Java 6 Decompiler

Joshua Cranmer
TJHSST Computer Systems Lab
2007-2008

Introduction

This project aims to create a decompiler capable of processing outputted Java 6 bytecode into fully-recompilable and functionally-equivalent source code.

Decompilers are nothing new; they have existed in some form almost to the beginning in programming (the first prototypes were made in the 1960s). However, the quality of decompilers has increased little in this time frame due to some fundamental problems with decompiling, most notably the theoretical impossibility of disassembly.

Since the advent of Java and other virtual machine-compiled languages, however, decompilers have improved. The nature of bytecode places it at the equivalent of an assembly language, bypassing the difficult disassembly step, and, additionally, provides plenty more metadata that aids in decompilation.

Several Java decompilers exist now, although most of these have become defunct. The only one in widespread use is Jad, the source code of which is not released. However, none of the existing stock can handle the new features introduced even in Java 1.4, let alone the mass changed by Java 5.

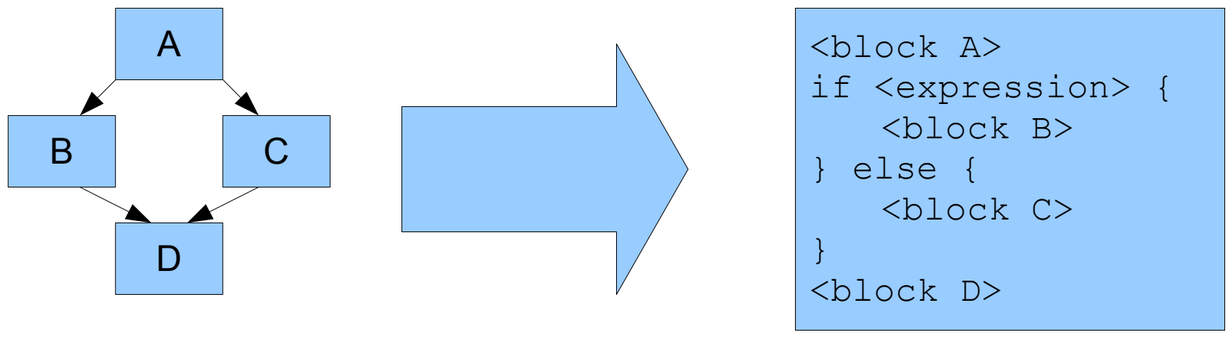
Reasons for Decompilation

- Finding bugs in program
- Finding vulnerabilities
- Finding malware
- Compiler code verification
- Comprehending algorithms
- Creating interoperability
- Induce customizability
- Porting code
- Create maintainable source code
- Fixing bugs without patching binaries
- Add features to a program

Control Flow Graph Recovery

The hardest portion of decompilation was the CFG recovery. I could not finish this by the end of the year, despite spending several months almost entirely on it and rewriting the entire module twice. The biggest block in implementing this was detecting and unifying loop blocks; the detection part was conceptually easy, but the implementation proved difficult, especially when trying to unify it, even after ignoring the difficulties posed by do-while loops, and break/continues. Not implementing loops left only if/else statements working, and try/catch/finally statements, switch statements, and synchronized blocks were not even considered due to the difficulty.

To the right, a diagram of a basic type unification.



Performing decompilation

Decompilation is performed in three parts, one of which ends up being simple. The three parts are, in order, signature recovery, code decompilation, and post-decompilation transforms. Code decompilation can be further split into three substeps: stack analysis and variable recovery, trivial transformations, control-flow graph recovery.

My decompiler can perform nearly all aspects of the first part (some objects, like enum fields, that logically belong in this category can only be discovered in the third part); the internal signatures are transformed using a simple recursive descent parser contained in a single 379-line file (out of 6082 lines of code).

Unfortunately, no post-decompilation transforms are done, since there is insufficient to implement them, and the barrier of the second step proved too difficult to finish by the second time.

Stack Analysis

Stack analysis is performed using single-static assignment, a translation of code where each variable is assigned to once. This technique is widely used in compilers, and is used in modern decompilers. The biggest difficulty posed by SSA is the type unification that needs to be done after the fact; this is an area of active research.

My program can do basic type inference and does pretty well at generating variables, but this feature was not stress-tested and likely does not work in the general case. It was turned off when working on CFG recovery for simplicity's sake.

jcranmer@loman:~/techlab/documents

```
import java.util.HashMap;
import java.util.LinkedList;
import util.Logger;

public final class ClassPool {
    private static HashMap<String, ClassInfo> classes;
    private static LinkedList<ClassSource> sources;
    private ClassPool() {
        super();
        return;
    }
    public static ClassInfo getClass(String className) {
        sources.iterator();
        Iterator var_1;
        store java.lang.Object, 1
        Logger.verbose("Retrieving class from source");
        new HashMap();
        putfield info.ClassPool.classes
        new LinkedList();
        putfield info.ClassPool.sources
        return null;
    }
}
```

Example screenshot of an example output. Note the use of proper indentation and (not seen here) proper 80-character overflow. Also note the lack of post-transformation in the private constructor.

Generic signatures are decompiled, as well as the recovery of new variables, and the decompilation of certain simple bytecodes.

Variables can be detected correctly, but assignments are not detected. The correct code for the second method would look as follows (several ports have been elided for brevity):

```
public static ClassInfo getClass(String className) {
    for (ClassSource source : sources) {
        if (source.hasClass(className)) {
            Logger.verbose("Retrieving class from source");
            if (classes == null)
                classes = new HashMap<String, ClassInfo>();
            ClassInfo c = source.getClass(className);
            classes.put(className, c);
            return c;
        }
    }
    return null;
}
```