Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL '99), San Antonio, Texas, USA, January 1999

JFlow: Practical Mostly-Static Information Flow Control

Andrew C. Myers

Laboratory for Computer Science
Massachusetts Institute of Technology
http://www.pmg.lcs.mit.edu/~andru

Abstract

A promising technique for protecting privacy and integrity of sensitive data is to statically check information flow within programs that manipulate the data. While previous work has proposed programming language extensions to allow this static checking, the resulting languages are too restrictive for practical use and have not been implemented. In this paper, we describe the new language *JFlow*, an extension to the Java language that adds statically-checked information flow annotations. JFlow provides several new features that make information flow checking more flexible and convenient than in previous models: a decentralized label model, label polymorphism, run-time label checking, and automatic label inference. JFlow also supports many language features that have never been integrated successfully with static information flow control, including objects, subclassing, dynamic type tests, access control, and exceptions. This paper defines the JFlow language and presents formal rules that are used to check JFlow programs for correctness. Because most checking is static, there is little code space, data space, or run-time overhead in the JFlow implementation.

1 Introduction

Protection for the privacy of data is becoming increasingly important as data and programs become increasingly mobile. Conventional security techniques such as discretionary access control and information flow control (including mandatory access control) have significant shortcomings as privacy-protection mechanisms.

The hard problem in protecting privacy is preventing private information from leaking through computation. Access control mechanisms do not help with this kind of leak, since they only control information release, not its propagation once released. Mandatory access control (MAC) mechanisms prevent leaks through propagation by associating a run-time *security class* with every piece of computed data. Every computation requires that the security class of the result value also be computed, so multi-level systems using this approach are slow. Also, these systems usually apply a security class to an entire process, tainting all data handled by the process. This coarse granularity results in data whose security class is overly restrictive, and makes it difficult to write many useful applications.

A promising technique for protecting privacy and integrity of sensitive data is to *statically* check information flows within programs that might manipulate the data. Static checking allows the fine-grained tracking of security classes through program computations, without the run-time overhead of dynamic security classes. Several simple programming languages have been proposed to allow this static checking [DD77, VSI96, ML97, SV98, HR98]. However, the focus of these languages was correctly checking information flow statically, not providing a realistic programming model.

This paper describes the new language *JFlow*, an extension to the Java language [GJS96] that permits static checking of flow annotations. JFlow seems to be the first practical programming language that allows this checking. Like other recent approaches [VSI96, ML97, SV98, HR98, ML98], JFlow treats static checking of flow annotations as an extended form of type checking. Programs written in JFlow can be statically checked by the JFlow compiler, which prevents information leaks through *storage channels* [Lam73]. JFlow is intended to support the writing of secure servers and applets that manipulate sensitive data.

An important philosophical difference between JFlow and other work on static checking of information flow is the focus on a usable programming model. Despite a long history, static information flow analysis has not been widely accepted as a security technique. One major reason is that previous models of static flow analysis were too limited or too restrictive to be used in practice. The goal of the work presented in this paper has been to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner.

This work has involved several new contributions: JFlow extends a complex programming language and supports many

This research was supported in part by DARPA Contract F30602-96-C-0303, monitored by USAF Rome Laboratory, and in part by DARPA Contract F30602-98-1-0237, also monitored by USAF Rome Laboratory.

language features that have not been previously integrated with static flow checking, including mutable objects (which subsume function values), subclassing, dynamic type tests, and exceptions. JFlow also provides powerful new features that make information flow checking less restrictive and more convenient than in previous programming languages:

- It supports the *decentralized label model* [ML97, ML98], which allows multiple principals to protect their privacy even in the presence of mutual distrust. It also supports safe, statically-checked *declassification*, or *downgrading*, allowing a principal to relax its own privacy policies without weakening policies of other principals.
- It provides a simple but powerful model of access control that allows code privileges to be checked statically, and also allows authority to be granted and checked dynamically.
- It provides *label polymorphism*, allowing code that is generic with respect to the security class of the data it manipulates.
- Run-time label checking and first-class label values provide a dynamic escape when static checking is too restrictive. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself.
- Automatic label inference makes it unnecessary to write many of the annotations that would otherwise be required.

The JFlow compiler is structured as a source-to-source translator, so its output is a standard Java program that can be compiled by any Java compiler. For the most part, translation involves removal of the static annotations in the JFlow program (after checking them, of course). There is little code space, data space, or run time overhead, because most checking is performed statically.

The remainder of this paper is structured as follows: Section 2 contains an overview of the JFlow language and a rationale for the decisions taken. Section 3 discusses static checking, sketches the framework used to check program constructs in a manner similar to type checking, and both formally and informally presents some of the rules used. This section also describes the translations that are performed by the compiler. Section 4 compares this work to other work in related areas, and Section 5 provides some conclusions. The grammar of JFlow is provided for reference in Appendix A.

2 Language overview

This section presents an overview of the JFlow language and a rationale for its design. JFlow is an extension to the Java language that incorporates the decentralized label model. In Section 2.1, the previous work on the decentralized label model [ML97, ML98] is reviewed. The language description in the succeeding sections focuses on the differences between JFlow and Java, since Java is widely known and well-documented [GJS96].

2.1 Labels

In the decentralized label model, data values are *labeled* with security policies. A label is a generalization of the usual notion of a security class; it is a set of policies that restrict the movement of any data value to which the label is attached. Each policy in a label has an *owner* O, which is a principal whose data was observed in order to create the value. Principals are users and other authority entities such as *groups* or *roles*. Each policy also has a set of *readers*, which are principals that O allows to observe the data. A single principal may be the owner of multiple policies and may appear in multiple reader sets.

For example, the label $L = \{\, o_1\colon r_1, r_2;\, o_2\colon\ r_2, r_3\}$ has two policies in it (separated by semicolons), owned by o_1 and o_2 respectively. The policy of principal o_1 allows r_1 and r_2 to read; the policy of principal o_2 allows r_2 and r_3 to read. The *effective reader set* contains only the common reader r_2. The least restrictive label possible is the label $\{\}$, which contains no policies. Because no principal expresses a privacy interest in this label, data labeled by $\{\}$ is completely public as far as the labeling scheme is concerned.

There are two important intuitions behind this model: first, data may only be read by a user of the system if all of the policies on the data list that user as a reader. The effective policy is an intersection of all the policies on the data. Second, a principal may choose to relax a policy that it owns. This is a safe form of *declassification* — safe, because all of the other policies on the data are still enforced.

A process has the authority to act on behalf of some (possibly empty) set of principals. The authority possessed by a process determines the declassifications that it is able to perform. Some principals are also authorized to *act for* other principals, creating a *principal hierarchy*. The principal hierarchy may change over time, but revocation is assumed to occur infrequently. The meaning of a label is affected by the current principal hierarchy. For example, if the principal r' can act for the principal r, then if r is listed as a reader by a policy, r' is effectively listed by that policy as well. The meaning of a label under different principal hierarchies is discussed extensively in an earlier paper [ML98].

Every variable is *statically bound* to a static label. (The alternative, *dynamic binding*, largely prevents static analysis and can be simulated in JFlow if needed.) If a value v has label L_1 and a variable x has label L_2, we can assign the value to the variable ($x := v$) only if L_1 can be *relabeled* to L_2, which is written as $L_1 \sqsubseteq L_2$. The definition of this binary relation on labels is intuitive: $L_1 \sqsubseteq L_2$ if for every policy in L_1, there is some policy in L_2 that is at least as restrictive [ML98]. Thus, the assignment does not leak information.

In this system, the label on x is assigned by the programmer who writes the code that uses x. The power to select a label for x does not give the programmer the ability to leak v, because the static checker permits the assignment to x only if the label on x is sufficiently restrictive. After the assignment, the static binding of the label of x prevents leakage. (Changes in who can read the value in x are effected by modifying the principal hierarchy, but changes to the principal hierarchy require appropriate privilege.)

Computations (such as multiplying two numbers) cause *joining* ($\sqcup$) of labels; the label of the result is the least restrictive label that is at least as restrictive as the labels of the values used in the computation; that is, the least upper bound of the labels. The join of two sets of policies is simply the union of the sets of policies. The relation $\sqsubseteq$ generates a lattice of equivalence classes of labels with $\sqcup$ as the LUB operator. Lattice properties are important for supporting automatic label inference and label polymorphism [ML97, ML98]. The notation $A \approx B$ is also used as a shorthand for $A \sqsubseteq B \land B \sqsubseteq A$ (which does not mean that the labels are equal [ML98]).

Declassification provides an escape hatch from strict information flow tracking. If the authority of a process includes a principal p, a value may be declassified by dropping policies owned by principals that p acts for. The ability to declassify provides the opportunity for p to choose to release information based on a more sophisticated analysis.

All practical information flow control systems provide the ability to declassify data because strict information flow control is too restrictive to write real applications. More complex mechanisms such as *inference controls* [Den82] often are used to decide when declassification is appropriate. In previous systems, declassification is performed by a *trusted subject*: code having the authority of a highly trusted principal. One key advantage of the new label structure is that it is *decentralized*: it does not require that all other principals in the system trust a principal p's declassification decision, since p cannot weaken the policies of principals that it does not act for.

2.2 Labeled types

This section begins the description of the new work in this paper (the JFlow programming language), which incorporates the label model just summarized. In a JFlow program, a label is denoted by a *label expression*, which is a set of *component expressions*. As in Section 2.1, a component expression of the form *owner: reader*$_1$*, reader*$_2$*, ...* denotes a policy. A label expression is a series of component expressions, separated by semicolons, such as $\{o_1\colon r_1, r_2;\ o_2\colon r_2, r_3\}$. In a program, a component expression may take additional forms; for example, it may be simply a variable name. In that case, it denotes the set of policies in the label of that variable. The label {a} contains a single component; the meaning of the label is that the value it labels should be as restricted as the variable a is. The label {a; o: r} contains two components, indicating that the labeled value should be as restricted as a is, and also that the principal o restricts the value to be read by at most r.

In JFlow, every value has a *labeled type* that consists of two parts: an ordinary Java type such as int, and a label that describes the ways that the value can propagate. The type and label parts of a labeled type act largely independently. Any type expression t may be labeled with any label expression $\{l\}$. This labeled type expression is written as $t\{l\}$; for example, the labeled type int{p:} represents an integer that principal p owns and only p can read (the owner of a policy is always implicitly a reader).

The goal of type checking is to ensure that the apparent, static type of each expression is a supertype of the actual, run-time type of every value it might produce; similarly, the goal of label checking is to ensure that the apparent label of every expression is at least as restrictive as the actual label of every value it might produce. In addition, label checking guarantees that, except when declassification is used, the apparent label of a value is at least as restrictive as the actual label of every value that might *affect* it. In principle, the actual label could be computed precisely at run time. Static checking ensures that the apparent, static label is always a conservative approximation of the actual label. For this reason, it is typically unnecessary to represent the actual label at run time.

A labeled type may occur in a JFlow program in most places where a type may occur in a Java program. For example, variables may be declared with labeled type:

```
int{p:} x;
int{x} y;
int z;
```

The label may always be omitted from a labeled type, as in the declaration of z. If omitted, the label of a local variable is inferred automatically based on its uses. In other contexts where a label is omitted, a context-dependent default label is generated. For example, the default label of an instance variable is the public label {}. Several other cases of default label assignment are discussed later.

```
int{public} x;
boolean{secret} b;
...
int x = 0;
if (b) {
   x = 1;
}
```

Figure 1: Implicit flow example

2.3 Implicit flows

In JFlow, the label of an expression's value varies depending on the evaluation context. This somewhat unusual property is needed to prevent leaks through *implicit flows*: channels created by the control flow structure itself.

Consider the code segment of Figure 1. By examining the value of the variable x after this segment has executed, we can determine the value of the secret boolean b, even though x has only been assigned constant values. The problem is the assignment x = 1, which should not be allowed.

To prevent information leaks through implicit flows, the compiler associates a *program-counter label* (<u>pc</u>) with every statement and expression, representing the information that might be learned from the knowledge that the statement or expression was evaluated. In this program, the value of <u>pc</u> during the consequent of the if statement is {b}. After the if statement, <u>pc</u> = {}, since no information about b can be deduced from the fact that the statement after the if statement is executed. The label of a literal expression (e.g., 1) is the same as its <u>pc</u>, or {b} in this case. The unsafe assignment

```
label{L} lb;
int{*lb} x;
int{p:} y;
switch label(x) {
   case (int{y} z) y = z;
   else throw new UnsafeTransfer();
}
```

Figure 2: Switch label

(x = 1) in the example is prevented because the label of x ({public}) is not at least as restrictive as the label of 1 in this expression, which is {b}, or {secret}.

2.4 Run-time labels

In JFlow, labels are not purely static entities; they may also be used as values. First-class values of the new primitive type label represent labels. This functionality is needed when the label of a value cannot be determined statically. For example, if a bank stores a number of customer accounts as elements of a large array, each account might have a different label that expresses the privacy requirements of the individual customer. To implement this example in JFlow, each account can be labeled by an attached dynamic label value.

A variable of type label may be used both as a first-class value and as a label for other values. For example, methods can accept arguments with run-time labels, as in the following method declaration:

```
static float{*lb} compute(int x{*lb}, label lb)
```

In this example, the component expression *lb denotes the label contained *in* the variable lb, rather than the label *of* the variable lb. To preserve safety, variables of type label (such as lb) may be used to construct labels only if they are immutable after initialization; in Java terminology, if they are final. (Unlike in Java, arguments in JFlow are always final.)

The important power that run-time labels add is the ability to be examined at run-time, using the switch label statement. An example of this statement is shown in Figure 2. The code in this figure attempts to transfer an integer from the variable x to the variable y. This transfer is not necessarily safe, because x's label, lb, is not known statically. The statement examines the run-time label of the expression x, and executes one of several case statements. The statement executed is the first whose associated label is at least as restrictive as the expression label; that is, the first statement for which the assignment of the expression value to the declared variable (in this case, z) is legal. If it is the case that $\{*lb\} \sqsubseteq \{p:\}$, the first arm of the switch will be executed, and the transfer will occur safely via z. Otherwise, the code throws an exception.

Since lb is a run-time value, information may be transferred through it. This can occur in the example by observing which of the two arms of the switch are executed. To prevent this information channel from becoming an information leak, the $\underline{pc}$ in the first arm is augmented to include lb's label, which is {L}. The code passes static checking only if the assignment from y to z is legal; that is, if $\{L\} \sqsubseteq \{y\}$.

```
class Account {
   final principal customer;
   String{customer:} name;
   float{customer:} balance;
}
```

Figure 3: Bank account using run-time principals

Run-time labels can be manipulated statically, though conservatively; they are treated as an unknown but fixed label. The presence of such opaque labels is not a problem for static analysis, because of the lattice properties of these labels. For example, given any two labels L_1 and L_2 where $L_1 \sqsubseteq L_2$, it is the case for any third label L_3 that $L_1 \sqcup L_3 \sqsubseteq L_2 \sqcup L_3$. This implication makes it possible for an opaque label L_3 to appear in a label without preventing static analysis. Using it, unknown labels, including run-time labels, can be propagated statically.

2.5 Authority and declassification

JFlow has capability-like access control that is both dynamically and statically checked. A method executes with some authority that has been granted to it. The authority is essentially the capability to act for some set of principals, and controls the ability to declassify data. Authority also can be used to build more complex access control mechanisms.

At any given point within a program, the compiler understands the code to be running with the ability to act for some set of principals, called the *static authority* of the code at that point. The actual authority may be greater, because those principals may be able to act for other principals.

The principal hierarchy may be tested at any point using the actsFor statement. The statement actsFor(p_1, p_2) S executes the statement S if the principal p_1 can act for the principal p_2. Otherwise, the statement S is skipped. The statement S is checked under the assumption that this acts-for relation exists: for example, if the static authority includes p_1, then during static checking of S, it is augmented to include p_2.

A program can use its authority to declassify a value. The expression declassify(e, L) relabels the result of an expression e with the label L. Declassification is checked statically, using the static authority at the point of declassification. The declassify expression may relax policies owned by principals in the static authority.

2.6 Run-time principals

Like labels, principals may also be used as first-class values at run time. The type principal represents a principal that is a value. A final variable of type principal may be used as if it were a real principal. For example, a policy may use a final variable of type principal to name an owner or reader. These variables may also be used in actsFor statements, allowing static reasoning about parts of the principal hierarchy that may vary at run time. When labels are constructed using run-time principals, declassification may also be performed on these labels.

```
public class Vector[label L] extends AbstractList[L] {
   private int{L} length;
   private Object{L}[ ]{L} elements;

   public Vector() ...
   public Object elementAt(int i):{L; i}
      throws (ArrayIndexOutOfBoundsException) {
         return elements[i];
      }
   public void setElementAt{L}(Object{} o, int{} i) ...
   public int{L} size() { return length; }
   public void clear{L}() ...
}
```

Figure 4: Parameterization over labels

Run-time principals are needed in order to model systems that are heterogeneous with respect to the principals in the system, without resorting to declassification. For example, a bank might store bank accounts with the structure shown in Figure 3, using run-time principals rather than run-time labels. With this structure, each account may be owned by a different principal (the customer whose account it is). The security policy for each account has similar structure but is owned by the principal in the instance variable customer. Code can manipulate the account in a manner that is generic with respect to the contained principal, but can also determine at run-time which principal is being used. The principal customer may be manipulated by an actsFor statement, and the label {customer:} may be used by a switch label statement.

2.7 Classes

Even in the type domain, *parameterizing* classes is important for building reusable data structures. It is even more important to have polymorphism in the information flow domain; the usual way to handle the absence of statically-checked type polymorphism is to perform dynamic type casts, but this approach works poorly when applied to information flow since new information channels are created by dynamic tests.

To allow usable data structures in JFlow, classes may be parameterized to make them generic with respect to some number of labels or principals. Class and interface declarations are extended to include an optional set of explicitly declared parameters.

For example, the Java Vector class is translated to JFlow as shown in Figure 4. Vector is parameterized on the label L, which represents the label of the contained elements. Assuming that secret and public are appropriately defined, the types Vector[{secret}] and Vector[{public}] would represent vectors of elements of differing sensitivity. Without the ability to parameterize classes on labels, it would be necessary to reimplement Vector for every distinct element label.

The addition of label and principal parameters to JFlow makes parameterized classes into simple *dependent types* [Car91], since types contain values. To ensure that these dependent types have a well-defined meaning, only immutable variables may be used as parameters.

Note that even if $\{\mathsf{secret}\} \sqsubseteq \{\mathsf{public}\}$, it is not the case that $\mathsf{Vector}[\{\mathsf{secret}\}] \leq \mathsf{Vector}[\{\mathsf{public}\}]$, since subtyping is invariant in the parameter L (the subtype relation is denoted here by $\leq$). When such a relation is sound, the parameter may be declared as a covariant label rather than as a label, which places additional restrictions on its use. For example, no method argument or mutable instance variable may be labeled using the parameter.

A class always has one implicit label parameter: the label {this}, which represents the label on an object of the class. Because $L_1 \sqsubseteq L_2$ implies that $C\{L_1\}$ acts like a subtype of $C\{L_2\}$, the label of this is necessarily a covariant parameter, and its use is restricted in the same manner as with other covariant parameters.

A class may have some authority granted to its objects by adding an authority clause to the class header. The authority clause may name principals external to the program, or principal parameters. If the authority clause names external principals, the process that installs the class into the system must have the authority of the named principals. If the authority clause names principals that are parameters of the class, the code that creates an object of the class must have the authority of the actual principal parameters used in the call to the constructor. If a class C has a superclass C_s, any authority in C_s must be covered by the authority clause of C. It is not possible to obtain authority by inheriting from a superclass.

2.8 Methods

Like class declarations, JFlow method declarations also contain some extensions. There are a few optional annotations to manage information flow and authority delegation. A method header has the following syntax (in the form of the Java Language Specification [GJS96]):

MethodHeader:
  Modifiers$_{opt}$ LabeledType Identifier
    BeginLabel$_{opt}$ (*FormalParameterList$_{opt}$*) *EndLabel$_{opt}$*
    Throws$_{opt}$ WhereConstraints$_{opt}$

FormalParameter:
  LabeledType Identifier OptDims

The return value, the arguments, and the exceptions may each be individually labeled. One subtle change from Java is that arguments are always implicitly final, allowing them to be used as type parameters. This change is made for the convenience of the programmer and does not significantly change the power of the language.

There are also two optional labels called the *begin-label* and the *end-label*. The begin-label is used to specify any restriction on pc at the point of invocation of the method. The end-label — the final pc — specifies what information can be learned by observing whether the method terminates normally. Individual exceptions and the return value itself also may have their own distinct labels, which provides fine-grained tracking of information flow.

In Figure 5 are some examples of JFlow method declarations. When labels are omitted in a JFlow program, a default label is assigned. The effect of these defaults is that often methods require no label annotations whatever. Labels may

```
static int{x;y} add(int x, int y) { return x + y; }
boolean compare_str(String name, String pwd):{name; pwd}
      throws(NullPointerException) { ... }
boolean store{L}(int{} x)
      throws(NotFound) { ... }
```

Figure 5: JFlow method declarations

be omitted from a method declaration, signifying the use of *implicit label polymorphism*. For example, the arguments of add and compare_str are unlabeled. When an argument label is omitted, the method is generic with respect to the label of the argument. The argument label becomes an implicit parameter of the procedure. For example, the method add can be called with any two integers x and y, regardless of their labels. This label polymorphism is important for building libraries of reusable code. Without it, a math routine like add would have to be reimplemented for every argument label ever used.

The default label for a return value is the end-label, joined with the labels of all the arguments. For add, the default return value label is exactly the label written ({x;y}), so the return value could be written just as int. The default label on an exception is the end-label, as in the compare_str example. If the begin-label is omitted, as in add, it becomes an implicit parameter to the method. Such a method can be called regardless of the caller's pc. Because the pc within the method contains an implicit parameter, this method is prevented from causing real side effects; it may of course modify local variables and mutate objects passed as arguments if they are appropriately declared, but true side effects would create static checking errors.

Unlike in Java, the method may contain a list of *constraints* prefixed by the keyword where:

WhereConstraints:
  where *Constraints*

Constraint:
  authority (*Principals*)
  caller (*Principals*)
  actsFor (*Principal* , *Principal*)

There are three different kinds of constraints:

- authority($p_1, \ldots, p_n$) This clause lists principals that the method is authorized to act for. The static authority at the beginning of the method includes the set of principals listed in this clause. The principals listed may be either names of global principals, or names of class parameters of type principal. Every listed principal must be also listed in the authority clause of the method's class. This mechanism obeys the principle of least privilege, since not all the methods of a class need to possess the full authority of the class.

- caller($p_1, \ldots, p_n$) Calling code may also dynamically grant authority to a method that has a caller constraint. Unlike with the authority clause, where the authority devolves from the object itself, authority in this case devolves from the caller. A method with a caller clause may be called only if the calling code possesses the requisite static authority.

  The principals named in the caller clause need not be constants; they may also be the names of method arguments whose type is principal. By passing a principal as the corresponding argument, the caller grants that principal's authority to the code. These dynamic principals may be used as first-class principals; for example, they may be used in labels.

- actsFor (p_1,p_2) An actsFor constraint may be used to prevent the method from being called unless the specified acts-for relationship (p_1 acts for p_2) holds at the call site. When the method body is checked, the static principal hierarchy is assumed to contain any acts-for relationships declared in the method header. This constraint allows information about the principal hierarchy to be transmitted to the called method without any dynamic checking.

```
class passwordFile authority(root) {
   public boolean
     check (String user, String password)
     where authority(root) {
       // Return whether password is correct
       boolean match = false;
       try {
           for (int i = 0; i < names.length; i++) {
               if (names[i] == user &&
               passwords[i] == password) {
                   match = true;
                   break;
               }
           }
       }
           catch (NullPointerException e) {}
           catch (IndexOutOfBoundsException e) {}
       return declassify(match, {user; password});
   }
   private String [ ] names;
   private String { root: } [ ] passwords;
}
```

Figure 6: A JFlow password file

2.9 Example: passwordFile

Now that the essentials of the JFlow language are covered, we are ready to consider some interesting JFlow code. Figure 6 contains a JFlow implementation of a simple password file, in which the passwords are protected by information flow controls. Only the method for checking passwords is shown. This method, check, accepts a password and a user name, and returns a boolean indicating whether the string is the right password for that user.

The if statement is conditional on the elements of passwords and on the variables user and password, whose labels are implicit parameters. Therefore, the body of the if statement has pc = {user; password; root:}, and the variable

```
class Protected {
   final label{this} lb;
   Object{*lb} content;

   public Protected{LL}(Object{*LL} x, label LL) {
      lb = LL;       // must occur before call to super()
      super();       //
      content = x;   // checked assuming lb == LL
   }
   public Object{*L} get(label L):{L}
     throws (IllegalAccess) {
      switch label(content) {
         case (Object{*L} unwrapped) return unwrapped;
         else throw new IllegalAccess();
      }
   }
   public label get_label() {
      return lb;
   }
}
```

Figure 7: The Protected class

match also must have this label in order to allow the assignment match = true. This label prevents match from being returned directly as a result, since the label of the return value is the default label, {user; password}. Finally, the method declassifies match to this desired label, using its compiled-in authority to act for root. Note that the exceptions NullPointerException and IndexOutOfBoundsException must be explicitly caught, since the method does not explicitly declare them. More precise reasoning about the possibility of exceptions would make JFlow code more convenient to write.

Otherwise there is very little difference between this code and the equivalent Java code. Only three annotations have been added: an authority clause stating that the principal root trusts the code, a declassify expression, and a label on the elements of passwords. The labels for all local variables and return values are either inferred automatically or assigned sensible defaults. The task of writing programs is made easier in JFlow because label annotations tend to be required only where interesting security issues are present.

In this method, the implementor of the class has decided that declassification of match results in an acceptably small leak of information. Like all login procedures, this method does leak information, because exhaustively trying passwords will eventually extract the passwords from the password file. However, assuming that the space of passwords is large and passwords are difficult to guess, the amount of information gained in each trial is far less than one bit. Reasoning processes about acceptable leaks of information lie outside the domain of information flow control, but in this system, such reasoning processes can be accommodated in a natural and decentralized manner.

2.10 Example: Protected

The class Protected provides a convenient way of managing run-time labels, as in the bank account example mentioned earlier. Its implementation is shown in Figure 7. As the implementation shows, a Protected is an immutable pair containing a value content of type Object and a label lb that protects the value. Its value can be extracted with the get method, but the caller must provide a label to use for extraction. If the label is insufficient to protect the data, an exception is thrown. A value of type Protected behaves very much like a value in dynamic-checked information flow systems, since it carries a run-time label. A Protected has an obvious analogue in the type domain: a value dynamically associated with a type tag (*e.g.*, the Dynamic type [ACPP91]).

One key to making Protected convenient is to label the instance variable lb with {this}. Without this labeling, Protected would need an additional explicit covariant label parameter to label lb with.

2.11 Limitations

JFlow is not completely a superset of Java. Certain features have been omitted to make information flow control tractable. Also, JFlow does not eliminate all possible information leaks. Certain covert channels (particularly, various kinds of *timing channels*) are difficult to eliminate. Prior work has addressed static control of timing channels, though the resulting rules are restrictive [AR80, SV98]. Other covert channels arise from Java language features:

Threads. JFlow does not prevent threads from communicating covertly via the timing of asynchronous modifications to shared objects. This covert channel can be prevented by requiring only single-threaded programs.

Timing channels. JFlow cannot prevent threads from improperly gaining information by timing code with the system clock, except by removing access to the clock.

HashCode. In Java, the built-in implementation of the hashCode method, provided by the class Object, can be used to communicate information covertly. Therefore, in JFlow every class must implement its own hashCode.

Static variables. The order of static variable initialization can be used to communicate information covertly. In JFlow, this channel is blocked by ruling out static variables. However, static methods are legal. This restriction does not significantly hurt expressive power, since a program that uses static variables usually can be rewritten as a program in which the static variables are instance variables of an object. The order of initialization of these objects then becomes explicit and susceptible to analysis.

Finalizers. Finalizers are run in a separate thread from the main program, and therefore can be used to communicate covertly. Finalizers are not part of JFlow.

Resource exhaustion. An OutOfMemoryError can be used to communicate information covertly, by conditionally allocating objects until the heap is exhausted. JFlow treats this error as fatal, preventing it from communicating more than a single bit of information per program execution. Other exhaustion errors such as StackOverflowError are treated similarly.

Wall-clock timing channels. A JFlow program can change its run time based on private information it has observed. As an extreme example, it can enter an infinite loop. JFlow does not attempt to control these channels.

Unchecked exceptions. Java allows users to define exceptions that need not be declared in method headers (*unchecked exceptions*), although this practice is described as atypical [GJS96]. In JFlow, there are no unchecked exceptions, since they could serve as covert channels.

Type discrimination on parameters. JFlow supports the run-time cast and instanceof operators of standard Java, but they may only be invoked using classes that lack parameters. The reason for this restriction is that information about the parameters is not available at run time. These operators could be permitted if the parameters were statically known to be matched, but this is not currently supported.

Backward compatibility. JFlow is not backward compatible with Java, since existing Java libraries are not flow-checked and do not provide flow annotations. However, in many cases, a Java library can be wrapped in a JFlow library that provides reasonable annotations.

3 Static checking and translation

This section covers the static checking that the JFlow compiler performs as it translates code, and the translation process itself.

3.1 Exceptions

An important limitation of earlier attempts to create languages for static flow checking has been the absence of usable exceptions. For example, in Denning's original work on static flow checking, exceptions terminated the program [DD77] because any other treatment of exceptions seemingly leaked information. Subsequent work has avoided exceptions entirely.

It might seem unnecessary to treat exceptions directly, since in many languages, a function that generates exceptions can be desugared into a function that returns a *discriminated union* or *oneof*. However, there are problems with this approach. The obvious way to handle oneofs causes all exceptions to carry the same label — an unacceptable loss of precision. Also, Java exceptions are actually *objects*, and the try. . . catch statement functions like a *typecase*. This model cannot be translated directly into a oneof.

Nevertheless, it is useful to consider how oneof types *might* be handled in JFlow. The obvious way to treat oneof types is by analogy with record types. Each arm of the oneof has a distinct label associated with it. In addition, there is an added integer field tag that indicates which of the arms of the oneof is active. The problem with this model is that every assignment to the oneof will require that $\{\mathsf{tag}\} \sqsubseteq \underline{\mathsf{pc}}$, and every attempt to use the oneof will implicitly read $\{\mathsf{tag}\}$. As a result, every arm of the oneof will effectively carry the same label. For modeling exceptions, this is unacceptable.

For each expression or statement, the static checker determines its *path labels*, which are the labels for the information transmitted by various possible termination paths: normal termination, termination through exceptions, termination through a return statement, and so on. This fine-grained analysis avoids the unnecessary restrictiveness that would be produced by desugaring exceptions. Each exception that can be raised by evaluating a statement or expression has a possibly distinct label that is transferred to the $\underline{\mathsf{pc}}$ of catch statements that might intercept it. Even finer resolution is provided for normal termination and for return termination; for example, the label of the value of an expression may differ from the label associated with normal termination. Finally, termination of a statement by a break or continue statement is also tracked without confusing distinct break or continue targets.

The path labels for a statement or expression are represented as a map from symbols to labels. Each mapping represents a termination path that the statement or expression might take, and the label of the mapping indicates what information may be transmitted if this path is known to be the actual termination path. The domain of the map includes several different kinds of entities:

- The symbol $\underline{\mathsf{n}}$, which represents normal termination.
- The symbol $\underline{\mathsf{r}}$, which represents termination through a return statement.
- Classes that inherit from Throwable. A mapping from a class represents termination by an exception.
- The symbols $\underline{\mathsf{nv}}$ and $\underline{\mathsf{rv}}$ represent the labels of the normal value of an expression and the return value of a statement, respectively. They do not represent paths themselves, but it is convenient to include them as part of the map. Their labels are always at least as restrictive as the labels of the corresponding paths.
- A tuple of the form $\langle \mathsf{goto}\ \mathcal{L} \rangle$ represents termination by executing a named break or continue statement that jumps to the target $\mathcal{L}$. A break or continue statement that does not name a target is represented by the tuple $\langle \mathsf{goto}\ \epsilon \rangle$. These tuples are always mapped to the label $\top$ since the static checking rules do not use the actual label.

Path labels are denoted by the letter X in this paper, and members of the domain of X (paths) are denoted by s. The expression $X[s]$ denotes the label that X maps s to, and the expression $X[s := L]$ denotes a new map that is exactly like X except that s is bound to L. Path labels may also map a symbol s to the pseudo-label $\underline{\emptyset}$, indicating that the statement cannot terminate through the path s. The label $\underline{\emptyset}$ acts as the bottom of the label lattice; $\underline{\emptyset} \sqcup L = L$ for all labels L, including the label $\{\}$. The special path labels $X_\emptyset$ map all paths to $\underline{\emptyset}$, corresponding to an expression that does not terminate.

3.2 Type checking vs. label checking

The JFlow compiler performs two kinds of static checking as it compiles a program: type checking and label checking. These two aspects of checking cannot be entirely disentangled, since labels are type constructors and appear in the rules for subtyping. However, the checks needed to show that a

$$\frac{\begin{array}{c}A^g[C] = \langle \mathsf{class}\ C\left[[..\mathcal{P}_i..]\right] \ldots \{\ldots\}\rangle \\ (A \vdash \mathcal{Q}_i \approx \mathcal{Q}'_i) \vee (\mathcal{P}_i = \langle \mathsf{covariant\ label}\ id\rangle \wedge A \vdash \mathcal{Q}_i \sqsubseteq \mathcal{Q}'_i)\end{array}}{A \vdash_T C[..\mathcal{Q}_i..] \leq C[..\mathcal{Q}'_i..]}$$

$$\frac{\begin{array}{c}A^g[C] = \langle \mathsf{class}\ C\left[[..\mathcal{P}_i..]\right]\ \mathsf{extends}\ t_s \ldots \{\ldots\}\rangle \\ T_s = \textit{interp-T}(t_s, \textit{class-env}(C[..\mathcal{Q}_i..])) \\ A \vdash_T T_s \leq C'[..\mathcal{Q}'_i..]\end{array}}{A \vdash_T C[..\mathcal{Q}_i..] \leq C'[..\mathcal{Q}'_i..]}$$

Figure 8: Subtype rules

statement or expression is safe largely can be classified as either type or label checks. This paper focuses on the rules for checking labels, since the type checks are almost exactly the same as in Java.

There are several kinds of judgements made during static checking. The judgment $A \vdash_T E : T$ means that E has type T in environment A. The judgment $A \vdash E : X$ is the information-flow counterpart: it means that E has path labels X in environment A. The symbol $\vdash_T$ is used to denote inferences in the type domain. The environment A maps identifiers (*e.g.*, class names, parameter names, variable names) to various kinds of entities. As with path labels, the notation $A[s]$ is the binding of symbol s in A. The notation $A[s := B]$ is a new environment with s rebound to B. In the rules given here, it is assumed that the declarations of all classes are found in the global environment, A^g.

A few more comments on notation will be helpful at this point. The use of large brackets indicates an optional syntactic element. The letter T represents a type, and t represents a type expression. The letter C represents the name of a class. The letter L represents a label, and l represents a label expression. τ represents an labeled type expression; that is, a pair containing a type expression and an optional label expression. The function *interp-T*(t, A) converts type expressions to types, and the function *interp-L*(l, A) converts label expressions to labels. The letter v represents a variable name. The letter $\mathcal{P}$ represents a formal parameter of a class, and the letter $\mathcal{Q}$ represents an actual parameter used in an instantiation of a parameterized class.

3.3 Subtype rules

There are some interesting interactions between type and label checking. Consider the judgment $A \vdash_T S \leq T$, meaning "S is a subtype of T". This judgement must be made in JFlow, as in all languages with subtyping. Here, S and T are ordinary unlabeled types. The subtype rule, shown in Figure 8, is as in Java, except that it must take account of class parameters. If S or T is an instantiation of a parameterized class, subtyping is invariant in the parameters except when a label parameter is declared to be covariant. This subtyping rule is the first one shown in Figure 8. The function *class-env*, used in the figure, generates an extension of the global environment in which the formal parameters of a class (if any) are bound to the actual parameters: $A^g[..\textit{param-id}(\mathcal{P}_i) := \mathcal{Q}_i..]$

Using this rule, Vector[L] (from Figure 4) would be a subtype of AbstractList[L'] only if $L \approx L'$. Java arrays (written as $T\{L\}[\,]$) are treated internally as a special type with two parameters, T and L. As in Java, they are covariant in T, but like most JFlow classes, invariant in L. User-defined types may not be parameterized on other types.

If S and T are not instantiations of the same class, it is necessary to walk up the type hierarchy from S to T, rewriting parameters, as shown in the second rule in Figure 8. Together, the two rules inductively prove the appropriate subtype relationships.

$$\frac{\textit{true}}{A \vdash \textit{literal} : X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}], \underline{\mathsf{nv}} := A[\underline{\mathsf{pc}}]]}$$

$$\frac{\textit{true}}{A \vdash \, ; : X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}]]}$$

$$\frac{\begin{array}{c}A[v] = \langle \mathsf{var}\ \left[\mathsf{final}\right]\ T\{L\}\ uid\rangle \\ X = X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}], \underline{\mathsf{nv}} := L \sqcup A[\underline{\mathsf{pc}}]]\end{array}}{A \vdash v : X}$$

$$\frac{\begin{array}{c}A \vdash E : X \\ A[v] = \langle \mathsf{var}\ T\{L\}\ uid\rangle \\ A \vdash X[\underline{\mathsf{nv}}] \sqsubseteq L\end{array}}{A \vdash v = E : X}$$

$$\frac{\begin{array}{c}A \vdash S_1 : X_1 \\ \textit{extend}(A, S_1)[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash S_2 : X_2 \\ X = X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2\end{array}}{A \vdash S_1; S_2 : X}$$

$$(X = X_1 \oplus X_2) \quad \equiv \quad \forall s\ (X[s] = X_1[s] \sqcup X_2[s])$$

Figure 9: Some simple label-checking rules

3.4 Label-checking rules

Let us consider a few examples of static checking rules. Space restrictions prevent presentation of all the rules, but a complete description of the static checking rules of JFlow is available [Mye99].

Consider Figure 9, which contains some of the most basic rules for static checking. The first rule shows that a literal expression always terminates normally and that its value is labeled with the current pc, as described earlier. The second rule shows that an empty statement always terminates normally, with the same pc as at its start.

The third rule shows that the value of a variable is labeled with both the label of the variable and the current pc. Note that the environment maps a variable identifier to an entry

$$\frac{\begin{array}{c} A \vdash E_a : X_a \\ A[\underline{\mathsf{pc}} := X_a[\underline{\mathsf{n}}]] \vdash E_i : X_i \\ A[\underline{\mathsf{pc}} := X_i[\underline{\mathsf{n}}]] \vdash E_v : X_v \\ X_1 = exc(X_a \oplus X_i \oplus X_v, X_a[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \\ X_2 = exc(X_1, X_a[\underline{\mathsf{nv}}] \sqcup X_i[\underline{\mathsf{nv}}], \mathsf{OutOfBoundsException}) \\ X = exc(X_2, X_a[\underline{\mathsf{nv}}] \sqcup X_v[\underline{\mathsf{nv}}], \mathsf{ArrayStoreException}) \\ A \vdash_T E_a : T\{L_a\}[\,] \\ A \ \vdash \ X_v[\underline{\mathsf{nv}}] \sqcup X[\underline{\mathsf{n}}] \ \sqsubseteq \ L_a \end{array}}{A \vdash E_a[E_i] = E_v : X}$$

$$\frac{\begin{array}{c} A \vdash E : X_E \\ A[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S_2 : X_2 \\ X = X_E[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_1 \oplus X_2 \end{array}}{A \vdash \mathsf{if}\ (E)\ S_1\ \mathsf{else}\ S_2 : X}$$

$$\frac{\begin{array}{c} L = \textit{fresh-variable}() \\ A' = A[\underline{\mathsf{pc}} := L,\ \langle \mathsf{goto}\ \epsilon \rangle := L] \\ A' \vdash E : X_E \\ A'[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S : X_S \\ A \ \vdash \ X_S[\underline{\mathsf{n}}] \ \sqsubseteq \ L \\ X = (X_E \oplus X_S)[\langle \mathsf{goto}\ \epsilon \rangle := \underline{\emptyset}] \end{array}}{\begin{array}{c} A \vdash \mathsf{while}\ (E)\ S : X \\ A \vdash \mathsf{do}\ S\ \mathsf{while}\ (E) : X \end{array}}$$

$$\frac{A \ \vdash \ A[\underline{\mathsf{pc}}] \ \sqsubseteq \ A[\langle \mathsf{goto}\ \mathcal{L} \rangle]}{\begin{array}{c} A \vdash \mathsf{continue}\ \mathcal{L} : X_\emptyset[\langle \mathsf{goto}\ \mathcal{L} \rangle := \top] \\ A \vdash \mathsf{break}\ \mathcal{L} : X_\emptyset[\langle \mathsf{goto}\ \mathcal{L} \rangle := \top] \end{array}}$$

$$\frac{\begin{array}{c} A \vdash S : X' \\ s \in \{\underline{\mathsf{n}}, \underline{\mathsf{r}}\} \\ \forall (s' \mid s' \in \textit{paths} \ \wedge \ s' \neq s)\ X[s'] = \underline{\emptyset} \\ X = X'[s := A[\underline{\mathsf{pc}}]] \end{array}}{A \vdash S : X}$$

$\textit{paths}$ = all symbols except $\underline{\mathsf{nv}}$, $\underline{\mathsf{rv}}$

$exc(X, L, C) = X \oplus X_\emptyset[\underline{\mathsf{n}} := L,\ \underline{\mathsf{nv}} := L,\ C := L]$

Figure 10: More label-checking rules

of either the form $\langle \mathsf{var}\ T\{L\}\ \textit{uid} \rangle$ or $\langle \mathsf{var\ final}\ T\{L\}\ \textit{uid} \rangle$, where T is the variable's type, L is its label, and *uid* is a unique identifier distinguishing it from other variables of the same name.

The fourth rule covers assignment to a variable. Assignment is allowed if the variable's label is more restrictive than that of the value being assigned (which will include the current $\underline{\mathsf{pc}}$). Whether one label is more restrictive than other is inferred using the current environment, which contains information about the static principal hierarchy. The complete rule for checking this statement would have an additional antecedent $A \vdash_T E : T$, but such type-checking rules have been omitted in the interest of space.

The final rule in Figure 9 covers two statements S_1 and S_2 performed in sequence. The second statement is executed only if the first statement terminated normally, so the correct $\underline{\mathsf{pc}}$ for checking the second statement is the normal path label of the first statement ($X_1[\underline{\mathsf{n}}]$). The function *extend* extends the environment A to add any local variable declarations in the statement S_1. The path labels of the sequence must be at least as restrictive as path labels of both statements; this condition is captured by the operator $\oplus$, which merges two sets of path labels, joining all corresponding paths from both.

Figure 10 contains some more complex rules. The rule for array element assignment mirrors the order of evaluation of the expression. First, the array expression E_a is evaluated, yielding path labels X_a. If it completes normally, the index expression E_i is evaluated, yielding X_i. Then, the assigned value is evaluated. Java checks for three possible exceptions before performing the assignment. The function *exc*, defined at the bottom, is used to simplify these conditions. This function creates a set of path labels that are just like X except that they include an additional path, the exception C, with the path label L. Since observation of normal termination ($\underline{\mathsf{n}}$) or the value on normal termination ($\underline{\mathsf{nv}}$) is conditional on the exception *not* being thrown, *exc* joins the label L to these two mappings as well. Finally, avoiding leaks requires that the label on the array elements (L_a) is at least as restrictive as the label on the information being stored ($X_v[\underline{\mathsf{nv}}]$).

The next rule shows how to check an if statement. First, the path labels X_E of the expression are determined. Since execution of S_1 or S_2 is conditional on E, the $\underline{\mathsf{pc}}$ for these statements must include the value label of E, $X_E[\underline{\mathsf{nv}}]$. Finally, the statement as a whole can terminate through any of the paths that terminate E, S_1, or S_2— except normal termination of E, since this would cause one of S_1 or S_2 to be executed. If the statement has no else clause, the statement S_2 is considered to be an empty statement, and the second rule in Figure 9 is applied.

The next rule, for the while statement, is more subtle because of the presence of a loop. This rule introduces a label variable L to represent the information carried by the continuation of the loop through various paths. L represents an unknown label that will be solved for later. It is essentially a loop invariant for information flow. L may carry information from exceptional termination of E or S, or from break or continue statements that occur inside the loop. An entry is added to the environment for the tuple $\langle \mathsf{goto}\ \epsilon \rangle$ to capture information flows from any break or continue statements within the loop. The rules for checking break and continue, shown below the rule for while, use these environment entries to apply the proper restriction on information flow.

Assuming that L is the entering $\underline{\mathsf{pc}}$ label, $X_S[\underline{\mathsf{n}}]$ is the final $\underline{\mathsf{pc}}$ label. The final condition requires that L' may be at

$$\frac{\begin{array}{c} A \vdash_T E : \mathsf{class}\ C\ \{\ldots\} \\ A \vdash E : X_E \\ X = exc(X_E, X_E[\underline{\mathsf{nv}}], C)[\underline{\mathsf{n}} := \emptyset] \end{array}}{A \vdash \mathsf{throw}\ E : X}$$

$$\frac{\begin{array}{c} A \vdash S : X_S \\ \underline{\mathsf{pc}}_i = exc\text{-}label(X_S, C_i) \\ A[\underline{\mathsf{pc}} := \underline{\mathsf{pc}}_i,\ v_i := \langle \mathsf{var\ final}\ C\{\underline{\mathsf{pc}}_i\}\, fresh\text{-}uid() \rangle] \vdash S_i : X_i \\ X = (\bigoplus_i X_i) \oplus uncaught(X_S, (.., C_i, ..)) \end{array}}{A \vdash \mathsf{try}\ \{S\}\ ..\mathsf{catch}(C_i\ v_i)\ \{S_i\}.. : X}$$

$$\frac{\begin{array}{c} A \vdash S_1 : X_1 \qquad A \vdash S_2 : X_2 \\ X = X_1[\underline{\mathsf{n}} := \emptyset] \oplus X_2 \end{array}}{A \vdash \mathsf{try}\ \{S_1\}\ \mathsf{finally}\ \{S_2\} : X}$$

$$exc\text{-}label(X, C) = \bigsqcup_{C' : (C' \leq C \vee C \leq C')} X[C']$$

$$(X' = uncaught(X, (.., C_i, ..))) \ \equiv$$
$$X'[s] = (\mathsf{if}\ (\exists i\ (s \leq C_i))\ \mathsf{then}\ \emptyset\ \mathsf{else}\ X[s])$$

Figure 11: Exception-handling rules

most as restrictive as L, which is what establishes the loop invariant.

The last rule in Figure 10 applies to any statement, and is important for relaxing restrictive path labels. It is intuitive: if a statement (or a sequence of statements) can only terminate normally, the $\underline{\mathsf{pc}}$ at the end is the same as the $\underline{\mathsf{pc}}$ at the beginning. The same is true if the statement can only terminate with a return statement. This rule is called the *single-path rule*. It would not be safe for this rule to apply to exception paths. To see why, suppose that a set of path labels formally contains only a single exception path C. However, that path might include multiple paths consisting of exceptions that are subclasses of C. These multiple paths can be discriminated using a try. . . catch statement. The unusual Java exception model prevents the single-path rule from being applied to exception paths.

However, Java is a good language to extend for static flow analysis in other ways because it fully specifies evaluation order. This property makes static checking of information flow simpler, because the rules tend to encode all possible evaluation orders. If there were non-determinism in evaluation order, it could be encoded by adding label variables in a manner similar to the rule for the while statement.

3.5 Throwing and catching exceptions

Exceptions can be thrown and caught safely in JFlow using the usual Java constructs. Figure 11 shows the rule for the throw statement, a try. . . catch statement that lacks a finally clause, and a try. . . finally statement. (A try statement with both catch clauses and a finally clause can be desugared into

```
y = true;
try {
   if (x) throw new E();
   y = false;
}
catch (E e) { }
```

Figure 12: Implicit flow using throw

a try. . . catch inside a try. . . finally.) The rule for throw is straightforward.

The idea behind the try. . . catch rule is that each catch clause is executed with a $\underline{\mathsf{pc}}$ that includes all the paths that might cause the clause to be executed: all the paths that are exceptions where the exception class is *either* a subclass or a superclass of the class named in the catch clause. The function *exc-label* joins the labels of these paths. The path labels of the whole statement merge all the path labels of the various catch clauses, plus the paths from X_S that might not be caught by some catch clause, which include the normal termination path of X_S if any.

The try. . . finally rule is very similar to the rule for sequencing two statements. One difference is that the statement S_2 is checked with exactly the same initial $\underline{\mathsf{pc}}$ that S_1 is, since S_2 is executed no matter how S_1 terminates.

To see how these exception rules work, consider the code in Figure 12. In this example, x and y are boolean variables. This code transfers the information in x to y by using an implicit flow resulting from an exception. In fact, the code is equivalent to the assignment y = x. Using the rule of Figure 11, the path labels of the throw statement are $\{\mathsf{E} \to \{\mathsf{x}\}\}$, so the path labels of the if statement are $X = \{\mathsf{E} \to \{\mathsf{x}\}, \underline{\mathsf{n}} \to \{\mathsf{x}\}\}$. The assignment y = false is checked with $\underline{\mathsf{pc}} = X[\underline{\mathsf{n}}] = \{\mathsf{x}\}$, so the code is allowed only if $\{\mathsf{x}\} \sqsubseteq \{\mathsf{y}\}$. This restriction is correct since it is exactly what the equivalent assignment statement would have required. Finally, applying both the try-catch rule here and the single-path rule from Figure 10, the value of $\underline{\mathsf{pc}}$ after the code fragment is seen to be the same as at its start. Throwing and catching an exception does not necessarily taint subsequent computation.

3.6 Run-time label checking

An interesting aspect of checking JFlow is checking the switch label statement, which inspects a dynamic label at run time. The inference rule for checking this statement is given in Figure 13. Intuitively, the switch label statement tests the equation $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i$ for every arm until it finds one for which the equation holds, and executes it. However, this test cannot be evaluated either statically or at run time. Therefore, the test is split into two stronger conditions: one that can be tested statically, and one that can be tested dynamically. This rule naturally contains the static part of the test.

Let L_{RT} be the join of all possible run-time-representable components (i.e., components that do not mention formal label or principal parameters). The static test is that $X_E[\underline{\mathsf{nv}}] \sqcup L_{RT} \sqsubseteq L_i \sqcup L_{RT}$ (equiva-

$$\frac{\begin{array}{c} A \vdash E : X_E \\ L_i = \textit{interp-L}(l_i, A) \\ A \;\vdash\; X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i \sqcup L_{RT} \\ A \vdash_T E : T \\ A \vdash_T T \le \textit{interp-T}(t_i, A) \\ \underline{\mathsf{pc}}_0 = X_E[\underline{\mathsf{n}}] \\ \underline{\mathsf{pc}}_i = \underline{\mathsf{pc}}_{i-1} \sqcup \textit{label}(X_E[\underline{\mathsf{nv}}] \sqcup L_i) \\ A[\underline{\mathsf{pc}} := \underline{\mathsf{pc}}_i, v_i := \langle \mathsf{var\ final}\ T_i\{L_i\}\ \textit{fresh-uid}()\rangle] \vdash S_i : X_i \\ X = X_E \oplus (\bigoplus_i X_i) \end{array}}{A \vdash \mathsf{switch\ label}(E)\{..\mathsf{case}\ (t_i\{l_i\}\ v_i)\ S_i..\} : X}$$

Figure 13: Inference rule for switch label

lently, $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i \sqcup L_{RT}$); the dynamic test is that $X_E[\underline{\mathsf{nv}}] \sqcap L_{RT} \sqsubseteq L_i \sqcap L_{RT}$. Together, these two tests imply the full condition $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i$.

The test itself may be used as an information channel, so after the check, the $\underline{\mathsf{pc}}$ must include the labels *of* $X_E[\underline{\mathsf{nv}}]$ and every L_i up to this point. This rule uses the *label* function to achieve this. When applied to a label L, it generates a new label that joins together the labels of all variables that are mentioned in L. However, the presence of *label* in constraint equations does not change the process of solving label constraints in any fundamental way.

3.7 Checking method calls

Let us now look at some of the static checking associated with objects. Static checking in object-oriented languages is often complex, and the various features of JFlow only add to the complexity. This section shows how, despite this complexity, method calls and constructor calls (via the operator new) are checked statically.

The rules for checking method and constructor calls are shown in Figures 14 and 15. Figure 14 defines some generic checking that is performed for all varieties of calls, and Figure 15 defines the rules for checking ordinary method calls, static method calls, and constructor calls.

To avoid repetition, the checking of both static and non-static method calls, and also constructor calls, is expressed in terms of the predicate *call*, which is defined in Figure 14. This predicate is in turn expressed in terms of two predicates: *call-begin* and *call-end*.

The predicate *call-begin* checks the argument expressions and establishes that the constraints for calling the method are satisfied. In this rule, the functions *type-part* and *label-part* interpret the type and label parts of a labeled type τ. The rule determines the begin label L_I, the default return label L_{RV}^{def}, and the argument environment A^a, which binds all the method arguments to appropriately labeled types. Invoking a method requires evaluation of the arguments E_j, producing corresponding path labels X_j. The argument labels are bound in A^a to labels L_j, so the line ($X_j[\underline{\mathsf{nv}}] \sqsubseteq L_j$) ensures that the actual arguments can be assigned to the formals. The begin-label L_I is also required to be more restrictive than the $\underline{\mathsf{pc}}$

$$\frac{\begin{array}{c} A \vdash \textit{call-begin}(C[\mathcal{Q}_i], (.., E_j, ..), \mathcal{S}, A^a, L_I, L_{RV}^{def}) \\ A \vdash \textit{call-end}(C[\mathcal{Q}_i], \mathcal{S}, A^a, L_I, L_{RV}^{def}) : X \end{array}}{A \vdash \textit{call}(C[\mathcal{Q}_i], (.., E_j, ..), \mathcal{S}) : X}$$

$$\frac{\begin{array}{c} \mathcal{S} = \langle \big[\mathsf{static}\big]\ \tau_r\ m\big[\{I\}\big](..\tau_j\ a_j..)\big[{:}\{R\}\big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ X_0 = X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}]] \\ A[\underline{\mathsf{pc}} := X_{j-1}[\underline{\mathsf{n}}]] \vdash E_j : X_j \\ L_j = \textit{fresh-variable}() \\ \textit{uid}_j = \textit{fresh-uid}() \\ A^c = \textit{class-env}(C[\mathcal{Q}_i]) \\ A^a = A^c[..a_j := \langle \mathsf{var\ final}\ \textit{type-part}(\tau_j, A^c)\{L_j\}\ \textit{uid}_j\rangle..] \\ L_I = (\mathsf{if}\ \big[\{I\}\big]\ \mathsf{then}\ \textit{interp-L}(I, A^a)\ \mathsf{else}\ X_{\max(j)}[\underline{\mathsf{n}}]) \\ A \vdash L_j \approx (\mathsf{if}\ \textit{labeled}(\tau_j)\ \mathsf{then}\ \textit{label-part}(\tau_j, A^a) \sqcup L_I\ \mathsf{else}\ L_j) \\ A \;\vdash\; X_j[\underline{\mathsf{nv}}] \sqsubseteq L_j \\ A \;\vdash\; X_{\max(j)}[\underline{\mathsf{n}}] \sqsubseteq L_I \\ L_{RV}^{def} = (\mathsf{if}\ (\tau_r = \mathsf{void})\ \mathsf{then}\ \{\}\ \mathsf{else}\ \bigsqcup_j X_j[\underline{\mathsf{nv}}]) \\ \textit{satisfies-constraints}(A, A^a, A[..a_j := E_j..], (..\mathcal{K}_l..)) \end{array}}{A \vdash \textit{call-begin}(C[\mathcal{Q}_i], (..E_j..), \mathcal{S}, A^a, L_I, L_{RV}^{def})}$$

$$\frac{\begin{array}{l} \mathsf{let}\ \textit{interp}(p) = \textit{interp-P-call}(p, A, A^a, A^m)\ \mathsf{in} \\ \quad \mathsf{case}\ \mathcal{K}_i\ \mathsf{of} \\ \qquad \mathsf{authority}(\ldots) : \textit{true} \\ \qquad \mathsf{caller}(..p_j..) : \forall(p_j)\exists(p' \in A[\underline{\mathsf{auth}}])\ A \vdash p' \succeq \textit{interp}(p_j) \\ \qquad \mathsf{actsFor}(p_1, p_2) : A \vdash \textit{interp}(p_1) \succeq \textit{interp}(p_2) \\ \quad \mathsf{end} \\ \mathsf{end} \end{array}}{\textit{satisfies-constraints}(A, A^a, A^m, (..\mathcal{K}_i..))}$$

$$\frac{\begin{array}{c} \mathcal{S} = \langle \big[\mathsf{static}\big]\ \tau_r\ m\big[\{I\}\big](..\tau_j\ a_j..)\big[{:}\{R\}\big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ L_R = L_I \sqcup (\mathsf{if}\ \big[:\{R\}\big]\ \mathsf{then}\ \textit{interp-L}(R, A^a)\ \mathsf{else}\ \{\}) \\ L_{RV} = L_R \sqcup (\mathsf{if}\ \textit{labeled}(\tau_r)\ \mathsf{then}\ \textit{label-part}(\tau_r, A^a)\ \mathsf{else}\ L_{RV}^{def}) \\ C_k = \textit{type-part}(\tau_k, \textit{class-env}(C[\mathcal{Q}_i])) \\ X' = (\bigoplus_j X_j)[\underline{\mathsf{n}} := L_R,\ \underline{\mathsf{nv}} := L_{RV}] \\ X = X' \oplus X_\emptyset[..C_k := \textit{label-part}(\tau_k, A^a) \sqcup L_R..] \end{array}}{A \vdash \textit{call-end}(C[\mathcal{Q}_i], \mathcal{S}, A^a, L_I, L_{RV}^{def}) : X}$$

Figure 14: Checking calls

after evaluating all of the arguments, which is $X_{\max(j)}[\underline{\mathsf{n}}]$.

The call site must satisfy all the constraints imposed by the method, which is checked by the predicate *satisfies-constraints*. The rule for this predicate, also in Figure 14, uses the function *interp-P-call*, which maps identifiers used in the method constraints to the corresponding principals. To perform this mapping, the function needs environments corresponding to the calling code (A), the called code (A^a), and a special environment that binds the actual arguments (A^m). The environment entry $A[\underline{\mathsf{auth}}]$ contains the set of principals that the code is known statically to act for. The judgement $A \vdash p_1 \succeq p_2$ means that p_1 is known statically to act for p_2. (The static principal hierarchy is also

$$\frac{\begin{array}{c} A \vdash_T E_s : C[..\mathcal{Q}_i..] \\ A \vdash_T E_j : T_j \\ \mathit{signature}(C[..\mathcal{Q}_i..], m(..T_j..), \mathcal{S}) \\ A \vdash E_s : X_s \\ A[\underline{\mathsf{pc}} := X_s[\underline{\mathsf{nv}}]] \vdash \mathit{call}(C[..\mathcal{Q}_i..], (..E_j..), \mathcal{S}) : X \end{array}}{A \vdash E_s \,.\, m(..E_j..) : X}$$

$$\frac{\begin{array}{c} T = \mathit{interp\text{-}T}(t, A) \\ A \vdash_T E_j : T_j \\ \mathit{signature}(T, m(..T_j..), \mathcal{S}) \\ A \vdash \mathit{call}(T, (..E_j..), \mathcal{S}) : X \end{array}}{A \vdash t \,.\, m(..E_j..) : X}$$

$$\frac{\begin{array}{c} T = C[..\mathcal{Q}_i..] = \mathit{interp\text{-}T}(t, A) \\ A^g[C] = \langle \mathsf{class}\ C \left[[..\mathcal{P}_i..]\right] \ldots \left[\mathsf{authority}(..p_l..)\right] \ldots \rangle \\ A \vdash_T E_j : T_j \\ \mathit{signature}(T, C(..T_j..), \mathcal{S}) \\ \mathcal{S} = \langle C\left[\{I\}\right](..\tau_j\ a_j..)\left[:\{R\}\right]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ \mathcal{S}' = \langle \mathsf{static}\ T\{\}\ \mathsf{m}\left[\{I\}\right](..\tau_j\ a_j..)\left[:\{R\}\right]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ A \vdash \mathit{call}(T, (..E_j..), \mathcal{S}') : X \\ \forall(\text{parameters } p_l)\ \exists(p \in A[\underline{\mathsf{auth}}])\ A \vdash p \succeq \mathit{interp\text{-}P}(p_l, \mathit{class\text{-}env}(T)) \end{array}}{A \vdash \mathsf{new}\ t(..E_j..) : X}$$

Figure 15: Rules for specific calls

placed in the environment.)

Finally, the predicate *call-end* produces the path labels X of the method call by assuming that the method returns the path labels that its signature claims. The label L_{RV}^{def} is used as the label of the return value in the case where the return type, τ_r, is not labeled. It joins together the labels of all of the arguments, since typically the return value of a function depends on all of its arguments.

The rules for the various kinds of method calls are built on top of this framework, as shown in Figure 15. In these rules, the function *signature* obtains the signature of the named method from the class. The rule for constructors contains two subtle steps: first, constructors are checked as though they were static methods with a similar signature. Second, a call to a constructor requires that the caller possess the authority of all principals in the authority of the class that are parameters. The caller does not need to have the authority of external principals named in the authority clause.

3.8 Constraint solving

As the rules for static checking are applied, they generate a constraint system of label variables for each method [ML97]. For example, the assignment rule of Figure 9 generates a constraint $X[\underline{\mathsf{nv}}] \sqsubseteq L$. All of the constraints are of the form $A_1 \sqcup \ldots \sqcup A_m \sqsubseteq B_1 \sqcup \ldots \sqcup B_n$. These constraints can be split into individual constraints $A_i \sqsubseteq B_1 \sqcup \ldots \sqcup B_n$ because of the lattice properties of labels. The individual terms in the constraints may be policies, label variables, label parameters, dynamic labels, or expressions $label(L)$ for some label L.

The constraints can be efficiently solved, using a modification to a lattice constraint-solving algorithm [RM96] that applies an ordering optimization [HDT87] shown to produce the best of several commonly-used iterative dataflow algorithms [KW94]. The approach is to initialize all variables in the constraints with the most restrictive label ($\top$) and iteratively relax their labels until a satisfying assignment or a contradiction is found. The *label* does not create problems because it is monotonic. The relaxation steps are ordered by topologically sorting the constraints and looping on strongly-connected components. The number of iterations required is $O(nh)$ where h is the maximum height of the lattice structure [RM96], and also $O(nd)$ where d is the maximum back edges in depth-first traversal of the constraint dependency graph [HDT87]. Both h and d seem likely to be bounded for reasonable programs. The observed behavior of the JFlow compiler is that constraint solving is a negligible part of run time.

$$\begin{array}{l} \mathbf{T}[\![\mathsf{actsFor}(p_1, p_2)\ S]\!]\ = \\ \quad \mathsf{if}\ (\mathsf{\_dynamic\_PH.actsFor}(\mathbf{T}[\![p_1]\!], \mathbf{T}[\![p_2]\!]))\ \ \mathbf{T}[\![S]\!] \\ \\ \mathbf{T}[\![\mathsf{switch\ label}(E)\ \{\ ..\mathsf{case}(\ t_i\{l_i\})\ S_i..\ \mathsf{else}\ S_e\ \}]\!]\ = \\ \quad T\ v\ =\ \mathbf{T}[\![E]\!]; \\ \quad \mathsf{if}\ (\mathbf{T}[\![X_E[\underline{\mathsf{nv}}] \sqcap L_{RT}]\!].\mathsf{relabelsTo}(\mathbf{T}[\![L_1 \sqcap L_{RT}]\!]))\ \{ \\ \quad\quad \mathbf{T}[\![S_1]\!] \\ \quad \}\ \mathsf{else}\ \ldots \\ \quad \mathsf{if}\ (\mathbf{T}[\![X_E[\underline{\mathsf{nv}}] \sqcap L_{RT}]\!].\mathsf{relabelsTo}(\mathbf{T}[\![L_i \sqcap L_{RT}]\!])\ \{ \\ \quad\quad \mathbf{T}[\![S_i]\!] \\ \quad \}\ \ldots\ \mathsf{else}\ \{\ \mathbf{T}[\![S_e]\!]\} \end{array}$$

Figure 16: Interesting JFlow translations

3.9 Translation

The JFlow compiler is a static checker and source-to-source translator. Its output is a standard Java program. Most of the annotations in JFlow have no run-time representation; translation erases them, leaving a Java program. For example, all type labels are erased to produce the corresponding unlabeled Java type. Class parameters are erased. The declassify expression and statement are replaced by their contained expression or statement.

Uses of the built-in types label and principal are translated to the Java types jflow.lang.Label and jflow.lang.Principal, respectively. Variables declared to have these types remain in the translated program. Only two constructs translate to interesting code: the actsFor and switch label statement, which dynamically test principals and labels, respectively. The translated code for each is simple and efficient, as shown in Figure 16. Note that the translation rule for switch label uses definitions from Figure 13. As discussed earlier, the run-time check is $X_E[\underline{\mathsf{nv}}] \sqcap L_{RT} \sqsubseteq L_1 \sqcap L_{RT}$, which in effect is a test on labels that are completely representable at run time.

The translated code uses the methods relabelsTo and actsFor of the classes jflow.lang.Label and jflow.lang.Principal, respectively. These methods are accelerated by a hash-table lookup into a cache of results, so the translated code is fast.

4 Related work

There has been much work on information flow control and on the static analysis of security guarantees. The lattice model of information flow comes from the early work of Bell and LaPadula[BL75] and Denning [Den76]. Most subsequent information control models use dynamic labels rather than static labels and therefore cannot be checked statically. The decentralized label model has similarities to the ORAC model [MMN90]: both models provide some approximation of the "originator-controlled release" labeling used by the U.S. DoD/Intelligence community, although the ORAC model is dynamically checked.

Static analysis of security guarantees also has a long history. It has been applied to information flow [DD77, AR80], to access control [JL78, RSC92], and to integrated models [Sto81]. There has recently been more interest in provably-secure programming languages, treating information flow checks in the domain of type checking. Some of this work has focused on formally characterizing existing information flow and integrity models [PO95, VSI96, Vol97]. Smith and Volpano have recently examined the difficulty of statically checking information flow in a multithreaded functional language [SV98], which JFlow does not address. However, the rules they define prevent the run time of a program from depending in any way on non-public data. Abadi [Aba97] has examined the problem of achieving secrecy in security protocols, also using typing rules, and has shown that encryption can be treated as a form of safe declassification through a primitive encryption operator.

Heintze and Riecke [HR98] have shown that information-flow-like labels can be applied to a simple language with reference types (the SLam calculus). They show how to statically check an integrated model of access control, information flow control, and integrity. Their labels include two components: one which enforces conventional access control, and one that enforces information flow control. Their model has the limitation that it is entirely static: it has no run-time access control, no declassification, and no run-time flow checking. It also does not provide label polymorphism or objects. Heintze and Riecke prove some useful soundness theorems for their model. This step would be desirable for JFlow, but important features like objects, inheritance and dependent types make formal proofs of correctness difficult at this point.

An earlier paper [ML97] introduced the decentralized label model and suggested a simple language for writing information-flow safe programs. JFlow extends the ideas of that simple language in several important ways and shows how to apply them to a real programming language, Java. JFlow adds support for objects, fine-grained exceptions, explicit parameterization, and the full decentralized label model [ML98]. Static checking is described by formal inference rules that specify much of the JFlow compiler. The performance of the label inference algorithm (the constraint solver) also has been improved.

5 Conclusions

Privacy is becoming an increasingly important security concern, and static program checking appears to be the only technique that can provide this security with reasonable efficiency. This paper has described the new language JFlow, an extension of the Java language that permits static checking of flow annotations. To our knowledge, it is the first practical programming language that allows this checking. The goal of this work has been to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner.

JFlow addresses many of the limitations of previous work in this area. It supports many language features that have not been previously integrated with static flow checking, including mutable objects (which subsume function values), subclassing, dynamic type tests, dynamic access control, and exceptions.

Avoiding unnecessary restrictiveness while supporting a complex language has required the addition of sophisticated language mechanisms: implicit and explicit polymorphism, so code can be written in a generic fashion; dependent types, to allow dynamic label checking when static label checking would be too restrictive; static reasoning about access control; statically-checked declassification.

This list of mechanisms suggests that one reason why static flow checking has not been widely accepted as a security technique, despite having been invented over two decades ago, is that programming language techniques and type theory were not then sophisticated enough to support a sound, practical programming model. By adapting these techniques, JFlow makes a useful step towards usable static flow checking.

Acknowledgments

I would like to thank several people who read this paper and gave useful suggestions, including Sameer Ajmani, Ulana Legedza, and the reviewers. Kavita Bala, Miguel Castro, and Stephen Garland were particularly helpful in reviewing the static checking rules. I would also like to thank Nick Mathewson for his work on the PolyJ compiler, from which I was able to steal much code, and Barbara Liskov for her support on this project.

A Grammar extensions

JFlow contains several extensions to the standard Java grammar, in order to allow information flow annotations to be added. The following productions must be added to or modified from the standard Java Language Specification [GJS96]. As with the Java grammar, some modifications to this grammar are required if the grammar is to be input to a parser generator. These grammar modifications (and, in fact, the code of the JFlow compiler itself) were to a considerable extent derived from those of PolyJ, an extension to Java that supports parametric polymorphism [MBL97, LMM98].

A.1 Label expressions

LabelExpr:
  { *Components*$_{opt}$ }

Components:
  Component
  Components ; *Component*

Component:
  Principal : *Principals*$_{opt}$
  this
  Identifier
  * *Identifier*

Principals:
  Principal
  Principals , *Principal*

Principal: Name

A.2 Labeled types

Types are extended to permit labels. The new primitive types label and principal are also added.

LabeledType:
  PrimitiveType LabelExpr$_{opt}$
  ArrayType LabelExpr$_{opt}$
  Name LabelExpr$_{opt}$
  TypeOrIndex LabelExpr$_{opt}$

PrimitiveType:
  NumericType
  boolean
  label
  principal

The *TypeOrIndex* production represents either an instantiation or an array index expression. Since both use brackets, the ambiguity is resolved after parsing.

TypeOrIndex:
  Name [*ParamOrExprList*]

ArrayIndex:
  TypeOrIndex
  PrimaryNoNewArray [*Expression*]

ClassOrInterfaceType:
  Name
  TypeOrIndex

ParamOrExprList:
  ParamOrExpr
  ParamOrExprList , *ParamOrExpr*

ParamOrExpr:
  Expression
  LabelExpr

ArrayType:
  LabeledType [ ]

ArrayCreationExpression:
  new *LabeledType DimExprs OptDims*

A.3 Class declarations

ClassDeclaration:
  Modifiers$_{opt}$ class *Identifier Params*$_{opt}$
    Super$_{opt}$ *Interfaces*$_{opt}$ *Authority*$_{opt}$ *ClassBody*

InterfaceDeclaration:
  Modifiers$_{opt}$ interface *Identifier Params*$_{opt}$
    ExtendsInterfaces$_{opt}$
    Interfaces$_{opt}$ *InterfaceBody*

Params:
  [*ParameterList*]

ParameterList:
  Parameter
  ParameterList , *Parameter*

Parameter:
  label *Identifier*
  covariant label *Identifier*
  principal *Identifier*

Authority:
  authority (*Principals*)

A.4 Method declarations

MethodHeader:
  Modifiers$_{opt}$ *LabeledType Identifier*
    BeginLabel$_{opt}$ (*FormalParameterList*$_{opt}$) *EndLabel*$_{opt}$
    Throws$_{opt}$ *WhereConstraints*$_{opt}$
  Modifiers$_{opt}$ void *Identifier*
    BeginLabel$_{opt}$ (*FormalParameterList*$_{opt}$) *EndLabel*$_{opt}$
    Throws$_{opt}$ *WhereConstraints*$_{opt}$

ConstructorDeclaration:
  Modifiers$_{opt}$ *Identifier*
    BeginLabel$_{opt}$ (*FormalParameterList*) *EndLabel*$_{opt}$
    Throws$_{opt}$ *WhereConstraints*$_{opt}$

FormalParameter:
  LabeledType Identifier OptDims

BeginLabel:
  LabelExpr

EndLabel:
  : *LabelExpr*

WhereConstraints:
  where *Constraints*

Constraints:
  Constraint
  Constraints , *Constraint*

Constraint:

Authority
caller (*Principals*)
actsFor (*Principal* , *Principal*)

To avoid ambiguity, the classes in a throws list must be placed in parentheses. Otherwise a label might be confused with the method body.

Throws:
throws (*ThrowList*)

A.5 New statements

Statement:
StatementWithoutTrailingSubstatement
. . . existing productions . . .
ForStatement
SwitchLabelStatement
ActsForStatement
DeclassifyStatement

SwitchLabelStatement:
switch label (*Expression*) { *LabelCases* }

LabelCases:
LabelCase
LabelCases LabelCase

LabelCase:
case (*Type LabelExpr Identifier*) *OptBlockStatements*
case *LabelExpr OptBlockStatements*
else *OptBlockStatements*

ActsForStatement:
actsFor (*Principal* , *Principal*) *Statement*

The declassify statement executes a statement, but with some restrictions removed from pc.

DeclassifyStatement:
declassify (*LabelExpr*) *Statement*

A.6 New expressions

Literal:
. . . existing productions . . .
new label *LabelExpr*

DeclassifyExpression:
declassify (*Expression* , *LabelExpr*)

References

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. Also appeared as SRC Research Report 47.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[HDT87] Susan Horwitz, Alan Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24:679–694, 1987.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[JL78] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5):358–367, May 1978.

[KW94] Atsushi Kanamori and Daniel Weise. Worklist management strategies for dataflow analysis. Technical Report MSR–TR–94–12, Microsoft Research, May 1994.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[LMM98] Barbara Liskov, Nicholas Mathewson, and Andrew C. Myers. PolyJ: Parameterized types for Java. Software release. Located at http://www.pmg.lcs.mit.edu/polyj, July 1998.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1998.

[MMN90] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC — defining new forms of access control. In *Proc. IEEE Symposium on Security and Privacy*, pages 190–200, 1990.

[Mye99] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999. In progress.

[PO95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[RM96] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *Proc. 3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996.

[RSC92] Joel Richardson, Peter Schwarz, and Luis-Felipe Cabrera. CACL: Efficient fine-grained protection for objects. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 154–165, Vancouver, BC, Canada, October 1992.

[Sto81] Allen Stoughton. Access flow: A protection model which integrates access control and information flow. In *IEEE Symposium on Security and Privacy*, pages 9–18. IEEE Computer Society Press, 1981.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Vol97] Dennis Volpano. Provably-secure programming languages for remote evaluation. *ACM SIGPLAN Notices*, 32(1):117–119, January 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.