McGill University
School of Computer Science
Sable Research Group

Metrics for Measuring the Effectiveness of Decompilers and Obfuscators

Sable Technical Report No. 2006-4

Nomair Naeem Michael Batchelder Laurie Hendren

June 2006

Contents

List of Figures

Abstract

Java developers often use decompilers to aid reverse engineering and obfuscators to prevent reverse engineering. Decompilers translate low-level class files to Java source and often produce "good" output. Obfuscators rewrite class files into semantically-equivalent class files that are either: (1) difficult to decompile, or (2) decompilable, but produce "hard-to-understand" Java source.

This paper presents a set of metrics we have developed to quantify the effectiveness of decompilers and obfuscators. The metrics include some selective size and counting metrics, a expression complexity metric and a metric for evaluating the complexity of identifier names.

We have applied these metrics to evaluate a collection of decompilers. By comparing metrics of the original Java source and the decompiled source we can show when the decompilers produced "good" code (as compared to the original code). We have also applied the metrics on the code produced by obfuscators and show how the metrics indicate that the obfuscators produced "hard-to-understand" code (as compared to the unobfuscated code).

Our work provides a first step in evaluating the effectiveness of decompilers and obfuscators and we plan to apply these techniques to evaluate new decompiler and obfuscator tools and techniques.

1 Introduction

Two popular tools in the development of Java applications are *decompilers* and *obfuscators*. Decompilers are used in reverse engineering to recreate Java source from class file binaries. Several Java decompilers are known to be quite good, especially when decompiling class files that have been produced using a known `javac` compiler. That is, Java decompilers often produce source code that *"looks good"*. Obfuscators, on the other hand, are used to prevent effective reverse engineering. Obfuscators convert class files into semantically-equivalent class files which either: (1) are difficult or impossible to decompile or (2) lead to decompiled source that is *"hard to understand"*.

The purpose of this paper is to provide metrics that can quantify these more abstract notions that *"the output of the decompiler looks good"* or that *"the output of the obfuscator is hard to understand"*. Historical and current software measurement techniques and metrics are heavily geared towards software engineering uses. In particular, it has been suggested that metrics can be used to measure programming effort and to detect error-prone software modules. In contrast, we are more interested in metrics that can help us compare two versions of the same program. Specifically, we want to compare an original source program against its decompiled version to determine if the decompiler is producing clear and understandable output. Further, we want to compare the obfuscated output to determine if it is harder to understand.

This paper provides two major contributions. First, we define a set of simple metrics aimed at measuring the effectiveness of decompilers and obfuscators. Second, in order to validate our metrics and study the effectiveness of a variety of tools, we have applied them to output of three decompilers including our Soot-based Dava decompiler [13, 14] and two obfuscators, including JBCO (Java Bytecode Obfuscator) which we are currently developing as part of the Soot toolkit [17, 20].

The structure of the paper is as follows. In Section 2 we provide an overview of some of the most related previous work on software metrics with a focus on why those metrics are or are not suitable for our purposes. In Section 3 we introduce our metrics and in Section 4 we provide an overview of our benchmarks and evaluate the metrics in the context of decompilers and obfuscators. Finally, conclusions and future work are given in Section 5.

2 Related Work

There has been much research into software complexity and many metrics have been proposed and embraced by the software engineering community throughout the years. Classic examples are McCabe's cyclomatic number [12], and Halstead's programming effort measures [6]. More recent efforts have been geared towards quality analysis for large-scale software projects and processes including design principle violation detection, tracking module evolution [11], and software engineering process improvement [4].

These complexity metrics are designed to measure effectiveness, code reliability, programming effort, and clarity (or cognitive expressibility) [19]. What this paper focuses on is the specific idea of cognitive expressibility. When a decompiler tries to recover the (higher-level representation) source code of a binary program it is effectively attempting

to recover a cognitive representation - a human-readable (or at least programmer-readable) version that is semantically equivalent to the binary. Likewise, when an obfuscator sets out to garble a program it is attempting to decrease the cognitive representability of the program by adding complexity of some kind.

Because the quality of the cognitive representation is our key interest, some well-developed metrics in the literature are nonetheless somewhat useless here. McCabe's Cyclomatic number, for example, shows the complexity of the control flow through a piece of code. It is the number of conditional branches in the flowchart. However, if a program segment S is compiled into a binary form B and then decompiled into a source code segment S' then S and S' will have the same cyclomatic number regardless of how the decompiler chooses to represent loops and other branching instructions in the program. Therefore the metric does not discern any differences between the cognitive representations of S and S'.

Similarly, Halstead's metrics are not all suitable for our case. They are most often used during code development in large projects in order to track complexity trends. A spike in Halstead metrics can signify a highly error-prone module, for example. However, this is not our concern. We wish to use metrics to compare two high-level representations of a program, both with the same semantics. Halstead's metrics do not lend themselves well to this problem.

Halstead's program volume metric is a measure of the minimum number of bits required for coding a program. In the case of Java, non-local variables (either class fields or static fields) and method names are preserved in the compiled bytecode. A common Java obfuscation technique is to rename these identifiers, often with shorter and more incomprehensible names. This reduces the program volume but also reduces the ability of a decompiler to recover the full cognitive representation of the original program.

Indeed, many metrics are designed to compare large software projects in a very abstract way in order to predict maintainability, reliability and/or programming effort. Most of these are not useful to the particular problem at hand.

However, some of the criticism that Halstead's measures have seen over the years - specifically the argument that they are a bad measure because they consider lexical and textual complexity rather than the structural complexity of a program [7] - is a key ingredient to our own proposed metrics. The high-level measures of textual structure and complexity are in fact exactly what we wish to measure, along with control flow complexity.

Finally, there has been a lot of work on developing metrics for the analysis of Object-Oriented projects [2]. Specifically, they try to measure the "Object-Orientedness" of the frameworks that projects employ, as opposed to the low-level instruction sequences found on at the method or function level. Examples include Chidamber and Kemerer's Depth of Inheritance Tree, Number of immediate sub-classes of a class, and Number of classes to which a class is coupled. In fact there has been fairly passionate debate as to the usefulness of older metrics that involve graph-oriented models, such as the cyclomatic number, when evaluating an object-oriented system [5]. Nevertheless, these OOD metrics are also of little use here because we are comparing original, decompiled, and obfuscated bytecode - none of these transformations currently affect the object-oriented nature of a program.[1]

We are much more interested in the high-level human-readable source code representation of a program's methods. This makes the approach in [16] a good starting point - they evaluate only source code and they measure such intricacies as identifier length, nesting depth, and decision node complexity. However, their goal is to identify outliers in the set of functions of a program (*i.e.*,, they try to answer the question "What functions don't fall within normal ranges"). They suggest that these functions may indicate areas of poor design or complex functionality. Our approach is not intended for highlighting problem areas of a program but rather to compare two different semantically-equivalent versions of an entire program as a measure of the effectiveness of code transformation tools.

3 Metrics

We experimented with a wide variety of metrics and in this section we present those metrics that we found to be most useful for the purposes of evaluating decompilers and obfuscators. We first present the simplest metrics for size (Section 3.1) and counting relevant constructs (Section 3.2). One of the key differences among decompilers is their treatment of conditional expressions and in Section 3.3 we define a *conditional complexity* metric designed to expose those differences. Finally, a special problem introduced in decompilation and and obfuscation is the naming of

[1] Although the more advanced techniques currently being implemented in our JBCO obfuscator will obfuscate the object-oriented design, in which case some OOD metrics may become applicable.

identifiers. We introduce an *identifier complexity* metric in Section 3.4.

All of the metrics were computed using specialized traversals over the abstract system tree (AST) representation of Java source as produced by the polyglot-based Soot frontend.

3.1 Program Size

A simple program size metric is useless in comparing two *different* programs other than to say one is larger than the other. However, this metric can be very useful in comparing two representations of the *same* program. Arguably, more verbose code is more complex and this metric is a good high-level measurement to see if decompilers produce unnecessarily verbose code and if obfuscators inserted useless code.

For our purposes, we define *program size* to be the number of nodes in program's AST representation. Measuring size in this way discounts comments, spurious parentheses and any program formatting issues.

3.2 Number of Java Constructs

Another simple metric for the comprehensibility of a Java program is the frequency of different Java constructs in the code. Of course it is necessary to identify which constructs are strong indicators of complexity. After considering empirical results, we narrowed our attention to four categories:

- `If` and `If-Else` statements (Simple Conditionals)
- Abrupt control flow (`break` and `continue`)
- Labeled blocks
- Local variables

Simple conditionals help to indicate the amount of decision-making in a program. A more complex program will have more branching and therefore more `If` and `If-Else` statements.

Abrupt control flow directives are even more indicative of complex programming. It is argued that the use of these statements decreases the tractability of control flow and therefore increases code complexity.

Labeled blocks are compound statements which are explicitly labeled. While programmers will often section their code using blocks, the existence of a label suggests the block is used for controlling execution flow (through the use of a explicitly labeled `break` or `continue`). Other than exception handling, this is one of the most unclear control flow mechanisms in Java.

Local variable counts can also indicate complexity. The more information one must consider when reading code the harder it is to understand. Programmers don't usually create unnecessary identifiers, but tools like decompilers and obfuscators often do.

3.3 Conditional Complexity

Boolean expressions which decide control flow in a program (*i.e.*,, those deciding `If`, `For`, and `While` branching) play a particularly crucial role in analyzing code. Aside from boolean constants (`true` or `false`), the simplest conditional expressions consist of a unary boolean literal - a boolean variable. This is assigned a complexity weight of 1. However, conditional expressions can be aggregations or nestings of simpler expressions. A boolean literal can be reversed with the negation operator, ! or relational operators ($<$, $>$, $<=$, $>=$, $==$) can be used to compare expressions. We argue that these operators, while more complex than a single boolean, are still fairly easy to understand and therefore we give them a weight of 0.5. Expression aggregation using the && or || operators requires the reader of code to evaluate the meaning of two subexpressions and then to combine the two - arguably a more complex task - so we define the weight for these operators to be 1.

The complexity for each boolean expression in a program is simply the sum of all the weights described above. Taking the subtree that represents the expression, the leaves of the tree are boolean literals (increasing the complexity

by 1 each) and every internal node is either an unary, relational, or binary operation (increasing the complexity by 0.5, 0.5, or 1, respectively).

Given this description, the expression `a<b && !done` would be assigned a complexity of 4. `a<b` refers to two variables (weight of 1 each) and the relational operator giving it a complexity of 2.5. `!done` is a boolean with a negation operator and is given 1.5. The aggregation (`&&`) adds another 1 to the overall complexity for a total of 5.

Average conditional complexity for a program is simply the average of the conditional complexities over all boolean expressions in the program.

3.4 Identifier Complexity

The name used for an identifier can provide valuable insight into the context in which the variable is used. This in turn can ease a programmer's task of understanding the code. Indeed, most obfuscators garble identifiers in a program. We compute the complexity of identifiers by calculating a sum of complexities for all identifiers where each is weighted by a relative importance. An identifier x has it's importance factor $I(x)$ defined as follows:

$$I(x) = \begin{cases} 4, & \text{if x is a Method} \\ 3, & \text{if x is a Class} \\ 2, & \text{if x is a Field} \\ 1.5, & \text{if x is a Formal} \\ 1, & \text{if x is a Local} \end{cases}$$

We argue that method names are particularly important for program understanding so we give them the highest importance value. Each identifier's complexity is computed as the sum of token and character complexities (described below) multiplied by their importance factor.

Token complexity is a measure of recognizable language. Alpha tokens are parsed by delimited by non-alphas and uppercase alphas. For example, `getASTNode` is split into `get`, `AST` and `Node`. Notice `ASTNode` is split into two tokens, the second one starting with a capital alpha). Similarly, `___Junk$$name` is broken into `Junk` and `name`. Tokens are then counted and the *token complexity* is defined as the ratio of total tokens to those found in a dictionary.[2] If the dictionary contains the tokens `get` and `Node` but not `AST` then token complexity for `getASTNode` will be 1.5.

Character complexity is a ratio of total characters to those classified as non-complex. Non-complex characters are those which are *not* part of a sequence of non-alphas of length greater than 1. The character complexity for the identifier `___Junk$$name`, for example, is 1.625 as there are five complex to 8 non-complex characters (_ , _ , _ , $, $ and J, u, n, k, n, a, m, e, respectively). Note that a sequence of non-alphas of length one is not considered as complex since it very likely exists as a word separator, as in `get_Socket`.

4 Results

In order to exercise our metrics we performed two sets of experiments on a set of small-to-medium sized benchmarks. In Section 4.1 we introduce the benchmarks, in Section 4.2 we present our first set of experiments aimed at examining decompilers and in Section 4.3 we examine our second set of experiments with obfuscators.

4.1 Benchmarks

The benchmarks have been culled from a graduate-level compiler optimizations course where students were required to develop interesting and computation-intensive programs for comparing the performance of various Java Virtual Machines. Each one was written in the Java source language and compiled with `javac`. The following is a brief description of each.

[2] The dictionary used in our experiments was a standard english language dictionary. However, one could use a special-purpose dictionary that also contained domain-specific identifiers.

Asac: is a multi-threaded sorter which compares the performance of the Bubble Sort, Selection Sort, and Quick Sort algorithms.

Chromo: implements a genetic algorithm, an optimization technique that uses randomization instead of a deterministic search strategy. It generates a random population of chromosomes. With mutations and crossovers it tries to achieve the best chromosome over successive generations.

Decode: implements an algorithm for decoding encrypted messages using Shamir's Secret Sharing scheme.

FFT: performs fast fourier transformations on complex double precision data.

Fractal: generates a tree-like (as in leaves) fractal image.

LU: implements Lower/Upper Triangular Decomposition for matrix factorization.

Matrix: performs the inversion function on matrices.

Probe: uses the Poisson distribution to compute a theoretical approximation to pi for a given alpha.

Sliding: solves the well-known Sliding Block Puzzle Problem.

Traffic: is an animation of a road intersection controlled by a traffic signal. It uses multithreading to simulate cars moving through the intersection.

Triphase: performs three separate numerically-intensive programs. The first is linpack linear system solver that performs heavy double precision floating-point arithmetic. The second is a heavily multithreaded matrix multiplication algorithm. The third is a multithreaded variant of the Sieve prime-finder algorithm.

The benchmarks we selected are not large (our size metric is shown in Figure 1), but are in fact quite varied and exhibit many different properties and coding styles.[3]

4.2 Decompiled Code

Decompilation is the process of retrieving a high-level representation from a lower-level representation of a program. In the case of Java the lowest level of representation is bytecode, a language for the Java Virtual Machine similar, but still higher level than, actual machine or assembler code. Since bytecode is already a higher-level representation than pure machine code, it is in fact possible to retrieve well-formed, valid, and compilable Java source code from it in most normal cases.

There exists a number of Java decompilers which perform well on bytecode specifically produced by Sun's Java `javac` compiler. The most popular of these are Jad [8] and SourceAgain [18]. When given bytecode produced by a known `javac` compiler, these decompilers produce very good output because they recognize the code patterns known to be created by `javac` and simply recreate the equivalent source code. If unknown patterns appear in the bytecode, then SourceAgain and particularly Jad are very often unable to fully decompile programs into valid source. These *javac-specific* decompilers are therefore often not able to handle bytecode produced from other sources such as optimizers, instrumenters, obfuscators and third-party compilers generating bytecode from non-Java source languages.

In contrast to javac-specific decompilers, our Soot-based Dava decompiler was created to handle arbitrary bytecode. Dava does not specifically look for patterns exhibited in Javac output; it operates on the idea that *any* valid bytecode should be decompilable. While this requires much more complicated decompilation techniques, it is a more robust approach since even simple obfuscators can transform programs into bytecode that is not recognizable as `javac` output. However, there is a price to pay. The output of Dava may not *"look good"*. In fact, the original version of Dava (henceforth referred to as *Dava(Original)*) produced semantically-correct, but often *"ugly"* code. More recently we have built a new backend for Dava that transforms the *"ugly"* code into code that *"looks better"* [15]. We refer to this new version of Dava as *Dava(Improved)*.

[3] We would have liked to experiment with some larger benchmarks as well, but in order to do so in a rigorous manner all of the decompilers and obfuscators would have to work correctly on those benchmarks. This appears not to be the case. As the other tools mature and become more robust on larger applications, it will be possible to experiment with larger programs.

Although we have previously given specific examples to compare the output of the decompilers, we have been unable to quantify the quality of their output. Thus, the purpose of the experiments in this section is to quantify how well the various decompilers perform and, in particular, examine how much better Dava(Improved) is compared to Dava(Original).

4.2.1 Program Size

Since each decompiler has its own source code formatting style, we normalized all output with a style formatter (JRefactory's JavaStyle [9]) in order to remove these differences. The formatter ensures that the AST contains the same number of AST nodes for the same constructs (an `If` block with one statement in its body is calculated the same whether brackets exist, distinguishing the block as a compound statement, or not). Figure 1 shows the number of nodes in the AST for all benchmarks. Traffic is largest with triphase, sliding, and chromo following it.

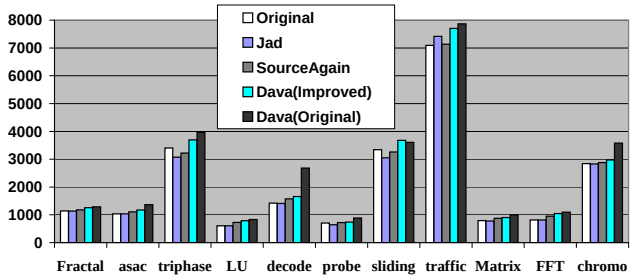

Figure 1: Program size for decompiled code

The outputs from the different decompilers do show some variance in the size of the code. Dava(Original) produces the largest ASTs, as was expected. However, Dava(Improved) is able to decrease program sizes considerably. Most of this size reduction can be attributed to the removal of abrupt jumps, labeled blocks, and the aggregation of conditional statements using the boolean && and || operators. The output produced by Jad and SourceAgain usually matches original source very closely, an expected result given their use of pattern matching to recognize constructs produced by `javac`.

4.2.2 Conditional Statements

Since Dava(Original) did not deal with short-circuit control flow created by && and || operators, it produces more `If` and `If-Else` statements. Dava(Improved) implements numerous aggregation transformations, greatly reducing the number of conditionals, as supported by the metrics in Figure 2 attests to this fact.[4]

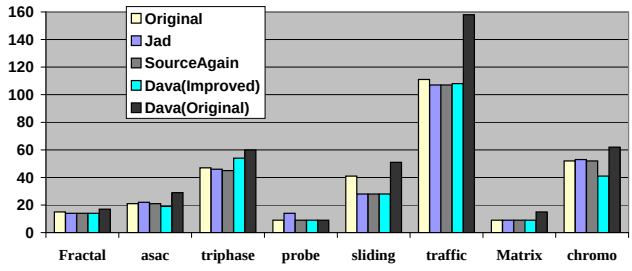

Figure 2: Conditional statement count for decompiled code

The largest peaks for the number of conditionals are from Dava(Original). With Dava(Improved), however, there is a drastic drop in these constructs which, in most cases, matches that of the other decompilers. Interestingly, all decompiler output (except Dava(Original)) for the sliding benchmark contain fewer conditionals than the original source.

[4]Note that in this and subsequent graphs we do not show results for benchmarks for which the metrics are the same, or nearly the same, for all versions of the benchmark. If the reviewers prefer we can include those in the final version of paper.

This would indicate that the benchmark's original code used very simple non-aggregated conditional statements and was perhaps written by a novice programmer. An examination of this benchmark proved this to be so. Another interesting observation is that the general strategies in Dava(Improved) sometimes find more aggregation opportunities than Jad and SourceAgain (asac and chromo), and sometimes finds fewer (triphase). This demonstrates that different decompilation strategies can impact the quality of the output.

4.2.3 Conditional Complexity

Conditional complexity is a measure of how complex the boolean expressions within conditional constructs (`If`, `If-Else`, and loop constructs) are. Conditional complexity increases as boolean subexpressions are aggregated using the && or || operators. At the same time the use of negations (!) also increases conditional complexity. Figure 3 shows conditional complexity for the benchmarks.

For most benchmarks Jad and SourceAgain produce code with almost the same measure as the original. Small variations occur when a boolean flag is represented using the negated flag and vice versa.

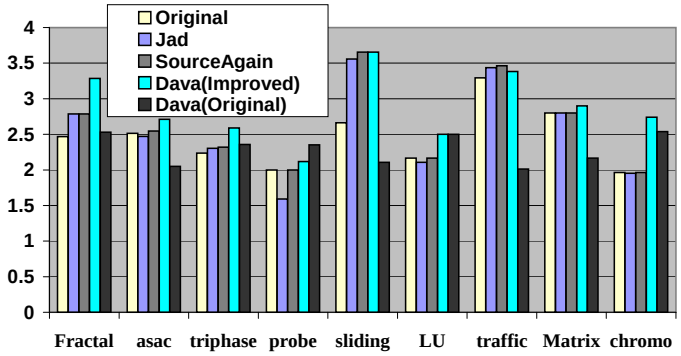

Figure 3: Average conditional complexity for decompiled code

An exception to this is the sliding benchmark. Here we see that all the decompilers increase the complexity by almost the same amount. The decompilers all detect the chance to aggregate multiple conditions and in doing so increase the conditional complexity, thereby reducing `If` and `If-Else` statements.

Comparing Dava(Improved) and Dava(Original) we see that apart from the probe benchmark there is a definite increase in conditional complexity implying the aggregation of conditions. When we investigated the probe code, we noticed that whereas Dava(Original) was creating conditions of the form "!flag" Dava-Improved was able to switch the bodies to have conditions of the form "flag". Further, there was no chance of aggregation in the code. Thus, the removal of negation decreases the complexity and we see this in the complexity values for probe.

By examining the metrics for the original metrics, we see that a conditional complexity between 2 and 3 is normal. In the future, a metric-aware Dava could use its aggregation transformation sparingly in an attempt to maintain this level.

4.2.4 Abrupt Control Flow

Eliminating `Break` and `Continue` statements is one of key transformations implemented in Dava(Improved). We argue that these abrupt control flow devices, of all Java constructs, add the most complexity to source code because they represent disjoint execution flow. The more abrupt edges there are in a program, the less the code reads sequentially. This makes it very difficult for a programmer because it increases the "problem space" by increasing the number of scoping levels that must be kept track of, as well as the cohesion of disparate code chunks.

Out of all the benchmarks, sliding and traffic were the only ones which had a sizable number of `break` statements. All decompilers end up introducing some abrupt flow but this number is usually very low for javac-specific decompilers, Jad and SourceAgain, as seen in Figure 4. Again, this is due to the matching of code patterns to obtain concise output.

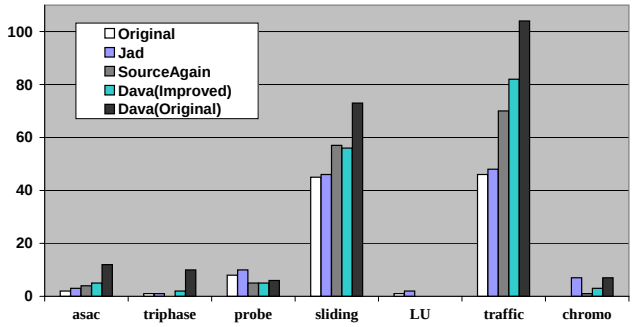

Figure 4: Abrupt control flow count for decompiled code

Dava(Original), on the other hand, suffers greatly by producing code with many complicated `break` statements nested within `Labeled-Block` constructs. This is because the low-level bytecode represents all of its control flow through only `If` and `Goto` instructions; a naive decompiler will take the simplest route and transform these into abrupt `breaks`. The impact of more complex abrupt flow transformations, as implemented in Dava(Improved), can be seen in the reduction of abrupt statements for Dava(Improved) as compared to Dava(Original). In many cases Dava is able to produce fewer, if not the same, number of abrupt statements as Jad and SourceAgain. However sliding and traffic are two benchmarks which still show there is room for improvement.

4.2.5 Labeled Blocks

Directly related to abrupt statements are the number of labeled blocks present in decompiled code. Labeled blocks are especially bad programming practice and, in fact, they exacerbate the previous problems with abrupt control flow by allowing more disjoint execution jumps than available with unlabeled `break` statements. Unsurprisingly, no labeled blocks appear in the original source of any of the benchmarks. Jad and SourceAgain are able to maintain this minimum. The general restructuring algorithm in Dava(Original), on the other hand, produces a high number (Figure 5). Luckily, Dava(Improved) shows a 75% reduction over Dava(Original).

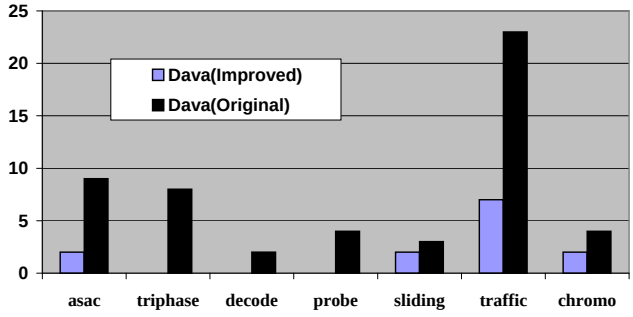

Figure 5: Labeled block count for decompiled code

4.2.6 Local Variables

Dava(Original) produces many local variables in its output. This is because Dava takes its input from `GRIMP` which has been computed from the low-level Soot IR which uses many local variables in order to get simple and precise compiler analyses. With Dava(Improved), copy elimination and constant propagation is used to considerably reduce the number of locals (Figure 6). Jad and SourceAgain output is, again, very close to the original for this metric.

An exception to this is triphase where we see a very high number for Jad. Inspection of Jad's output for this benchmark shows that it is unable to handle aggregated floating point and double precision calculations. These are broken down into 3-address form where each statement introduces a new local variable.

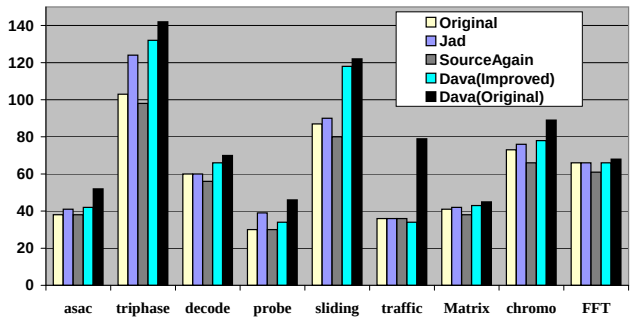

Figure 6: Local variable count for decompiled code

4.2.7 Overall Complexity

In order to provide one summary metric, we experimented with a variety of composite metrics. We found a good overall complexity metric which is defined by first expressing each component metric as a normalized value with respect to the value for the original Java benchmark, and then combining the normalized values, each component multiplied by a constant representing that metric's importance. The sum of the constants is 1, so that when comparing the original javac source to itself will always result in an overall metric of 1.

For example, for the size component we compute the normalized value by (size of decompiled benchmark)/(size of original benchmark) and we multiply this normalized value by 0.2. Figure 7 gives the result using 0.2 * size + 0.2 * if_count + 0.2 * cond_complexity + 0.1 * num_abrupt + 0.1 * num_labeled + 0.2 * num_locals, where each component of this metric corresponds to normalized values of the metrics as presented in subsections 4.2.1 through 4.2.6.

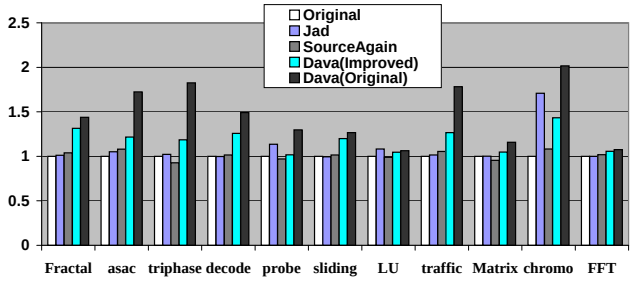

Figure 7: Overall complexity for decompiled code

Using this overall metric we can see that Jad and SourceAgain produce decompiled code that is close to the original code (remember that these benchmarks have not been obfuscated and thus javac-specific decompilers work well for them). We can also observe that Dava(Original) does in fact produce (ugly) code that is not as similar to the original code, but that the additional transformations implemented in Dava(Improved) do improve upon this substantially.

4.3 Obfuscated Code

Obfuscation deals with altering a program in order to make it more obscure or confusing to understand. An obfuscator may work on a high-level or low-level representation of the program, or both. Obscuring high-level code contributes to a decrease in overall comprehensibility and by obscuring the low-level code, obfuscators can make it more difficult for decompilers to regenerate source code at all.

Although for a long time Java programs were mostly obfuscated by changing the names of identifiers (JShrink, RetroGuard), new techniques have emerged which perform control flow obfuscation. One such obfuscation is the introduction of opaque predicates [3]. These predicates can not be statically decided through analyses such as copy propagation and therefore they introduce a level of complexity (such as undecidably dead code) in the program that a

reverse-engineering tool such as a decompiler is unable to remove or simplify. Zelix Klassmaster is a notable example in this category of second-generation tools [10]. We use Klassmaster for most of our experiments in this section.

Finally, new ideas in Java-specific obfuscations is leading to a new third generation. JBCO (Java ByteCode Obfuscator) is an obfuscator that we are currently working on which performs such convoluted transformations as to render current decompilers useless. These transformations take advantage of the fact that there are many valid, but obscure, uses of Java bytecode that do not translate naturally to high-level Java. Thus, decompilers are not able to handle these obfuscations. However, for this paper we require correct decompiled source code in order to calculate our metrics. Thus, for the purposes of this paper we enable only the the first-generation name obfuscation and add some method indirection with JBCO, so that all of the decompilers will still be able to work correctly.

The experiments in this section were performed as follows. We created our baseline by first compiling the application using an ordinary `javac` compiler to produce the class files and then decompiled those class files with our Dava decompiler, with all of the advanced transformations turned on. This option is labeled Dava(Improved) in subsequent figures. We used the Dava decompiler because it is robust enough to be able to decompile code after first- and second-generation obfuscations, whereas the other decompilers often fail to decompile after the obfuscations.

To create the obfuscated versions of the source code we first applied the obfuscators (Klassmaster and JBCO) to the class files to produce obfuscated class files. We then decompiled the obfuscated class files using Dava. We used Dava in two configurations, the *Original* one, and the *Improved* one where all simplifications are applied. In the subsequent figures JBCO(Improved) refers to the case where we obfuscated with JBCO and then decompiled with Dava(Improved) and JBCO(Original) refers to the case where we obfuscated with JBCO and then decompiled with Dava(Original). Similarly, we created two versions for the Klassmaster obfuscator.

By comparing the Dava(Improved) versions with JBCO(Improved) and Klassmaster(Improved) one can observe the impact the the two obfuscators had on the metrics. By comparing the Klassmaster(Improved) to Klassmaster(Original), and similarly comparing JBCO(Improved) to JBCO(Original), we can observe the impact of the advanced Dava simplifications in undoing some of the obfuscations introduced by the obfuscators. These include some identifier renaming optimizations, control-flow simplifications, copy elimination and advanced dead-code elimination.

Although we computed all the metrics for both obfuscators, Klassmaster and JBCO, we only show results for Klassmaster in many of the figures. This is because JBCO has no effect on some of the metrics since we enable only two obfuscations: renaming identifiers and moving library calls into new methods with obfuscated names.

4.3.1 Program Size

Figure 8 shows the program size metric. It is clear that both JBCO and Klassmaster increase the size in all cases. Comparing the two obfuscators we see that the size increase is greater for Klassmaster. This is expected because Klassmaster adds dead code guarded by opaque predicates which can therefore not be removed by the static analyses performed by Dava. JBCO size increases are due to the addition of methods which are used to invoke library calls through an extra level of indirection. Therefore, the difference between the unobfuscated Dava(Improved) case with the JBCO(Improved) case is directly proportional to the number of unique library methods called in the program. A smart decompiler could apply a refactoring algorithm to overcome this obfuscation through re-inlining these unneeded indirections.

Also interesting is the difference between Klassmaster with and without Dava's advanced simplification analyses, Klassmaster(Original) versus Klassmaster(Improved). This difference is most obvious for the decode and chromo benchmarks. In these cases the Dava dead code elimination removes a large amount of code introduced by Klassmaster. Nevertheless, not all dead code is removed because much of it is guarded by opaque predicates. Dava is unable to statically detect the values of these predicates and hence the code remains. A much more powerful context sensitive flow analysis would be required to remove the remaining dead code.

4.3.2 Conditional Statements

Figure 9 demonstrates a large increase in the number of conditional statements after obfuscation by Klassmaster. This is consistent with Klassmaster's technique of introducing redundant or dead code enclosed by simple `If` statements. Dava attempts to aggregate many of the conditionals and can sometimes remove some redundancies, as illustrated by

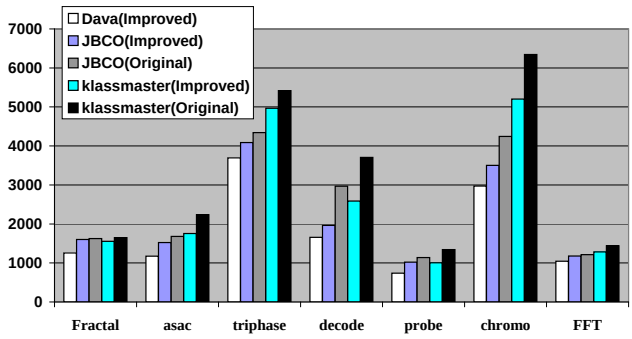

Figure 8: Program size for obfuscated code

the difference between Klassmaster(Original) and Klassmaster(Improved). However, a large number of these conditions still remain.

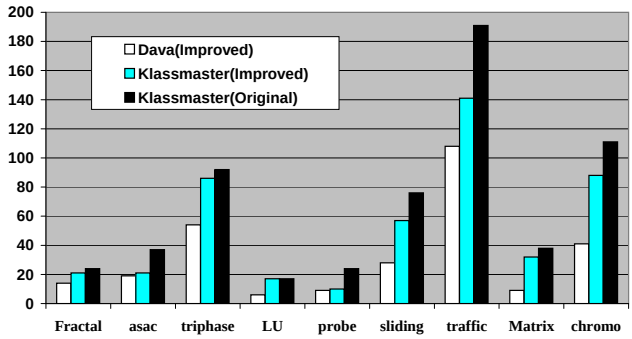

Figure 9: Simple conditional statement count for obfuscated code

4.3.3 Conditional Complexity

Conditional complexity is shown in Figure 10. Here, the decrease in complexity is mainly due to the fact that Klassmaster introduces its own conditional constructs which are simple un-aggregated boolean expressions. Hence, although the number of conditional constructs increases, the average conditional complexity decreases. An additional possible reason for the drop in complexity is that the original bytecode is intermixed with obfuscation code. This inhibits the pattern-based simplifications and therefore results in fewer conditional aggregations. The increase seen in Klassmaster(Improved) versus Klassmaster(Original) is due to the aggregation of conditions. Some benchmarks show a decrease which most likely occurs due to removal of dead code which included complex conditionals.

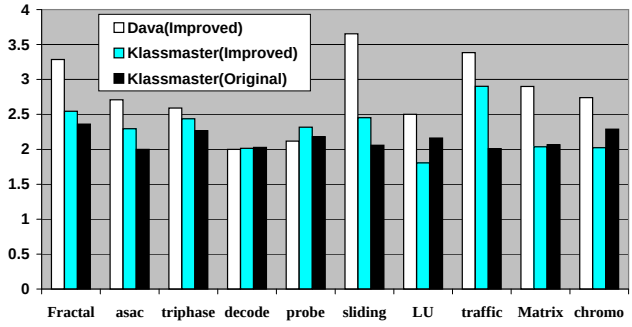

Figure 10: Average conditional complexity for obfuscated code

4.3.4 Abrupt Control Flow

The count of abrupt statements (`break` and `continue`) for the obfuscated code as compared to the un-obfuscated code is shown in Figure 11. We can see a marked increase in abrupt statements (particularly in triphase, decode and chromo).

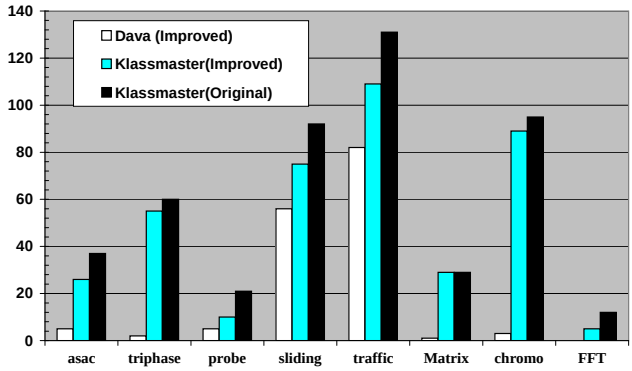

Figure 11: Abrupt control flow count for obfuscated code

The abrupt metric is particularly useful in identifying obfuscated code. Abrupt edges in the flow graph of a program are a direct result of control-flow obfuscation techniques and it clearly worsens the readability. As stated earlier, a programmer has a lot to keep track of when trying to follow abrupt control, especially when execution jumps directly out of multiple nesting levels. Thus, programmers tend to make sparse use of complex abrupt control-flow, whereas obfuscators intentionally add them in to complicate the control flow.

It is interesting to note that javac-specific decompilers such as Jad and Sourceagain often fail to decompile such code because the control-flow in the class files does not correspond to any known structured Java control flow pattern. Dava succeeds in decompiling and reducing the number of abrupt control flow statements due to its use use of graph-based restructurings.

As demonstrated by comparing Klassmaster(Original) to Klassmaster(Improved), the Dava simplifications are able to restructure some of the code to reduce abrupt control flow in many of the benchmarks, but not all cases of abrupt control-flow can be removed.

4.3.5 Labeled Blocks

Labeled blocks are shown in Figure 12, correlating closely with the number of abrupt statements. The Klassmaster(Original) case has a large number of labels but Klassmaster(Improved) shows that Dava's simplifications can reduce these to a more acceptable level. For some benchmarks (FFT and probe) all labeled blocks can be removed. Over the whole benchmark suite 65% of the labeled blocks are removed.

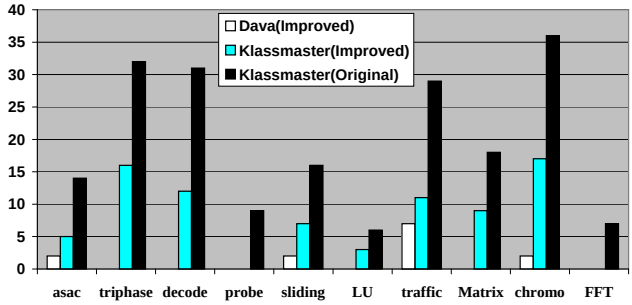

Figure 12: Labeled block count for obfuscated code

4.3.6 Identifier Complexity

Identifier obfuscation is a very important metric for evaluating obfuscators. Nearly all obfuscators perform identifier obfuscation and it is perhaps the only technique that is truly irreversible [1]. Figure 13 shows that JBCO performs identifier obfuscation extremely well based on our metric. Klassmaster also does well, though a difference between the Klassmaster(Original) and Klassmaster(Improved) values can be seen due to a basic local variable renaming algorithm implemented in Dava. Also, removal of dead code reduces the local variable count, some of which have complex names, hence decreasing the complexity.

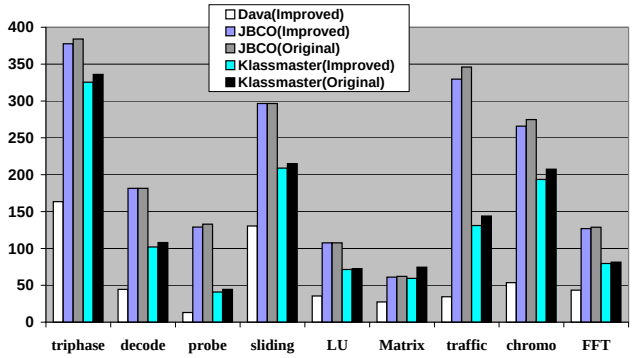

Figure 13: Identifier complexity for obfuscated code

4.3.7 Overall Complexity

Figure 14 reports the same overall complexity metric as we introduced in Section 4.2.7. Note that this metric does not include identifier complexity, so one should really consider both the identifier complexity presented in figure 13 and the overall metric in figure 14 which summarizes control-flow like obfuscations, when considering the effect of obfuscators.

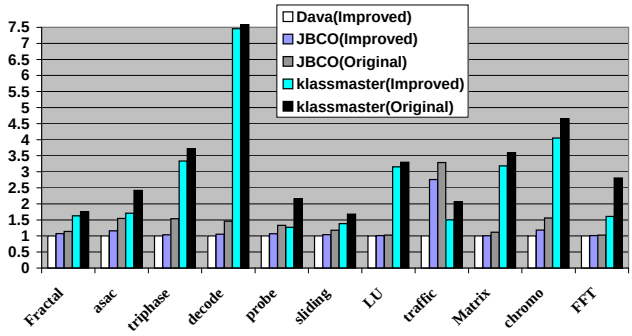

Figure 14: Overall complexity for obfuscated code

Considering these two figures we can see that, as expected, the effect of JBCO is mostly on identifier obfuscation, whereas Klassmaster shows significant impacts on the structure of the code. It is also interesting to note that the Klassmaster(Improved) is closer to the unobfuscated code than Klassmaster(Original), indicating that the advanced transformations in Dava do help to clean up the code somewhat.

5 Conclusions and Future Work

The purpose of this paper was to provide metrics to evaluate the effectiveness of decompilers and obfuscators in order to help quantify if a decompiler produces code that is similar to the original source and if an obfuscator effectively

produces code very different from the original.

We first defined a set of metrics that were specifically designed to distinguish between two semantically-equivalent programs. These metrics included a simple size metric, several metrics for counting key constructs, a conditional complexity metric and an identifier complexity metric. We also suggested an overall metric which combines the individual metrics.

In order to evaluate the metrics we examined a collection of decompilers including the javac-specific decompilers Jad and SourceAgain and two versions of the tool-independent Dava decompiler, the original Dava and a new improved Dava version which includes transformations to improve the quality of the output source. The metrics demonstrated the general belief that javac-specific decompilers produce code that is very close to the original source, when used on unobfuscated class files. The metrics also demonstrated that the improved Dava decompiler produces code more like the original source than the original Dava.

We also used the metrics to evaluate two obfuscators, a simple version of our JBCO obfuscator and the KlassMaster obfuscator. The results show that obfuscators are quite effective in producing Java source that is different and more complicated than the original source. Size metrics showed that additional code was added, an increase in abrupt statements and labeled blocks showed an increase in control-flow complexity and increases in identifier complexity metrics demonstrated that, while simple to perform, name-changing is one of the most effective obfuscation techniques available.

In the future we plan to use our metrics to evaluate the next phases of development for both the Dava decompiler and the JBCO obfuscator. We would also like to identify some useful object-oriented metrics for the use in investigating third-generation obfuscators.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:1–??, 2001.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[3] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[4] R. Conn. A reusable, academic-strength, metrics-based software engineering process for capstone courses and projects. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 492–496, New York, NY, USA, 2004. ACM Press.

[5] L. O. Ejiogu. On diminishing the vibrant confusion in software metrics. *SIGPLAN Not.*, 32(2):35–38, 1997.

[6] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[7] P. G. Hamer and G. D. Frewin. M.h. halstead's software science - a critical examination. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 197–206, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[8] Jad - the fast JAva Decompiler. `http://www.geocities.com/SiliconValley/Bridge/8617/jad.html`.

[9] JavaStyle - JRefactory's Pretty Printer. `http:// www.jrefactory.sourcforge.net`.

[10] Zelix KlassMaster - The second generation Java Obfuscator. `http://www.zelix.com/klassmaster`.

[11] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM Press.

[12] T. J. McCabe. A complexity metric. *IEEE Trans. Software Eng.*, 2(4):308–320, December 1976.

[13] J. Miecnikowski and L. J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag, 2002.

[14] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 368–374, October 2001.

[15] N. A. Naeem and L. Hendren. Programmer-friendly decompiled Java. In *Proceedings of the 14th IEEE International Conference on Program Comprhension*, 2006.

[16] P. N. Robillard, D. Coupal, and F. Coallier. Profiling software through the use of metrics. *Softw. Pract. Exper.*, 21(5):507–518, 1991.

[17] Soot - a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[18] Source Again - A Java Decompiler. `http://www.ahpah.com/`.

[19] K.-C. Tai. A program complexity metric based on data flow information in control graphs. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 239–248, Piscataway, NJ, USA, 1984. IEEE Press.

[20] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.