Metrics for Measuring the Effectiveness of Decompilers and Obfuscators*

Nomair A. Naeem Michael Batchelder Laurie Hendren
School of Computer Science, McGill University, Montreal, Canada
{nnaeem, mbatch, hendren}@cs.mcgill.ca

Abstract

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

We present a set of metrics developed to quantify the effectiveness of decompilers and obfuscators. The metrics include some selective size and counting metrics and an expression complexity metric. We have applied these metrics to evaluate a collection of decompilers and obfuscators. By quantitatively comparing original Java source against decompiled and obfuscated code respectively, we show which decompilers produce "good" code and whether obfuscations result in "hard-to-understand" code.

1 Introduction

Two popular tools in the development of Java applications are *decompilers* and *obfuscators*. Decompilers are used to recreate Java source from class file binaries. They can be used to aid program comprehension in reverse engineering tools or as a way to display the effect of low-level optimizers or aspect weavers. For example, it is easier for a programmer to see the effect of an optimizer if the resulting program is displayed as clear source code, rather than the low-level bytecode representation. Several Java decompilers are known to be quite good, especially when decompiling class files that have been produced using Sun's `javac` compiler and 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 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. We are interested in metrics that can help us compare two versions of the same program. We want to compare an original source program against its decompiled version to determine if the decompiler is producing clear and understandable output. Furthermore, 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 the output of three decompilers and two obfuscators, including our Dava decompiler [8, 9] and JBCO obfuscator [1].

The structure of the paper is as follows. Section 2 gives an overview of some related work on software metrics with a focus on why those metrics are or are not suitable for our purposes. In Section 3 we introduce a set of metrics and in Section 4 we evaluate these in the context of decompilers and obfuscators. Section 5 discusses some study constraints and future work and in Section 6 we conclude.

2 Related Work

The effectiveness of decompilers is often measured in terms of whether it produces compilable source code. Similarly, an obfuscator is considered good if it thwarts decompilation attempts or produces garbled source code. To the best of our knowledge no formal work has ever been carried out to quantitatively classify the effectiveness of these tools. It is the aim of this paper to propose an initial set of metrics and then use them to evaluate decompiled and obfuscated benchmarks in order to quantitatively measure whether these tools are doing what they claim to do.

There has been extensive research into software complexity and many metrics have been proposed and embraced by the software engineering community. Classic examples are McCabe's cyclomatic number [7] and Halstead's programming effort measures [3]. More recent efforts have been geared towards quality analysis for large-scale software projects and processes including design principle vi-

*This work was supported, in part, by NSERC and FQRNT.

olation detection, module evolution tracking, and software engineering process improvement.

These complexity metrics are designed to measure effectiveness, code reliability, programming effort, and clarity [13]. This paper focuses on the specific idea of code comprehension. When a decompiler recovers the source code of a binary program, it is attempting to recover a human-readable version that is semantically equivalent to the binary. Likewise, when an obfuscator garbles a program, it is attempting to decrease the cognitive representability of the program by adding complexity of some kind.

McCabe's Cyclomatic number [1] is an often used metric to show the complexity of a piece of code. This metric, however, is too coarse grained for our purpose. Although the metric gives an overall complexity value to the program, it does not show where this complexity stems from. For instance, the number of edges, could stem from branching statements as well as abrupt control flow. We would like to differentiate between the two. Furthermore, 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 number of connected components regardless of how the decompiler chooses to represent loops.

Because the quality of human-readability is our key interest, 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 these 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 an entire program, both with the same semantics. Halstead's metrics do not lend themselves well to this problem.

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. Since we are interested in the high-level human-readable source code representation of a program, most of these are not useful to the particular problem at hand.

3 Metrics

We experimented with a wide variety of metrics and present here those that we found to be most useful. We present a metric for size (Section 3.1) and counting relevant constructs (Section 3.2) and then define a *conditional complexity* metric (Section 3.3). The metrics were computed using specialized traversals over the Abstract Syntax Tree representation of Java source.

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, it can be useful in comparing two representations of the *same* program. When used in tandem with other metrics it can be used as a normalization factor. This metric is a good test to see if decompilers produce verbose code and if obfuscators insert useless code.[2]

3.2 Number of Java Constructs

The frequency of different Java constructs in code can be a useful metric as well. After considering empirical results, we narrowed our attention to two categories that are strong indicators of high complexity:

Conditional statements 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 (`break` and `continue`) along with explicitly labelled compound statements are even more indicative of complex programming since they represents disjoint execution flow. The more abrupt edges there are, 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. [3]

3.3 Conditional Complexity

Boolean expressions, especially those deciding control flow, play a particularly crucial role in analyzing code. We assign different weights to these expressions based on how elaborate they are. Boolean constants (`true`/`false`) and boolean variables, the simplest conditional expression, are assigned a weight of 1. Conditional expressions using relational operators or the unary negation operator, while more complex than a single boolean, are still fairly easy to understand and get a weight of 0.5. Aggregation, using && or ||, requires the code reader to evaluate the meaning of two subexpressions and then to combine the two - arguably a more complex task - so we give them a weight of 1.

An expression's complexity is the sum of all the weights described above. Given the expression subtree, each leaf is a boolean literal (increasing the complexity by 1) and every internal node is either a binary, relational, or unary operation (increasing the complexity by 1, 0.5, or 0.5, respec-

[1] McCabe's Cyclomatic Number = # of edges - # of nodes + # of connected components

[2] We define *program size* to be the number of nodes in a program's AST hence discounting comments, spurious parentheses and any program formatting issues. Since each decompiler has its own source code formatting style, we normalized all output with a style formatter (JRefactory's JavaStyle [5]). This ensures that the AST contains the same number of AST nodes for the same constructs.

[3] Notice that the sum of abrupt and conditional statements is a good indication of the number of edges in the code. Since we have the number of nodes (program size metric), and since connected components remain unchanged, we are in a position to calculate changes in McCabe complexity. Instead we have chosen to define our own overall complexity metric.

tively) [4]. *Average conditional complexity* is the average of all boolean expression complexities in the program.

4 Results

In order to exercise our metrics we performed two sets of experiments on a suite of eleven benchmarks. These Java benchmarks have been culled from a graduate-level compiler optimizations course where students were required to develop interesting and computation-intensive programs. [5]

We present our first set of experiments, on decompiled output, in Section 4.1 and our second set, examining obfuscated code, in Section 4.2.

4.1 Decompiled Code

Decompilation is the process of retrieving a high-level representation of a program from a low-level one. In the case of Java the lowest level of representation is bytecode. Because bytecode maintains some high-level information and is simpler than pure machine code, it is possible to retrieve well-formed, compilable Java source code from it.

There exists a number of Java decompilers which perform well on bytecode specifically produced by Sun's `javac` compiler. The most popular of these are Jad [4] and SourceAgain [12]. When given bytecode produced by the `javac` compiler, these decompilers produce very good output because they recognize the code patterns created by `javac` and simply recreate the equivalent source code. If unknown patterns appear in bytecode, then SourceAgain and particularly Jad are often unable to fully decompile these sequences into valid source. These *javac-specific* decompilers are therefore often thwarted by bytecode produced from other sources such as optimizers, instrumenters, obfuscators and third-party compilers.

In contrast, the 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"*. Although we have previously given specific examples comparing output of the decompilers [11], we have been unable to quantify this. Experiments in this section, quantify the performance of various decompilers with a focus on how a tool-independent decompiler fairs against javac-specific decompilers.

[4] For example, the expression `a<b && !done` has a weight of 5. `a<b` has two variables (weight of 1 each) and a relational operator, giving it a weight of 2.5. `!done`, a boolean with a negation operator, is given 1.5. The aggregation (&&) adds another 1 for an overall complexity of 5.

[5] Benchmarks available at: http://www.sable.mcgill.ca/dava/#download

4.1.1 Program Size

There is some very small variance in the number of AST nodes between the different decompilers (Figure 1.a). Jad and SourceAgain produce output very close to the original source, an expected result given their use of pattern matching to recognize constructs produced by `javac`. Dava produces larger ASTs but the difference seems insignificant.

4.1.2 Conditional Statements

Conditional statements produced by the decompilers (Figure 1.b) show minor differences [6]. Although bytecode represents conditionals one comparison at a time, Dava creates short-circuit control flow using && and || aggregation. It appears that these transformations sometimes find more aggregation opportunities than Jad and SourceAgain (Asc and Chr), and sometimes fewer (Trp) indicating that different decompilation strategies can impact the quality of output.

Interestingly, all decompilers produced fewer conditionals than the original source for Sld. This indicates that the original code used non-aggregated conditional statements and was perhaps written by a novice programmer.

4.1.3 Abrupt Control Flow and Labelled Blocks

Figure 1.c indicates that all decompilers introduce some abrupt flow but this number is usually very low for the javac-specific decompilers, Jad and SourceAgain. This can be explained by the code pattern-matching performed by those tools. Out of all the benchmarks, Sld and Trf were the only ones that had a sizable number of abrupt statements.

Considering that bytecode represents all control flow only through `If` and `Goto` instructions; a naive tool-independent decompiler will take the simplest route and transform these into `Labelled-Blocks` containing `breaks`. An artifact of this is visible in Dava in the form of 13 labelled blocks created over all benchmarks as compared to none by other decompilers (the original source did not have any labelled blocks either). Dava does not create as many abrupt statements as one would expect considering its generic algorithms. In some cases it even produces fewer abrupt statements than Jad and SourceAgain.

4.1.4 Conditional Complexity

Conditional complexity increases as boolean subexpressions are aggregated using the && or || operators since it is a measure of the level of conditional nesting. Figure 1.d shows that, 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

[6] The graphs do not show results for benchmarks for which the metrics are the same, or nearly the same, for all versions of the benchmark.

using the negated flag and vice versa. An exception to this is the Sld benchmark. All the decompilers increase the complexity by almost the same amount. They all detect and exploit conditional aggregation opportunities and, in doing so, increase the complexity.

Comparing Dava to other decompilers we see higher conditional complexity for Dava. This is of concern since a higher conditional complexity indicates that the decompiler is representing the same conditions in a more complicated way, hence making it harder to understand the code.

The metrics of the original sources indicate that a conditional complexity between 2 and 3 is normal. Indeed, an inspection of Sld and Trf (conditional complexities higher than 3) showed very convoluted conditional expressions.

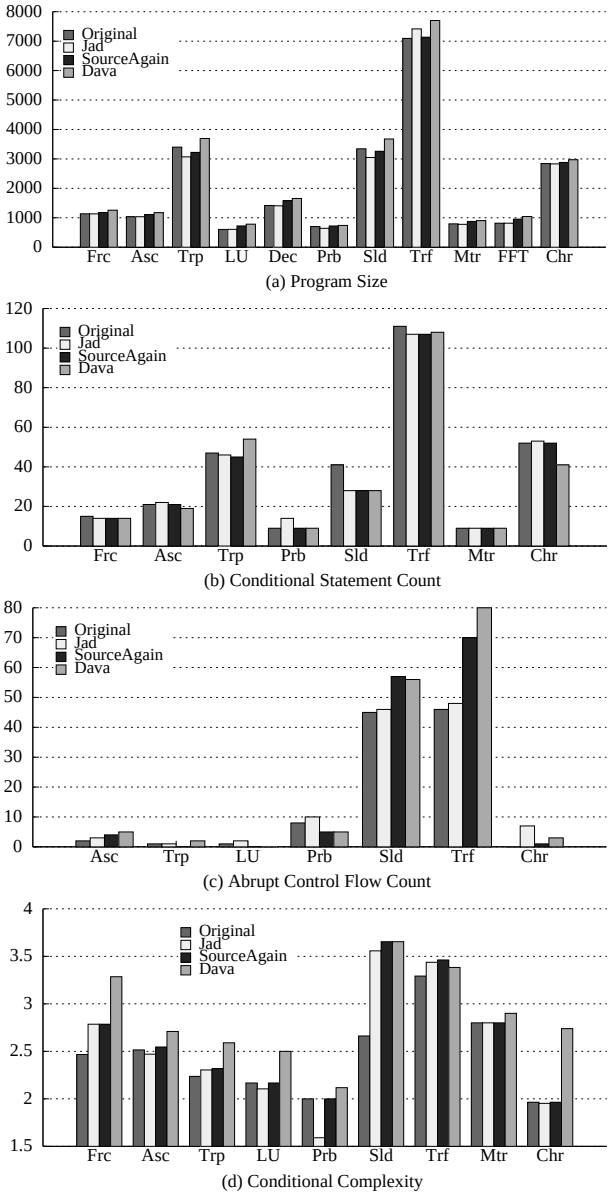

Figure 1. Decompiled Code Metrics

4.2 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 representation, decreasing the overall comprehensibility, but not necessarily confusing a decompiler. Obfuscating techniques directed at low-level code, however, can often 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 complicated dead code guarded by opaque predicates [2]. This introduces a new level of complexity in the program that a decompiler is unable to remove or simplify through static program analyses. We use Zelix KlassMaster [6], a notable example in this category of second-generation tools.

Finally, new ideas in Java-specific obfuscations is leading to a new third generation of tools. JBCO (Java Byte-Code Obfuscator) [1] 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.

To perform the experiments in this section we created our baseline by compiling the application using the `javac` compiler and then decompiling the produced class files using Dava. We used Dava because it is the only decompiler robust enough to be able to decompile code after first- and second-generation obfuscations [10]. To create the obfuscated versions of the source code we first applied the obfuscators (KlassMaster and JBCO) to the class files and then decompiled the obfuscated classes using Dava.

By comparing the Dava versions with JBCO and KlassMaster one can observe the impact of the two obfuscators on the metrics. It is interesting to note here that Dava contains advanced analyses targeting obfuscated code. These include some control flow simplifications, copy elimination and advanced dead-code elimination. [7]

4.2.1 Program Size

Figure 2.a shows the program size metric. It is clear that both JBCO and KlassMaster increase the size in all cases, with a greater increase in size for KlassMaster. This is expected because KlassMaster adds dead code guarded by opaque predicates which can therefore not be removed by

[7] Although we computed all the metrics for both obfuscators we only show results for KlassMaster in many of the figures. Since computing our metrics requires decompilable bytecode, we had to disable JBCO's advanced obfuscations which render even our tool-independent decompiler useless. Because of this JBCO has no effect on some of the metrics and we omit these from the presentation. More on this in Section 5.

the static analyses performed by Dava. JBCO size increases are due to the addition of methods that are used to invoke library calls through an extra level of indirection.

4.2.2 Conditional Statements

Figure 2.b demonstrates a large increase in conditional statement count after obfuscation by KlassMaster. This is consistent with KlassMaster's technique of introducing redundant or dead code enclosed by simple `If` statements.

4.2.3 Abrupt Control Flow and Labelled Blocks

There is a marked increase in abrupt statements (particularly in Trp and Chr) in the code obfuscated by KlassMaster (Figure 2.c). This metric seems particularly useful in identifying obfuscated code. Whereas programmers make sparse use of abrupt statements, control flow obfuscation techniques intentionally add these to complicate the control flow, as depicted by a sharp increase in our metric. This clearly indicates decreased code readability. Correlating closely with the number of abrupt statements is an increase in labelled blocks with a count of 11 in unobfuscated and 80 in the obfuscated code. It is interesting to note that it is the complicated control flow that causes most javac-specific decompilers to fail. Dava succeeds, due to its use of graph-based restructurings, but needs to resort to labelled blocks.

4.2.4 Conditional Complexity

Conditional complexity (Figure 2.d) shows a decrease in complexity mainly due to the opaque predicates (conditional statements with un-aggregated boolean expressions) introduced by KlassMaster. Although the number of conditional constructs increases, the average conditional complexity decreases. Another reason for the drop is that the original bytecode is intermixed with obfuscated code inhibiting Dava's pattern-based code simplifications.

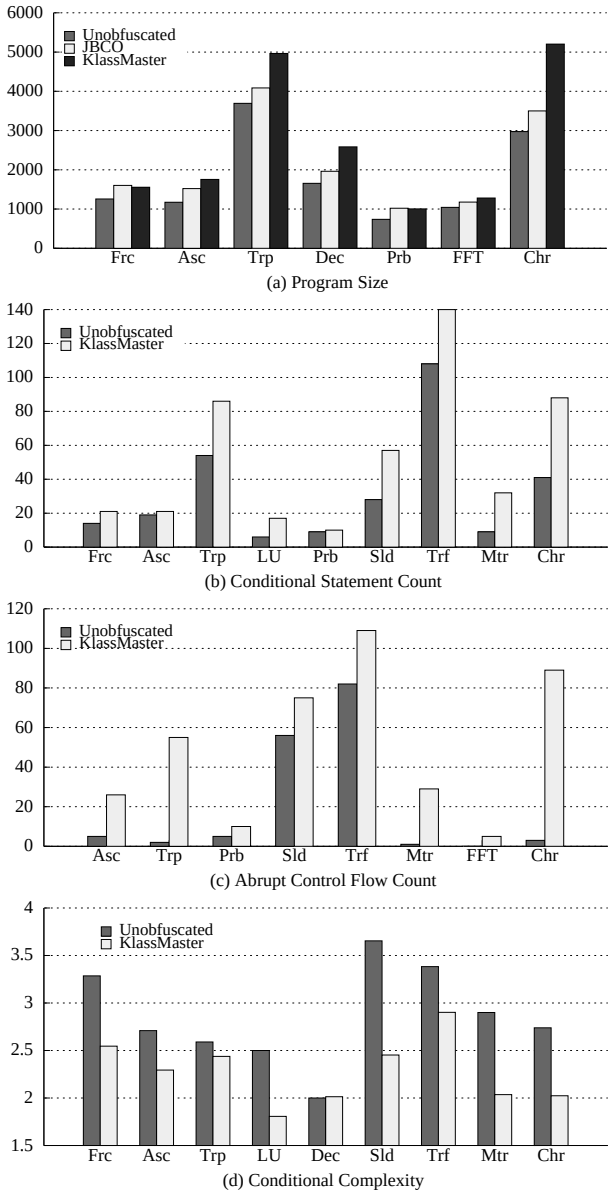

Figure 2. Obfuscated Code Metrics

5 Threats to Validity

We use lexical and textual complexity along with the structural complexity of a program as a key ingredient in our proposed metrics. We have also experimented with other metrics like the number of local variables and identifier complexity. These also provide some interesting results[8]. Our set of metrics is an attempt to see if quantitative assessment of quality of decompilation and obfuscation makes sense. As appropriate metrics are identified we intend to include those in our metrics suite.

For the decompiler experiment the bytecode was produced by javac. Since both Jad and SourceAgain are tuned to decompile javac-generated bytecode we are unable to classify their use in decompiling arbitrary bytecode. Javac-specific decompilers tend to perform much worse when bytecode is not from `javac` [10, 11]. In such cases our metrics should show an increase in complexity for Jad and SourceAgain whereas Dava's metrics should remain the same. However, the reality is that Jad and SourceAgain often do not work at all on this sort of bytecode.

To further validate the use of our metrics we would like to use programmer opinion. Given different decompiler outputs, programmers will be asked to classify their choice of output in terms of code complexity and ease of understanding. They will also be asked to rank the different outputs in terms of our metrics and the relative importance of

[8] A detailed version of this paper, including more metrics and results, is available at www.sable.mcgill.ca/publications/techreports/#report2006-4

the metrics in their decision. These subjective judgments will help to demonstrate the effectiveness of our metrics suite in capturing code complexity.

6 Conclusions

This paper is a first attempt 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 designed to distinguish between two semantically-equivalent programs. To concisely represent the result we experimented with a variety of combinations of these metrics, ultimately deriving it as a weighted summation. Each metric (program size, conditional statements, abrupt statements, labelled blocks and expression complexity) was normalized with respect to the value for the original code and multiplied by a constant of 0.2 [9]. This gives us the property that comparing the original source to itself has an overall metric of 1.

To evaluate the metrics we examined two javac-specific decompilers: Jad and SourceAgain and the tool-independent Dava decompiler. Figure 3 gives the results for our overall complexity metric. Apart from a few outliers for Jad (LU, Prb and Chr)[10], the metrics demonstrate the general belief that javac-specific decompilers produce code that is very close to the original source (these benchmarks have not been obfuscated and thus javac-specific decompilers work well). Although, compared to other decompilers Dava produces slightly complex code, this is a small price to pay for the decompiler's increased applicability.

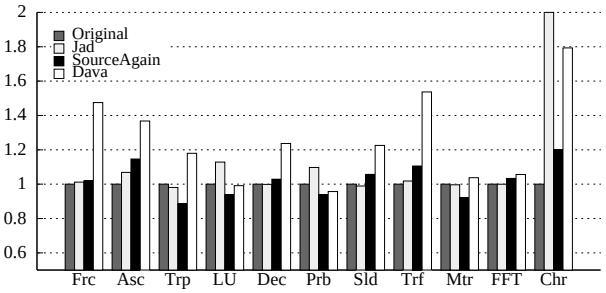

Figure 3. Overall complexity (decompiled code)

We also used the metrics to evaluate two obfuscators: a simple version of our JBCO obfuscator and KlassMaster. The overall complexity metric for obfuscators (Figure 4) shows that obfuscators are quite effective at producing complicated bytecode that decompilers are unable to transform back into anything resembling the original source. Even with Dava's simplification analyses, the results for KlassMaster show that the impact of obfuscation is severe enough to have a drastic impact on code comprehension.

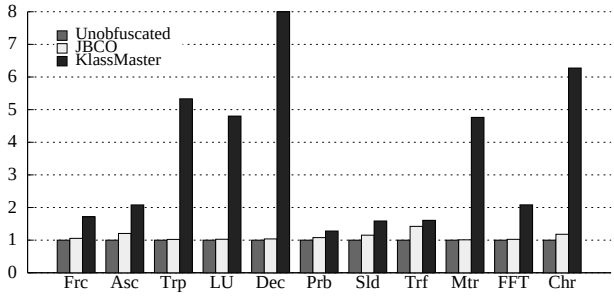

Figure 4. Overall complexity (obfuscated code)

A limitation of our approach is that we need decompilable bytecode to compute our metrics. We had to disable most of JBCO's obfuscations since they thwart decompilation attempts even from a tool-independent decompiler like Dava. This is apparent from the insignificant increase in the overall complexity of JBCO obfuscated code as seen in Figure 4. To this end we have come to the conclusion that to measure obfuscated code additional metrics need to be developed that operate on a lower-level representation than Java source code. This would enable measuring complexity from partially decompilable obfuscated code.

References

[1] M. Batchelder and L. Hendren. Obfuscating Java: the most pain for the least gain. In *Compiler Construction*, pages 96–110, 2007.

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

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

[4] Jad. `http://www.kpdus.com/jad.html`.

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

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

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

[8] J. Miecnikowski and L. J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In *Compiler Construction*, pages 111–127, 2002.

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

[10] N. A. Naeem. Programmer-friendly decompiled Java. Master's thesis, School of Computer Science, McGill University.

[11] N. A. Naeem and L. Hendren. Programmer-friendly decompiled Java. In *International Conference on Program Comprehension*, pages 327–336, 2006.

[12] Source Again. `http://www.ahpah.com/`.

[13] K.-C. Tai. A program complexity metric based on data flow information in control graphs. In *International Conference on Software Engineering*, pages 239–248, 1984.

[9] We intend to improve on this heuristic using the study proposed in Section 5

[10] Jad produces really complicated code in the case of Chr because of its inability to properly handle floating point operations