Computer Systems Research Project
Java Decompiler

Joshua Cranmer

June 11, 2008

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 1. 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. [2, 3]

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,[1] the best open-source C decompiler and the only one easily obtainable.
- JODE,[4] 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 1 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

Later prototypes developed the full capability to correctly reproduce annotations that were retained in the class file. Annotations have some confusing rules: in particular, an annotation element cannot have a type of `Object` or multidimensional arrays. The decompiler can decompile both the declaration of an annotation type and the use of an annotation on classes, methods, method parameters, and fields. Annotations in other places are not passed down to the source code, so their proper decompilation is impossible.

4 Static Stack Analysis

The Java VM specification specifies a total of 202 opcodes, as well as reserving three more (a breakpoint and two implementation-defined opcodes). Of this list of 202, 6 are not used in the Java 5 or Java 6 compiler. The opcodes `jsr` and `ret` were deprecated because of their incompatibility with the new stack-frame verification; `goto_w` and `jsr_w` are 4-byte instructions that are unused since code is limited to 2-bytes; `nop` is unused for obvious reasons; and, finally, `invokedynamic` was created for the ease of dynamic programming languages, and its semantics are not concisely represented in Java syntax.

The Java VM is ultimately stack-based. An array of local variables is maintained, and operands are pushed onto or off of the stack by the various operators. Other bytecode-interpreted languages have similar semantics, but most modern processors are register-based, where the operator declares which registers it operates on. This stack makes decompilation easier, since the stack starts empty and is required to end empty, unlike real processors where the registers may be used as arguments or return parameters.

Of the previously mentioned 202 opcodes, many are shortcuts for what could be larger opcodes. `iload_1`, for example, is semantically identical to `iload 1`, both of which are very close to `aload 1` or even `istore 1`. Taking advantage of these similarities, the opcodes are simplified into the following 22 classes:

`LoadConstantInstruction` Push a constant value onto the stack

`LoadStoreInstruction` Push a local variable to the stack or pop the stack to a local variable

`ArrayLoadStoreInstruction` Store or load a variable to an array

`PopInstruction` Pops one or two values from the stack

`DupInstruction` Duplicates potentially several values on the stack

`SwapInstruction` Swaps the top two values on the stack

`ArithmeticInstruction` Performs an arithmetic operation on the top one or two variables on the stack

`CompareInstruction` Performs a comparison on the top two variables on the stack

`IncrementInstruction` Increments an integer by a specified count

`PrimitiveConversionInstruction` Converts between the primitive types

`IfInstruction` Conditionally jumps based on a variety of conditions

`GotoInstruction` Unconditionally jumps

`JSRInstruction` Unconditionally jumps, but pushes the current address on the stack

`RetInstruction` Jumps to the address popped from the stack

`SwitchInstruction` Executes a switch instruction

`ReturnInstruction` Returns control from the function

`FieldAccessInstruction` Accesses a field of a class

`InvokeInstruction` Invokes a method

`NewInstruction` Instantiates a new class

`ArrayNewInstruction` Instantiates a new array

`ArrayLength` Pushes the length of the array onto the stack

`ThrowInstruction` Throws an exception

`CastInstruction` Checks the type of a class

`MonitorInstruction` Enters or exits a synchronization monitor

Every opcode, except `nop`, `goto`, and `goto_w`, involves operating on the stack. The first stage of decompilation is to therefore analyze the stack. Analysis of the stack involves determining a few components: ensuring that the VM types (integer, float, long, double, and object) are proper at all times, construction of a few invariants for later analysis, and the determination of variable typing and scope. All three operations are carried out simultaneously in the first analytic pass through the bytecode (the initial construction technically counts as a pass, although the only analysis it performs is the insertion of pseudo-bytecodes and the removal of `GotoInstruction`. The first operation—VM typing—is trivial and carried out normally as part of

the VM's bytecode analysis; it is the last two operations that present the problems.

The main invariant that is needed for further decompilation is the invariant that any instruction with two parent nodes should have an empty stack. This requirement makes the decompilation of many of the opcodes simpler by ensuring that the operand can only be one thing and not several. For the most part, Java code already satisfies this assumption, with one exception: the ternary operator. Since the bytecode is more liberal in this aspect than Java permits, some sort of heuristic needs to be used to determine whether or not this branch pair is a conditional expression or an optimized branch.

4.1 Variable Analysis

Variable typing and scoping is the most difficult aspect of decompiling, and it is a problem shared by all types of decompilers, be they native-code decompilers or bytecode decompilers. An interesting aspect is that if either type or scope is known, the other becomes simple to find, whereas finding both together is much harder.

The harder problem by far is scoping. To see its difficulty, one needs to learn a little about the structure of the Sun Java compiler. Each variable is mapped to one spot in the local array buffer, or two if it is a `long` or `double`. This spot is reserved for it from the point of declaration to the end of its scope. The naïve implementation would be to rely on the scope of control structures to dictate variable scope, but this can lead to some over-scoping of variables. Even worse is the fact that Java allows the introduction of arbitrary variable scope. Therefore, a decompiler cannot rely solely on control structures even if code only compiled from the Sun Java compiler was input (there is one simplifying caveat: the argument variables' scope is that of the entire program, so the Sun Java compiler will never reuse these slots).

The solution to variable scoping is by using a form known as SSA, or static single assignment. More commonly known for its use in compilers, SSA form is a modified form of code where each variable is limited to only one assignment.

Currently, only a mild portion of SSA is formed. The stack analyzer is incapable of handling branches, and control structures with it.

5 Control Flow Structure Recreation

Historically, the earliest decompilers used pattern recognition for this portion of decompiling. As one might imagine, however, this type of decompiling was very fragile and does not scale at all, especially in the modern days of optimizing compilers and complex features: discussion of how try/catch features work in g++ is difficult enough, let alone trying to discover it based on the posted assembly code. Pattern recognition does, however, hold a place in modern decompilers: this is how the post-decompilation transformation step is performed; this part, however, does not impact the compilability of code (from a theoretical standpoint, although the nature of compile-time hacks may dictate otherwise).

In Java bytecode, as with much of machine code, there are essentially only two control structure commands: unconditional branching and conditional branching.[3] It is from these two commands that if-else statements and all loops are constructed. However, because goto is not permissible in Java code, it is possible to construct arbitrary constructs in the bytecode which do not have direct Java-representable formats; for this reason, Java code obfuscators rely very heavily on such control flow mangling.

Recovery of control flow is performed in multiple stages. First, a graph of "blocks" is created (where a block represents all sequential instructions that are not the target of a branch or the source of one). Transformations are performed iteratively until the code cannot do any better or only one block remains (hopefully the latter case is trigged). Finally, the blocks are deconstructed and the resulting node lists replace the old ones. The interesting part is the iterative transformation sequence.

The most basic transformation detects an if-else block. A block which has two children whose sole child are the same node becomes and if-else block. An if block is detected by similar transformations.

The more complex transformations are loop blocks. Conceptually, a loop is distinguished by a circle in its control flow graph, with a single entry point and multiple potential exit points. Unfortunately, although the detection of a loop is relatively simple, the combination of loop detection and simple if-statement detection proved impossible to perform under the current structure recreation architectures, and the necessary modifications to allow such

[3]There are a few more commands related to try-catch-finally blocks and switch blocks, but those are outside the scope of this project for now.

a mechanism proved too daunting to be able in the timeframe. Such a modification would have been the third refactoring necessary in the project (an earlier redesign of the process, a second redesign of stack analysis, as well as other rather-large modifications in varying parts of the code).

6 Post-Decompilation Transformations

Surprisingly, much of Java since Java 1.0 is essentially a hack in terms of the bytecode. The only real changes to the bytecode are the now-allowed use of class in the `ldc` instruction, a slight change in the lookup of the nonvirtual method lookup, and the deprecation of the `jsr`, `jsr_w`, and `ret` instructions due to the difficulty of verifying them under the new Java 6 stack verification model. Everything since then–including, but not limited to, generics and inner classes–is merely a compile-time hack, see example 2 for a partial comparison of a case involving enums and inner classes. It is also possible that proposed Java features—including most notably closures—will be similarly solely compile-time hacks, reification of generic types.

These advanced constructs can be detected by simple analyses. Each synthetic method is not reproduced to output if its existence can be explained and rectified. So far, no explanations or rectifications are performed, due mostly to a lack of time and complexity.

7 Future Work and Applications

I will be the first to admit that my code as it stands is not very impressing. Internally, it is a mess, mostly due to the convolutions needed to get the CFG recovery portions (located in the `code.*` hierarchy of my code) to work even at all. From my basic inspection of other decompilers, this problem is shared between other decompilers. As I continue to improve my decompiler in the coming years, I will be relying on much more in-depth analysis of these other decompilers as a basis for better writing this code, if not actually building off of these compilers directly.

The biggest future application I foresee would be to open up the core of the decompiler to allow several different bytecode languages or maybe even native languages to use the same basic core as a decompiler backend. Unfortunately, this relies on me finishing the CFG code. I will also add a

```
public enum EnumTest {
  A, B;
  private static int foo = 5;
  static class Bar {
    public String toString() {
      return Integer.toString(5);
    }
  ]
}
```

```
public class EnumTest {
  private EnumTest A, B;
  private static int foo = 5;
  public static int access$100() {
    return foo;
  }
}
public class EnumTest$Bar {
  static class Bar {
    public String toString() {
      return Integer.toString(EnumTest.access$100());
    }
  }
}
```

Example 2: Simplified example of how some constructs are handled internally

graphical front-end to the program, as well as documenting it better.

All future work will be (eventually) developed on an open-source project server, probably `www.sourceforge.net`; I'm thinking of ultimately calling it the JBCA, Java Bytecode Architecture. All who wish to help or be informed are welcome to email me or visit the page.

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

[5] Polygot Java Compiler Architecture. Vers. 2.3.0. 11 June 2008 http://www.cs.cornell.edu/Projects/polyglot