Verifying mostly-static information flow control in Java Bytecode

by

Nicholas A. Mathewson

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

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

Certified by ....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

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

Verifying mostly-static information flow control in Java Bytecode

by

Nicholas A. Mathewson

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 Computer Science and Engineering

Abstract

When people use untrusted computer programs, they would often like to give secret information as input, but at the same time remain confident that unauthorized parties will not learn their secrets. Jif—a programming language developed by Andrew Myers for his PhD thesis [**?**]—enables them to have this confidence.

This describes the design and implementation of a component of Jif: a bytecode verifier that can check whether a piece of compiled Jif code is flow-safe. Users can be confident that the verifier will only approve programs that will not reveal their secrets without their permission. The verifier's architecture incorporates several novel features to help ensure its correctness.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I'd like to thank the academy, and the mad scientists who grew me from human DNA, without whom I would not be here today.

[[**Write a real set of acknowledgements here!**]]

Contents

Chapter 1

Introduction

As people store and transmit more of their information electronically, they face two conflicting goals: they want to allow computers to process their confidential information, but at the same time they want to keep their information private.

For example, consider an online tax-preparation service. The tax preparer's software needs to use financial records from the taxpayer and a proprietary database from the tax preparer, but the taxpayers do not want anybody to learn more than is necessary about their financial affairs, whereas tax preparers do not want anyone to learn the results of their proprietary research. In such a transaction. however, both parties are forced to give their valuable information as input to a computer program.

1.1 This thesis: a bytecode verifier for Jif

In his thesis, Andrew Myers [?] describes one solution to this problem: we can verify statically that a program will not leak data: first by labeling data with the privacy policies of its owners, and then tracking its use through the program to ensure that those policies are obeyed. Myers also describes a Java-based language, Jif, that implements this label model. This thesis concerns the implementation of a component of the Jif system: a bytecode verifier.

The Jif language itself has already been implemented as a Jif-to-Java translator and information-flow checker. Nevertheless, a translator is not enough. With only a translator, every party who wants to verify a program needs access to the original source. Because developers are often unwilling to reveal their proprietary source code, and because compilation is computationally expensive, Jif code is an impractical way to transmit programs. Furthermore, a translator is a large and complex piece of software; if the Jif translator has a security hole, it may be difficult to discover.

We can solve these problems by providing a bytecode verifier to verify the information flow safety of the Jif compiler's output. The verifier works as follows: it takes as input regular Java bytecode, annotated with Jif's information-flow labels. It then checks whether the bytecode obeys all of Jif's relabeling rules, and obeys all of its privacy policies. If so, it *approves* the bytecode as flow-safe; otherwise, it *rejects* it as unsafe.

Because it can verify flow properties in annotated bytecodes, a bytecode verifier does not need the extra resources or extra complexity of a full compiler, and so can be deployed with less overhead and greater confidence.

1.1.1 Design Goals

Although the design goals of the bytecode verifier are similar to those of any security-sensitive software project, they have several important consequences for the system's architecture. These goals are:

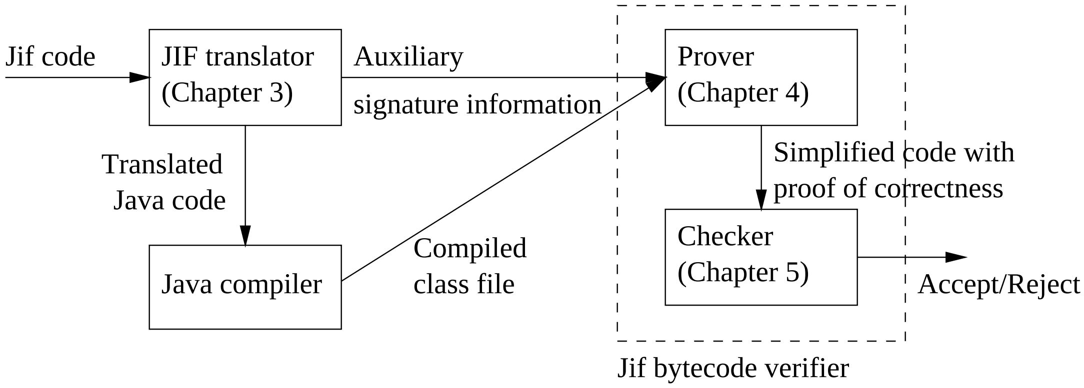
- **Safety:** The verifier must never approve *unsafe* bytecode that might violate a user's privacy policies. Otherwise, the trusted platform could run code that would leak information.
- **Liveness:** The verifier should approve all safe bytecode. Otherwise, the trusted platform would refuse to run code that was in fact harmless.
- **Confidence:** It should be as easy as possible to argue that the verifier will behave correctly.
- **Compatibility:** The verifier should be, as much as possible, a plug-in replacement for the Java verifier.
- **Compactness:** The verifier should not require excessive amounts of memory or processor time.
- **Flexibility:** The verifier should be as portable and adaptable as possible.

It is important to note that a failure in safety is far costlier than a failure in liveness: if the verifier has a bug affecting its safety, then users' secrets may be leaked, whereas if the verifier has a bug affecting its liveness, users will simply be unable to run their code until the verifier is repaired.

Therefore, it follows that whereas confidence in the verifier is important, confidence in the *critical* components of the verifier—those whose failure would jeopardize safety—is more critical than confidence in the system as a whole. If we are sure that the critical components work, then we can be sure that the system as a whole is safe.

In order to achieve confidence in a program, it helps to make it as simple as possible; a simple program is easier to reason about than a complex one. Because of the importance of safety, however, it is acceptable to add a little complexity to the verifier as a whole, if doing so will appreciably simplify the verifier's critical components.

Figure 1-1: Path of code through the Jif system



1.1.2 The Two-phase model

Our Jif bytecode verifier is composed of two major parts: the *Prover* and the *Checker*. When the system is about to run the program, it asks the Prover to demonstrate the program's safety. If the Prover can not provide such a demonstration, the verifier assumes that the program is unsafe. Otherwise, it passes the Prover's demonstration to the Checker, which ensures that the demonstration is correct. If so, the program is safe to run. (This approach owes much to previous work on Proof-Carrying Code [?].)

Why separate the Prover from the Checker? Because a Java bytecode verifier really performs two functions at once. First, it uses a dataflow algorithm to generate a "proof" of type safety by trying to find a type for every variable at every point of execution. By doing so, it implicitly ensures that the types it finds are consistent with one another, and with the operations that the code invokes on them.

In practice, running the dataflow algorithm is more complex than verifying its results afterwards. Therefore, the Jif bytecode verifier is split into two pieces; the Prover, which tries to find labels for the local variables and stack for every point of execution; and the Checker, which verifies that the discovered labels are safe, consistent with the code, and consistent with one another. Because the Checker only approves valid labelings, and because code has a valid labeling only if it is safe, we do not need to trust that the Prover is correct in order to be confident that our complete system is safe.

1.2 Integration with other parts of Jif

Jif will be deployed as a trusted execution platform on which parties with divergent security interests can run their code. The bytecode verifier described in this proposal will need to work as a part of this system, which is currently under development. Specifically, it will need to be invoked by the trusted JVM's ClassLoader, use the hierarchy information provided by the trusted base, and return whatever dependency information the trusted base may require.

1.3 Outline

The rest of this thesis is structured as follows. Chapter 2 provides an overview of the features of Jif, focusing on those that present a challenge for verification. Chapter 3 describes the code and annotations that the Jif translator must generate. Chapter 4 describes the Prover algorithm in detail. Chapter 5 describes the Checker. Finally, Chapter 6 examines the success of our two-phase approach to verification, discusses some related work, and suggests some directions for future research.

Chapter 2

Jif and the Decentralized Label Model

Andrew Myers in his work [?] has developed a label model to describe the privacy interests of diverse principals, and a language, Jif, to implement this model. Because our verifier is based on Myers' work, we will discuss the security model and the language as background material.

2.1 The Decentralized Label Model

In the Decentralized Label Model, every data item has a *label* determining its secrecy properties. Each label has set of *policies*, each of which has an *owner* and a set of *readers*. An owner corresponds to any party whose information has been used to construct that data item; a reader corresponds to a principal authorized by the owner to learn the datum. Only if every policy allows a given reader is the reader actually granted access.

For example, if Alice trusts Bob with a piece of information x, she may label it as {Alice: Alice, Bob}, signifying that, according to Alice, only Alice and Bob may receive the value of x. If Bob trusts only himself with the value of y, he would label it as {Bob: Bob}. If z is then set to some value depending on both x and y, it receives the *join* of the two labels: {Alice: Alice, Bob; Bob: Bob}. Since Bob is a reader in both policies, he is allowed to access z, but since Alice is not allowed to read in the policy owned by Bob, she can not access z.

The model also includes the notion that principals may relate to one another in a *principal hierarchy*. At a hospital, for example, every doctor may *act for* the principal Doctors; if Alice labels her contact information as {Alice: Doctors}, then any doctor may read it.

2.2 Jif: mostly-static information flow control

The Jif language implements the decentralized label model described above. Labels Jif programs are checked statically, at compile time. For every calculation listed in a program's source code, Jif determines the label for the result of that calculation, and ensures that the result is never sent to an unauthorized reader. If the code attempts to leak information, the Jif translator will refuse to compile it at all.

Jif is *mostly-static* because under some circumstances, the label of a piece of data can not be known until runtime; in those cases, Jif checks the relevant flows dynamically. For example, if a piece of data is received over a secure network channel, or read from a database, there is no good way for a programmer to be sure of its label in advance. To solve this problem, Jif provides *runtime labels*: if the variable `label x;` contains a label, then another variable `Object y;` may be declared to have the label stored in `x` by writing `Object y{*x};`. The actual value of a runtime label may be examined with Jif's `switch label` statement.

Unlike other projects implementing static or mostly-static information flow control, Jif is based on Java, a popular, object-oriented programming language with rich exception semantics [?]. Past work had focused on functional languages, or languages with other easy-to-analyze properties. While these languages simplify the work of their designers, they lack features that many programmers desire. The Jif language, however, incorporates almost all of Java's semantics.

Additionally, Jif allows *safe declassification*. A principal may sign a piece of trusted code to run under its *authority*, thereby allowing the code to modify policies owned by that principal. In the tax-preparer example, the taxpayer's financial records would be labeled {Taxpayer: Taxpayer}, and the preparer's database would be labeled {Preparer: Preparer}. The prepared tax return form—a combination of the two parties' data—would then be labeled {Taxpayer: Taxpayer; Preparer: Preparer}. Ordinarily, no reader would be allowed to learn this information, because no reader appears in both of the label's policies. Because of this problem, Jif allows the preparer to sign a piece of trusted code, which is then authorized to *declassify* the tax return, relabeling the return as {Taxpayer: Taxpayer} by removing the preparer's policy. Jif checks that `declassify` operations will only modify policies owned by a principle under whose authority the code is running.

Finally, Jif requires a *trusted execution platform* (TEP). This requirement arises because, even though all parties may agree that a particular piece of code leaks no information, they can never be sure that an arbitrary remote machine executes that code correctly. For example, if the tax preparation program were running on machines under the preparer's control, the taxpayers could never be certain that their information was secure. Because of this concern, Jif specifies that programs must run on a *trusted execution platform*—a system trusted by all relevant parties.

Figure 2-1: Example of an implicit flow

```
int {public} phone;
int {secret} income;
...
// pc = {}
if (income > 100000) {
   // pc = {secret}
   phone = 5551212; // This assignment fails, since it would leak information about income
}
```

2.2.1 The program counter

?? In order to trap *implicit flows* – flows where information is carried by the execution of a statement or expression – Jif keeps track of a label for the program counter, or pc.

As shown in figure 2-1, it is possible to use control statements to try to leak secret information. To solve this problem, Jif maintains a pc label for every point of execution. The pc label is defined to be at least as restrictive as all the information that could be learned by discovering that the program had reached a given point of execution.

In Jif, every method is labeled with a pc label for its entry and each of its exits, so that the translator can ensure that a method invocation never leaks information.

Myers gives a general rule for establishing the pc label for a basic block [?]:

> At any point during execution, various vaules v_i have been observed in order to decide to arrive at the current basic block; therefore, the labels of these values affect the current pc:
>
> $$\mathsf{pc} = \sqcup_i \{v_i\} = \{v_1\} \sqcup \{v_2\} \sqcup ...$$
>
> ...This label $\sqcup_i\{v_i\}$ can be determined through straightforward analysis of the program's basic block diagram. The decision about which exit point to follow from a basic block B_i depends on the observation of some value v_i. The label pc for a particular basic block B is the join of some of the labels $\{v_i\}$. A label $\{v_i\}$ is included in the join if it is possible to reach B from B_i, and it is also possible to reach the final node from B_i without passing through B. If all paths from B_i to the final node pass though B, then arriving a B conveys no information about v_i. (pp. XXXX-XXXX)

In section 4.3.3, we discuss a further modification to this rule.

2.2.2 Other features

In addition to the features described above, Jif provides many other features to allow programmers to write flow-safe programs more easily. Some of these features are summarized in figure 2-2.

Figure 2-2: Summary of Jif's features and syntax

Feature	Syntax
Labeled types	`int {alice: alice} secret;`
Using runtime labels	`int{*L} add(Label L, int {*L}a, int{*L}b)` `{ return a + b; }`

Inspecting runtime labels	`int i{*L};`
	`switch label(i) {`
	`case (int {alice:alice}) aliceChannel.println(i);`
	`else throw new AccessException(); }`

Safe declassification	`int{} declassifyTotal() where authority(alice) {`
	`int{alice} total = calculateTotal();`
	`return declassify(total, {}); }`

Checking actsFor	`String{bob: bob; doctors: doctors} patientName = "Bob";`
	`String{bob: bob; alice: alice, doctors} alicePatient;`
	`actsFor(alice, doctors) {`
	`alicePatient = patientName;`
	`}`

Parameterized types	`class Vector[Label L] { ... }` `v = new Vector[{secret}]();`

Method constraints	`void sendAliceInformation() where authority(alice);`
	`void sendInformation(principal P) where caller(P);`
	`void sendMoreInformation() where actsFor(alice,bob);`

Considering the features in figure 2-2 in turn: We have already discussed labels, runtime labels, and declassification. The actsFor statement is used to inspect the principal hierarchy dynamically. Jif's parameterized types allow classes to be written as generic with respect to generic labels and principals. Finally, Jif includes method constraints to indicate that a method must run under a principal's authority; that a method must be called by a method with a given principal's authority; or that a method must be called from an environment where one principal is known to act for another.

Chapter 3

Translating from Jif to Java

In the original Jif documentation, Myers provides rules to translate Jif to Java [?]. This translation, however, discards some information that the verifier needs in order to complete its analysis. Here we describe an alternative set of rules to enable verification

Additionally, the original Jif work states that some auxiliary signature information needs to pass from the translator to the verifier. Later in this chapter, we specify this information in more detail.

3.1 Translating Jif to Java

The original translation rules for Jif [?] discard all label information unnecessary for running Jif programs. Because we are running a verifier on the code, however, we need to retain some information that would otherwise be discarded.

For example, consider the declassify expression. In the original rules, declassify(expr, label) is translated to (expr). This translation is inadequate for input to the bytecode verifier, however, since it would prevent us from determining where declassifications are to occur. To solve this problem, we introduce a call to a *dummy function* — a method that does nothing, but whose presence tells the checker that a declassification is to occur. Under this scheme, declassify(expr, label) translates to jif.vmimpl.Declassify.declassify(expr, label). (See figure 3-1.) Later, when the verifier sees a call to the static method jif.vmimpl.Declassify.declassify, it can deduce that a declassify operation should occur.

We use a similar trick to translate Jif's added statements. For example, Jif includes a declassify statement, which executes a piece of code with a relaxed pc. After label analysis, the original translator would have translated declassify(label) Statement; as Statement;. To indicate the declassify statement's presence to the verifier, however, we translate it as if (jif.vmimpl.Declassify.declassifyS()) Statement. When the verifier sees call to declassifyS (which always returns true), it can check to see if the result is used in any if conditionals, and declassify the pc within those conditionals.

Figure 3-1: Translating from Jif to Java

Expressions:

Jif		Java	
declassify(expr, label)	$\Longrightarrow$	jif.vmimpl.Declassify.declassify(expr, label)	(3.1)
new ParamClass[A,B](arg1, arg2)	$\Longrightarrow$	new ParamClass(A, B, arg1, arg2)	(3.2)
PClass[A,B].staticMethod(arg1, arg2)	$\Longrightarrow$	PClass.staticMethod(A, B, arg1, arg2)	(3.3)
new T{label1}[n]	$\Longrightarrow$	jif.vmimp.Labels.noteArray(label,new T[n])	(3.4)

Statements:

Jif		Java	

declassify(label)
  Statement
	$\Longrightarrow$	if (jif.vmimpl.Declassify.declassifyS())
		  Statement
			(3.5)

actsFor(P1, P2)
  Statement1
else
  Statement2
	$\Longrightarrow$	if (Principal.actsFor(P1,P2))
		  Statement1
		else
		  Statement2
			(3.6)

switch label(expr) {
case(Type1{Label1} id)
  Statement1
else Statement2 }
	$\Longrightarrow$	if (*label-of-expr*.relabelsTo(Label1)) {
		  id = expr; Statement1
		} else Statement2
			(3.7)

Catch blocks:

Jif		Java	

try { ... }
catch (Type{Label} id) {
  Statements
}
	$\Longrightarrow$	try { ... }
		catch (Type id) {
		  jif.vmimpl.Labels.noteFormal(id,Label);
		  Statements
		}
			(3.8)

Labels:

Jif		Java	
new label{o1: r1, r2}	$\Longrightarrow$	new Label(o1, r1, r2)	(3.9)
new label{o1: r1; o2: r2}	$\Longrightarrow$	new Label(o1, r1).join(o2, r2)	(3.10)
label v = *expr*; new label{o1: r1, *v}	$\Longrightarrow$	Label v = *expr*; new Label(o1,r2).join(v)	(3.11)

Labels are easy to translate: we create Label objects explicitly, and pass them around as needed. For runtime labels, we join in the component, and rely on the verifier to find the dependency.

We handle other Jif language features similarly. More complete information is given in figure 3-1.

3.2 Signature information

Besides the translations we describe above, the Jif verifier requires information about class, field, and method signatures. (Figure 3-2 describes this information in detail.) For example, when the verifier encounters a call to a method, it needs to know the labels on the method's arguments in order to determine whether the method will leak information. Since these labels are lost in translation, the

Figure 3-2: Structure of auxiliary information

For each class:
- A list of that class's parameters, if any.
- The principals in the class's authority declaration, if any.
- The actual types of the class's superclass and interfaces.

For each field:
- The field's label, and actual type.

For each method:
- The labels and types for the method's arguments and return value.
- The begin and end labels.
- A label for each of the method's exceptions.
- Optionally, a list of authority, caller, and actsFor constraints.

Figure 3-3: The Protected class in Jif

```
class Protected {
  Object{*lb} content;
  final label{this} lb;
  public Protected{LL}(Object{*LL} x, label LL)
    { lb = LL; super(); content = x; }
  public Object get(label L):{L} throws (IllegalAccessError) {
    switch label(content) {
      case (Object{*L} unwrapped)
        return unwrapped;
      else throw new IllegalAccess();
    }
  }
  public label get_label()
    { return lb; }
}
```

translator must pass them on to the verifier as auxiliary information.

Furthermore, because Jif's parameterized types lose their parameters during translation, we need to include full information about *type* signatures with every class.

3.3 An example

Figures 3-3 through 3-5 show the translation of the Protected class from [?]. The original class in figure 3-3 class manages runtime labels by carrying an Object and its label together. When a client wishes to extract the data, it calls the get method to attempt to extract the object with a given new label. If the relabeling is safe, the object is returned. Otherwise, an IllegalAccess exception is thrown.

Figure 3-4 gives the translation of Protected into Java. Most obviously, the labels on the types are gone. Also, note that the switch label statement has been translated to a relabelsTo call as described

Figure 3-4: The Protected class, translated to Java

```
class Protected {
  Object content;
  final Label lb;
  public Protected (Object x, Label LL)
    { lb = LL; super(); content = x; }
  public Object get(label L) throws (IllegalAccessError) {
    if (lb.relabelsTo(L)) {
      unwrapped = content;
      return unwrapped;
    } else throw new IllegalAccess();
  }
  public label get_label()
    { return lb; }
}
```

Figure 3-5: Signature information for Protected

Parameters	(None)
Authority	(None)
Supertypes	java.lang.Object
Fields	Object{*lb} content; label{this} lb
Methods	public Protected{LL}(Object{*LL} x, label LL) public Object get(label L):{L} throws (IllegalAccessError) public label get_label()

above.

Finally, figure 3-5 gives the signature information which the translator must generate for Protected.

Chapter 4

Verifying JifVM code: The Prover

The Prover is the most essential portion of the Jif bytecode verifier, and the most complex. It must distinguish valid code from invalid code, and — when it encounters valid code — construct an appropriate proof for the checker.

In this chapter, we discuss the Prover's algorithms. We describe them partially in terms of a notional Jif Virtual Machine (or JifVM) that differs from the ordinary JVM in that it incorporates features from Jif.

This document does not give all of the conditions necessary for a class's safety; instead, it omits the conditions that are identical to ones in Jif, and present no challenge to verification. For example, Jif's restrictions on the labels of methods in subclasses are unchanged, and therefore unremarkable.

4.1 Overview of the Java Virtual Machine

Because at its core the JifVM is derived from the JVM, we will begin with a description of the Java Virtual Machine itself.

The JVM is, fundamentally, a stack machine with local variables. To a first approximation, each stack or variable slot holds either an integer, a floating-point number, or a reference to a Java object.

Every JVM method consists of a list of instructions. These instructions manipulate values on the stack, access fields, call methods, and perform all the work needed to implement Java code.

To implement `catch` statements, a JVM method may optionally provide list of exception handlers. A handler indicates that a certain block of code is to be run whenever an instruction of a given type occurs within a certain range of code.

In figure 4-1, we show a possible compilation of a simple Java method that computes the sum of an array of numbers, and then prints the sum to `System.out`. Note that local variables are indexed by number, and that method invocation is handled as a single instruction. On the rightmost column, we describe the effect of each instruction.

Figure 4-1: Compiling a simple Java method

```
static void printSum(int[] els) {
  int n = els.length;
  int sum = 0;
  for(int i=0;i¡n;++i) {
    sum += els[i];
  }
  System.out.println(sum);
}
```

$\Longrightarrow$

	Instruction	**Stack**/*Effect*
	aload_0	els
	arraylength	els.length
	istore_1	*n = els.length*
	iconst_0	0
	istore_2	*sum = 0*
	iconst_0	0
	istore_3	*i = 0*
	goto 19	
10:	iload_2	sum
	aload_0	els sum
	iload_3	i els sum
	iaload	els[i] sum
	iadd	(els[i]+sum)
	istore_2	*sum = els[i]+sum*
	iinc 3 1	*i++*
19:	iload_3	i
	iload_1	n i
	if_icmplt 10	*if i<n, goto 10*
	getstatic System.out	System.out
	iload_2	sum System.out
	invokevirtual println	
	return	

In order to pass an ordinary Java bytecode verifier, a JVM method must obey certain correctness conditions. Most notably, at any given point of execution, there must be a single consistent type assignment for the stack and local variables, such that the subsequent instructions may be validly invoked on those types. This condition ensures type safety.

4.2 The Prover algorithm

A complete proof for a method consists of:

1. A complete set of block aliases. (See sections 4.2.3 and 4.3.3.)

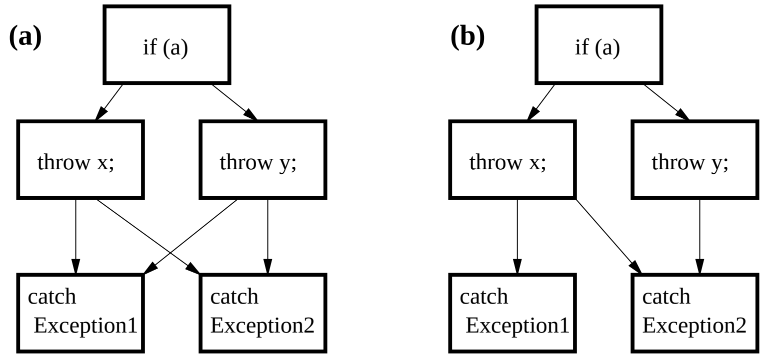
2. A control flow graph[1] (CFG) for the method's instructions.

3. A label for the formal parameter of each exception handler.

4. The pc label for every basic block.

[1]A **control flow graph** is a directed graph that indicates all transitions between a method's basic blocks. A basic block is a series of 1 or more contiguous instructions that is only entered at the first instruction, and only exited after the last instruction.

Figure 4-2: Conservative and accurate control flow graphs



Before type analysis, we have no way to know which exception may be caught by which handler, and so we build a conservative CFG.

Once we know the types of the exceptions being thrown, we can determine which handler can catch which exceptions.

5. For every instruction: a list of all the types and labels of all the stack elements and local variables at the time when the instruction is about to be invoked.

6. The static principal hierarchy at every point in the method.

It is not possible to construct this information directly, because of a few annoyances in the JVM's instruction set. First, in order to determine the `pc` labels at every instruction, we need to know the method's complete CFG. In order to do that, we need to know which `athrow` instructions correspond to which exception handlers. But there is no reliable way to tell the type of an `athrow` instruction without performing type analysis — which itself requires the CFG!

We escape from this obstacle as shown in figure 4-2: at first by using an overconservative CFG (a) to perform type analysis; next by using the deduced type information to build a more accurate CFG (b); and finally by using the accurate CFG to perform label analysis. [2]

4.2.1 Step 1: Building a conservative control flow graph

First, we build a conservative control flow graph (figure 4-2b) for the method. The conservative graph differs from the actual one only in that it does not consider actual exception types, but instead assumes that every exception-generating instruction may transfer control to an exception handler, or may exit the method.

In order to build the conservative control flow graph, we proceed as shown in figure 4-3. For the most part, this algorithm is relatively straightforward, except for the fact that in step 1b, we

[2] As an alternative, one might want to attempt to perform type and label analysis at the same time via a single dataflow algorithm. We have attempted this, with no success. It may be that no such algorithm is possible, since every type of change to a CFG can result in some `pc` labels becoming more strict and other becoming less strict, and so therefore it would be difficult to prove that such an algorithm would converge.

Figure 4-3: Building a conservative control flow graph

1. Initialize $Succ[i] = \{\}$ for all i.
2. Consider all instructions in turn.
   (a) If the instruction at i is:
   - an unconditional jump to j, let $Succ[i] = \{j\}$.
   - a conditional branch to j: let $Succ[i] = \{i+1, j\}$.
   - a switch instruction with targets $j_1 \dots j_n$: let $Succ[i] = \{j_1 \dots j_n\}$.
   - If the instruction at i If it is a return instruction, let $Succ[i] = \{\textbf{exit}\}$. If it is an athrow instruction, let $Succ[i] = \{\}$.
   - Otherwise: let $Succ[i] = \{i+1\}$.

   (b) If the instruction at i may throw any exception,[a] add to $Succ[i]$ the start address of every exception handler that handles exceptions from i.
3. If any instruction i is contained in $Succ[j]$ for some j, and $i \neq j+1$, then i is the beginning of a basic block that extends from i up to the beginning of the next basic block. The successors of the block is $Succ[k]$, where k is the last instruction in the block.

[a]Exceptions may be thrown by putfield, getfield, arrayload, arraystore, divide, remainder, invoke, newarray, arraylength, athrow, and checkcast instructions.

introduce an edge from every instruction that *might* throw *any* exception to the beginning of *every* exception handler that is active when the instruction is executed.

4.2.2 Step 2: Type checking

The type checking algorithm is almost identical to the ordinary dataflow algorithm described in the JVM specification [?]. Of course, Jif's features introduce new issues beyond those encountered in ordinary Java. These are as follows:

1. Most significantly, we must handle the additional complexity of Jif's type system, which includes parameterized types. To do so, we use Jif's subtyping rules [?].
2. We need to track constant-valued strings, labels, and principals, so that we can track their use in certain statements and expressions. For example, upon encountering a call to a constructor of a parameterized type, we must know the *actual values* of the arguments that represent the type's parameters.
3. When a boolean is the result of an actsFor or relabelsTo query, or when it signals the onset of a declassify statement (see section 3.1), we track its source, so that we can later determine which blocks correspond to which of Jif's new statements.

Figure 4-4: Exceptions thrown by JVM instructions

Instruction	**Exception thrown**
athrow	Type on top of stack
invoke	Exceptions thrown by invoked method
putfield, getfield arrayload, arraystore athrow, invoke	NullPointerException
arrayload, arraystore	ArrayIndexOutOfBoundsException
divide, remainder	ArithmeticException
newarray	NegativeArraySizeException
checkcast	ClassCastException

Figure 4-5: Relationships between CFG nodes

$$
\begin{aligned}
\textsc{Path}(A,B) &= \{N_1, N_2, \ldots, N_n \mid N_1 = A \text{ and } N_n = B \text{ and } \forall i, N_i + 1 \in \textsc{Succ}(N_i)\} \\
\textsc{Follows}(A) &= \{B \mid \textsc{Path}(A,B) \neq \emptyset\} \\
\textsc{Dom}(A) &= \{B \mid \forall P \in \textsc{Path}(\textsc{Start}, A), B \in P\} \\
\textsc{PDom}(A) &= \{B \mid \forall P \in \textsc{Path}(A, \textsc{Exit}), B \in P\}
\end{aligned}
$$

4.2.3 Step 3: Final control flow graph

Before the final label checking phase, we use the information gained in step 2 to determine our final control flow graph, pc labels, and static principal hierarchy.

To find the accurate flow graph (figure 4-2b), we follow the same process as in the original CFG algorithm of figure 4-3, except that in step 2b, we only add an exception handler to $Succ[i]$ if the instruction at i might throw an exception to that handler: if the handler is in reach; and if either the thrown exception type is a subtype of the handler type, or the handler type is a subtype of the thrown exception type.[3]. We determine the possible thrown exception types as shown in figure 4-4 If any instruction may throw an exception that is not definitely caught by an exception handler, that instruction also has a transition to the exit node.

Establishing pc labels

Once we've found the successor relations for the basic blocks, we need to find the FOLLOWS, DOM, and PDOM sets so that we can calculate the pc labels. These relationships are commonly used in compilers [?]; we define them in figure 4-5. In general, A *follows* B if a path exists from B to A; A *dominates* B if *every* path from the start node to B must pass through A; and A *postdominates* B if every path from B to the exit node must pass through A.

[3]For example, if the exception we're throwing is apparently an Exception, it would certainly be caught by catch (Throwable t), and possibly caught by catch (RuntimeException e)

We find values for these relations with any standard procedure, such as those given in Muchnick's *Advanced Complier Design and Implementation* [?].

We then find the pc-INFLUENCERS sets of each basic block. These are all the basic blocks whose exit conditions influence the pc label of a given block. By the ordinary algorithm discussed in section 2.2.1, the pc-INFLUENCERS of block B^* are all B such that $B^* \in \text{FOLLOWS}(B)$ and $B^* \notin PDom(B)$.

In English, this formula states that, by reaching a block, we learn about the exit-conditions of all the blocks that might have come before, except when were certain to reach this block no matter what the value of the preceding block.

4.2.4 Step 4: Special Jif statements

At this point, we determine the effects of Jif's new statements, and establish the static principal hierarchy for the method.

First, we use actsFor checks to establish the static principal hierarchy. We begin by initializing the static principal hierarchy to that implied by the method's actsFor constraints. Next, we find all the conditional branch instructions in the method that branch on the result of a Principal.actsFor(P1, P2) check. Suppose that such a branch instruction in block B^* branches on true to block T. Let the set of blocks $S = \{B | B \in \text{DOM}(T)\}$. For every block B in S, the relationship P1 actsFor P2 is inserted into the static principal hierarchy.

Next, we compute the labels for the exception formals. This is fairly simple: the first statement of any exception handler is requred to be a call to Labels.noteFormal. We use the literal String argument to noteFormal as the label for that handler's formal argument.

Finally, we need to handle the switch label statement. We do this analogously to actsFor: we find the conditional branch instructions that depend on the result of L1.relabelsTo(L2), and – within the blocks dominated by their true-branches – note that L1 relabelsTo L2.

At this point we are finally ready to find a valid label assignment for the method.

4.2.5 Step 5: Label checking

In order to find the labels, we proceed according to the following dataflow algorithm.

1. Set the initial state so that, at the start of the first block:
   (a) The pc label is the method's begin label.
   (b) The stack is empty.
   (c) The labels on the local variables are {}, except for the labels of the arguments, which are given by the method's signature.

Set the pc for all *other* blocks, and the labels for the stack and locals at all *other* positions, to $\{\}$.

2. To process a block, start with the block's initial state, and simulate each of its instructions. For more information on the effects of various instructions, see figure 4-6 below.

3. Upon reaching the end of a block, recalculate the incoming state for every successor of that block. The incoming state of a block is the join of the outgoing states of all preceding blocks.

4. After processing a block B ending in a conditional branch or possible exception, let pc_b be the label of the value(s) influencing the branch. Let $\mathsf{pc}\ (B^*) \leftarrow \mathsf{pc}\ (B^*) \sqcup \mathsf{pc}_b$ for all B^* such that $B \in \mathsf{pc}\text{-}\textsc{influencers}\ (B^*)$.

5. Continue processing blocks until no block has any change to its incoming state.

Simulating JVM instructions

A great deal of the prover's work lies in step 2 above: simulating JVM instructions on a set of label states. Fortunately, the simulation is relatively straightforward: nearly all of the instructions are trivial to simulate. In figure 4-6, we show the transitions for some of the most common types of JVM instructions.

The simplest instruction types are arithmetic, constant, stack, type, and local-variable instructions, as shown at top of the table; here, the labels of the instructions' outputs depend trivially on the input state.

To determine the types labels on the results of a method call, we follow exactly the same rules as Jif. Additionally, it is possible that a method call cannot be permitted according to Jif; in this case, we reject.

Array construction presents complications. It occurs in two steps: first, the newarray instruction creates an array to hold elements with an unspecified label, and second, the noteArray method tells the verifier the actual label for the array's elements. Note that in the stack transitions shown in the figure, the array initially contains elements of an unspecified label, represented by $\{?\}$. Any attempt to access an array with unknown element type must be forbidden.

The declassify operation requires that the top of the stack contain a label with a known value. This label is used as the label of the new expression. Of course, we only allow this new label if it is safe to declassify L2 into L1, given the current authority.

Managing labels themselves requires some ingenuity. While static (compile time) labels are easy enough to track, runtime labels require more work. Since Jif guarantees that every Label reference used in a runtime label will be final, we can track which labels on the stack come from final variables or fields. Then, when they're inserted into existing labels, we can recognize the resulting runtime label, as shown in the last section of the figure.

Figure 4-6: Effects of JVM instructions on the stack

	Stack before	Instruction	Stack after
Arithmetic:	A, B, int{L1}, int{L2}	iadd	A, B, int{L1 $\sqcup$ L2}
Constants:	A, B	iconst_0	A, B, int{}
Stack:	A, B	dup	A, B, B
Types:	A, B{L}	instanceof	A, boolean{L}
Locals:	A, B	aload var	A, B, (label of var)
	A, B, C{}	astore var	A, B
		[Local variable var receives label C{}]	
Invocation:	A, B, C	invoke B{D m(C)}	A, D
Arrays:	A, B{L1}[]{L2}, int{L3}	aaload	A, B{ L1 $\sqcup$ L2 $\sqcup$ L3 }
	A, B, int{L1}	anewarray B	A, B{?}[]{L1}
	A, Label=L1, B{?}[]{L2}	invoke noteArray	A, B{L1}[]{L2}
Declassify:	A, B{L2}, Label=L1	invoke declassify	A,B{L1}
Labels:	A, Label=?, String=S1	invoke Label.<init>	A, Label=S1
	A, Label=L1, String=S2	invoke Label.join	A, Label={L1 $\sqcup$ S1}
	A, Label=L1, Label=L2<var v>	invoke Label.join	A, Label={L1 $\sqcup$ *v}

Note that all of the above stack transition assume that the pc label is {}. If this is not so, the actual pc label is included in all of the output values.

Transitions and Exceptions

A final source of complexity in the prover is the fact that, when an instruction throws an exception, the pc label of the exception does not always depend on all the instruction's arguments. For example, the aload (load from an array) instruction can throw an exception either when the array is null, or when the index is out of bounds. In the first case, the exception only tells us about the value of the array, while in the second case, the exception tells us about both the array and the index. Step 4 of the algorithm above must take this distinction into account.

4.3 Removing JSR instructions

The algorithm above only works for a simplified set of JVM instructions that does not include Java's jsr and ret. Therefore, the Prover desugars them before it begins. Although the Prover and Checker are simplified by omiting these instructions, desugaring them is not completely trivial.

This section discusses our motivation for removing these instructions, the algorithm we use to remove them, and the ramifications of doing so.

4.3.1 Motivation

The jsr and ret instruction pair was originally added to the JVM so that Java compilers could generate the code for try/finally statements without having to duplicate the finally block at every possible exit point from the try block.

In practice, however, the benefits of these instructions are small. The savings in code size is typically no more than [some very small number] ([?], [?] and see section ??). The complexity added to the bytecode verifier, on the other hand, is enormous. The presence of jsr blocks prevents a verifier from forming a reliable control flow graph for any method that contains them; instead, the verifier must first infer the location and extent of each finally block in a method, and analyze each one independently from the rest of the method. Therefore, by removing the jsr instruction from the JifVM's repertoire, we can decrease the amount of complexity required of our verifier.

4.3.2 JSR desugaring algorithm

The desugaring algorithm is as shown in figure 4-7. It takes as its input a valid JVM method with jsr and ret instructions, and returns a new JVM method without any such instructions.

Note that this algorithm results in some dead code: every instance of the finally block is copied to the end of the method, and the original location of each block becomes dead. At the expense of an increase in code size, this strategy makes it easier to check that the appended blocks really do contain the same instructions. This makes block aliases (section 4.3.3) easier to determine.

4.3.3 JSR desugaring and pc the label

In the process of removing jsr instructions, we duplicate the code in the finally blocks. This duplication, however, makes it harder to establish their pc labels.

Consider the situation in figure 4-8. Before removing the jsr instructions, the pc labels are as shown on the left. The finally block has the label {}, because it will be reached no matter what the value of a is.

After desugaring, however, the finally block has turned into two separate blocks in the CFG. Since they are now 'distinct,' the value of a influences which one will be reached, and so each one must have its pc labeled {a}. This change in the pc label would alter the interpretation of the rest of the program, since it might require a change in the label of y.

The solution, as shown to the right, is to recognize that two distinct blocks that contain the same instructions are *aliases* for one another. We declare that two blocks B_1 and B_2 may be aliases only if they contain identical instructions. We then let $\mathsf{Aliases}(B)$ be the set including B, and every block identical to B.

For the purposes of finding the pc label, if two blocks are aliases, there is no way to distinguish

Figure 4-7: Algorithm for desugaring JSR instructions

DESUGARJSR: inlines all finally blocks in a method.

1. Identify all jsr targets in the method.
2. As in the JSR verifier, find for every instruction a set of all jsr targets required to reach that instruction. [a] Call the set for instruction i be $T(i)$.
3. Pick a target t from among the targets with the greatest $\#T(t)$. Identify the associated ret instruction.
4. Let J be the set of jsr instructions targeted to t. For every $j \in J$:
   (a) JCOPY all instructions i with $t \in T(i)$ to the end of the method, prefixing them with an ipush_0 instruction.
   (b) Replace the jsr instruction j with an unconditional jump to the start of the copied block.
   (c) Replace the ret instruction in the copied block with an unconditional jump to the instruction after j.

JCOPY: copies a finally block of Java bytecodes.

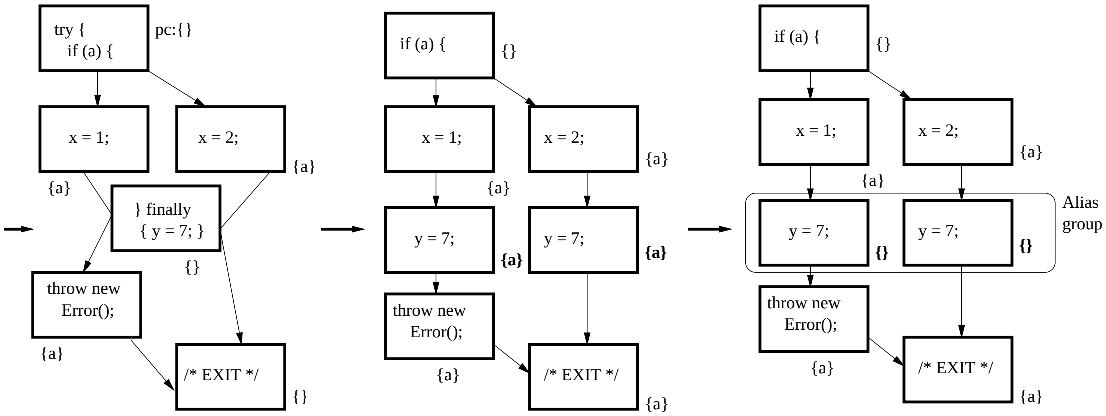
1. Let s be the first source and d the first destination address.
2. Create a mapping D from source to destination addresses. Let D[s] = d.
3. For every instruction i to be copied:
   (a) Let D[i] = d.
   (b) Increment d by the length of i.
4. Copy every instruction i to $D[i]$. If instruction i contains a relative branch of length k, change it to $D[i+k] - D[i]$. If instruction i contains an unconditional branch to k, change it to $D[k]$.

[a] The JVM specification [?] guarantees that this list will exist for all correct code, and that such lists will be strictly nested.

whether the program is in one block or another. We therefore alter the definition of the pc label from section 4.2.3 as follows: The pc label of a BB is the join of every B_l such that BB may be reached from B, and such that the exit node may be reached from B without passing through Aliases(BB).

Another restriction must apply to alias sets, in order to avoid a potential information leak. In the pseudocode of figure 4-9, it is clear that once we are executing the if statement, we are bound to execute both block a and block b. It is also clear that we can tell which block we executed first by inspecting the value of x, and thereby learn information about the variable secret. To solve this problem, we require that the groups of nodes postdominated by alias sets must be properly nested. This nesting restriction would render the code in figure ?? illegal, and thereby block the associated information leak.

Figure 4-8: JSR desugaring and the pc label



(CFG before JSR desugaring)

CFG after JSR desugaring, but before alias detection. Note that the pc label on the finally block has changed.

CFG after alias detection. Note that the pc label on the finally block is back to its original value.

Figure 4-9: Hostile use of misnested finally blocks

```
if (secret > 0) {
  jsr a; jsr b;
} else {
  jsr b; jsr a;
}
return x;
a: x = 1; ret;
b: x = 2; ret;
```

4.3.4 JSR desugaring and flow safety

Because of the complexity of the JSR desugaring process and its associated analysis, it is critical to discuss its effect on flow safety.

First, we consider the correctness of the desugaring algorithm itself. It follows that because the algorithm takes the form of ordinary function inlining, it will not change a function's semantics.

Our argument for the safety of the block-aliasing rules is more complicated, and runs as follows. First note that we only identify blocks as aliases *exactly when* they could be combined into a single jsr statement, and that we only allow these aliases (according to the nesting restriction above) when they would correspond to properly-nested try-finally blocks. Because Myers' rules **??** are valid for Java's try-finally blocks, and because our rules only allow block aliases that are isomorphic to try-finally, it follows that our rules are safe.

4.4 Encoding the proofs

Because the Prover must send proofs to the Checker over a potentially bandwidth-limited medium, we need to ensure that proofs are encoded as efficiently as possible.

Some of our arguments below hinge on average-case proof size. We analyze the relative proof sizes for samples of production code in section 6.1.1.

Encoding the CFG

To encode the CFG, we first mark all of the instructions that begin basic blocks. Next, we associate with each basic block a list of its successors. The maximum number of successors of a basic block is on average small, because nearly all instructions have an upper bound of one to four successors. (The only unbounded cases are switch statements and calls to methods with many exceptions.)

The proof does not transmit DOM, PDOM, FOLLOWS, or pc-INFLUENCERS—first, because these relations are not much easier to check than they are to derive from SUCC, and second, because FOLLOWS and pc-INFLUENCERS are difficult to encode efficiently.

Encoding pc

We encode types and labels using the same method the JVM uses to encode types in class files. Specifically, we provide a list of all the types and of all the labels used in the proof, and describe types and labels as indices into that table.

We encode the pc label for each block simply as the index to the appropriate label.

Encoding type/label assignments

Because the stack depth in a Java method is in the worst case proportional to the method's length, a naive encoding for type assignments would risk possible instructions $\times$ depth $= O(n^2)$ proof size. By optimizing for average-case methods, however, we can improve on this.

The main idea is to list explicitly all of the types and labels for the start node, and for all the nodes with more than one predecessor (join nodes). For all other nodes, we encode the types and labels of the stack and locals by their difference from the previous instruction's state. We encode known constant values for the stack and locals in the same way.

This approach ought to be efficient, because Io instruction pushes more than a two values onto the stack, or changes more than one local variable; therefore, the memory needed for those instructions' type state is proportional to the number of such instructions. The efficiency depends, however, on the number of join nodes being relatively few, and the stack depth at join nodes being relatively shallow. The space needed is $O(\text{n\_join\_insns} \times \text{stack\_depth} + \text{n\_insns})$. In section 6.1.1 we will discuss the average fraction of join instructions and stack depths found in a typical method.

The rest of the proof

We encode exception handler formals as a simple list of $\langle \text{location}, \text{type} \rangle$ pairs.

Since there is little variation in the static principal hierarchy throughout a method, we encode it list of various values of the principal hierarchy, and give each basic block an index into the list.

4.5 An example proof

To conclude our discussion of the Prover, we provide an example proof for a small method, and discuss its derivation. [[**XXXX Write this.**]]

Chapter 5

Verifying JifVM code: The Checker

The Checker represents the more interesting of the two components in our two-phase model. It is designed to be as simple as possible, and to thereby ensure the safety of the system.

Because the Checker shares several components in common with the Prover, we will not describe those components in detail. Instead, we will describe the checker in outline, and go into detail only about those components that differ significantly from their counterparts in the Prover.

5.1 The Checker algorithm

The checker receives proof objects (as described in the previous chapter) and class files as its input, and either accepts or rejects the class files based on the validity of the proofs. It works as follows:

1. Make sure that the proof is syntacticly well-formed.
2. Validate the block aliases. (Section 5.1.1)
3. Check that the CFG matches the method. (Section 5.1.2)
4. Check the static principal hierarchy. (Section 5.1.3)
5. Check that the `pc` labels are consistent. (Section 5.1.4)
6. Check the type and label assignments for the method. (Section 5.1.5)

5.1.1 Checking block aliases

First, we provide a general means to see whether two blocks are aliases for one another. Next, we note a special case that simplifies our verifier, given that the desugaring algorithm from section 4.3.2

is used.

In the general case, we build control flow graphs for each block, with branch targets outside of the blocks represented as separate exit nodes. Then we determine whether the instructions within each basic block are equivalent, and that the graphs themselves are identical.

When the jsr desugaring algorithm from 4.3.2 is used, however, we have a special case: each aliased block consists entirely of contiguous instructions, in the same order. Therefore, it is sufficient to ensure that candidate aliases for a block contain the same contiguous instructions, and that branch targets within the blocks are appropriately offset.

5.1.2 Checking the CFG

To check the CFG, the Checker first ensure that every branch instruction, or instruction that might throw an exception, is at the end of a basic block, and that the successors of that instruction are the successors of that block.

The Checker then regenerates FOLLOWS, DOM, and PDOM as needed.[1]

5.1.3 Checking the static principal hierarchy

In order to determine whether the static principal hierarchy is consistent with the method, the Checker tries to recompute it according to the rules of section 4.2.4, using the type and constant information in the proof.

5.1.4 pc consistency

To check the pc labels, we first recompute the pc-INFLUENCERS set for every block. Next, we find verify that every block's pc label is actually the join of the outgoing pc labels of its pc-INFLUENCERS. Finally, we make sure that each block's outgoing pc labels correspond correctly to the proper labels on the stack at the end of the block.

Note that, in order to check that $L = L_1 \sqcup L_2$, it is sufficient to check that $L_1 \sqsubseteq L$ and that $L_2 \sqsubseteq L$. This implies that the Checker does not need to include the code for $\sqcup$, so long as it has the code for $\sqsubseteq$.

[1] We do not pass these relations as part of the proof, mainly because (a) these relations do does not seem to be any easier to check than they are to recompute, and (b) nearly any encoding for FOLLOWS would need to be quadratic in size.

Parenthetically, we note that if we *did* try to check these relations, it would not be necessary to show that they were exactly correct; rather, it would be sufficient to show that FOLLOWS was not missing any nodes, and that DOM and PDOM did not have any extra nodes.

Our reasoning is as follows: pc-INFLUENCERS must be as large as needed, so that the pc is always at least as restrictive as necessary. On the other hand, there is no danger to safety if pc-influencers is *too* large: if it is, then some pc labels ill be too restrictive, and the class will be rejected. Therefore, to make sure that pc-influencers is not missing any blocks–given the formula from section 4.2.3 we must make sure that FOLLOWS is not missing any blocks, and that PDOM does not contain any extra blocks.

The reasoning for DOM follows similarly from its usage.

5.1.5 Checking the type and label assignments

Here, we start by checking that the method's initial and final states are really as implied by the method signature. Next, we simulate each block exactly once (as in 4.2.5), and make sure that, beginning with the block's initial state, we end with its final state. Finally, we check that the incoming state of each block really is the join of the outgoing states of its immediate predecessors.

5.2 Examining the Checker's correctness

Although we will not provide a formal proof, if is valuable to consider how we might argue for the Checker's correctness.

We take our correctness conditions from Myers' work on Jif [?]: (a) the apparent label of every expression must be at least as restrictive as the actual labels of its possible values; and (b) the actual label of a value is at least as restrictive as the actual label of all the values that could have affected it.

To demonstrate the Checker's correctness, we need to argue that *if* the Checker accepts a proof for a piece of code, then that proof represents an assignment for the 'apparent labels' that satisfies the conditions above.

If we assume that Jif is correct, then the Checker's correctness depend on the correctness of the pc rules given previously, and on the correctness of the instruction simulation rules from section 4.2.5. If the pc rules truly determine the values we can learn by discovering that we have executed a given block, and it the instruction simulation rules really yield labels at least as restrictive as the output of their JVM instructions, then it follows that—when the checker agrees that a given label assignment is consistent with the method's instructions—that assignment will be at least as restrictive as it needs to be.

The correctness of our pc rule has been discussed more fully in sections 2.2.1 and 4.3.3, and by Myers [?]. Because the JVM has so many instructions, it is not practical to demonstrate the validity of our simulation for each one; instead, we refer to the arguments of section 4.2.5.

5.3 An example

[[**Continue example from ch. 4**]]

Chapter 6

Conclusions

In this chapter, we analyze some of the performance implications of our approach, discuss some related approaches and techniques, and suggest some possible directions for future research.

6.1 Results

Throughout this document, we have made claims for the performance and simplicity of our approach. In this section we will give experimental results to argue for these claims.

In discussing experimental results, we will use a corpus of examples to run our tests. For this, we choose the core class files from version 1.2.2 of Sun's Java Developer Kit. We have run these same tests on other large systems, including the checker itself, but not found significant variation in the results.

6.1.1 Results: Proof encoding

Before our system can be deployed, it is essential that its proofs can be encoded in a fairly compact fashion. In particular, we would like for the size of each proof to vary at most linearly with the length of a method's code.[1]

As we argued in section 4.4, the CFG, the `pc` labels, the static principal hierarchy, and the formals for the exception handlers are necessarily linear in code size. The only uncertain factor is the length of the type and label assignments, which is O(the number of instructions, + the number of join nodes × the stack depth at the join nodes[2]).

Upon analysis, we found that the average method has 27 instructions, out of which 1 begins a join node. Even in a worst-case scenario, where the stack is at its maximum depth within the method

[1] For our data, method code accounts for 30-40% of the class files' length.

[2] In practise, the stack depth depends only on the complexity of a method's most complex expression. In all of Sun's class files, the stack is never deeper than 20 words.

at the point of the join, the average join only incurs a cost of 12.5 words. Therefore, join nodes add (on average) less than $12.5/27 \approx 46\%$ to the cost of a method's type and label information.

Therefore, although the worst-case proof length is quadratic in the method's length, it seems that in practice, the relation will be linear.

6.1.2 Results: Two-stage verification

Our principal claim for our two-stage architecture has been that, because the Checker is significantly simpler than a full verifier, it is far easier to be confident in the Checker's correctness. In this section, we examine Checker's actual simplicity.

Unfortunately, we do not have the code necessary for a full, direct comparison, because we have not written an ordinary verifier for Jif bytecodes. Therefore, we will list the features which a full verifier would have to contain that the Checker does not, as well as the features which the Checker needs that a full verifier would not.

Verifier features not needed by the Checker:

- Type join

6.1.3 Results: Removing JSR's

x

6.2 Related work

x

6.3 Future work

x