Computer Systems Project Proposal
Java Decompiler

Joshua Cranmer

November 12, 2007

Abstract

This project is a decompiler capable of processing outputted Java bytecode into fully-recompilable and functionally-equivalent source code. Several people could benefit from being able to decompile source code, including potentially large companies.

Keywords: decompiler, static analysis, reverse engineering

1 Introduction

1.1 Scope of Study

This project is designed to write a Java decompiler. At first, the goal will be to process class files, one at a time, that were generated by the Java SE 6 compiler. Later, the program will be modified to be able to handle whole JAR or ZIP files at once, and (if time permits) through multithreading to take advantage of multiple-core processors. It is also planned to support some of the older compilers, and, hopefully, optimized code as well.

1.2 Type of research

This project is partially applied research and partially pure research: the goal is to produce a final product, but some of its features may be more esoteric in their application.

2 Background of current research

At this time, there exists several Java decompilers. In the early days of Java, several decompilers were written that took advantage of the ease of decompiling bytecode, prompting several articles to be written detailing the scope of issue, including fooling decompilers. Most of these early decompilers are helpless at modern code, and several no longer exist. Furthermore, very few decompilers exist for non-Java programs. A search of SourceForge revealed one Flash decompiler, a Python decompiler, one C decompiler incapable of decompiling even simple code (although it is innovative in its usage), Boomerang (another C decompiler), and several defunct Java decompilers. Aggregating all together, there are currently only four decompilers of note:

- Jad, the best Java decompiler currently out there (although closed source and written in C).
- Boomerang, the best open-source C decompiler and the only one easily obtainable.
- JODE, the best open-source Java decompiler, but seems to be more-or-less abandonware.
- Hex-Rays, a decompiler that plugs into the popular IDA program. Closed source, expensive, and requires another expensive program to use.

Ilfak Guilfanov, the developer of Hex-Rays, has a blog with several discussions of automated decompiling and reverse engineering. Since he is using direct executable code, there is a heavier focus on dealing with non-trivial compiler optimizations (division of 64-bit numbers is an example), but much is still relevant for decompiling Java code.

3 Procedures and Methodology

The plan for writing a decompiler is iterative. At each stage, new features, progressively harder, will be added. So at first, for example, only a simple function that does nothing will be decompiled. Then functions that call other functions, will be handled, followed by functions with simple arithmetic transformations. Control flow will be added on afterwards, handling if/else statements first, followed by while/do-while/for loops and finally loops with break/continue statements. After that, more complex statements will be added: the ternary statement, try/catch/finally blocks, switch blocks, and synchronized statements.

While handling the different levels of decompilation, various filters will be set it. These filters will transform the source code from compilable to readable. First, the default return statements at the end of functions would be removed. Later, default constructors would be excised if unnecessary, and default static initializers for enums would be eliminated as well. Eventually, the class files will fully handle inner classes by removing several of the accessory `access$XXX` functions.

Testing will consist of a directory of files containing various methods representing different constructs to test the code. The testing process will consist of compiling the source code, decompiling it, and comparing the sources. Once recompilable code is produced, the test will actually recompile the code and compare the differences in the bytecode. If that proves to be too stressful, the test will merely consist of executing the resulting program to test functional equivalence. It is expected that for full stress-testing, portions of the actual Java source code will be used for testing (for example, using java.awt.Component to test the limits of decompiling large files).

4 Expected results

Decompiling is actually very important in several cases. One statistic says that approximately 10% of source code is presumed to have been lost; decompiling can help recover source code when it is lost from the binary executables. In addition, decompilation can be used to analyze malware to try and defend against them. Another case is for reverse engineering closed protocols for purposes of interoperability. What my project would also fulfill would be investigation into more extensive type analysis (recovering the generic arguments from the executable code). A final reason for decompiling would be helpful to large corporations: decompilation could be used to detect patent and copyright infringement code.