

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

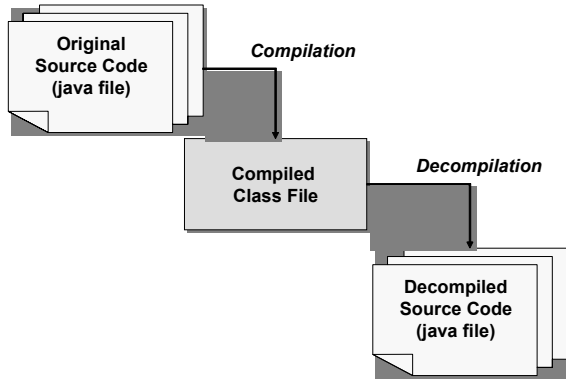
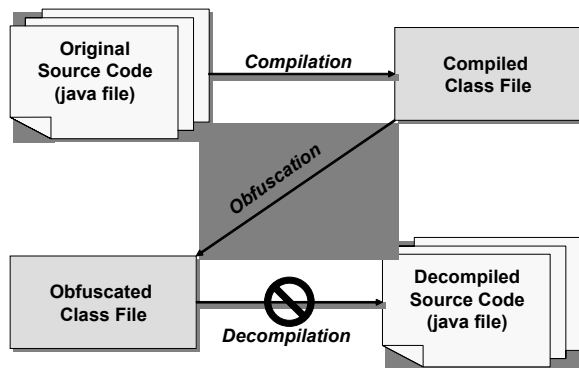


Figure 1B. Framework of obfuscation



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

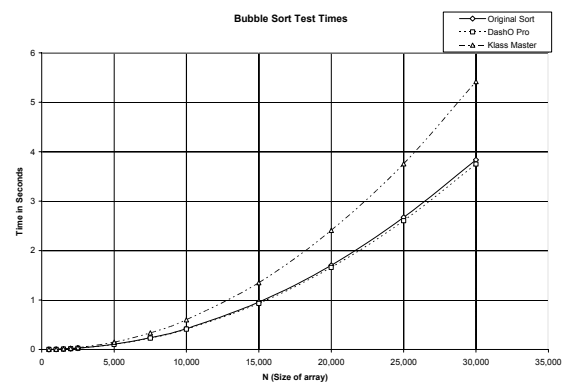


Figure 4B. Close-up of the first 1000 data points of bubblesort algorithm

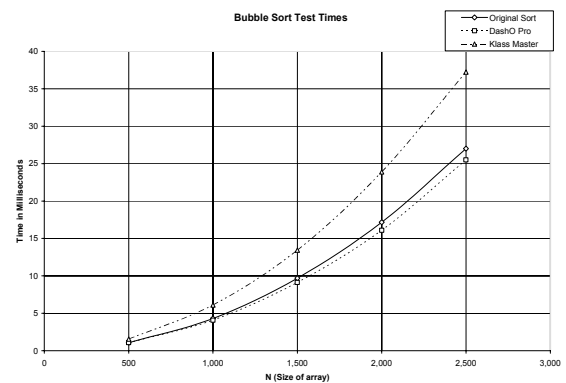


Table 1 shows the three versions of the Bubblesort algorithm and their structures. As shown in the second column, the original sorting possesses only two *for* loops and a single *if* statement. This simple coding structure is converted to eleven *goto* and *label* statements after being obfuscated with 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

using DashO-Pro, except the code is not altered to the extent that KlassMaster altered the code.

Table 1. Bubblesort Structure Type Comparison

Bubblesort	Original Sort	DashO-Pro	KlassMaster
If	1	1	4
Goto / Labels	0	0	11
While	0	1	0
For	2	1	0
Invalid Decompilation	0	0	

4.2.2 Quicksort

As stated earlier, the equation to approximate the performance curves of quicksort is $k \cdot n \cdot \log n$. The k values computed for the original code, and the obfuscated versions from DashO-Pro and KlassMaster are $4.0241 \cdot 10^2$, $4.1095 \cdot 10^2$, $4.9588 \cdot 10^2$, respectively. From these values, there can be determined a 2.12% performance loss from DashO-Pro and a 23.23% performance loss from KlassMaster as shown in Figure 5A. The first 10,000 data points are re-plotted with a smaller scale in Fig. 5B.

Figure 5A. Performance curves for quicksort algorithm

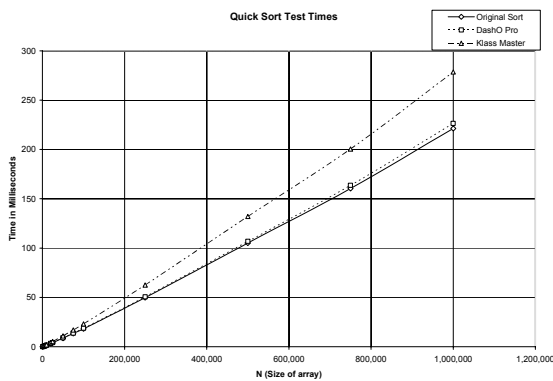


Figure 5B. Close-up of the first 10,000 data points of quicksort algorithm

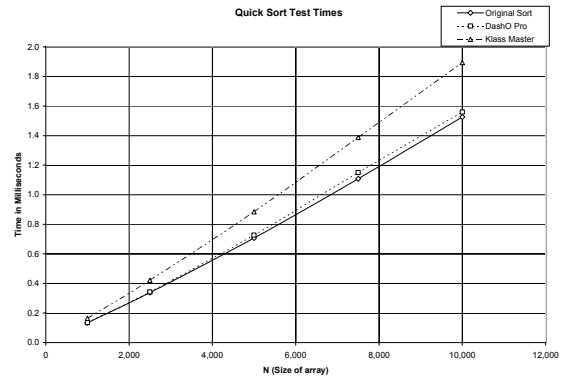


Table 2 shows the structure of the code for the three versions of the quicksort algorithm. Compared to the bubblesort algorithm the implementation of the code is augmented which is apparent from Tables 1 and 2. Again each of the obfuscators adds a degree of complexity to the code. However, the number of lines of obfuscated code inserted to the original one is slightly reduced.

Table 2. Quicksort Structure Type Comparison

Quicksort	Original Sort	DashO-Pro	KlassMaster
If	3	5	11
Goto / Labels	0	8	24
While	4	11	1
For	0	1	0
Invalid Decompilation	0	0	5

4.2.3 Radixsort

Figure 6A displays the performance curves of the radixsort algorithm that are best fit to a first-order polynomial. The variance between the three performance curves is small enough that any performance increase or decrease can be regarded as a statistical error in the testing procedure. At some points in Figure 6A, the curves intersect each other which is another consequence of the variances being so small. Figure 6B displays a close-up of the first 10,000 data points of Figure 6A. This shows the dependence of the array size.

Figure 6A. Performance curves for radixsort algorithm

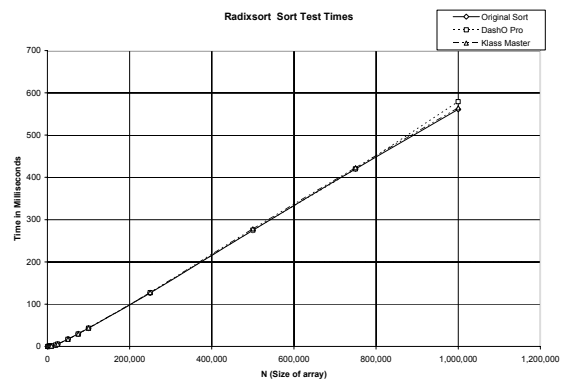


Figure 6B. Close-up of the first 10,000 data points of radixsort algorithm

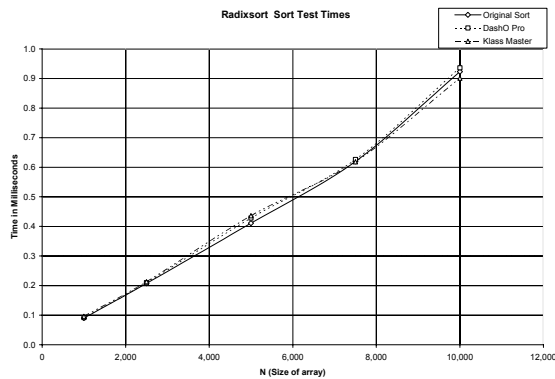


Table 3 illustrates how the structure of the original algorithm is transformed after using the two obfuscator programs. It is apparent, from the second and third rows, that there is a significant increase in the number of *if* statements and *goto* labels added to the original structure after obfuscation. Specifically, KlassMaster does the most to alter the original algorithm. However, not evident from Table 3 is the complexity of the original algorithm. Out of the three sorting algorithms, Radixsort contains the most code complexity. This gives a rationale as to why the three performance curves are closely related.

Table 3. Radixsort Structure Type Comparison

Radixsort	Original Sort	DashO-Pro	KlassMaster
If	1	3	8
Goto / Labels	0	9	17
While	0	1	1
For	4	1	0
Invalid Decompile	0	0	4

5 Conclusion

This paper presents a qualitative measurement of the capability of two commercially available obfuscators, DashO-Pro and KlassMaster. Specifically, this work addresses control flow obfuscation in terms of the computational overhead added to the obfuscated applications.

Overall, the two obfuscators used for the analysis both cause variations in the performance of the algorithms used for testing. The greatest performance losses occur when obfuscating the bubblesort algorithm using KlassMaster. The progression from the bubblesort algorithm to the radixsort shows an increase in the complexity of the coding routines. Also by going in the same direction it can be seen that the performance loss decreases significantly. Therefore, it can be concluded that the performance loss and the complexity of the code for KlassMaster possess an inverse relationship. The obfuscator has to do more work to make simplistic code appear code indecipherable. Therefore, it must add more code to make the algorithm harder to read, in effect causing a much greater performance loss. With regard to DashO-Pro, even as the complexity of the code increased, the performance loss occurred does not vary drastically.

To develop more analytical metrics in evaluating the scalability and overheads of current obfuscators is our future research.

References

- [1] Chan, J. T. and Yang, W., "Advanced obfuscation techniques for java bytecode," *Journal of systems and software*, Vol. 71, 2001, pp. 1-10.
- [2] Collberg, C. S. and Thomborson, C., "Watermarking, tamper-proofing, and obfuscation – tools for software protection," *IEEE Transactions on software engineering*, Vol. 28, No. 8, 2002, pp. 735-746.
- [3] Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A., "Software obfuscation on a theoretical basis and its implementation," *IEICE Transactions on Fundamentals*, Vol. E86-A, NO. 1, 2003, pp. 1-11.
- [4] Van Oorschot, P. C., "Revisiting software protection," *Proceedings of the 6th International Information Security Conference*, Bristo2001, UK, Oct. 2003, pp. 1-13.
- [5] "DashO – the Premier Java Obfuscator and Efficiency Enhancing Tool" from Preemptive Solutions, [Online document], [cited 2 Mar 2005], Available [HTTP: http://www.preemptive.com/products/dasho/index.html](http://www.preemptive.com/products/dasho/index.html)
- [6] "The Second Generation Java Obfuscator" from Zelix, [Online document], [cited 2 Mar 2005], Available [HTTP: http://www.zelix.com/klassmaster/index.html](http://www.zelix.com/klassmaster/index.html)
- [7] "obfuscation: a whatis.com definition" from SearchVBTechTarget, 19 Jul 2004, [cited 2 Mar 2005], http://searchvb.techtargt.com/sDefinition/0..sid8_gci967845.00.html
- [8] Collberg, C. S., Thomborson, C., and Low, D., "A taxonomy of obfuscating transformation," *Technical Reprot #148*, 1997.
- [9] Sosonkin, M., Naumovich, G., and Memon, N., "Obfuscation of design intent in object-oriented applications," *2003 ACM Workshop on Digital Right Management*, pp. 142-153.
- [10] "Java Technology", from Sun Microsystems, [Online document], [cited 2 Mar 2005], <http://java.sun.com/>
- [11] Black, P.E. "big-O-notation" from National Institute of Science and Technology, [Online document], 3 Jan 2005, [cited 2 Mar 2005], <http://www.nist.gov/dads/HTML/bigOnotation.html>
- [12] Minoura, T. "Bubble Sort Program," from CS261 Data Structures, [Online document], [cited 2 Mar 2005], <http://web.engr.oregonstate.edu/~minoura/cs261/java/Progs/sort/Bubblesort.html>
- [13] Singh, H. "Quicksort", [Online document], 2000, [cited 2 Mar 2005], <http://www.seeingwithc.org/topic2html.html>
- [14] Standish, T.A., *Data Structures, Algorithms, and Software Principles*, New York: Addison Wesley, 1994, pp 563-565.