Including Native Methods in Static Analysis of Java Programs

by

Victor L. Hernandez

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by .....
Chris Terman
Senior Lecturer
Massachusetts Institute of Technology
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Including Native Methods in Static Analysis of Java Programs

by

Victor L. Hernandez

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most static analyses of Java programs treat native methods conservatively due the analyses' dependence on bytecodes or source code for the syntactic and semantic analysis of the method bodies. This thesis presents a less-conservative approach that extracts sufficient information from native methods and transforms them into a representation that facilitates their inclusion in the analysis of Java programs. The approach focuses on analyses of Java programs that are concerned with the propagation of Java object references, such as escape analysis and mutability analysis. The Java Native Interface has become the standard for including C/C++ source in Java applications. With object reference propagation being the focus, the task of extracting information from native methods becomes that of separating the Java semantics associated with calls to the Java Native Interface (JNI) from the semantics of the remainder of the C/C++ code. The implementation of a conservative native method translator produces updated Java class files whose native methods contain non-executable Java bytecode streams which sufficiently characterize the native methods' invocation of JNI methods and whose control flow is determined by the non-JNI parts of the C/C++ source.

Thesis Supervisor: Chris Terman
Title: Senior Lecturer
Massachusetts Institute of Technology

Acknowledgments

I would like to thank Larry Koved for being a great supervisor during the completion of this thesis. I owe a lot to his support and his ability to track down the best resources. One such resource was Mark Chu-Carroll, who contributed large sections of the Native Method Translator's implementation.

I would also like to thank Steve Holton and Dave Bantz who supported me throughout my summers at IBM T.J. Watson Research Center and gave me great opportunities for professional growth.

I would also like to thank Dr. Chris Terman for his support during my five years at MIT and also for being instrumental in the completion of this document.

Finally, I cannot forget to thank my parents, Victor and Patricia. I appreciate their support and involvement in my education over the past 17 years. They have made great sacrifices to give me the educational and professional opportunities that I have enjoyed. For this I am truly grateful.

Para Mami, Papi, Patricia, Lucia, y Daniel

Contents

Chapter 1

Introduction

This thesis introduces a methodology for including native methods in the static analysis of Java Programs. The static analysis of Java Programs generally takes one of two approaches to dealing with native methods: native methods are ignored or treated conservatively. Neither is an adequate approach for systems that rely heavily on native source. This is because static analysis is used to improve the performance of the system or to determine that the system adheres to predefined characteristics.

1.1 Multilingual Programs

The development of modern computer systems requires the creation of programs whose parts have been written in more than one language. One reason for this is that the new system incorporates a block of legacy code, written in an out-dated language. Another reason is that some languages are better suited to do certain tasks, forcing developers to swap languages depending on the necessary functionality of the system's component.

This need for multiple programming languages is evident in the Java programming language [6]. Java is an interpreted, object-oriented language with a standardized library of classes that facilitates the development of programs. For Java developers to take advantage of any system that was developed before Java gained popularity, Java must provide a standard interface for interacting with other programming languages. Java developers also require the ability to interface other programming languages due to the interpreted nature of Java.

An interpreted language suffers in slower performance than compiled languages due to the extra level of indirection. Interpreted languages avoid providing a convenient interface for system calls because the system- dependent nature of system calls goes against the machine-independent goal of an interpreted language like Java. The use of a system-dependent compiled programming language for system-dependent tasks leads to more efficient system implementations.

All of these reasons make it evident that an interpreted language like Java needs an interface to use programs written in other languages. Java provides this functionality with its Java Native Interface (JNI) [10]. The Java Native Interface allows methods of Java classes and objects to be implemented as compiled procedures written in C/C++. JNI is specifically designed for Java programmers to include native functionality in their programs, but the design is general enough to provide the means for a Java runtime environment to be created from a C/C++ program. Specifically, JNI allows Java classes to declare certain methods as native, meaning that their implementation is not a bytecode stream, but instead it is a compiled method body. JNI provides native implementations of procedures that can be called from a native method, allowing it to emulate the functionality of Java methods. This functionality includes the ability to call Java methods, access Java fields, change Java fields, and create Java objects.

1.2 Static analysis of Java programs

The static analysis of programs provides the developer of a program with a means of understanding the behavior of his program. This analysis has traditionally been used in performance optimization in compilers and in systems that prove the correctness of an implementation. Program understanding is just as important in modern object-oriented languages like Java [17][12]. Object-oriented languages offer a new behavior to understand: the propagation of object references. An object in an object-oriented language can be passed around and stored within other objects. Since anyone with possession of a reference to an object may update that object's fields or invoke any of its methods, it is important to know who has access to an object at all times.

Knowledge of who has access to an object helps determine the dependencies between objects. Knowledge of these dependencies is exploited in a variety of ways by various object-oriented static program analyses. One such analysis is mutability analysis [16]. Mutability analysis makes a conservative estimate about what static fields can be mutated after instance initialization by the execution of any code on a given code base. The understanding of object interdependencies is crucial for this analysis because if one object depends on another (it contains a reference to that object), then a mutation of the second object results in a mutation of the dependent object. Mutability analysis labels a class as mutable if there exists a propagation or dependency path that exposes the values of that class' fields to mutation after instance initialization. This information is very important in designing an optimized shared class library for multiple Java Virtual Machines running simultaneously [15]. Specifically, Java programs running on a scalable Java Virtual Machine need to be given read-only access to classes that are determined immutable by mutability analysis, avoiding the need for class library management that maintains a coherent view of the class library amongst the parallel Java programs.

Another analysis that needs to know the propagation of object references in a program is escape analysis [5][1][18]. Escape analysis does the opposite computation to mutability analysis, in that it identifies object references that do not escape a certain scope. For example, escape analysis identifies object references that do not leave the method or the thread in which they are created. The results of escape analysis have been used to improve performance by identifying when objects can be allocated directly on the stack and when unshared objects do not have to be synchronized [7].

1.3 Including native methods in static analysis

The static analyses described in the previous section do not have a means of extending their program understanding across the multilingual boundary that exists in Java programs with native methods. Native methods are quite common in many Java systems. For example, a Java transaction server contains a lot of preexisting middleware written as compiled programs that perform optimally for server implementations targeted to a particular machine. The

analyses treat native methods as a black-box, making no assumptions about their behavior. This leads to unsatisfactory interpretations of the propagation of objects that occurs in a native method. One interpretation is to ignore native methods altogether, assuming that no interesting object reference propagation occurs in them. Another more conservative interpretation is that every possible object reference propagation that could happen inside the native method, does happen. The first interpretation relies on an unrealistic assumption, and the second interpretation is too conservative.

This thesis introduces a third way of handling native methods in static analysis of Java programs that more accurately reflects the real behavior of a native method. The goals of this approach are:

1. Loosen the conservative assumptions made about native methods' behavior.

2. Whenever a part of a native method's behavior cannot be accurately understood, make conservative estimates of the object propagation that occurs in that part.

3. The representation of a native method's behavior needs to accurately reflect only its object propagation behavior.

4. Simplify the incorporation of the native method analyzer into preexisting static Java program analysis implementations.

The rest of this thesis introduces a native method analyzer based on these goals that produces a representation of a native methods' Java object reference propagation that can be included in static analysis of Java programs. Chapter 2 discusses the Java bytecode representation that is used to represent a native method's behavior. Chapter 3 discusses how a preexisting Java class file can be modified to include the output of the native method analyzer. Chapter 4 gives a detailed description of how the semantics of the Java Native Interface and C/C++ can be represented as Java bytecodes. Chapter 5 discusses the implementation of the Java native method analyzer. Chapter 6 provides results of analyzing native methods and how they affect the results of static Java program analyses. Finally, Chapter 7 assesses our design and implementation decisions and discusses what more work can be done on native method analysis.

Chapter 2

Object Reference Propagation in Java

Native methods in Java are allowed to pass around object references in the same ways that non-native methods pass them around. That is one of the key features of the Java Native Interface (JNI). The JNI allows native methods to interact with Java objects just like non-native methods. This chapter focuses on exactly how object references can propagate inside native and non-native Java methods. Various possible representations of this object reference propagation are discussed and the choice of representation is explained.

2.1 Control flow in multilingual Java programs

Control of a program's execution is passed to a native method whenever it is invoked. The stack of the Java Virtual Machine [11] is used to pass values for all of the formal parameters of the native method, just as if it were a non-native Java method. There are three types of values that can be passed to a Java method:

1. *Implicit object or class references:* A method is always passed a reference to the object to which it belongs (for instance methods) or to the class to which it belongs (for class methods). The type of this reference can be determined by looking at the class file.

2. *Explicit object references:* These are all the arguments of the method that are instances of a Java class. The type of these references can be determined by reading the formal argument declaration of the method in its class file.

3. *Scalar values:* These are all non-object values used inside of the Java Virtual Machine. These include integers, floating point numbers, booleans, characters, etc. It is important to note that the value of these arguments do not influence the understanding of object reference propagation. Although the propagation of object references is used to understand when objects are mutated, the analysis of this behavior does not concern itself with the scalar values used in the object's mutation.

Once a native method encounters a return statement or the end of its body, execution control returns to the Java method that invoked the native method. A native method can return three types of values:

1. *Object reference:* This is a reference to a Java object that is type-compatible with the return types' class in the native method's declaration.
2. *Scalar values:* These include all scalar values supported by the Java Virtual Machine.
3. *No return value:* A method does not return a value if it is declared to have a void return type.

2.2 Entry and exit points of Java object references

An important part of understanding how object references can be propagated during the execution of a Java method is to understand how object references can enter the scope of the method and also how object references can be passed back to the Java runtime environment. A method's propagation of object references can be seen as the relationship between entering object references and the exit points through which they leave the method.

There are four ways that object references can enter the scope of a Java method:

1. The method recieves the object reference as an actual parameter.
2. The method retrieves the object reference from a Java class or object field.
3. The method has the object reference returned to it by a method it invokes.
4. The method creates a new instance of a Java class.

5. The method receives the object reference in the catch clause of an exception handler.

There are four ways that object references can exit a Java method:

1. The method returns the object reference as its return value.
2. The method assigns the object reference to a Java class or instance field.
3. The method passes the object reference to a Java method it invokes.
4. The method passes the object reference to the constructor of a Java class while creating a new Java object.

2.3 Representing the propagation of object references in a method

To accurately describe the propagation of object references inside of a Java native method, the chosen representation of the behavior must include the relationship between the entry and exit points of all object references. There are three representations that were considered: an object propagation intermediate representation; Java source; Java bytecode stream.

2.3.1 Object Propagation Intermediate Representation

The Object Propagation Intermediate Representation (OPIR) is a graph that represents the propagation paths between entry points of object references into a method and their exit points. The nodes of the graph are exit and entry points described in the previous section. The graph has directed edges indicating that the object reference that entered the method through a particular entry point was propagated back to the Java Runtime Environment through a particular exit point. Edges are also numbered to distinguish parameters to a method or constructor. The OPIR is able to represent the object reference propagation of all methods, not just native methods.

Using the Object Propagation Intermediate Representation has advantages and disadvantages. An advantage of this representation is that is that it is very simple and light-weight,

containing only the information necessary for analyzing a method's propagation of object references. A serious disadvantage of this representation is that a complicated interface to the OPIR is necessary for it to be used by static program analyses that don't represent all methods with OPIR.

2.3.2 Java Source Representation

Another way of representing the propagation of object references is to translate the native source of the method body to the equivalent Java source. Of course, the Java source would have to be a conservative estimate of the native sources' behavior, since native source is allowed to do more things (such as pointer arithmetic) than Java source. The propagation of object references is determined based on the use of the Java Native Interface inside of the method. As long as Java source is able to emulate the JNI's functionality, then Java source is a good representation. Unfortunately, JNI provides greater functionality than Java source. Specifically, native methods are allowed to ignore access modifiers of fields and methods. For example, a native method can directly modify another class' *private* static field. This functionality can in fact be represented using the syntax of the Java Programming Language. Unfortunately, Java compilers perform semantic verification of Java programs prior to generating Java class files. A Java source representation of a native method can contain semantically-unacceptable code that would prevent a Java compiler from generating the Java class file representation of the Java program. This is a problem for static analysis implementations that take Java class files (not Java source) as input.

2.3.3 Java Bytecode Representation

The final way to represent the propagation of object references in a native method is to translate the method's native source to a non-executable Java bytecode representation. There are many advantages to this representation. First, the Java Native Interface is modelled on the functionality of Java bytecodes, making it easy to translate calls to JNI methods. Second, bytecode streams can be added to Java class files in place of the empty body of a native method, making it easy to generate the necessary Java class files that are the input of most Java static analyzers. The main drawback of using Java bytecodes as a representation is

that all of native source needs to be translated to ensure logical control flow in the bytecode stream even if alot of the native source is not involved in propagating object references. For example, if a native method contains multiple returns, then listing the returns sequentially will result in a control flow where later method returns can be ignored. The bytecode representation cannot just contain a listing of all JNI method calls made by the native method. Operations involving scalar values are an example of parts of native source that may be unneccesarily translated.

2.3.4 Representation decision

Of the three possible representations of a native methods' propagation of object references, the Object Propagation Intermediate Representation is the best-suited representation. Unfortunately, the native method analyzer was implmented as part of an implementation of mutability analysis. The implementation of mutability analysis required bytecode streams as input. For this reason the third representation, Java bytecodes, was chosen for the native method analyzer. It should be noted that mutability analysis could be implemented efficiently by representing all Java methods, native and non-native, using OPIR.

Chapter 3

Updating Java class files with translated native methods

Having decided to use a Java bytecode stream to represent the propagation of object references in a native method, I will now discuss how the bytecode representation is generated and how that representation is then stored in preexisting Java class files.

3.1 Identifying native methods

The first step in analyzing native methods is to identify all of the native methods to be analyzed. The native method analyzer aims to understand the behavior of all native methods that can be invoked from a Java method. The standard way of transporting a collection of Java class files is the JAR file. The native method analyzer processes each of the Java class files separately. The native method analyzer goes through all of the methods of a class and keeps track of which methods have the *native* access modifier turned on. These are all of the classes' native methods that can be called from the Java Runtime Environment. These native methods have the exact same format as non-native methods except for two characteristics. A native method has no code attribute in which to store a bytecode stream. Also, the native method has an extra access modifier which identifies it as native. Therefore, to make the class file appear as having no native methods to a static analyzer requires adding a code attribute to the method and removing the native access modifier.

The updated class files are not intended to be executed in a Java Virtual Machine. They are intended for static analysis of the class files. The methods of the class files are non-verifiable because they could possibly emulate some behavior that is only allowed for native methods. An example of this is the new method calling a method whose access modifier prevents that from happening. Static analysis does not need its bytecode stream to be verifiable; it only needs the bytecode stream to be syntactically well-formed.

It is important to note that this native method analyzer only analyzes those native methods that can be reached from the Java Runtime Environment. It is possible for the Native Environment of a Java program to include entry points into native procedures that do not originate in from the Java program. These entry points are considered to be outside of the scope of the native method analyzer and the semantic analysis of the native source will take this issue into account.

3.2 Adding synthetic bytecodes to Java class files

Java class files are made up of three main parts: fields, methods, and a constant pool. The translation of a native method body into a bytecode stream will modify all three of these parts.

The methods will be modified in two ways:

1. Methods identified as being native will have a code attribute added to them and their native access modifier will be turned off. Adding a code attribute requires specifying a bytecode stream, the maximum depth of the method's stack usage, the maximum amount of local variables that the method uses, and the length of the bytecode stream.

2. New methods will be added to the class file to facilitate in the translation of native methods that call other native procedures.

The fields will be modified in the following way:

1. New fields will be added to help emulate the indeterminate propagation of an object reference out of the method. This feature is necessary to make the generated bytecode

stream a conservative representation of native methods whose propagation of object references cannot be accurately specified.

Finally, the constant pool will be modified to include all constants introduced by new bytecode streams, new methods, and new fields. The constant pool contains string constants representing literal values, names of methods and fields, and dynamic references to class definitions. The addition of dynamic references to the constant pool will comprise a large part of the work required in converting Java Native Interface procedure calls to their equivalent bytecode representations. This is because whenever a Java class or instance field or method is used from inside a native method, the JNI procedure that is called uses variables to hold the dynamic references to those fields and methods, whereas the bytecodes that perform the same function require the dynamic reference identification strings to be statically defined in the constant pool.

3.3 Producing new method bodies

The process for generating the bytecode stream for a native method reflects the traditional approach used by compilers to do code generation. The source language is C/C++ and the target Instruction Set Architecture is the Java Virtual Machine's Bytecode Instruction Set. The native source will first be parsed and then an Abstract Syntax Tree (AST) will be generated representing the body's statements and subexpressions. Then the AST will be traversed with each node being translated into the semantically-equivalent node in a Java Bytecode Intermediate Representation (JBIR). Given a class file, the JBIR makes the necessary updates to the constant pool and the list of fields, and it also updates the native method with the JBIR's generated bytecode stream.

Of course, the semantics of C/C++ cannot be fully emulated using the JVM's Bytecode Instruction Set. If that were the case, there would be no need for Java programs to use native methods. Therefore, the generated bytecode stream produces semantically-equivalent nodes in the JBIR when possible, and otherwise adds nodes to the JBIR that serve as a conservative emulation of the object propagation behavior of the native source. The next two chapters concentrate on the semantic equivalent of native source as Java bytecode streams.

Chapter 4

Translating Native Semantics into Java Bytecodes

The Java Native Interface (JNI) provides a set of native types and procedures that allow a native method to interact directly with the Java Runtime Environment. The translation of these methods to Java bytecodes is simplified due to the fact that the functionality of encompass the actions that a method could perform if it were implemented as a bytecode stream.

Propagation of object references can only be accomplished in a native method by using the Java Native Interface. Therefore, the key to successfully understanding the object reference propagation that occurs in a native method is the accurate determination of what each JNI procedure call is doing. The translation of all of the other source found in the native method only serves to provide a logical control flow in which to the object reference- propagating bytecodes are housed. This chapter introduces a native method translation strategy, dividing it between the translation of general C/C++ syntax and the translation of calls to procedures in the Java Native Interface.

4.1 Dealing with C/C++ semantics

C/C++ semantics are not as limited as the semantics of Java. Therefore, it is not possible to fully translate all of a native method's body into bytecodes. In spite of this, it is possible

to generate a satisfactory bytecode representation by aiming to accurately represent only the interaction of the native method with the Java Runtime Environment which is done through invocations of procedures in the Java Native Interface. The following two sections describe how C/C++ statements and expressions were translated by the JNI translator.

4.1.1 Statements

1. *Declaration statements:* There are two types of declarations in the native source that the JNI translator looks at: variable declarations and procedure declarations. Whenever a variable declaration is encountered, that variable is assigned a uinque Java variable and is both are added to the variable table. The JNI translator uses a variable table to keep track of the Java variables assigned to all native variables. The variable table also aids in the constant propagation required by JNI variables, which will be discussed later in this chapter.

   The JNI translator looks at the procedure declaration of all native methods and also all native procedures can be reached from a native method. The procedure declaration contains three pieces of information: the formal types of the arguments, the formal type of the return value, and the body of the procedure. The types of the arguments are used to initialize the variable table that will be used while translating the body. Also, the method's first bytecodes are generated to move the actual parameters from the stack into the formal arguments' unique Java variables.

2. *Lexical block statements:* The JNI translator needs to introduce a new lexical scope that inherits from the old scope whenever a lexical block is encountered. This requires the JNI translator's variable table be implemented as a stack of scopes. The structure of lexical blocks is a tree, but since the JNI translator only needs the variable information while translating the present lexical block, a stack is a sufficient data structure. The information of a variable is that associated with the first location of the variable starting at the top of the stack of scopes. Once the lexical block ends, the top scope on the variable table's stack is removed.

3. *Control-flow statements:* The two kinds of control-flow statements in C/C++ are

those that implicitly use destination labels and those that explicitly use destination labels. The implicit control-flow statements are *do*, *for*, *if*, *while*, and *switch*. The JNI translator generated bytecode streams for these statements just like a traditional compiler [14]. There are two tricky parts to generating this code. The first is that the *break* and *continue* statements have special meaning in each of these statements that translates to a *goto* bytecode with a target either before or after the statement. The intermediate representation of the *break* and *continue* statements needs to know the value of these two labels. The JNI translator provides these values by keeping a stack of before/after labels, with the top of the stack being the labels for the control-flow statement that is presently being translated. The second difficult issue is the translation of the *switch* statement, which translates to use of the *tableswitch* Java bytecode, which only allows byte-long values as indices into its table, while a *case* statement in the native method can contain any value. This can be solved by arbritarily transforming the *case* values into byte-long values. This approach works because it does not hide the propagation of any object references in the generated bytecode stream.

The other kind of control flow statement, *goto*, uses an explicit reference to a target destination, a label. The *goto* statement is translated to a *goto* Java bytecode whose target is the index in the bytecode stream associated with the target label. This target is calculated by all intermediate representations of a *goto* statement having their target indices calculated once the target label has been encountered.

4. *Expression statements:* The description for how the JNI translator handles expressions are found in the next section of this chapeter.

5. *Return statement* The JNI translator generates the bytecodes that load the value of return's subexpression onto the stack followed by the *return* or *areturn* bytecode depending on what type of value is being returned.

4.1.2 Expressions

The general approach to translating C/C++ expressions into Java bytecodes is to an intermediate representation that contains the type of the expression, the intermediate represen-

tation of the expression's subexpressions, and a unique target JVM variable for the value of the expression. The association of a target JVM variable for all expressions simplifies the flattening of the intermediate representation for bytecode generation, since subexpressions can be executed independently and then combined through their target variables. Such an approach is not appropriate if their were a limited number of variables, but the Java Virtual Machine allows 65,355 variables, making this a non-issue.

The JNI translator is able to successfully handle all C/C++ expressions that it is unable to translate to bytecodes. Any expression that cannot be translated has a special intermediate representation that identifies this property and also keeps track of any subexpressions with side-effects, specifically procedure calls, so that the bytecode stream will still reflect the invocation of these methods.

1. *Binary and Unary Expressions:* Most C/C++ binary and unary expressions are assignment or arithmetic operations that are available in the Java Virtual Machine as long as they are not being used to do pointer arithmetic. The type of these expressions' subexpressions can be analyzed to determine if they are referring to memory locations instead of values. The JNI translator properly translates all of these value-based binary expressions using the code generation approach of a compiler [14]. The binary operations are =, +=, -=, *=, /=, %=, ≫=, ≪=, &=, ^=, |=, ||, &&, |, ^, &, ==, !=, ≥, ≤, <, >, ≪, ≫, +, -, *, /, and %. The unary operations are ++, --, ~, !, -, and +. On the other hand, if these expressions are operating on references instead of values, the binary expressions are in fact involved in pointer arithmetic. These expressions cannot be translated because pointer arithmetic is not supported by the semantics of Java. Pointer arithmetic expressions are ignored, except that any JNI method invocations contained in their subexpressions are translated and correctly inserted in the resulting intermediate representation where the pointer arithmetic operation would have existed. Whenever a reference is dereferenced, conservative type analysis is used to determine the value of the expression. It should be noted that this same approach can be used for translating *all* binary and unary expressions. This is due to the fact that the exact scalar values propagating in the program are not necessary to conservatively understand the propagation of object propagation. The

only parts of binary or unary expressions that must be accurately translated are the invocations of JNI methods that they contain.

2. *Data Structure Expressions:* Expressions that access native data structures explicitly can be conservatively translated into bytecodes that access equivalent Java data structures. For example, arrays can be represented as Java arrays, and structs can be represented with a Java object. The operators of translatable data structure expressions are *[]*, $\rightarrow$, *&*, ***, and .. The operators, *sizeof*, *typedId*, *__vastart*, and *__vaarg*, are used in pointer arithmetic and are handled just like the pointer arithmetic expressions of the previous section.

3. *Cast Expressions:* Casting is the explicit conversion of a value reference from one type to another. Even though casting exists in the semantics of Java, the JNI translator does attempt to emulate the casting behavior of the native method body. This is because the only casts that the JNI translator should care about are those involving Java object references. This sort of casting, converting from one Java class to another, cannot be accomplished from inside a native method. Casts inside a native methods only involve native types. Therefore simple casts (T), static casts ($static\_cast < T >$), dynamic casts ($dynamic\_cast < T >$), reinterpret casts ($reinterpret\_cast < T >$), and constant casts ($const\_cast < T >$) of native types are treated like the non-translatable expressions discussed earlier.

   Whenever a JNI variable is being casted, this should indicate that the native method is trying to something that might break our assumptions that a native method interacts with the Java Runtime Environment only through Java Native Interface. This limited interaction does not require JNI variables to be casted, indicating that the casting of JNI variables needs to be flagged as behavior that the JNI translator does not support. The JNI translator reports whenever a JNI variable of type *jclass*, *jobject*, *jfieldId*, or *jmethodId* is casted.

4. *C++ Object-Oriented Expressions:* The C++ *delete*, *new*, and *this* expressions all require the JNI translator to be able to handle C++ classes. All of the expressions are treated as untranslatable, but any procedure calls contained in their subexpressions

are kept track of.

5. *Conditional Expressions:* These are expressions of the type A ? B : C. The JNI translator translates these expressions into an intermediate representation that executes A, stores it in an intermediate variable, tests if that variable is true, and then executed either B or C and stores their value in the target variable of the whole conditional expression.

6. *Function Call Expressions:* There are three types of function calls that a native method can make. The first type is a call to a procedure of the Java Native Interface. The next section of this chapter describes how JNI calls are handled.

   Another kind of function that can be called is a native procedure. These procedures are allowed to make JNI calls as well, requiring the JNI translator to handle them as well. The JNI translator keeps track of all native procedures that are called by a native method, with the goal of translating them into Java methods after the translation of the present native method is complete. This approach is more appealing than inlining the procedures because it avoids bloating of the method bodies. Unfortunately, the native method can pass to the native procedure JNI values that cannot be represented as arguments to Java procedures, namely class, field, and method references. These JNI values will be discussed later in the chapter. This means that the translation of the native procedures is context- sensitive, and all necessary constant propagation of JNI values needs to be represented statically. This means that every call to a native procedure results in a unique Java method, which is statically specialized to reflect the actual values of the JNI arguments to the native procedure. This requires the JNI translator to not only keep track of what native procedures are called, but also the values of the JNI variables at that calling context.

   The final kind of function is one where the function target cannot be resolved. An example of when this might occur is when a C++ virtual method is invoked. The determination of the target of a virtual method call requires sophisticated C++ type analysis and is outside of the scope of this native method analyzer. If the target of a virtual method call could be determined, that target could be translated exactly the

same way as a call to a non-virtual native procedure.

7. *Literal Expressions:* All native literal expressions are parsed directly, converted into the Java Virtual Machine representation of the literal string, and placed in the constant pool.

8. *Name Expressions:* Names either identify a variable or the target of a method call. Variable name expressions translate to the appropriate *load* or *store* bytecode that moves that variable's value between the Java stack and the variable's unique Java variable. Method name expressions are used to determine the target type of a procedure call (JNI procedure, native procedure, or virtual method) telling the JNI translator how to translate a procedure call.

9. *Throw Expressions:* A throw statement is used by a native method to throw a C++ exception. This statement translates into the athrow Java bytecode. The object that is being thrown is an instance of generic C++_Exception class that the JNI translator creates just for this situation. The generated bytecode stream will throw an exception in the same location as the original native method.

4.2 Dealing with JNI semantics

There are two parts to the Java Native Interface. The first part is the native types that represent Java value types and also static references to Java classes, methods, and fields. Native variables of a JNI type that represent a Java value that can be stored in the Java Virtual Machine are treated simply as variables generally are in traditional code generation. Each one of these variables will be assigned a unique variable number in the Java Virtual Machine. Native variables that represent static references are much harder to deal with because their equivalent representation in the Java Virtual Machine is a static string sitting in the constant pool. Therefore, the translation of these native variables requires the propagation of the string value associated with the static reference from its creation to its use in a JNI procedure call. The problem of translating these static references becomes a problem of constant propagation to aliases of the native value in the native source.

The other part of the JNI is all of the native procedures that define the interaction between the native method and Java Runtime Environment. The translation of these procedure calls consists of determining the equivalent series of bytecodes that exhibit the same semantics. It also consists of accurately passing the Java arguments to the procedure. This means pushing the Java variables that correspond to the values passed in the native procedure call and also referring to the correct static reference in the constant pool. The data structures developped to handle JNI variables will make this easy.

4.2.1 JNI Types

The Java Native Interface defines types for all value types that are legal inside of the Java Virtual Machine.

These types include all of the Java scalar types like *jint*, *jfloat*, etc. These JNI variables are very easy to deal with for two reasons. The first reason is that code generation of variables who have an immediate representation in the target machine's ISA can be translated to variable numbers (the JVM's equivalent of a register) just like in a traditional compiler. The Java Virtual Machine is a stack machine, meaning that operations are done on values at the top of the stack. This feature was introduced in the JNI translator's code generation by making abstracting all stack operations to be variable operations. This means that an operation explicitly pushes its input values onto the stack and explicitly pops the resulting value into a unique JVM variable as well. This abstraction, though inefficient, makes code generation for a stack based machine as easy at it is for a register machine.

There is one non-scalar JNI variable type that is legal in the Java Virtual Machine. That type is jobject, which represents a reference to a Java object. The way of passing an object reference to an operation is just like what was just described for scalar values. The reason that I separate the jobject JNI type from these other value types is that an object reference can also have its methods invoked and its fields accessed. In the Java Virtual Machine, the bytecodes that perform these operations require the object reference to be on the stack, just like the behavior of scalar operations. But the JVM also requires a static reference to a string constant in the class files' constant pool that identifies the class the object is an instance of. This use of the constant pool for object operations also is needed for class

operations. The JNI variable type jclass represents the static reference to the class identifier that resides in the constant pool. Likewise, method and field operations in the Java Virtual Machine require similar string constants in the constant pool that identify the method or field. Static method and field references are represented in JNI using the jmethodID type and the jfieldID type, respectively. All four of these JNI value types emulate behavior in the Java Virtual Machine that requires all of their associated identifier strings to be known at the time that the class file is created. This means that the JNI translator needs to know all of these identifier strings when it updates the class files in which a native method is found.

This requires the JNI translator to keep a variable table while it is translating the body of a native method. This variable table was needed for JNI scalar values in order to map JNI variables to their corresponding variable number in the Java Virtual Machine. The variable table did not need to keep track of any values associated with these variables. On the other hand, when a jobject, jclass, jmethodID, or jfieldID variable is added to the variable table, its associated identifier string needs to be stored with it. This approach allows the JNI translator to perform constant propagation from the creation of the JNI variable to its use. It is at the use (a JNI procedure call that uses an object, class, method, or field reference) that the equivalent JVM bytecode will need to have a reference to the a string in the constant pool. This also requires all variables of type *const char** to have their values propagated as well.

The JNI translator implements a systematic approach to adding string constants to the variable table. The variable table needs to be initialized with the type identifiers of all object references that are passed to the native method as arguments. The class of each object reference can be determined from the formal parameter declaration in the class file's native method declaration, and the name of the variable can be determined from the formal parameter declaration in the native implementation of the native method. Also, another variable that needs to be added to the variable table is the *this* variable whose associated string will be the name of the class the method is found in, no matter if the native method is a class or object method. Once the variable table has been initialized, the method body can be traversed. Object, class, method, and field references can be introduced inside of the method body by calls to procedures in the Java Native Interface.

1. *jclass:* Values of JNI type jclass are returned by the following JNI procedures:

   - `jclass DefineClass(JNIEnv *env, jobject loader, const jbyte *buf, jsize bufLen);`

     The JNI procedure DefineClass loads the class defined in the variable *buf*. This JNI procedure does not make the class' identifier string parseable, so it's use is considered outside of the scope of the JNI translator. Fortunately, DefineClass is rarely used.

   - `jclass FindClass(JNIEnv *env, const char *name);`

     The jclass' class identifier is the second argument of this procedure.

   - `jclass GetSuperclass(JNIEnv *env, jclass clazz);`

     To get the class identifier of this jclass requires looking up the class identifier of the variable *clazz* in the variable table. Then the class identifier is used to find a class file in the JAR file containing all of the programs class files. The class file will be able to tell us who the superclass is.

   - `jclass GetObjectClass(JNIEnv *env, jobject obj);`

     The class identifier associated with this jclass value is found by looking up the second argument in the variable table.

2. *jobject:* Values of JNI type jobject are returned by the following JNI procedures:

   - `jobject AllocObject(JNIEnv *env, jclass clazz);`
   - `jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, ...);`

     Both of these JNI procedures explicitly pass the class identifier to the procedure in the third argument *clazz*, making it immediately parseable.

3. *jfieldId:* Values of JNI type jfieldId are returned by the following JNI procedures:

   - `jfieldID GetFieldID(JNIEnv *env, jclass clazz, const char *name, const char *sig);`

This JNI procedure explicitly passes the class identifier to the procedure in the second argument *clazz*, making it immediately parseable. The field reference is made up of the value of the *name* and *sig* actual arguments.

4. *jmethodId:* Values of JNI type jfieldId are returned by the following JNI procedures:

   - `jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz, const char *name, const char *sig);`

     This JNI procedure explicitly passes the class identifier to the procedure in the second argument *clazz*, making it immediately parseable. The method reference is made up of the value of the *name* and *sig* actual arguments.

By keeping track of the string identifiers associated with jclass, jobject, jfieldId, and jmethodId JNI variables in the variable table, the string constants can be propagated to the JNI procedures where they are used.

4.2.2 JNI Methods

The Java Native Interface provides a native method with a set of procedures that let it interact with the Java Runtime Environment. The JNI methods that introduce JNI-specific variables into a native method were discussed in the previous section. This section discusses the rest of the JNI methods, specifically those that deal with Java objects, methods, fields, arrays, exceptions, and object synchronization.

1. *Java Method Invocation:* There are three ways to invoke a Java method from a native method:

   (a) `j_Type CallStatic__Type__Method(JNIEnv *env, jclass clazz, jmethodID methodID, ...);`

   Translates to the *invokestatic* bytecode.

   (b) `j_Type Call__Type__Method(JNIEnv *env, jobject obj, jmethodID methodID, ...);`

   Translates to the *invokevirtual* bytecode.

(c) `j_Type CallNonvirtual__Type__Method(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, ...);`

Translates to the *invokespecial* bytecode. This is used by a typical Java program to invoke the $< init >$ method of an object, which has the semantics that the target of the invocation is an object method, but it is not virtual. Those are the same semantics of the CallNonvirtual_Type_Method JNI method.

2. *Java Fields:* A native method can access either a static or instance field:

(a) `j_Type GetStatic__Type__Field(JNIEnv *env, jclass clazz, jfieldID fieldID);`

Translates to the *getstatic* bytecode.

(b) `void SetStatic__Type__Field(JNIEnv *env, jclass clazz, jfieldID fieldID, j_Type value);`

Translates to the *putstatic* bytecode.

(c) `j_Type Get__Type__Field(JNIEnv *env, jobject obj, jfieldID fieldID);`

Translates to the *getfield* bytecode.

(d) `void Set__Type__Field(JNIEnv *env, jobject obj, jfieldID fieldID, j_Type value);`

Translates to the *putfield* bytecode.

3. *Java Arrays:* A native method can create, access, and modify a Java array.

(a) `jarray NewObjectArray(JNIEnv *env, jsize length, jclass elementClass, jobject initialElement);`

Translates to the *anewarray* bytecode.

(b) `j_Type_Array New__Type__Array(JNIEnv *env, jsize length);`

Translates to the *newarray* bytecode.

(c) `jsize GetArrayLength(JNIEnv *env, jarray array);`

Translates to the *arraylength* bytecode.

(d) `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index);`

Translates to the *aaload* bytecode.

(e) `void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value);`

Translates to the *aastore* bytecode.

4. *Java Exceptions:* Native methods can throw Java exceptions:

(a) `jint Throw(JNIEnv *env, jthrowable obj);`

Translates to the *athrow* bytecode.

(b) `jint ThrowNew(JNIEnv *env, jclass clazz, const char *message);`

Translates to a *new* bytecode to create an instance of the exception, and then an *athrow* bytecode.

5. *Java Object Synchronization:* Native methods can perform synchronization of shared Java objects:

(a) `jint MonitorEnter(JNIEnv *env, jobject obj);`

Translates to the *monitorenter* bytecode.

(b) `jint MonitorExit(JNIEnv *env, jobject obj);`

Translates to the *monitorexit* bytecode.

6. *Java Object Boolean Operations:* Native methods can test the equality and types of Java objects:

(a) `jboolean IsInstanceOf(JNIEnv *env, jobject obj, jclass clazz);`

Translates to the *instanceof* bytecode.

(b) `jboolean IsSameObject(JNIEnv *env, jobject ref1, jobject ref2);`

Translates to the *is_acmpeq* bytecode.

7. *Global Java Object References:* A native method can inform the Java Runtime Environment that it is going to keep a reference to an object for a long period. This is required because the JRE assumes that it can garbage collect based on object references stored in the Java Runtime Environment. Garbage collection ignores object references that are stored in the native environment, meaning that an object can get garbage-collected if there is no reference to it any in the Java side of the runtime environment, even if a reference to it exists in the native part of the runtime environment. A native method informs the Java Runtime Environment that it has a reference to a Java object using these procedures:

   (a) `jobject NewGlobalRef(JNIEnv *env, jobject obj);`

   (b) `jobject NewLocalRef(JNIEnv *env, jobject ref);`

   (c) `jweak NewWeakGlobalRef(JNIEnv *env, jobject obj);`

   A native method relinquishes the reference to the Java object by calling the following JNI methods:

   (a) `void DeleteGlobalRef(JNIEnv *env, jobject globalRef);`

   (b) `void DeleteLocalRef(JNIEnv *env, jobject localRef);`

   (c) `void DeleteWeakGlobalRef(JNIEnv *env, jweak obj);`

   If a native method informs the Java Runtime Environment that it is going to store an object reference in the native runtime environment, the JNI translator concludes that the object reference is capable of propagating anywhere. The JNI translator represents this behavior by generating the *putstatic* bytecode that assigns this object reference to a newly-added static public field of the class in which the native method is found. Since all objects and classes have access to all static public fields, this representation describes the possibility of that object reference propagating anywhere in the Java Runtime Environment.

8. *Other JNI Methods:* All other Java Native Interface methods are not discussed because they require the JNI translator to handle native arrays, which it is unable to do.

4.3 Handling indeterminate JNI variables

In the previous section, all of the translations of the JNI procedure calls assumed that all of the necessary information about the JNI variables was available at translation time. This is not always the case. There are two things that can go wrong while trying to resolve the Java equivalent of JNI variables. The first thing that can go wrong is that the string constant associated with a JNI variable was not determined. This can occur in the following situations:

1. The string value was stored in global variable or a C++ field.
2. The string was introduced into the present native method as the return value of call to a native procedure.
3. The string has been stored indirectly, either in C data structure or being cast to a variable of a different type.

The second thing that can go wrong is that the expression identifying what JNI variable to pass to a JNI procedure cannot be resolved. This occurs when:

1. The expression is a global variable.
2. The expression is a C++ class or instance field.
3. The expression is a call to another native procedure.
4. The expression is a reference into C data structure.
5. The expression is a cast expression.

If the JNI variable that we have lost track of is a scalar variable, then it is not a problem since the native method translator does not need to provide scalar accuracy. The correct

translation for this is to pass as an argument a generic literal that is type-compatible with the formal type of either the Java method's argument or the Java field.

Whenever JNI identifier strings or object references cannot be accurately determined, the reaction of the JNI translator is to produce bytecodes that conservatively behave like the all logical values of the indeterminate values. For example, if an instance method is being invoked and we don't know which object it is being invoked on, then the JNI translator checks all variables of type *jobject* in the variable table that is an instance of a class that has a method identified by the invokeVirtual's *jmethodId*. The conservative representation of this behavior includes a method invocation of the methods of all *jobject* variables that contain that a method of the name and signature identified by the invokeVirtual's *jmethodId*.

It seems as if the conservative reaction to dealing with indeterminism of JNI variables leads to behavior understanding of native methods that is as weak as the understanding afforded by the black-box approach of present static analyzers. This is true, but only in situations where indeterminism occurs. Indeterminism of JNI variables is not a very common occurrence. For example, in the native source in a Java Virtual Machine, 92% of all *jmethodId* uses can have their associated strings directly parsed from the source files. The goal of the JNI translator is to maximize the percentage of object propagation actions in a native method that can be accurately represented. There is no guarantee that this percentage will reach 100%, and there is no doubt that native methods can be written in ways that make this native method analyzer not very useful. The native translator needs to provide at least as conservative behavior representation as the black-box approach in those cases where there is JNI variable indeterminism. The native translator uses the type declarations of Java methods and fields to provide a less conservative behavior representation. So although, there are C/C++ semantic constructs that are outside the scope of the JNI translator, the generated bytecode streams are much closer estimates of the native method's behavior than the blind estimates of the black-box native method approach.

Chapter 5

Implementation Overview

A conservative native method translator was implemented to modify the entries of native methods in Java class files with a synthetic bytecode stream that conservatively describes the propagation of object references in the native method.

5.1 Parsing native methods

The first step in the generation of the bytecode stream is the parsing of the native method's source. A C++ source analyzer called Montana [8][4] was used to parse the native source and also to generate an Abstract Syntax Tree (AST). The AST generated by Montana had a lot of great functionality such as the ability of searching for a procedure declaration based on the procedure's name and the automatically-generated type information of all expressions. A lot of the rules described in the previous chapter relied heavily on these features.

5.2 Java Bytecode Intermediate Representation

The second step in the generation of a bytecode representation of a native method is the generation of a Java Bytecode Intermediate Representation(JBIR) based on the AST representation of the native source. The JBIR contains an intermediate representation for all C++ statements and expressions and all JNI methods described in the previous chapter. The structure of the JBIR reflects the tree structure of the AST. Since the bytecode stream

is a linear (non-tree) structure, the JBIR was designed to allow the all nodes in the tree to generate linear bytecode streams. Also, all JBIR nodes were given default schemes for supplying the value of a subexpression that was unable to be translated by the JNI translator. The JBIR is similar to a compiler's low-level intermediate representation [14]. The bytecode generation and Java class modification functionality of the JBIR was implemented using IBM's CFParse [3].

5.3 Implementation difficulties

It proved very difficult to successfully integrate the Montana C++ parser (written in C++) with the Java Bytecode Intermediate Representation (written in Java). Initially the generator of JBIR was written in Java and interacted with the Montana AST navigator through the Java Native Interface. This approach failed because the libraries containing the Java native methods that interfaced with Montana were unable to be loaded by the Java Runtime Environment.

This problem was solved by separating these two modules' processes and having them communicate over the network. The modularization of the Montana C++ parser and the JBIR generator was implemented using the Manitoba distributed programming environment [2]. Manitoba consists of a server with a Montana back-end where the native source can be loaded, parsed, and navigated. It also consists of a client implemented in Java that communicates with the Montana server. The communication layer transfers data represented as streams of the Extensible Markup Language (XML). The parsing and construction of XML was done using IBM's XML4C and XML4J. The implementation required adding a Java native method translator that was driven from the Java client. The client used CFParse to access the JAR file containing all Java class files to be analyzed. The client requests the server to load the native source that contains the implementations of these Java classes' native methods. The client keeps track of all native methods that need to be translated (initially these are all Java methods identified by the their *native* flag). For every native method that the client requests to be translated by the server, the server returns the XML representation of that native method's bytecode representation and a list of all native procedures invoked

from the method. The XML representation of these invoked native procedures contains the calling context information that will be used to create the initial variable table for the native procedures' translation. These native procedures are added to the list of procedures to be translated. The largest part of the client implementation consists of the Java class that creates a method's JBIR based on the XML input stream. The server is modified to know how to traverse the Montana-generated Abstract Syntax Tree (AST). The AST traversal implements the translation rules of the previous section and the generation of the XML representation of C/C++ statements and expressions and JNI method calls. The XML representation is a tree structure just like the AST and the JBIR, making it easy to create an XML stream from the AST and the JBIR from the XML stream. The implementation of the JNI translation rules in the server and of the JBIR in the client were the largest pieces of code, each being a few thousand lines long.

Chapter 6

Results

This chapter shows examples of the native method translator translating a wide variety of class files with native methods. Each example consists of the original Java class declaration and the native source of its native method. Each example also includes the resulting Java class declaration (with bytecode streams in the bodies of all native methods) that is the result of native method translation. The impact of the class files' updated native methods on the accuracy of mutability analysis and escape analysis is also discussed.

6.1 Translation examples

6.1.1 Simple example

The first example is a native method that is precisely translated by the native method translator.

1. *Input: Class declaration*

```
class example {

    private BankAccount myAccount;

    public example() {
        myAccount = new BankAccount(0);
```

```
    }

    public native Object unknown();
}
```

2. *Input: Native method source*

```
JNIEXPORT jobject JNICALL
JAVA_example_unknown(JNIEnv* env, jobject this)
{
    const char* classString = (*env)->GetObjectClass(env, this);
    const char* fieldString = "myAccount";
    const char* sigString = "LBankAccount";
    jfieldId field = (*env)->GetFieldID(env, classString,
                                        fieldString, sigString);
    return (*env)->GetObjectField(env, this, field);
}
```

3. *Output: Modified class declaration*

```
class example {

    private BankAccount myAccount;

    public example() {
         myAccount = new BankAccount(0);
    }

    public Object unknown() {
        aload 0
        getfield example (LBankAccount)myAccount
        astore 1
```

```
        aload 1
        areturn
    }
}
```

The output of the native method translator is an updated class file. The method *unknown* is no longer *native*, and it contains a bytecode stream. The native method got the value in the *myAccount* field (the call to *GetObjectField*()) and then returns that value (the *return* statement). This behavior is precisely represented in the generated bytecode stream, as it gets the value in the *myAccount* field (the *getfield* bytecode) and then returns it (the *areturn* bytecode).

6.1.2 Translation of native procedure invocations

The second example is a native method that calls a native procedure twice, each time with different JNI-specific values being passed as actual parameters.

1. *Input: Class declaration*

```
class example {

    private int balance1;
    private int balance2;

    public example() {
        balance1 = 0;
        balance2 = 0;
    }

    public native void unknown();
}
```

2. *Input: Native method source*

```
JNIEXPORT jobject JNICALL
JAVA_example_unknown(JNIEnv* env, jobject this)
{
    const char* classString = (*env)->GetObjectClass(env, this);
    const char* field1 = "balance1";
    const char* field2 = "balance2";
    const char* sigString = "I";
    jfieldId field = (*env)->GetFieldID(env, classString,
                                        field1, sigString);
    nativeProc(env, this, field);
    field = (*env)->GetFieldID(env, classString,
                               field2, sigString);
    nativeProc(env, this, field);
    return;
}

void nativeProc(JNIEnv* env, jobject obj, jfieldId field)
{
    (*env)->SetIntField(env, obj, field, 15);
}
```

3. *Output: Modified class declaration*

```
class example {

    private int balance1;
    private int balance2;

    public example() {
        balance1 = 0;
        balance2 = 0;
```

```
    }

    public void unknown() {
        aload 0
        aload 0
        invokevirtual example (Lexample;)V nativeProc_1
        aload 0
        aload 0
        invokevirtual example (Lexample;)V nativeProc_2
    }

    private void nativeProc_1(example obj) {
        aload 0
        ldc 15
        istore 1
        iload 1
        putfield example (I)balance1
        return
    }

    private void nativeProc_2(example obj) {
        aload 0
        ldc 15
        istore 1
        iload 1
        putfield example (I)balance2
        return
    }
}
```

The modified class file contains two new *private* methods, one for each of the two calls to the *nativeProc* native procedure. The two new methods are exactly the same except for the name of the field passed as a reference to *putfield*, which is the difference between the two procedure invocations. Also, the *unknown* method is no longer *native* and it calls the procedures once each (equivalent to twice in the native version).

6.1.3 Translation of indeterminate JNI variables

The third example shows how the native method translator deals with an indeterminate JNI-specific variable. In this case, the indeterminate variable is a *fieldId*.

1. *Input: Class declaration*

```
class example {

    private int balance;
    private Object ref;
    private String str;

    public example() {
        balance = 0;
        ref = null;
        str = null;
    }

    public native void unknown(String newVal);
}
```

2. *Input: Native method source*

```
JNIEXPORT jobject JNICALL
JAVA_example_unknown(JNIEnv* env, jobject this, jobject newVal)
{
```

```
    const char* classString = (*env)->GetObjectClass(env, this);
    const char* field1 = "str";
    const char* sigString = "Ljava/lang/String";
    jfieldId[10] field;
    field[3] = (*env)->GetFieldID(env, classString,
                                  field, sigString);
    (*env)->SetIntField(env, obj, field[3], newVal);
}
```

3. *Output: Modified class declaration*

```
class example {

    private int balance;
    private Object ref;
    private String str;

    public example() {
        balance = 0;
        ref = null;
        str = null;
    }

    public void unknown(String newVal) {
        aload 0
        aload 1
        putfield example (Ljava/lang/Object)ref
        aload 0
        aload 1
        putfield example (Ljava/lang/String)str
        return
```

```
    }
}
```

The native implementation of the *unknown* method only updates one field, specifically the *str* field. The generated bytecode stream for the method *unknown* updates two separate fields. The reason for this is that the third argument to *SetInField* in the native method could not be resolved because the native method translator is unable to tell what are the contents of an array (or another intermediate data structure). The translator is forced to make a conservative guess as to what field is the one being updated. The translator knows that it must be a field of an instance of the class *example* and that the class of the field's new value is *Ljava/lang/String*. There are two fields in *example* that are type-compatible with that type, namely the fields *ref* and *string*. The translator generates bytecodes that assign the value to both of those fields, since either of those actions is possible. The conservative translation is accurate because the real action is contained in the possible actions represented by the bytecode stream.

6.1.4 Translation of Java object references stored in native environment

The fourth example shows how the native method translator deals with object references that the native method tells the Java Runtime Environment it will keep for a while.

1. *Input: Class declaration*

```
class example {

    private BankAccount myAccount;

    public example() {
        myAccount = new BankAccount(0);
    }
```

```
    public native void unknown();
}
```

2. *Input: Native method source*

```
JNIEXPORT jobject JNICALL
JAVA_example_unknown(JNIEnv* env, jobject this)
{
    const char* classString = (*env)->GetObjectClass(env, this);
    const char* fieldString = "myAccount";
    const char* sigString = "LBankAccount";
    jfieldId field = (*env)->GetFieldID(env, classString,
                                        fieldString, sigString);
    jobject anAccount = (*env)->GetObjectField(env, this, field);
    (*env)->NewGlobalRef(env, anAccount);
    return;
}
```

3. *Output: Modified class declaration*

```
class example {

    public static Object escapeReference;
    private BankAccount myAccount;

    public example() {
        myAccount = new BankAccount(0);
    }

    public void unknown() {
        aload 0
        getfield example (LBankAccount)myAccount
```

```
            astore 1
            aload 0
            aload 1
            putfield example (Ljava/lang/Object)escapedReference
            return
        }
    }
```

The native method informs the Java runtime environment that it is going to keep the object it got from the *myAccount* field for a while (a call to *NewGlobalRef*). This behavior is conservatively represented as meaning that the object reference can now propagate anywhere. The generated bytecode stream assigns the object reference to a new public field called *escapedReference*. This representation reflects the previously- described behavior because any method in the Java runtime environment can access a public static field.

6.2 Effect on mutability analysis

Mutability analysis can get more accurate results if it knows more about the behavior of native methods. Mutability analysis determines what fields can be mutated by any code using a given set of classes.

The simple example from earlier in this chapter shows the importance of knowing more about native methods. Without native method translation, the input to mutability analysis is the following class:

```
class example {

    private BankAccount myAccount;

    public example() {
        myAccount = new BankAccount(0);
    }
```

```
    public native Object unknown();
}
```

With only this much information, mutability analysis would conclude that the field *myAccount* cannot be modified since the only method that has access to it has no body to be analyzed. If mutability analysis used native method translation, its input would instead be:

```
class example {

    private BankAccount myAccount;

    public example() {
        myAccount = new BankAccount(0);
    }

    public Object unknown() {
        aload 0
        getfield example (LBankAccount)myAccount
        astore 1
        aload 1
        areturn
    }
}
```

The body of the modified *unknown* method now tells mutability analysis that any method that has a reference to an instance of the class *example* can get access to the object reference stored in *myAccount* by calling the method *unknown*. Such a method might in fact call a *BankAccount* mutator method, resulting in *myAccount*'s object being modified.

6.3 Effect on escape analysis

Escape analysis can also get more accurate results if it knows more information about its native methods. Escape analysis determines if an object reference escapes the scope it was

created in, for example a method body or a thread.

The following example shows how escape analysis' results improve by using the native method translator. Without native method translation, the input to escape analysis are the following class files:

```
class escapedTo{
    public static Object escapeSpot;

    public escapedTo() {}
}

class example {
    public run() {
        Bankaccount account1 = new BankAccount(0);
        Bankaccount account2 = new BankAccount(0);
        unknown(account1, account2);
    }

    public native void unknown(BankAccount account1, BankAccount account2);
}
```

Based on these class files, escape analysis would make one of two conclusions. The first conclusion is that both *account*1 and *account*2 escape the lifetime of an invocation of *run*() because they are both passed to *unknown*(), and since it is native, the escape analysis doesn't know what *unknown*() does with those object references, so it assumes that it lets them escape. The other conclusion, that both *account*1 and *account*2 do not escape, is based on the assumption that the native method *unknown* does not let them escape. These conclusions are both based on blind assumptions. Translating the native method allows escape analysis to base its conclusions on knowledge, not assumptions. These are the Java classes that escape analysis would take as input if it used native method translation:

```
class escapedTo{
```

```
    public static Object escapeSpot;

    public escapedTo() {}
}

class example {

    public run() {
        Bankaccount account1 = new BankAccount(0);
        Bankaccount account2 = new BankAccount(0);
        unknown(account1, account2);
    }

    public void unknown(BankAccount account1, BankAccount account2) {
        aload 2
        putstatic escapedTo (LBankAccount)escapeSpot
        return
    }
}
```

The bytecode stream of the *unknown* method says that only object reference passes as the second argument to *unknown*, that being *account*2, is allowed to escape. This extra knowledge allows escape analysis to accurately conclude that the object stored in the variable *account*1 does not escape.

Chapter 7

Conclusion

This thesis has described a conservative approach at representing the propagation of object references that occurs in a Java native method. The behavior is represented as a bytecode stream, which simplifies preexisting static analyzer's ability to use the results of this research. The specification of the native method translator implies that some native source is better designed for translation than others.

7.1 Effectiveness of Native Method Translator

The accuracy of the Native Method Translator depends on the source contained in the native method. A native method will have its object reference propagation accurately described by the Native Method Translator if it does not use any C/C++ statements or expressions or JNI methods that are not handled well by the Native Method Translator. This means that native source that never requires the Native Method Translator to make conservative estimates of the method's behavior will be accurately translated.

In the context of the native method translator, there is a concept of *well-behaved* implementations of a native method. A *well-behaved* native method has the following characteristics:

1. All use of Java objects and values is limited to calls to the Java Native Interface. This means that no pointer arithmetic can be performed on Java object references, Java values, and JNI-specific values.

2. All string arguments to JNI methods that create JNI-specific values (*jobject*, *jclass*, *jfieldId*, and *jmethodId*) are easily parseable. This means that actual argument is a string literal or it is a variable of type *const char** that has been assigned to a string literal.

3. No native procedures that return a JNI-specific value are ever called. The native method translator only does forward constant propagation of the string literals associated with the JNI-specific values.

4. JNI-specific values cannot be stored in data structures such as records or arrays.

5. JNI-specific values cannot be stored global variables or C++ object fields.

6. All procedures called by the native method can have their target accurately resolved.

Well-behaved native method implementations can be accurately represented by the bytecode stream generated by the native method translator. The native method translator described in this thesis has been run on a limited number of native methods.

7.2 Future work

The previous section describes characteristics of programs that are not handled ideally by the Native Method Translator. Future work on native method analysis should concentrate on increasing the number of programs that can be analyzed successfully. Suggestions for further work include:

1. Adding C++ type analysis will make it possible to resolve the target of all procedure calls.

2. Adding pointer alias analysis will allow JNI-specific values to be stored in intermediate data structures [13].

3. Adding fixed-point iteration to our native interprocedural analysis will let the analysis propagate JNI-specific values backwards from procedure calls.

4. Adding a preprocessing stage to the native method analysis that resolves the values of JNI-specific values that are stored in global variables and class fields.

5. Developing the methodology to analyze the effects of asynchronous native objects that execute in the native environment even when no native method is being invoked.

7.3 Evaluation of Java bytecodes as chosen representation

The Native Method Translator generates a bytecode representation of the native method's behavior. This choice of representation was motivated by the goal of making the results of this research easily incorporated into preexisting implementations of static Java program analysis, specifically the implementation of mutability analysis developed at IBM Research [16]. By deciding to generate bytecodes, the problem of describing the object reference propagation in a native method also became a problem in how to compile C/C++ source in Java bytecodes. This thesis has attempted to simplify the second problem by taking advantage of the reality that the native method translator aims to describe behavior, not fully emulate it. The generated bytecode stream is well-formed, not executable. There is no doubt that the code-generation issues dealt with during the development of the native method translator could have been avoided if a simpler representation had been chosen. The best choice is the Object Propagation Intermediate Representation. This representation works just as well for describing Java methods as it does for representing native methods. Future implementations of static analysis of Java programs that rely on the propagation of object references should base their analysis on the Object Propagation Intermediate Representation of the Java programs' methods.

Bibliography

[1] Bruno Blanchet. Escape analysis for object-oriented languages: Application to java. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver. Colorado, 1999.

[2] Mark Chu-Carroll. An architecture for decoupling interactive applications. Submitted to *ACM SIGSOFT Conference on Foundations of Software Engineering*, 2000.

[3] Matt Greenwood. Cfparse, an interactive class file editor. Information available at http://www.alphaworks.ibm.com/tech/cfparse.

[4] Michael Hind and Anthony Pioli. Traveling through dakota: Experiences with an object-oriented program analysis system. IBM Research Report 21674, IBM Research Division, 1999.

[5] M. Serrano V.C. Sreedhar J-D. Choi, M. Gupta and S. Midkiff. Escape analysis for java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[6] Bill Joy James Gosling and Guy Steele. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[7] Emin Gun Sirer Johnathan Alridch, Craig Chambers and Susan Eggers. Static analysis for eliminating unnecessary synchronization from java programs. In *Proc. Sixth International Static Analysis Symposium*, Venezia, Italy, 1999.

[8] Michael Karasick. The architecture of montana: an open and extensible programming environment with an incremental c++ compiler. In *Proc. ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering*, pages 131–142, 1998.

[9] Chris Laffra and Ilya Tilevich. C2j, automatic c++ to java translator. Information available at http://sol.pace.edu/ tilevich/c2j.html.

[10] Sheng Liang. *The Java Native Interface Specification.* Addison-Wesley, Reading, Massachusetts, 1999.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 1997.

[12] Stephen Fink David Grove Michael Hind Vivek Sarkar Mauricio J. Serrano V. C. Sreedhar Harini Srinivasan Michael G. Burke, Jong-Deok Choi and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proc. ACM SIGPLAN 1999 Java Grande Conference*, 1999.

[13] Paul Carini Michael Hind, Michael Burke and Jong-Deok Choi. Interprocedural pointer alias analysis. In *ACM Transactions on Programming Languages and Systems*, number 4, pages 848 – 894, 1999.

[14] Steven Muchnik. *Advanced Compiler Design and Implementation.* Morgan Kaufman, Reading, Massachusetts, 1997.

[15] D. Dillenberger D. Durand D. Emmes O. Godha S. Howard M. Oliver F. Samuel R. St John R. Bordawekar, C. Clark. Building a java virtual machine for server applications: The jvm on 390. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[16] L. Koved B. Mendelson S. Porat, M. Biberstein. Mutability analysis in java. Submitted to *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[17] Bilha Mendelson Sara Porat and Irina Shapira. Sharpening global static analysis to cope with java. In *Proc. of CASCON 1998*, pages 303 – 316, 1998.

[18] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver. Colorado, 1999.