RJ 10022 (89113) May 9, 1996
Computer Science

Research Report

Jive: A Java™ Decompiler

Daniel Ford

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

RJ 10022 (89113) May 9, 1996
Computer Science

Jive: A Java™ Decompiler

Daniel A. Ford

Department of Computer Science
IBM Almaden Research Center
650 Harry Road
San Jose, California. 95120-6099
daford@almaden.ibm.com

Abstract

This paper describes a decompiler for Java™ virtual machine instructions (bytecodes). It describes how the stack architecture of the Java™ virtual machine and the symbolic contents of Java™ class files, combine to make decompilation of Java™ executables particularly straightforward. It describes the processing required to parse Java™ machine instructions into Java™ source, giving a parser organization and various semantic rules needed to guide the process. The paper then describes the implementation of the Java™ decompiler "Jive" which is capable of reconstructing source code from a Java™ class file retrieved from the local file system or from across the internet. The paper concludes with a discussion on the likely effectiveness of possible countermeasures to make decompilation harder.

1. Introduction

The introduction of the programming language known as Java™ [1] to the World Wide Web (WWW) has created a new application development and delivery platform that seems posed to compete directly with the conventional personal computer Desktop. The central characteristic of this new "Webtop", is the downloading of platform independent executable code written in Java™ from remote servers. Much attention and discussion have focused on the security issues that surround such an arrangement, principally the danger of unwittingly importing malicious programs such as viruses.

This basic security problem faced the designers of Java™ as they originally strove to create a software environment for embedded systems ("set top boxes"). They needed to permit code to be downloaded from potentially untrusted hosts without the danger that malicious code could easily take over or disrupt the system. Their solution was the creation of a restricted execution environment embodied in the definition of the *Java™ Virtual Machine* [2]. Due to the simple architecture of this virtual machine and the high-level nature of many of its instructions, the execution environment it provides has limited flexibility for expressing computations. This characteristic is deliberate as it allows the code to be examined more easily to determine if it functions in a malicious manner.

The designers of Java™ also faced a second related problem: if code is downloaded, there is the potential that its parts may come from more than one host. If the code on one host relied upon the physical layout of code on another, then changes on the first could require changes (recompilation) of code on the second. Known as the "fragile super-class" problem, this feature is characteristic of languages such as C++ that require recompilation of sub-classes whenever their super-classes are modified.

[1] Java™ is a trademark of Sun Microsystems Inc.

Unfortunately, if code is distributed around a network, there is no guarantee that all components will be updated in a timely fashion, if ever. The solution adopted for Java™ is to delay linking and physical layout until run-time by retaining symbolic information after compilation. As a result, Java™ class files contain the symbolic names and types of all methods defined in the file as well as all externally referenced fields and methods.

Solving the security and fragile super-class problems has made Java™ suitable not only for embedded systems, but also for the secure delivery of distributed executable content on the World Wide Web. In this paper, however, we show that while Java™ appears to address the security concerns of a web user, it creates potential security problems for a web content provider. The very characteristics that make Java™ safe and useful for users of the web inadvertently allow the easy reproduction of the original Java™ source code by decompiling Java™ executables.

In the next section we discuss the structure of the Java™ Virtual Machine, its instruction set, and the Java™ class file contents. In section 3, we describe a parsing strategy for a decompiler and parser organization. We also describe the implementation of Jive, a Java™ decompiler capable of reconstructing Java™ source code from Java™ class files. In section 5, we illustrate the parsing process with an example. In section 6, we discuss the effectiveness of countermeasures that can make decompilation more difficult. In section 7, we present our summary.

2. The Java™ Development and Execution Environment

The Java™ development environment consists of three main components: the Java™ compiler, *javac*, the Java™ interpreter, *java*, and a library of class files that provide various platform independent abstractions for system and user interface functions.

The compiler transforms Java™ source code into *class files*. With Java™, there is one separate class file for each class, regardless of the number of class definitions that may be given in any individual source file. These are obtained by the interpreter through a *class loader* utility that retrieves class files from either the local machine's file system or from a remote machine via a network (i.e., the Internet). This process also performs a verification check that looks for malicious code in the class file. The exact definition of what constitutes malicious code is somewhat vaguely documented [3], but in general it is code that violates the rules of the architecture, for instance, trying to use an integer as an object reference or other incompatible type combinations.

The interpreter scans the class file to obtain the information required for physical layout and linking of the class, and to obtain the machine instructions for each method. Future versions of the interpreter may call a "just in time" compiler to transform further the virtual machine instructions into the instruction set of the native platform for improved performance.

The library of interface classes act as a front-end for a platform specific set of utilities that implement various functions such as process thread control, I/O, and graphical user interface construction.

2.1. The Java™ Virtual Machine

The Java Virtual Machine and its instruction set define the architecture of the Java™ runtime environment. The virtual machine is an abstract stack processor that maintains a unique operand stack, stack pointer, program counter (PC) and an array of "local variables," for each method invocation, Figure 1.

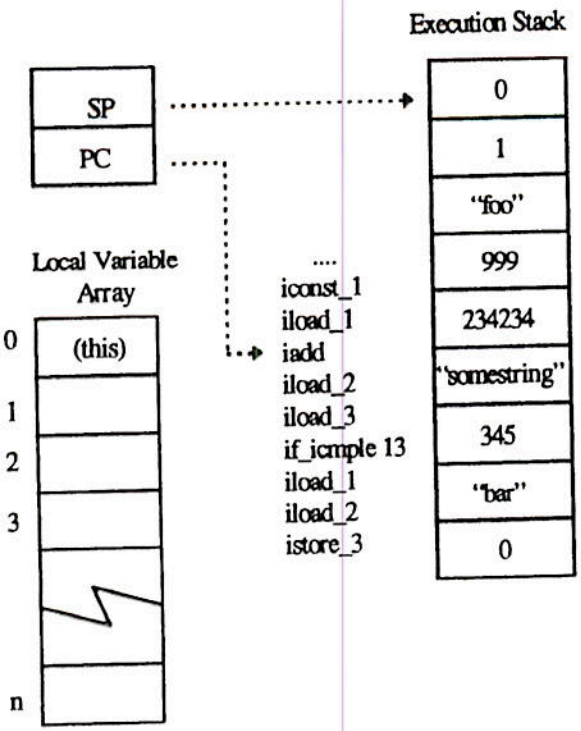

Figure 1 Virtual Machine Architecture

The local variable array serves the role of a general register set. For non-static method invocations, the first element of this array (index 0) is an object reference (*this* pointer) to the method's class instance in a heap of dynamically allocated (and garbage collected) storage. In succession, the remaining entries hold, first, the values of the method's arguments and then, the values of variables local to the method. Each entry in the array is either of built-in type, (byte, char, short, integer, float, double, long) or is an object reference (pointer into the heap).

The instruction set of the virtual machine includes the usual complement of mathematical, logical and stack operators as well as a number of high-level instructions that perform dynamic memory allocation (`new`), function invocation (`invokevirtual`, `invokenonvirtual`, `invokestatic`, `invokeinterface`), array indexing operations (`aaload`), exception processing (`throw`), type casting (`checkcast`), and monitors (`monitorenter`, `monitorexit`). The machine also contains instructions for conditional and unconditional branching, including two variations of conditional multiway branch (`tableswitch`, `lookupswitch`).

The virtual machine contains no registers, flags, status or other state information that persists after the execution of an instruction or method. This feature greatly simplifies static analysis. With such information, computation can be easily obfuscated by intermingling instruction sequences or using processor registers to pass values across subroutine invocations. Instead, the Java™ virtual machine makes computations more observable by forcing all state information and intermediate results to be stored on the method specific execution stack. The access constraints imposed by the semantics of the stack data structure (i.e., push, pop) force the Java™ compiler (or other code generator) to generate sequences of machine instructions that conform to these semantics. This makes them clear and relatively systematic, and hence, easier to analyze. Analysis is further facilitated by forcing all arguments and return values passed across method invocations to be placed on the stack. This discipline ensures that all subroutine calls and the values being passed are explicit and open to examination.

2.2. Java™ Class Files

Java™ class files are the principle unit for storing the structure and executable content of a Java™ class. The logical structure of a class consists of the definitions of its single super-class (no multiple inheritance in Java™) and the type and visibility of its fields. The executable content consists of the prototypes (name, return value and argument types) and sequences of virtual machine instructions that make up each of the class' methods.

The first entry in every Java class file is a four byte "magic number" with the hexadecimal value "0xCAFEBABE". It serves as a quick identification tag

for those utilities that process class files. This number is followed by an four byte number that specifies the version of the compiler that produced the file.

Immediately following those two values is an array of symbolic values and constants called the *constant pool*. It contains the symbolic information required for dynamic linking, as well as various numeric and string literals that are referenced by machine instructions during execution. The information required to perform dynamic linking is stored in the constant pool as method *signature* constants. These are short character strings that encode the name, type, and argument types of each field and method defined by the file or referenced by its methods.

After the constant pool comes a series of short entries that specify various access flags (e.g., public, private) for the class along with its name and the name of its superclass. The later two being specified by indices into the constant pool of character string constants that give the class names.

The remainder of the class file is reserved for three arrays, one for the *interfaces* (basically a function prototype) defined by the class, one for the class fields, and one for the class methods. The first two arrays supply indices into the constant pool for their names and type information (signatures).

An entry in the class method array may be either a method written in Java™ or, less frequently, one implemented with *native* (platform specific) code in an external library. Each method definition specifies a signature, and for non-native methods, the virtual machine instructions that implement the method.

To aid symbolic debuggers, the designers of the Java™ class file format also included provision for specifying line number information and the original names of a method's arguments and local variables. This information is optional, but will be included in the class file when the source is compiled with debug flags enabled.

3. Parsing and Decompilation

As described previously, the restricted nature of the Java™ virtual machine, particularly its stack architecture, constrains the sequences of machine instructions that can be generated to implement a particular computation. This constraint imposes a type of "computational grammar" to which valid sequences Java virtual machine instructions must conform. Other architectures impose similar grammars, but the one for Java, dictated by stack LIFO access semantics, is unusually simple. By codifying the grammar in a parser, it is possible to scan a sequence of virtual machine instructions and reconstruct, piece-by-piece, the computations it performs and a source-level representation of those computations expressed in Java™.

A grammar consists of a set of production rules, each of which is identified by a single *non-terminal*. Each rule specifies how to construct the rule's non-terminal by forming a particular ordered sequence of non-terminals and *terminals*. A terminal is a leaf in the parse tree and corresponds to basic units read from the input stream. In our case, the terminals are the Java™ virtual machine instructions that make up the program.

There are two basic techniques for parsing, *top-down* and *bottom-up*. The first approach attempts to construct a parse tree from the root, the second from the leaves. We describe a bottom-up parsing strategy.

A bottom-up parser functions by reading successive terminals (often called *tokens*) from the input stream and storing them, and any recognized non-terminals, on a *parse stack*. This stack serves to store state information for the parser. As each terminal is encountered in se-

quence, the parser performs one of two actions: *shift*, or *reduce*. A shift simply pushes the terminal onto the parse stack, and is done when the sequence of members on the top of the stack is not yet recognized by the parser. A reduction occurs when the compiler does recognize a sequence for some production rule in the top members of the stack. The reduction action replaces the recognized sequence on the stack by the production rule's single non-terminal.

The ability to decide whether to shift or reduce is a property of the grammar being parsed. The portion of the grammar that describes Java™ instruction sequences for arithmetic and logical computations is LALR(1). This means that the actions of the parser can be decided by looking at only the next terminal from the input stream. A formal proof of this is beyond the scope of this paper, but the intuitive reason is simply that the manipulations made by the arithmetic and logical operators to the execution stack, mirror exactly the shift/reduce actions the parser must make to its own parse stack.

The rest of the grammar that expresses more general code sequences, such as loops, and conditional execution is not LALR(1). An intuitive explanation for this is that the need for conditional branching to implement these constructs disrupts the correspondence between manipulations of the execution and parse stacks.

The complexity of the grammar for Java machine instruction level programs prevents a bottom-up parser from reconstructing the Java source code in a single pass as is usually case. The parser needs to store many simultaneously partially recognized execution branch structures that cannot be completely reduced until other simpler reductions are made. These reductions may precede, follow or be intermixed with the more complicated branches. The single parse stack of a conventional bottom-up parser is insufficient to maintain such "intertwined" state information. The solution is to construct a multi-pass bottom-up parser that integrates its parse state information with the sequence of machine instructions it is parsing. This is simpler than it sounds. All that is required is for the multi-pass parser to insert the non-terminals it recognizes into the instruction sequence at the point they are recognized. This organization allows the parser to maintain simultaneously the state of multiple partially recognized constructs.

On each pass in this organization, the parser scans both terminals and non-terminals. If it cannot make a reduction with the current terminal or non-terminal being scanning, the parser "shifts" by moving on to the next in the sequence. If it can make a reduction, it reduces the recognized contiguous sub-sequence of terminals and non-terminals to a single non-terminal and inserts it into the sequence in their place.

This integration exploits the correspondence between the parse stack and the runtime execution stack. Straight instruction sequences (no branching) that compute values and leave them on the execution stack will be recognized by the parser and reduced to some type of "expression" non-terminal that occupies the same relative position in the sequence. Thus, as the parser makes its passes, a machine instruction that operates on the top element of the execution stack will eventually be preceded by an expression non-terminal that represents the computation of the value it needs.

3.1. Branching and Stack Operations

The reason that the grammar for Java™ machine instruction programs cannot be parsed with a single bottom-up pass is that both branching instructions and instructions that manipulate the contents of the stack, obscure the relationship between the integrated parse stack and

the sequence of instructions. With branching, it is possible that the state of the stack at an address that is a *branch target* could be determined by computations in several places in the sequence, all of which compute some value and leave it on the stack before they branch to that one target location. Similarly, this correspondence is disrupted by instructions that manipulate the stack by duplicating, swapping or popping values. These are dynamic housekeeping operators that maintain the state of the stack during execution and require special attention during parsing.

There are two semantic rules needed to handle branching. The first is that the stack is not determined for an operator if the operator or any of the expressions that precede it, that compute values it operates on, is a branch target. The exception to this rule is that the bottommost required expression can be a target because the stack contents will be defined up to the depth required by the operator. The second semantic rule is that when a reduction is made that removes a branch, the branch is truly removed. When the last branch to a particular address is removed, that address is no longer a branch target.

The stack manipulation operators are necessary during dynamic execution to maintain correct values on the stack. For instance, it is necessary to duplicate the top of the stack when computing expressions with sub-expressions that assign to fields or variables, for example "a = 1+(b=3)". The assignment sub-expression, "b=3", actually removes the value "3" from the top of the stack, therefore during dynamic execution it is necessary to duplicate the value so that it will be available on the stack for subsequent computations.

It is possible at any point in the parsing process to encounter a duplicate instruction (e.g., `dup`, or `dup2`) where an expression is expected. They function as expressions as far as all production rules are concerned, but when participating in a reduction, both they, and the value they duplicate must both be part of a reduction. This seems counterintuitive, but dynamically computing an expression requires you to leave its value on the stack, while parsing requires you to leave the non-terminal. In the case of the assignment in the sub-expression, the assignment is retained, not the value being a ssigned.

The `dup2` instruction duplicates the top two words of the execution stack. Parsing it correctly requires each expression to carry along the size of its type. If the expression on top of the stack has a size of 64 bits (long, double), then `dup2` is equivalent to a `dup` instruction. If the size is 32 bits or less, then the top two expressions must be duplicated, equivalent to two successive `dup` instructions.

Parsing `swap` instructions does not result in any reductions, but instead requires a parser to exchange the order of the previous two expressions in the sequence. If the bottom expression is originally a branch target, then the bottom expression after the swap must also be a branch target.

The `pop` instruction removes and discards the top value of the stack. It is encountered after a method invocation to discard a return value for a method that was only called for its side-effects. It is generally not encountered elsewhere in code sequences as there is little point to computing a value that will be discarded. The presence of the pop instruction helps to distinguish function invocations (methods with non-void return values) that are procedure calls from those that are embedded in an expression.

3.2. Exception Parsing

Java™ exception handling is specified using source level *try* and *catch* blocks. These blocks, which may be nested, function such that an exception in a try block branches to the code in its corresponding catch block, Figure 2.

```
try {
     foo = 1;
     if ( bar == foo )
            throw new Exception;
     }
} // try
catch ( Exception e ) {
     System.out.print("trouble!");
} // catch
```

Figure 2 Java try/catch blocks

The relationships between these blocks at the machine instruction level are recorded in an *exception table* for each method. The table matches the address ranges of the machine instructions that implement the try blocks with the starting addresses of their corresponding catch blocks.

The presence of exception handling code complicates parsing. To avoid mingling code from different blocks, it becomes necessary to mark the boundaries of the try/catch blocks in the sequence and prohibit reductions that cross these boundaries. It is also necessary to eventually combine try and catch blocks together to create source-level *try* statements. One approach to organizing the parser to handle exception parsing is to divide it into three phases. The first, makes a single pass and uses the exception table to mark the boundaries of the blocks. The second phase makes as many reductions as possible, with as many passes as needed, restricting each from crossing the block boundaries. When no further reductions are possible, the parser enters its third phase, where it makes a single pass looking for a single try/catch it can reduce. The parser alternates between the second and third phases until neither results in a reduction

4. Jive

Jive is an implementation of a multi-pass bottom-up parser for Java™ virtual machine instructions. It is written in Java and is capable of completely regenerating Java source code from any Java class file retrieved either from the local file system or from the Internet.

Jive functions as either an Applet or stand-alone application. It places a small window on the user's screen containing two buttons, one to decompile a file on the local system, and one to decompile a file specified by a URL on a remote system, Figure 3. Jive reads the class file, either from the local file system or from a network connection specified by a URL (Uniform Resource Locator) and parses its contents. It then creates another window, Figure 4, that displays the magic and version numbers, the original source file name, the name of the super class and in a scrollable text window, the source code.

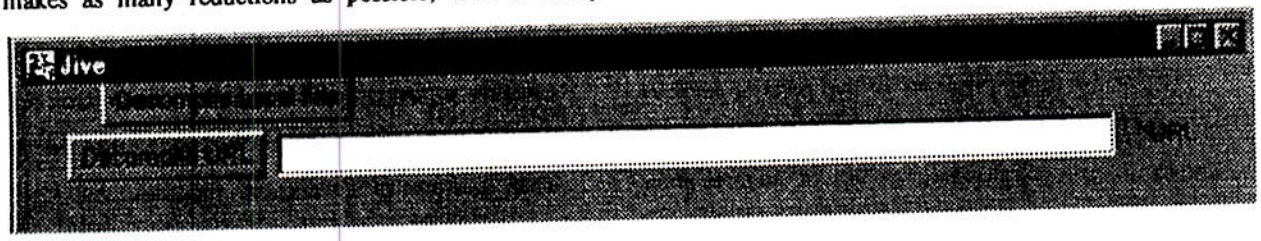

Figure 3 Jive Prompt Window

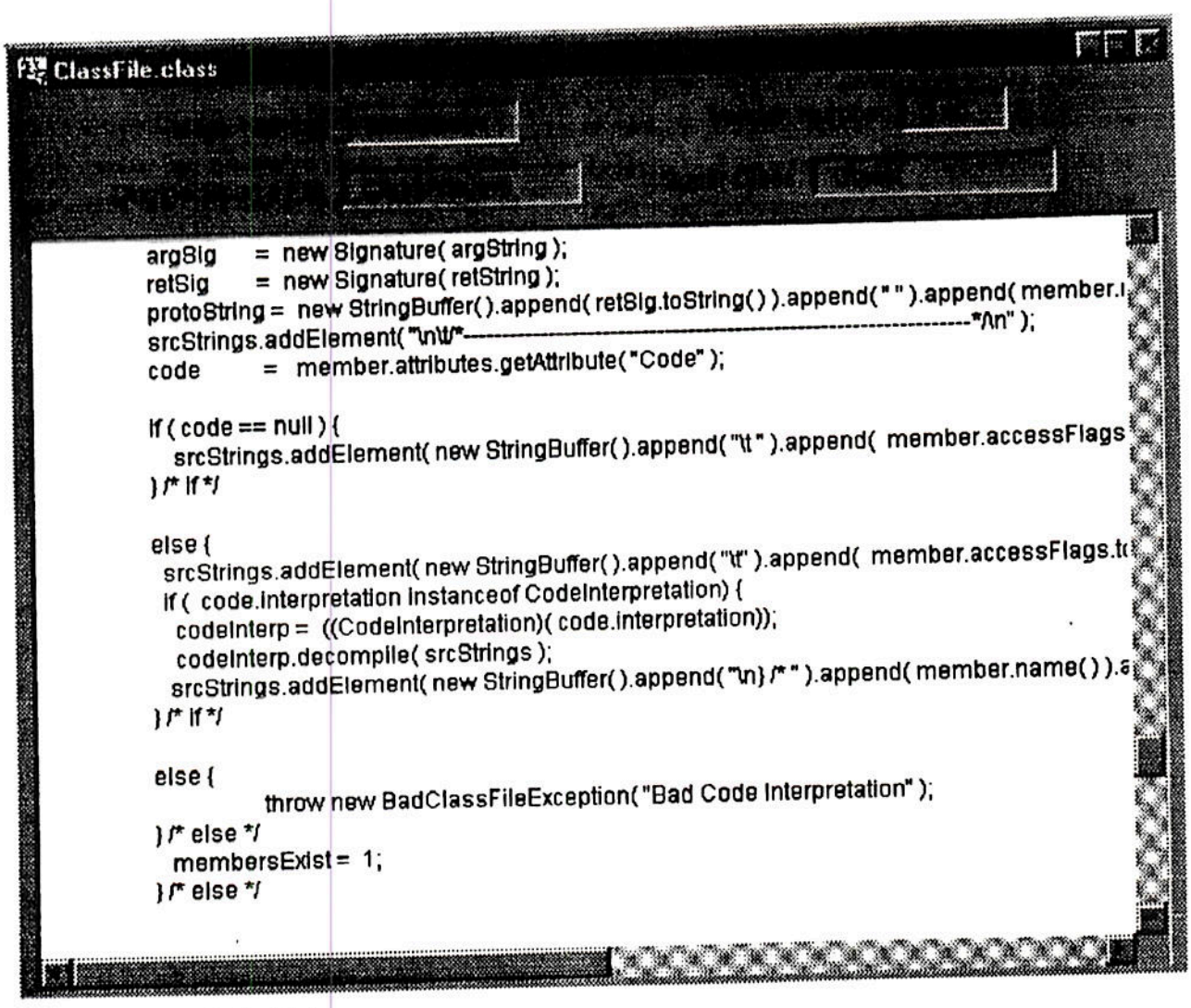

Figure 4 Jive Decompilation Window

The code illustrated in Figure 4 is the actual output produced by Jive for one of its own class files ("ClassFile.class"). The code has been manually indented to improve readability (the pretty printer for Jive is not yet implemented), but is otherwise unaltered.

4.1. Parser Implementation

The parser in Jive consists of two parts, a *parse engine*, and a class hierarchy representing non-terminals. The parse engine implements the logic of the three phases of the parser, while the class hierarchy records the grammar and semantic actions of the parser.

The class hierarchy is illustrated in Figure 5. Each class defined in the hierarchy represents a terminal or non-terminal. Instances of these classes are inserted into the sequence as they are recognized. The root of the hierarchy is "ExecComponent" or "executable component". It is the overall abstraction for terminals and non-terminals as they span the spectrum from machine instruction to source-level statement. Each ExecComponent contains logic, written in Java™, that recognizes when it is part of a reduction, and additional logic that actually performs the reduction by manipulating the sequence. Each ExecComponent also maintains the range of

original machine addresses that it spans, a count of the number of branches that target the first of this addresses range, and a flag that indicates if it is the start of a try block. These attributes, and links to the next and previous ExecComponent's in the sequence, are all inherited from the root of the class hierarchy and are maintained and retained through each redu ction.

With most of the logic stored in the class hierarchy, the job of the parse engine simplifies to one of scanning the sequence of ExecComponent's, asking each if it can be reduced, and for those that say "Yes", asking them to perform the reduction.

4.2. Discussion

The approach taken in implementing Jive of using the class hierarchy to store the grammar, semantic rules and actions proved to be flexible, robust. and very extendible. The later helped during development as it allowed the grammar to be developed and refined incrementally. A re-implementation of Jive, however, would benefit from having a "table drive" parser that allowed the grammar and semantic rules to be specified without writing Java source code.

While Jive is capable of decompiling almost all class files, it is not perfect. The source-level constructs that give it the most trouble are `break` and `continue` statements in multiply nested loops. In the right combination, these can produce very complex branching at the machine instruction level. Jive usually parses the branching correctly, even for elaborately construed examples, but if pressed, it can finish without completely re-assembling the source code. In these cases, which are rare, its output is still fairly complete and at a level that a human can easily stitch together. This limitation has more to do with Jive's current level of implementation than being a fundamental limit on Java bytecode decompilation.

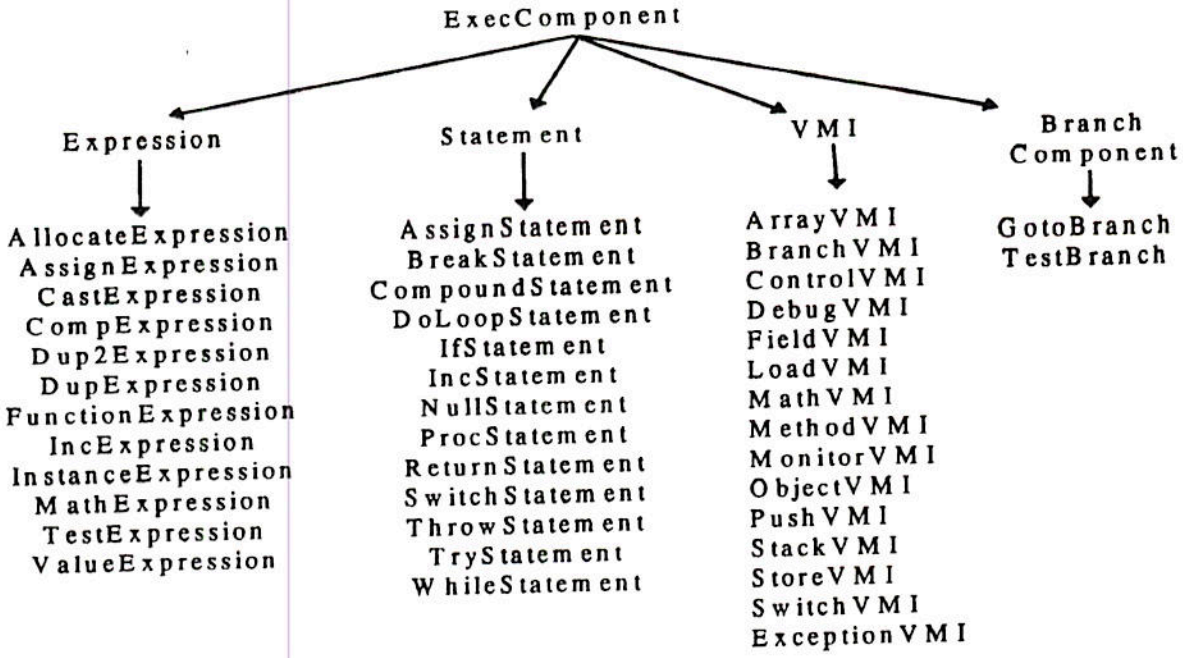

Figure 5 Class Hierarchy

5. Parsing Example.

We illustrate the parsing process for Java™ bytecode just described by showing successive passes of the parser for the bytecodes produced for the Java™ source code in Figure 6.

The output of the parser after disassembly and the first phase which marks the boundaries of the try and catch clauses is shown in Figure 7. Each line of the disassembly corresponds to one virtual machine instruction, and includes a parser assigned non-terminal label, the address of the instruction and the instruction's mnemonic.

For example, the first line is labeled <PushVMI> by the parser meaning that it is a virtual machine instruction that pushes a value on to the stack (PushVMI is a class derived from VMI). Its address is "0" and the mnemonic is `iconst_0` (push constant zero). The fifth instruction has an extra label `T` that indicates that it is the first instruction of a try block. The label on the instruction at address 41 is "`T 1->`", this indicates that it is both the first instruction of a try block, and that it is the target of one branch. If it was the target of two branches, the label would be "`T 2->`".

After the single initial phase one pass, the parser enters phase two and performs its first scan looking for reductions. In this case it is able to complete 36 reductions, the resulting mix of non-terminals and instructions is shown in Figure 8. After just this single pass, we already see the emergence of snippets of source code. For instance, the

```
int a = 0;
int b = 0;
try {
        if ( a > b+2 ) {
           a = foo( b ) > 0 ? 5 : 6;
        }
        else if ( a == 0 ) {
                throw
           new Exception( "problem 1");
        }
        else try {
                foo( 2 );
        } // try
        catch( Exception e2 ) {
            System.out.println("catch2");
        } // 2
} // try
catch( Exception e1 ) {
        System.out.println("catch1");
} // catch 1
```

Figure 6 Sample Java Source Code

first two non-terminals are completely recognized as source level assignment statements. They will be combined in the next pass to form a single compound statement. The logical expressions tested by the conditional branches have also all been recognized. Note, that they are the logical complement of the expressions in the original source. This is simply an artifact of the way in which compilers typically generate code for source-level conditional execution. The non-terminal <SingleTBranch> ("Single Test Branch") retains two versions of the expression, the original and its complement. The complement is displayed at this stage as it reflects the actual machine-level computations. When a source level construct is recognized such as an "if" statement, the original expression will be used.

```
<PushVMI>            [0]    iconst_0
<StoreVMI>           [1]    istore_1
<PushVMI>            [2]    iconst_0
<StoreVMI>           [3]    istore_2
<LoadVMI>     T      [4]    iload_1
<LoadVMI>            [5]    iload_2
<PushVMI>            [6]    iconst_2
<MathVMI>            [7]    iadd
<BranchVMI>          [8]    if_icmple 27
<LoadVMI>            [11]   aload_0
<LoadVMI>            [12]   iload_2
<MethodVMI>          [13] invokenonvirtual
                            int foo(int )
<BranchVMI>          [16]   ifle 23
<PushVMI>            [19]   iconst_5
<BranchVMI>          [20]   goto 25
<PushVMI>     1->    [23]   bipush 6
<StoreVMI>    1->    [25]   istore_1
<ControlVMI>         [26]   return
<LoadVMI>     1->    [27]   iload_1
<BranchVMI>          [28]   ifne 41
<allocateVMI>        [31]   new
                            java/lang/Exception
<StackVMI>           [34]   dup
<PushVMI>            [35]   ldc1 1
                            <String 38>
<MethodVMI>          [37]   invokenonvirtual
    boolean Exception(java/lang/String)
<ExceptionVMI>       [40]   athrow
<LoadVMI>     T 1->  [41]   aload_0
<PushVMI>            [42]   iconst_2
<MethodVMI>          [43]   invokenonvirtual
                            int foo(int )
<StackVMI>           [46]   pop
<ControlVMI>         [47]   return
<CatchStart>         [48]   catch
                            (Exception)
<FieldVMI>           [49]   getstatic
                            java/lang/System.out
<PushVMI>            [52]   ldc1 3
                            <String 42>
<MethodVMI>          [54]   invokevirtual
    boolean println(java/lang/String )
<ControlVMI>         [57]   return
<CatchStart>         [58]   catch (Exception)
<FieldVMI>           [59]   getstatic
                            java/lang/System.out
<PushVMI>            [62]   ldc1 2
                            <String 43>
<MethodVMI>          [64]   invokevirtual
    boolean println(java/lang/String )
<ControlVMI>         [67]   return
```

Figure 7 After Disassembly and Phase 1

A procedure call statement was recognized at address range 41-46. This is an example of the pop instruction

```
<AssignStatement>          [0-1]   a = 0;
<AssignStatement>          [2-3]   b = 0;
<SingleTBranch>     T      [4-10]Goto 27
                               if: a<=b+2
<SingleTBranch>            [11-18] Goto 23
                               if: foo(b)<=0
<ValueExpression>          [19]    5
<GotoBranch>               [20-22] Goto 25
<ValueExpression>   1->    [23-24] 6
<StoreVMI>          1->    [25]    istore_1
<ReturnStatement>          [26]    return;
<SingleTBranch>     1->    [27-30] Goto 41
                               if: a!=0
<ThrowStatement>           [31-40]
          throw new Exception("problem 1");
<ProcStatement>     T 1->  [41-46]foo( 2 );
<ReturnStatement>          [47]    return;
<CatchStart>               [48]
                           catch ( Exception )
<ProcStatement>            [49-56]
                  out.println("catch2");
<ReturnStatement>          [57]    return;
<CatchStart>               [58]
                           catch (Exception)
<ProcStatement>            [59-66]
                  out.println("catch1");
<ReturnStatement>          [67]    return;
```

Figure 8 After first pass, Phase 2

differentiating between a function call in an expression (address 11-13) and one that is a statement.

The pop instruction at address 46, removes the return value of the class member function "foo" from the stack;

```
<CompoundStatement> [0-3]   a = 0;
                            b = 0;
<SingleTBranch> T   [4-10] Goto 27 if: a<=b+2
<CompoundStatement> [11-26]
                    a=(foo(b)>0 ? 5 : 6);
                            return;
<IfStatement> 1->   [27-40] if (a==0){
            throw new Exception("problem 1");
            }
<CompoundStatement> T [41-47]       foo( 2 );
                                    return;
<CatchStart>        [48]    catch ( Exception )
<CompoundStatement> [49-57]
                    out.println( "catch2" );
                    return;
<CatchStart>        [58]    catch ( Exception )
<CompoundStatement> [59-67]
                    out.println( "catch1" );
                    return;
```

Figure 9 After Pass 2, Phase 2

this discards its return value and indicates that it is called as a procedure.

Having completed its first pass, the parser returns to the beginning of the sequence and begins its second. The output of the second pass, which includes 8 further reductions, is shown in Figure 9. Reductions include the grouping of several sequential statements into compound statements, the recognition of an *arithmetic if* in an assignment statement (addresses 11 to 26) and an *if* statement (27-40).

We see that after pass two most of the high-level blocks have been identified and the conditional branching that links them together is beginning to be recognized. At this point the parser has little to reduce until the try and catch exception blocks have been reduced. Consequently, the next pass, whose output is shown in Figure 10 has only 2 reductions.

After these reductions, the next pass is unable to perform any further reductions. The parser then enters its third phase where it looks for a single try/catch block reduction. The result of this pass is shown in Figure 11.

```
<CompoundStatement> [0-3]   a = 0;
                            b = 0;
<CompoundStatement> T 0->[4-40] if (a>b+2) {
                     a = ( foo( b ) > 0 ?  5 :  6 );
                     return;
                    }
                    if ( a == 0) {
                    throw new Exception( "problem 1");
                    }
<CompoundStatement> T  0->[41-47]  foo( 2 );
                           return;
<CatchStart>               [48]    catch ( Exception )
<CompoundStatement> [49-57]
                    out.println( "catch2" );
                    return;
<CatchStart>               [58] catch ( Exception )
<CompoundStatement> [59-67]
                    out.println( "catch1" );
                    return;
```

Figure 10 After Pass 3, Phase 2

```
<CompoundStatement> [0-3]   a = 0;
                            b = 0;
<CompoundStatement> T  0->[4-40]   if (a>b+2){
                    a =  ( foo( b ) > 0 ?  5 :  6 );
                    return;
                    }
                    if ( a == 0) {
                    throw new Exception( "problem 1");
                    }
<TryStatement>             [41-57] try {
                            foo( 2 );
                            return;
                           }
                    catch ( Exception lv1 ) {
                           out.println( "catch2" );
                           return;
                    }
<CatchStart>                       [58]
                           catch ( Exception )
<CompoundStatement> [59-67]
                           out.println( "catch1" );
                           return;
```

Figure 11 After try/catch reduction

Once it finds the reduction it re-enters the second phase and begins scanning for further reductions with the new statement. From this point on, the parser simply combines sequential compound statements and reduces the final try/catch block. The final result is shown in Figure 12. It compares quite favorably with the original source code. The only major difference is the missing *else* clause of the *if* statement. The same computational effect is obtained in the decompiled version by using a return statement in the first *if* statement. Both are correct and result in the same computation.

6. Countermeasures

There are several approaches to making decompilation harder for a parser like Jive and its users. One is to obfuscate the code in some manner, either by systematically modifying symbol names (e.g., "foo" becomes "sys0001") or by complicating the actual sequence of machine instructions defined for a method. Other approaches attempt to deny access to the class file contents altogether,

```
<CompoundStatement> [0-67]
    a = 0;
    b = 0;
    try {
      if ( a > b + 2) {
            a = ( foo( b ) > 0 ?  5 : 6 );
            return;
      }
      if ( a == 0) {
            throw new Exception("problem 1");
      }
      try {
            foo( 2 );
            return;
      }
      catch ( Exception lv1 ) {
            out.println( "catch2" );
            return;
      }
    }
    catch ( Exception lv1 )
            out.println( "catch1" );
            return;
    }
```

Figure 12 Final Decompiled Source Code

either by using encryption or a proprietary class file format.

Symbol name transformation certainly makes decompiled code harder to understand, particularly when compared to same code with the author's original symbol names. But, for a determined party, who knows the area and what they are looking for, the structure and nature of computations may still be quite accessible.

In general, adding extraneous branching and superfluous code blocks that are never executed could complicate code or confuse a decompiler. Doing so to Java™ bytecode however, maybe ineffective. In the Java™ virtual machine, all targets of all branches are statically determined, there are no dynamically computed branch addresses. It would be a simple matter to augment a decompiler with a flow analysis preprocessor that eliminated code that could not be reached and reorganized instructions to remove branches that perform no useful function.

Creating a secure delivery mechanism that hides the contents of the class files seems promising, but there is a wrinkle. Usually, encryption is used to stop a third party from obtaining information that is being transferred between two parties. To prevent decompilation, encryption would need to hide the information from the recipient as well. However, the recipient will still need a technique to decrypt the class files for their Java™ interpreter. It is not clear what would prevent a decompiler from masquerading as an interpreter and using the same technique.

A proprietary class file representation seems as it might prevent decompilation simply by being too difficult for a potential decompiler author to understand. The community of users that constitutes the web, however, places a premium on accessibility; it is unlikely that anything that reduces accessibility will be widely adopted.

7. Summary

In this paper, we showed that the features needed by a language like Java™ to facilitate the secure distribution of executable content in a distributed environment such as the World Wide Web, also make it easy to decompile that content and produce source level representations. The restricted stack architecture and high-level instruction set that make the code easy to analyze for malicious behavior also make it easy to parse bottom-up and reconstruct the original source. The need to dynamically link the code at runtime requires large amounts of symbolic information to be carried along with the machine instructions. These symbols give the regenerated source code the original names used by the author of the code.

We then described an implementation of a decompiler called Jive that is capable of generating source code from

arbitrary Java™ class files. Further enhancements for Jive include a flow analysis preprocessor that will eliminate unreachable code, a better pretty printer, and a table driven parser.

The main implication of the existence of Jive is that it predicts the coming of Java™ decompilers to the Web in the very near future. Its very existence shows that it is quite possible to construct a Java™ decompiler that does an effective job of generating source code from an arbitrary Java™ class file. Possessed by an undergraduate with a first course in compiler theory, the skill level required to construct a decompiler like Jive is low enough that it is likely only a matter of time before such tools are implemented by other people. Given the "hacker" cachet associated with such a tool, it is probable that a dedicated core of individuals will devote the time to craft and tune a decompiler that is every bit as capable as Jive, if not more so.

The simple implication of such tools becoming available on the Web is the complete exposure of the source code of any Java™ Applet publicly available on the Web. In one scenario, a decompiler Applet is placed on a Web server in country A. It is accessed by a user in country B and is used to decompile an Applet in country C. In this environment, source code will flow like water through the plumbing of the Web.

Easy access to source code may or may not be a problem for source code owners. Many applications are trivial and consequently their source code is of little real value. Computer programs are also notoriously hard to understand, even by the authors with their own commented versions of the source. There is, however, knowledge to be gained from source code. It contains algorithms, and ideas, and answers to questions that a knowledgeable and unwelcome reader may have no trouble extracting. Easier access to ideas is usually the fuel for the creation of new knowledge, so a "damn the torpedoes" attitude by Web Java™ developers may spark a flowering of innovation and creativity on the Web. Alternatively, we may see rampant paranoia that stunts development. One thing is certain: the Desktop and the Webtop are different.

8. References

[1] Sun Microsystems, "The Java Language Specification", Version 1.0, Beta, October 30, 1995.

[2] Sun Microsystems, "The Java Virtual Machine Specification", March 15, 1995.

[3] Sun Microsystems, "The Java Language Environment, A White Paper", May 1995.