England, 2004.

[5] H. Lai, "A Comparative Survey of Java Obfuscators Available on the Internet," The University of Auckland, Auckland, New Zealand, Proj. Rep. 415.780, 2001.

[6] Mascioli, C., Marks, S., MacBride, J., Tang, Y., Head, M. L., Ramachandran, R. P., "Analysis of Java Obfuscators," Rowan University, technical report, 2005.