Computer Systems Research Project
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

It is very commonly asked if it is possible to get working source code from the executable binaries, and the most common replies are along the lines of "It's impossible," often including references to turning hamburgers to cows or similar. These replies are, simply put, false in every way. Compiling is not anything like turning cows to hamburgers. The latter process involves ripping out large portions of the cow and recombining much of the rest, in essence, fundamentally changing every aspect of the cow. Compiling, instead, only discards structural information and rewrites the rest in a way that a processor can understand better. Executable code is essentially source code without the structure.

The primary problem with decompiling is that the problem tends to be ill-defined. Executable binaries represent the main input,[1] the other inputs

[1]There are some decompilers that might use other inputs as bases, the most common being assembly code. This will be treated later.

being auxiliary information that can help the decompiler or mere stylistic guidelines. But the output is hard to define. Three definitions quickly come to mind: the original source code used to compile the binary; a source code that will, fed into the original compiler with the same options, produce the same binary code; and source code, when fed into a compiler, will produce not necessarily identical but functionally equivalent source code. The first one is obviously impossible, the second one (surprisingly) is often unfeasible or undesirable, so the latter is what most people focus on.

This project's primary goal is to be able to produce fully recompilable Java code given the class files input into a Java virtual machine. The original versions will focus on being able to handle only the output of the most recent Java compiler, Sun Java 6 SE, in unoptimized format. Time permitting, support for handling older versions (mostly limited to the more complex `finally` handling) will be added. Further improvements to handling optimized code and code not produced from any standard Sun Java compiler will be next, followed by improvements for handling entire JAR or ZIP files at once.

2 Background

In the realm of computer code, there are several layers of code. Amongst the so-called *high-level programming languages* (like Java, C#, Python, or LISP), there are variations in expressiveness and readability. These variations make ranking the languages in any hierarchal order next to impossible. Java, for example, as string capabilities that far outreach those of LISP, but LISP is more effective at dynamic interpretation than Java. However, it is possible to classify these languages by levels of expressiveness—loosely defined, how much the language impedes certain tasks. This same metric can be applied to lower-level languages and even executable code representations.

In this hierarchy of languages by expressiveness, the most expressive languages would be machine code as executed by a native processor, such as Intel's i686 instruction set or the instructions for a MIPS processor. Corresponding almost exclusively one-to-one with these 'languages' are the respective assembly languages, mostly a series of mnemonics for the actual instructions (although certain operations are prohibited in the assembly language that are permitted in the machine code). Slightly less expressive are various types of portable assembly, like GCC's RTL. Older languages like C or FORTRAN are the next level, representing easier representations of

	Machine code	Assembly	Source code
Machine code	Porting	Disassembling	Decompiling
Assembly	Assembling	Porting	Depends on author
Source code	Compiling	Depends on author	Source code transformation

Table 1: Nomenclature of various transformations

the same information with a thin veil of type-checking. In the next tier are bytecode languages, like that of Java or Python, which retain significantly more structure and have much more intensive sanity checks. In the top tier lies many of the modern languages, with complex features like static type-checking or stack-unrolling exception handling.

This hierarchy is generally collapsed into four segments: machine code, bytecode,[2] assembly, and source code. Transforming code between these various classes has different names, specified in Table ??. Conversions between source code and assembly are not typically used, so their common names will vary considerably, mostly depending on whether or not the context dictates where assembly falls on the line between machine code and source code. In terms of decompiling, determining the assembly from machine code is much more difficult than the source code from assembly, so these lines are more viewed as source level analyses, albeit more difficult than the more common ones. [?, ?]

Several examples of decompilers exist at the present time. 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

[2]In most circumstances, bytecode and machine code are considered identical classes. The distinction is only important when classifying difficulties between various transformations.

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.

3 Class File Parsing and Signature Handling

The first stage of the decompiler is to parse the incoming files. Most of the internal representations are handled in the `info` package. Internally, the class files are handled through a service architecture: a central class, `ClassPool`, contains a pool that manages the various known classes. Class files are given to this pool by registering various `ClassSource` interfaces that can produce an input stream for a requested class. The main shell registers a source based on the files passed into the command line arguments and then proceeds to find outputs for all of these classes, by requesting fully-decompiled versions from `ClassPool`.

Whenever a class (internally represented using `ClassInfo` references) is requested, a level of decompilation is requested. If the class has not been handled yet, the class pool grabs the input stream and starts parsing it to the required level. If it has been handled, the internal decompilation level is compared to the requested level and proceeds until the requested level is reached. For all but the last two levels (`PROCESSED` and `FILTERED`), the stream is parsed to the given point. The possible levels to parse to are the header information, constant pool, class metadata, fields, methods, and annotations; all parsing is done by the `ClassParser` class, which has intimate access to the internals of `ClassInfo`.

Should the parsing run into any problem that violates the Java VM specification, it will try to continue whenever possible, logging a verification error. Examples of these errors are mismatched magic numbers, illegal flags, and improper versioning. Should continuation prove impossible, the decompiler stops attempting to parse the class, printing out an error. An example would be illegal constant pool tags; other examples include I/O errors.

When the processing stage is reached, the input stream is discarded to conserve memory and the class then focuses on trying to make sense out of attributes. Signatures are attached to methods and fields at this stage, and code is actually physically decompiled here. Signatures are not parsed here, but are lazily evaluated when the class is being printed.

The first prototypes of the decompiler ignored the `Code` attribute (where all the instructions are actually stored) and focused on printing out the full signatures, including generics. These signatures are decoded through the hand-written `SignatureParser` class. This class is currently not optimized for speed: it switches from using a `StringBuilder` to using a `String` several times, a process which can incur very large overhead costs. This class has five entry points; two for the internal field and method types and three for the stored generic signatures. It is also capable of returning full generic signatures, as example **??** shows. The string on the bottom is the actual signature stored in the class file while the above code is the returned output of the decompiler.

```
abstract class GenericsTest {
  public abstract <T extends java.lang.Object, E extends
      java.lang.Throwable> T foobar(T var_0) throws E;
}
```

```
<T:Ljava/lang/Object;E:Ljava/lang/Throwable;>(TT;)TT;^TE;
```

Example 1: Generic method example

References

[1] Boomerang. Vers 0.3 alpha 31 Oct. 2007 http://boomerang.sourceforge.net/download.php

[2] Emmerik, Mike Van. “PhD Confirmation Report: Type Inference Based Decompilation.” U of Queenland, 2003. 31 Oct 2007 http://www.itee.uq.edu.au/ emmerik/_confirmation/confirmation.ps.gz

[3] Guilfanov, Ilfak. “Portable output for Assembler.” Weblog entry. 24 Apr. 2006. Hex Blog. 30 Oct. 2007 http://hexblog.com/2006/04/portable_output_for_assembler.html

[4] Java Optimize and Decompile Environment (JODE). Vers. 1.1.1. 31 Oct. 2007 http://jode.sourceforge.net/download.html