Securing Data at Java IO Boundaries

Scott F. Smith and Mark Thober

The Johns Hopkins University
{scott,mthober}@cs.jhu.edu

Abstract

We present an information flow type system for Featherweight Java, taking a programmer-centered view by requiring minimal program annotations, and focusing on IO points, the most critical flow boundary. Our static type inference system automatically infers information flow labels, thus eliminating the need for explicit program annotations. We prove type soundness and a noninterference property using an extensible operational approach. On top of this system, we provide an analysis that extracts all information flowing in and out of the IO points of a program. Global program flows can then be observed, and policies can be set to control these flows. We argue that controlling data at input and output points is ultimately the only data security borders that matter, and our system allows programmers to focus on this dimension.

1. Introduction

Effective tracking of information in programs begins and ends at the program's runtime data boundaries, the IO points. If secret data appears to be abused in a program, yet the program generates no observable output, by definition no secrets are lost. For this reason, we focus here on making IO points the center of information flow policy declarations, since that is the only place in the end that a policy is needed. We believe bringing the issue of IO to the center of information flow research will lead to systems that work better in practice.

Existing work on information flow [4, 12, 24, 19, 31] shows how type systems can be defined that statically guarantee that high security data will not affect low security data. A *noninterference* [11] property is usually shown for well-typed programs: low security outputs are not affected by any high security inputs. While the foundations of information flow are solid, existing information flow systems have several usability problems. The overhead for adding information flow security to programs is potentially large, since existing systems usually require that security annotations be added to the code, making the coding process more time consuming. With large numbers of annotations, the likelihood of having incorrect annotations also increases: a mistake can get lost in the noise of so many annotations.

Instead of focusing on securing the boundaries, most current systems focus on internal control, by placing static labels on memory locations. For example, the variable `h` may have the label *high*, while the variable `l` may have the label *low*. A type system then ensures that the value of a lower level variable is not influenced in any way by the value of a higher level variable. While this is an effective means to control information flow, it does not make clear what the top-level policy being enforced is: the flow policy is buried in the variable declarations.

Our primary goal is to provide practical data secrecy and integrity protection to aid programmers in securing programs they write. In particular, we focus on input and output points as *the* important boundaries for securing data. Thus, our focus is on controlling information flows upon entry and exit from a program, and not unnecessarily controlling flows internally. Additionally, we do not want to require significant new syntax or require any program annotations, in order to ease the necessary overhead for programmers. In general, we should speak of securing the *component interface* [30] of software, since some runtimes are composed of multiple independent components with distinct security policies; here we focus on IO for simplicity.

Our approach associates all program values with flow labels. Labels are explicitly placed on input data and checked at output points; for points in between, the type system can automatically infer the labels and so programmers do not need to add declarations.

We provide program constructs `Label(e,L)` to explicitly label data and `Forbid(e,L)` and `Ensure(e,L)` to check data secrecy and integrity, respectively; we additionally allow the ability to downgrade (*declassify*) labels when deemed safe to do so. We prove a type soundness result, and a noninterference property.

One weakness of Java (along with many other languages)

is how the IO points get buried in the code through subclassing, method calls, *etc.*. This in turn makes it difficult to observe where the actual IO is occurring without digging through the whole program. This lack of a clear top-level IO interface means anyone who wants to understand the information flow properties of a whole program must have intimate knowledge of the code in order to understand what information flows occur through IO. To help programmers obtain a top-level picture of the IO and thus the information flow of a program, we provide a simple tool that extracts program input and output points along with the corresponding information flows at these points. With this tool, programmers can observe *all* input and output points in a program, along with what information flows through these points. This can then be used to set policies for these points, or as a way of double-checking the accuracy of the program's information flow security controls. A more elegant solution would be a language design that focuses more explicitly on the runtime interface of code; some language designs have begun to address this topic, *e.g.* [18]. Rather than proposing a new language here, however, we show how it is not difficult to derive this interface, obtaining a workable solution without need for language redesign.

The rest of this paper is organized as follows. Section 2 is a high-level overview of our system. Section 3 describes in more detail how our system can be used to control information flows at IO points, via an example program. Section 4 presents the EFL language, our extension of Featherweight Java [13], and our label type system for typing information flows. Section 4.3 presents the small-step operational semantics for our system and soundness and noninterference results. Our IO extraction tool is detailed in Section 5, and we show how to create policies for enforcing secrecy and integrity properties in section 5.3. We then discuss related work in section 6.

2. System Overview

Our syntax is based on Featherweight Java [13], extended with labeling, checking, and declassifying syntax as well as other minor additions. Labels are placed on data with the statement `Label(e,L)`, which labels an expression `e` with label set `L`. We track sets of labels as opposed to defining a security lattice [6, 10]. Information flows may be controlled by check statements, which ensure that the labels inferred for a particular value satisfy the secrecy or integrity property as defined by the check statement. These checks allow programmers to create well-defined security boundaries, and the flexibility to only check the labels that are really important for a particular piece of code. For secrecy, we use `Forbid` checks as in the following example

```
int f (int x) {return Forbid(x,{high});}
```

Now, any application `f(h)`, where data flowing into `h` has the label `high` will cause a type failure since the check forbids data labeled `high` from passing through the check statement. These statements accommodate IO security, where data is labeled upon input, and checked immediately before output.

We can also enforce integrity properties using a check, `Ensure(e,L)`, which requires the expression `e` to carry all of the labels in the set `L`. For example, consider a program that changes a user's password. Integrity checks can be placed on the password output channel to ensure low security data does not flow into the system password file. While most research correctly states that integrity is a dual to secrecy [7], there are subtle differences [14, 16], and for this reason we model both secrecy and integrity in detail.

We additionally provide a `Declassify(e,L)` statement, which removes secrecy labels `L` from `e`. This serves to declassify data in infrequent, explicitly allowable instances [21, 34]. For example, in a program where a password is being checked, the result of a password comparison may be declassified, so the resulting boolean will not carry the high security label of the password. Programmers must be very careful when using declassify operations, which may reveal too much information and compromise security.

We define a static constraint-based type inference system, with a form of automatic label polymorphism inference that is related to 1CFA [27], and related concrete class analyses [3, 28, 33]. The need for label polymorphism inferences will become evident when we study the example program of Section 3.

2.1. Information Flow Analysis for IO

Regardless of program internals, data is only truly compromised during input or output, whether writing to a file, sending information over a socket connection, or reading user input from the keyboard. For this reason, we are focusing on the IO points. We define an IO point extraction analysis in section 5. This analysis extracts *every* input and output point in a program along with the respective information flow details. Figure 1 shows the general form of such an extraction. Input operations are shown with the labels placed on the input data. Output operations are shown with the labels on the data being output, the checks on the data, and any declassifications which have occurred on data flowing to the output point. We extract the declassifications because of their potentially hazardous nature. This analysis gives programmers a top-level view of the program information flow, allowing them to produce the proper security policies or double check the program for accuracy.

Beyond extracting the global information flow, it is useful to be able to write the *entire* information flow policy, *independent* of the program. We describe such a system in Section 5.3. These policies automatically insert the ap-

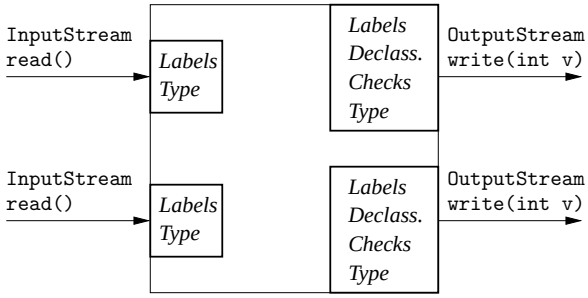

Figure 1. Information Flow Analysis for IO

propriate information flow statements into regular Java programs. This allows programmers to specify information flow policies at the top level, without altering program internals.

2.2. Program Constants

The use of security-critical constants in a program can create security holes: hard-coded secret data may be mislabeled and leak out of a program through output operations, or by an unauthorized agent reading the source code itself. Similarly, program constants may adversely affect data integrity, *e.g.* if a rogue string constant is inadvertently written as a user's password. Remarkably, programmers continue to make such mistakes, even in recent commercially available programs and devices [25, 2, 9], where hard-coded passwords resulted in security problems. This underscores the dangers of hard-coding data and the need for tools to help programmers avoid such mistakes.

We take the approach that hard-coding of secret data or low-integrity data simply should not happen: the only reasonable way to view program constants are as low secrecy but high integrity data, and this is how our type system treats all constants. Since abuse of constants is potentially a programming pitfall, it is useful to also point out to programmers the constants involved in various outputs, so they can verify they are low secrecy, high integrity data. An analysis producing all such constants is easily defined by treating program constants as a special form of input channel; since it is straightforward we do not define the actual algorithm.

3. An Example of IO-based Information Flow

In this section we elaborate on how information flow is controlled at IO points in our system, by the study of a simple example. IO channels in Java are created through subclassing, as in `FileInputStream`, `DataOutputStream`, `SocketInputStream`, etc. We build on this principle, and define different information flow policies using different subclasses of these IO classes. For each distinct security category of IO, a different subclass is created. This 1-1 relationship between class definitions and security policies makes for an "object oriented" approach to information flow policies, harmonizing with the existing language structures.

We now focus on an example program for changing passwords, where data security is important in both secrecy and integrity dimensions. Firstly, we want to provide secrecy for the user name and password information contained on the system, making sure this information is not leaked to a public channel, i.e. the screen. Secondly, we want to ensure the integrity of the system password file by not allowing it to be tainted by improper data, thereby altering user names and passwords on the system. These are two well-defined goals for a programmer of a password changing application.

We take some liberties with syntax that is not described in our calculus, such as the use of local variables, `super()`, and a `while` loop. We make some abbreviations to shorten the presentation, `IS` for `InputStream`, `OS` for `OutputStream`, `PS` for `PrintStream`. `B` abbreviates `Buffer`, so `BR` is `BufferedReader`. Other obvious abbreviations have been made, and some code is omitted for lack of space.

The first segment of code defines the `PwdFile` class, which has fields for the password file name and a temporary file name. The primary method here is `ChangePwd`, which changes the password of a user in the password file. The second segment of code defines some input and output streams for this program. The input streams label data from the system or the user accordingly. Similarly, the output streams check to make sure sensitive data is not output to the screen, and that low integrity data is not output to the password file. The final segment is the `main` method that takes user input to change a password and outputs whether the password change succeeded or not.

```
class PwdFile extends Object {
 String fileName; String tempName;

 Reader getPwdReader() {
  SysFileIS fin = new SysFileIS(fileName);
  return new BR(new ISReader(fin)); }

 Writer getWriter() {
  PwdFileOS fout = new PwdFileOS(tempName);
  return new PrintWriter(fout); }

 bool isUser(String line, uname, oldpwd) {
  // parse line and return true if uname and oldpwd match
   }

 bool ChangePwd(String uname,oldpwd,newpwd){
  bool succ = false; String line;
  BR passIn = getPwdReader();
  PrintWriter tempOut = getWriter();
  while((line = passIn.readLine()) != null) {
   if (isUser(line,uname,oldpwd)) {
```

```
    tempOut.println(uname + ":" + newpwd);
    succ = true;
   } else { tempOut.println(line) }
  }
  // rename tempFile to fileName
  return Declassify(succ,{Secret,Sys,User}); }
}

public class SysFileIS extends FileIS {
 public int read() {
  return Label(super.read(),{Secret,Sys}); }}

public class UserBIS extends BIS {
 public int read() {
  return Label(super.read(),{Secret,User});}}

public class PwdFileOS extends FileOS {
 public void write(int v) {
  Ensure(v,{Secret}); super.write(v); }}

public class ScreenPS extends PS {
 public void println(String s) {
  Forbid(s,{User,Sys,Secret});
  super.println(s); }}

void main(){
 String fileName = "/etc/passwd";
 String tempName = "/tmp/tmppasswd";
 PwdFile pf = new PwdFile(fileName,tempName);

 //read uname,oldpwd,newpwd from a UserBIS.
 ScreenPS screen = new ScreenPS(System.out);

 bool succ = pf.ChangePwd(uname,oldpwd,newpwd);
 if (succ) { screen.println("Success"); }
 else { screen.println("Failure"); }
}
```

The modifications needed to support information flow analysis here are minor. The most significant requirement is to define distinct subclasses of `InputStream` and `OutputStream` for each distinct IO policy. In this case we are defining four new IO policies, in the classes `SysFileIS` and `UserBIS` (for input), and `PwdFileOS` and `ScreenPS` (for output). In each case, the `read` or `write` method labels (input) or checks (output) the appropriate information flow. (Note that IO can occur with other methods such as file `rename`, but we are simplifying a bit in this example). There is also a declassification of labels at the end of the `ChangePwd` method. Programs require no explicit parametric types, and no label type declarations on variables – both type parametricity and variable information flow is automatically inferred.

Note the password file and temporary password filenames are hard-coded in the program. This does not violate our assumption that secret data is not hard-coded in programs, as neither of these are secret. The constant extraction algorithm we described in the previous section would still extract these constants for inspection, to verify that indeed they were not secret information.

Proper typing of this example imposes some requirements on the type system: the type of the `read` and `write` methods simply *cannot* be the same across all subclasses, otherwise all of the work we made to separate the policies in separate classes would be for nothing since their information flow would all merge. So, a form of parametric polymorphism is needed to distinguish between subclasses. It is even more subtle because a variable declared to be an `InputStream` can at runtime be any of its subclasses such as `SysFileIS` or `UserBIS`, and so it may look very difficult to type these methods distinctly. Our solution is to use a polymorphic form of concrete class analysis [3]: we use a constraint-based type system that specializes the type of an object at each method call site for each different type of object that it could be. This technique leads to a very accurate typing [3, 33], and allows the methodology of placing different security policies in different subclasses to be sound yet expressive.

To better illustrate the expressiveness of our polymorphic type system we show an alternate implementation of the `ChangePwd` method, one that takes an `InputStream` and `OutputStream` as arguments for reading from and writing to the password file, respectively.

```
bool ChangePwd(IS in,OS out,String
 uname,oldpwd,newpwd){
 bool succ = false; String line;
 BR passIn = new BR(new ISReader(in));
 PrintWriter tempOut = new PrintWriter(out);

 // ... same code as above
}
```

The following code uses this new implementation. `TopFileOS` is subclassed from `FileOS`, and the `write` method of the new class checks the output data for the integrity label `TopSecret`. In the `main` portion, two different calls are made to `ChangePwd`, one with a `PwdFileOS`, as before, and one to a `TopFileOS`.

```
public class TopFileOS extends FileOS {
 public void write(int v) {
  Ensure(v,{TopSecret}); super.write(v); }}

void main() {
 // ... same code as above

 String ts = "/etc/topsecret";
 SysFileIS in = new SysFileIS();
 PwdFileOS pout = new PwdFileOS(tempName);
 TopFileOS tout = new TopFileOS(ts);

 pf.ChangePwd(in,pout,uname,oldpwd,newpwd);
 pf.ChangePwd(in,tout,uname,oldpwd,newpwd);
}
```

Our polymorphic type system is expressive enough to support this new `ChangePwd` method, with no annotations. Additionally, since we are statically inferring the concrete classes of objects, we can create different security policies for overriding methods, and the type system will know the correct policy to use. In this example, the first call to `ChangePwd` will type properly, but the second call will cause a type error, since the data passed to the `write` method of the `TopFileOS` is not labeled with `TopSecret`. Note that this approach allows the use of behavioral subtyping [17], by adding flow restrictions in the subclasses, but does not require it.

4. Types for Data Tracking and Checking

In this section, we present the formal type system, operational semantics, and soundness properties. In order to simplify the reasoning and presentation of the system, we define a *label type inference system* solely for typing data flows, and use the existing FJ type system for normal FJ typechecking not related to information flow. Our label type system is strong enough to handle any valid FJ program, including those with mutually recursive class definitions, and method recursion. A program type checks if and only if it type checks in both the FJ type system and the label type system.

4.1. The Language

Our language is an extension of Featherweight Java (FJ) [13]. FJ contains the basic object constructs of Java, and is small enough to still allow formal properties to be established. However, FJ does not contain primitive data types or conditional statements, and these are critical to the study of information flow. We thus add the following Java constructs: *constants* (`int, bool, string`), *operators* (`+,-` etc.), *conditionals*, and *sequencing*. The grammar for our Extended FJ (EFJ) language is given in Figure 2.

CL	::= class C extends C $\{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}$	*classes*
K	::= $\texttt{C}(\bar{\texttt{C}}\ \bar{\texttt{f}})\ \{\texttt{super}(\bar{\texttt{f}}); \texttt{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\}$	*constructors*
M	::= $\texttt{C}\ \texttt{m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\ \{\texttt{return e};\}$	*methods*
L	::= $\{\bar{\texttt{l}}\}$, where l are unique labels.	*labels*
CO	::= c | b | s	*constants*
e	::= x | CO | e.f | $\texttt{e.m}(\bar{\texttt{e}})$ | new $\texttt{C}(\bar{\texttt{e}})$ |	*expressions*
	if e then e else e | (C) e | e $\oplus$ e |	
	e; e | Label(e, L) | Forbid(e, L) |	
	Ensure(e, L) | Declassify(e, L)	

Figure 2. Grammar

To track data flows, we also add explicit constructs for that purpose: `Label(e,L)`, which labels `e` with label `L`; `Ensure(e,L)`, which ensures that `e` carries at least all of the integrity labels in label set `L`; `Forbid(e,L)`, which forbids `e` from carrying any of the secrecy labels in label set `L`; and `Declassify(e,L)`, which removes the secrecy labels in `L` from those on `e`. We here omit the integrity dual of declassification, `Endorse(e,L)`, which is a simple extension that adds integrity labels `L` to those on `e`.

We assume some familiarity with FJ, and do not reproduce its typing or semantic definitions; see [13] for the details. The small-step operational semantics and type soundness for EFJ are presented in section 4.3, following a discussion of the label type system. Note that EFJ follows FJ and types expressions with respect to a global class table, CT, that contains the types of all classes. At the top level an expression `e` corresponding to the `main` method is typechecked with respect to this table.

In FJ type assertions are of the form $\Gamma \vdash_T \texttt{e} \in \texttt{C}$, meaning in environment Γ, expression `e` has type `C`. In addition to the standard type rules for FJ, we add the type rules corresponding to the EFJ extensions; they are mostly straightforward, and are omitted for lack of space. For $\texttt{Label}(\texttt{e}, \texttt{L})$, $\texttt{Forbid}(\texttt{e}, \texttt{L})$, $\texttt{Ensure}(\texttt{e}, \texttt{L})$, and $\texttt{Declassify}(\texttt{e}, \texttt{L})$, the resulting type of each expression is the same type as `e`, since the value of each expression is `e` and the label tracking is only handled in the label typing rules.

4.2. Label Types

EFJ values are either objects or primitive constants. Objects may be labeled, as may the internal fields of an object. Thus, Label types, τ, are four-tuples $\langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$; $\mathcal{S}$ is a set of secrecy labels for the current object, $\mathcal{I}$ is a set of integrity labels for the object, $\mathcal{F}$ is a label record for the internal fields of the object, and $\mathcal{A}$ is an α-type, a type representing the concrete class of the object. The type notation is summarized in Figure 3.

Each field of an object has its own labels, $\mathcal{F}$, $\{\texttt{f}_1 \mapsto \tau_1, \ldots, \texttt{f}_n \mapsto \tau_n\}$. The individual labels may be accessed by a dot notation: $\mathcal{F}.\texttt{f}.S$ is the secrecy label on the `f` field of the object. Primitive constants int/bool/string are labeled as objects with no fields.

The α-types are used to express a form of parametric polymorphism over the inheritance hierarchy, allowing the superclass and subclass to differ in their labeling. Without α-types, we would lose a critical degree of expressiveness. For example, suppose `SecretOutput` is a subclass of `Output`, where the `write` method of `SecretOutput` contains a check, and the `write` method of `Output` does not. It is imperative to statically distinguish `SecretOutput` objects from `Output` objects. The usual Java type declaration is insufficient for determining the class of an object, as it may be an object of a subclass, which contains different

$$
\begin{array}{lcll}
\tau & ::= & \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle | t & \textit{types} \\
\mathcal{S} & ::= & \{\bar{\mathtt{l}}\} | s | \mathcal{S} \cup \mathcal{S} | \mathcal{S} - \mathcal{S} | \mathcal{F}.\mathtt{f}.S | \emptyset & \textit{secrecy types} \\
\mathcal{I} & ::= & \{\bar{\mathtt{l}}\} | i | \mathcal{I} \cap \mathcal{I} | \mathcal{F}.\mathtt{f}.I | \emptyset & \textit{integrity types} \\
\mathcal{F} & ::= & \{\bar{\mathtt{f}} \mapsto \bar{\tau}\} | f | \mathcal{F}.\mathtt{f}.F | \emptyset & \textit{field types} \\
\mathcal{A} & ::= & \mathtt{C} | \alpha | \mathcal{F}.\mathtt{f}.A & \textit{alpha types} \\
t & ::= & \langle s, i, f, \alpha \rangle & \textit{type variables} \\
\kappa & ::= & \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} & \textit{method types} \\
 & & \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} & \\
c & ::= & \mathcal{S} <: \mathcal{S} | \mathcal{I} <: \mathcal{I} | \mathcal{F} <: \mathcal{F} & \textit{constraints} \\
 & & | \mathcal{A} <: \mathcal{A} | FC(\mathtt{L}, \mathcal{S}) | EC(\mathtt{L}, \mathcal{I}) & \\
 & & | \mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r) & \\
\mathcal{C} & ::= & \{c\} | \mathcal{C} \cup \mathcal{C} | \emptyset & \textit{constraint sets} \\
s, i, f, \alpha, s_p, i_p & & & \textit{variables} \\
pc, pc_i & & & \textit{program counters} \\
u & & & \textit{top integrity label}
\end{array}
$$

Figure 3. Notation

checks, or returns different labels. Consider the following example:

```
void foo(Output out,int i) { out.write(i); }
```

Here the function `foo` takes an `Output` object, and calls the `write` method of the object with a `String` argument `s`. Now, the object `out` may actually be a `SecretOutput` object, which performs a check on the data to be written. For this example our type system gives a polymorphic type to `out`, and this type is instantiated to a concrete class for each call site of the method. Since our system instantiates object types at each method call site, it is a form of 1CFA analysis [27] and is also closely related to CPA [3, 28, 33]. This expressiveness of α types allows programmers to define new security requirements on IO channels, and because the parametric types are inferred, it requires no additional overhead on the part of the programmer.

We use $\mathtt{l}$ to represent a concrete label, and s, i for label variables in the secrecy and integrity domains, respectively. Notation $\mathtt{L}$ refers to a set of concrete labels $\{\bar{\mathtt{l}}\}$, and label sets $\mathcal{S}, \mathcal{I}$ may contain both concrete label sets and label variables, the latter used when the concrete label is not yet known. For example, when typing methods, the argument labels are variables since the actual labels are not instantiated until the method is invoked. Additionally, f is a field variable referring to an abstract field, and $\mathcal{F}$ is either an abstract or a concrete field mapping; α is a variable referring to an unknown class, and $\mathcal{A}$ is either an abstract class α or a concrete class $\mathtt{C}$.

We implicitly work over a simple equational theory of sets in typing and constraint closure. Concrete unions, $\mathcal{S} \cup \mathcal{S}'$, where $\mathcal{S} = \{\bar{\mathtt{l}}\}$ and $\mathcal{S}' = \{\bar{\mathtt{l}'}\}$ are considered equivalent to the unioned set, $\mathcal{S} \cup \mathcal{S}' = \{\bar{\mathtt{l}}, \bar{\mathtt{l}'}\}$. An analogous equivalence holds for $\mathcal{I} \cup \mathcal{I}'$ when $\mathcal{I}$ and $\mathcal{I}'$ are concrete label sets. $\mathcal{S} - \mathcal{S}'$ is also equivalent to the obvious set difference when both are concrete label sets. For field access, $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.S$ is equivalent to $\mathcal{S}_i$, where $\mathtt{f}_i \mapsto \langle \mathcal{S}_i, \mathcal{I}_i, \mathcal{F}_i, \mathcal{A}_i \rangle$. A similar equivalence analogously holds for any $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.I$, $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.F$, or $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.A$.

We use a label table, LT, to keep track of the label types of all classes when typing expressions. This is analogous to the class table CT of the FJ type system that keeps track of all class types. However, since we are inferring label types here, we must build up the label table while typing the classes.

Label type rules are of the form $\Gamma, pc, pc_i \vdash \mathtt{e} : \tau \backslash \mathcal{C}$, meaning in label environment Γ, with program counters pc and pc_i, expression $\mathtt{e}$ has label type τ with constraint set $\mathcal{C}$. Γ binds variables to label types, so $\Gamma(x) = \tau$. The variables pc and pc_i are program counters that track implicit flows through programs and are a standard feature of information flow type systems.

The constraint set, $\mathcal{C}$, contains normal subtyping constraints $<:$ for secrecy, integrity, field, and α types. In addition, check constraints of the form $FC(\mathtt{L}, \mathcal{S})$, and $EC(\mathtt{L}, \mathcal{I})$, for Forbid and Ensure checks, respectively, are placed in $\mathcal{C}$ and the closure process will need to verify their correctness. Method constraints $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r)$ contain the necessary information to tie up method invocations with the labels of the resulting method call. Methods in the label table are universally quantified, $\forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}$ so they may vary parametrically. For each method invocation, the type variables are separately instantiated in the constraint closure; this ensures that labels do not pollute independent method calls.

We discuss the typing of expressions in section 4.2.1 followed by details of the constraint closure in section 4.2.2 and inconsistencies in section 4.2.3.

4.2.1. Expression Typing

The Label type inference rules for expressions are given in Figure 4. Label type rules for methods, classes, and whole programs are given in figure 5.

Here are a few highlights of the rules. (Const) types constants as label types containing only pc for secrecy, and pc_i for integrity, reflecting our view that constants should by default have no secrecy and full integrity as discussed in section 2.2.

The type of an object field access is the type of the field within the object, along with the labels the object itself carries (Field). For example, if $\mathtt{o}$ has type $\langle \mathtt{Sec}, \mathtt{Sec}, \{f_1 \mapsto \langle \mathtt{User}, \mathtt{User}, \emptyset, \alpha_1 \rangle; f_2 \mapsto \langle \mathtt{Pwd}, \mathtt{Pwd}, \emptyset, \alpha_2 \rangle\}, \alpha \rangle$ then $\mathtt{o}.\mathtt{f}_1$ has type $\langle \{\mathtt{Sec}, \mathtt{User}\}, \emptyset, \emptyset, \alpha_1 \rangle$, that is, the labels on the object $\mathtt{o}$, and the label on the field $\mathtt{f}_1$.

In (Invoke), the constraint $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r)$ is added to the constraint set. The method type eventually needs to

$$\frac{\Gamma(\mathtt{x}) = \langle\, s, i, f, \alpha \,\rangle}{\Gamma, pc, pc_i \vdash \mathtt{x} : \langle\, s \cup pc, i \cap pc_i, f, \alpha \,\rangle \backslash \emptyset}\text{(Var)} \qquad \frac{}{\Gamma, pc, pc_i \vdash \mathtt{c} : \langle\, pc, pc_i, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\text{(Const)}$$

$$\frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash \mathtt{e.f} : \langle\, \mathcal{S} \cup \mathcal{F}.\mathtt{f}.S, \mathcal{I} \cap \mathcal{F}.\mathtt{f}.I, \mathcal{F}.\mathtt{f}.F, \mathcal{F}.\mathtt{f}.A \,\rangle \backslash \mathcal{C}}\text{(Field)}$$

$$\frac{\begin{array}{c}\Gamma, pc, pc_i \vdash \mathtt{e} : \tau = \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C} \qquad \Gamma, pc, pc_i \vdash \bar{\mathtt{e}} : \bar{\tau} \backslash \bar{\mathcal{C}} \\ t_r = \langle\, s, i, f, \alpha \,\rangle \qquad s, i, f, \alpha \text{ are fresh variables}\end{array}}{\Gamma, pc, pc_i \vdash \mathtt{e.m}(\bar{\mathtt{e}}) : \langle\, \mathcal{S} \cup s, \mathcal{I} \cap i, f, \alpha \,\rangle \backslash \mathcal{C} \cup \bar{\mathcal{C}} \cup \{\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau \xrightarrow{pc, pc_i} t_r)\}}\text{(Invoke)}$$

$$\frac{\Gamma, pc, pc_i \vdash \bar{\mathtt{e}} : \bar{\tau} \backslash \bar{\mathcal{C}} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}{\Gamma, pc, pc_i \vdash \mathtt{new\ C}(\bar{\mathtt{e}}) : \langle\, pc, pc_i, \{\bar{\mathtt{f}} : \bar{\tau}\}, \mathtt{C} \,\rangle \backslash \bar{\mathcal{C}}}\text{(New)} \qquad \frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash (\mathtt{C})\ \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}\text{(Cast)}$$

$$\frac{\begin{array}{c}\Gamma, pc, pc_i \vdash \mathtt{e} : \tau \backslash \mathcal{C} \\ \Gamma, pc, pc_i \vdash \mathtt{e}' : \tau' \backslash \mathcal{C}'\end{array}}{\Gamma, pc, pc_i \vdash \mathtt{e}; \mathtt{e}' : \tau' \backslash \mathcal{C} \cup \mathcal{C}'}\text{(Seq)} \qquad \frac{\begin{array}{c}\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \emptyset, \mathcal{A} \,\rangle \backslash \mathcal{C} \\ \Gamma, pc, pc_i \vdash \mathtt{e}' : \langle\, \mathcal{S}', \mathcal{I}', \emptyset, \mathcal{A}' \,\rangle \backslash \mathcal{C}'\end{array}}{\Gamma, pc, pc_i \vdash \mathtt{e} \oplus \mathtt{e}' : \langle\, \mathcal{S} \cup \mathcal{S}', \mathcal{I} \cap \mathcal{I}', \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C} \cup \mathcal{C}'}\text{(Op)}$$

$$\frac{\begin{array}{c}\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C} \qquad \Gamma, pc \cup \mathcal{S}, pc_i \cap \mathcal{I} \vdash \mathtt{e}' : \tau' \backslash \mathcal{C}' \\ \Gamma, pc \cup \mathcal{S}, pc_i \cap \mathcal{I} \vdash \mathtt{e}'' : \tau'' \backslash \mathcal{C}'' \qquad s, i, f, \alpha \text{ are fresh variables}\end{array}}{\Gamma, pc, pc_i \vdash \mathtt{if\ e\ then\ e'\ else\ e''} : \langle\, s, i, f, \alpha \,\rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'' \cup \{\tau' <: \langle\, s, i, f, \alpha \,\rangle, \tau'' <: \langle\, s, i, f, \alpha \,\rangle\}}\text{(If)}$$

$$\frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash \mathtt{Label(e,L)} : \langle\, pc \cup \mathtt{L}, pc_i \cap \mathtt{L}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}\text{(Label)} \qquad \frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \tau = \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash \mathtt{Forbid(e,L)} : \tau \backslash \mathcal{C} \cup FC(\mathtt{L}, \mathcal{S})}\text{(Forbid)}$$

$$\frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \tau = \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash \mathtt{Ensure(e,L)} : \tau \backslash \mathcal{C} \cup EC(\mathtt{L}, \mathcal{I})}\text{(Ensure)}$$

$$\frac{\Gamma, pc, pc_i \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc, pc_i \vdash \mathtt{Declassify(e,L)} : \langle\, pc \cup (\mathcal{S} - \mathtt{L}), \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}\text{(Declassify)}$$

Figure 4. Label Type Rules for Expressions

be looked up in the global label table LT. However, since $\mathcal{A}$ may not be a concrete class we postpone this decision until more information is known about $\mathcal{A}$, at constraint closure. The above type constraint records the method call information so it can be propagated in the closure once the concrete class of $\mathcal{A}$ is known.

In (New), the names of the fields in the class `C` are looked up using *fields*. The $\mathcal{F}$ element of the type contains all the labels on the object fields. The α-type is given the concrete class name of the object being created. pc and pc_i are the secrecy and integrity labels on the new object, respectively. Like constants, objects are assumed to have no secrecy and full integrity by default.

The expression `Label(e,L)` places `L` on both the secrecy and integrity labels of `e` via (Label). `Declassify(e,L)` removes `L` from the secrecy labels of `e` in (Declassify).

(Ensure) adds a constraint $EC(\mathtt{L}, \mathcal{I})$ to the constraint set; similarly, (Forbid) adds a constraint $FC(\mathtt{L}, \mathcal{S})$. Since some concrete labels are abstract variables during type inference, these constraints serve as markers,and the actual checks will be made during closure.

Type inference rules for typing programs, classes, and methods are found in Figure 5. Programs are typed by typing each class definition, which types each method definition, which are in turn typed according to the expression rules in Figure 4. Methods require the type variables to be set in an initial label table in order to support recursive class definitions and mutually recursive methods. Method typing fills in the constraint types in the full label table, where the return type of the method body flows into the return label variable of the method.

4.2.2. Label Closure

The closure rules for label constraint sets are given in Figure 6. The rules add new constraints based on transitivity, obvious set propagations, and field labels. The closure rule (Method) is important for tying up the types of method calls. As discussed above, method constraints are added during method invocation, when the actual class of the object on which the method is being called may be unknown. Thus, for all constraints $\mathtt{C} <: \mathcal{A}$, where `C` is a concrete class, the method `m` is looked up in LT via *mtype*, which returns

Initial Lable Table:

$\bar{t}, t_t, t_r, s_p, i_p$ consist of fresh variables.
Each use of $InitialMethod()$ creates distinct variables.

$$\frac{}{InitialMethod() = \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset} \qquad \frac{\bar{\kappa}_i = InitialMethod()}{InitialLT = LT[(\mathtt{C_0}, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, (\mathtt{C_1}, \bar{\mathtt{M}}_1) : \bar{\kappa}_1, \ldots]}$$

Method Typing:

$$\frac{\mathtt{C_0} = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{M} = \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{return\ e};\} \qquad InitialLT(\mathtt{C_0}, \mathtt{M}) : \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset \quad \Gamma[\bar{\mathtt{x}} : \bar{t}, \mathtt{this} : t_t], s_p, i_p \vdash \mathtt{e} : \tau \backslash \mathcal{C} \qquad \bar{t'} = FreeTypeVar(\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\})}{InitialLT \vdash_M (\mathtt{C_0}, \mathtt{M}) : \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Class Typing:

$$\frac{InitialLT \vdash_M (\mathtt{C_0}, \bar{\mathtt{M}}_0) : \bar{\kappa}_0 \qquad InitialLT \vdash_M (\mathtt{C_1}, \bar{\mathtt{M}}_1) : \bar{\kappa}_1 \qquad \ldots}{\vdash_C LT[(\mathtt{C_0}, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, (\mathtt{C_1}, \bar{\mathtt{M}}_1) : \bar{\kappa}_1, \ldots]}$$

Program Typing:

$$\frac{\vdash_C LT[(\mathtt{C_0}, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, (\mathtt{C_1}, \bar{\mathtt{M}}_1) : \bar{\kappa}_1, \ldots] \qquad \emptyset, u \vdash \mathtt{e} : \tau \backslash \mathcal{C} \qquad Closure(LT[(\mathtt{C_0}, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, (\mathtt{C_1}, \bar{\mathtt{M}}_1) : \bar{\kappa}_1, \ldots], \mathcal{C}) \text{ is consistent}}{\vdash_P \{\mathtt{C_0}, \mathtt{C_1}, \ldots\};\ \mathtt{e} : \tau \backslash \mathcal{C}}$$

Fields:

$$\frac{}{fields(\mathtt{Object}) = \emptyset} \qquad \frac{}{fields(constants) = \emptyset} \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad fields(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}}{fields(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}, \bar{\mathtt{C}}\ \bar{\mathtt{f}}}$$

Figure 5. Continuation of Label Type Rules

a typing for that method as found either in C or in a superclass if not defined in C. We then substitute the labels in the *method* constraint into this constraint set from the label table, and replace all local label variables with fresh ones. This allows us to add the constraints from the method being called, without mixing any labels from two separate calls to the same method.

We define a constraint closure as follows.

Definition 4.1 (Constraint Closure) *$Closure(LT, \mathcal{C})$ is the least set that includes $\mathcal{C}$ and any constraint that can be derived from $\mathcal{C}$ by the rules of figure 6, and with the additional constraint that the (Method) rule is only applied once in the closure for each unique set of premises.*

If we did not constrain (*Method*) rule as above, it could be applied arbitrarily many times, generating different fresh variables each time.

4.2.3. Inconsistencies

Inconsistencies in the label constraint sets come from EC and FC constraints. Constraint consistency is defined as follows.

Definition 4.2 (Inconsistent Constraints) *An inconsistent constraint is any constraint $FC(\mathtt{L}, \mathtt{L}')$, where $\exists \mathtt{l} \in \mathtt{L}'$ such that $\mathtt{l} \in \mathtt{L}$; or any constraint $EC(\mathtt{L}, \mathtt{L}')$, where $\mathtt{L} \not\subseteq \mathtt{L}'$*

If $Closure(LT, \mathcal{C})$ contains an inconsistent constraint, then the closure is inconsistent, and type inference fails.

Forbid checks are enforced by FC constraints. $FC(\mathtt{L}, \mathtt{L}')$ is consistent if $\forall \mathtt{l} \in \mathtt{L}', \mathtt{l} \notin \mathtt{L}$. Thus, the sets L and L′ should have no labels in common. For example, the constraint $FC(\{\mathtt{Secret}\}, \mathtt{Public})$ is consistent, while $FC(\{\mathtt{Secret}, \mathtt{TopSecret}\}, \mathtt{Secret})$ is not.

Ensure checks are enforced by EC constraints. $EC(\mathtt{L}, \mathtt{L}')$ is consistent if $\mathtt{L} \subseteq \mathtt{L}'$. The set L are the labels we are requiring, and L′ are the labels on the data, as inferred by the type system. Thus, we are enforcing that the labels required must be a subset of the labels on the data. For example, $EC(\{\mathtt{Secret}\}, \{\mathtt{Secret}, \mathtt{TopSecret}\})$ is consistent, since we are ensuring the label is at least Secret.

4.3. Semantics and Soundness

We now present a small-step operational semantics for our system. Figure 7 shows the necessary definitions for the semantics. Values, v, are constants or objects. We propagate label sets explicitly in the operational semantics; it allows us to show noninterference for the operational semantics independent of types. We are also interested in run-time information flow extensions of our system as future work. Although run-time checks cannot capture all information flows in the presence of state or concurrency, they do narrow the flow.

A configuration $[\![\ \mathtt{e}\ ]\!]_I^S$ consists of an expression with two

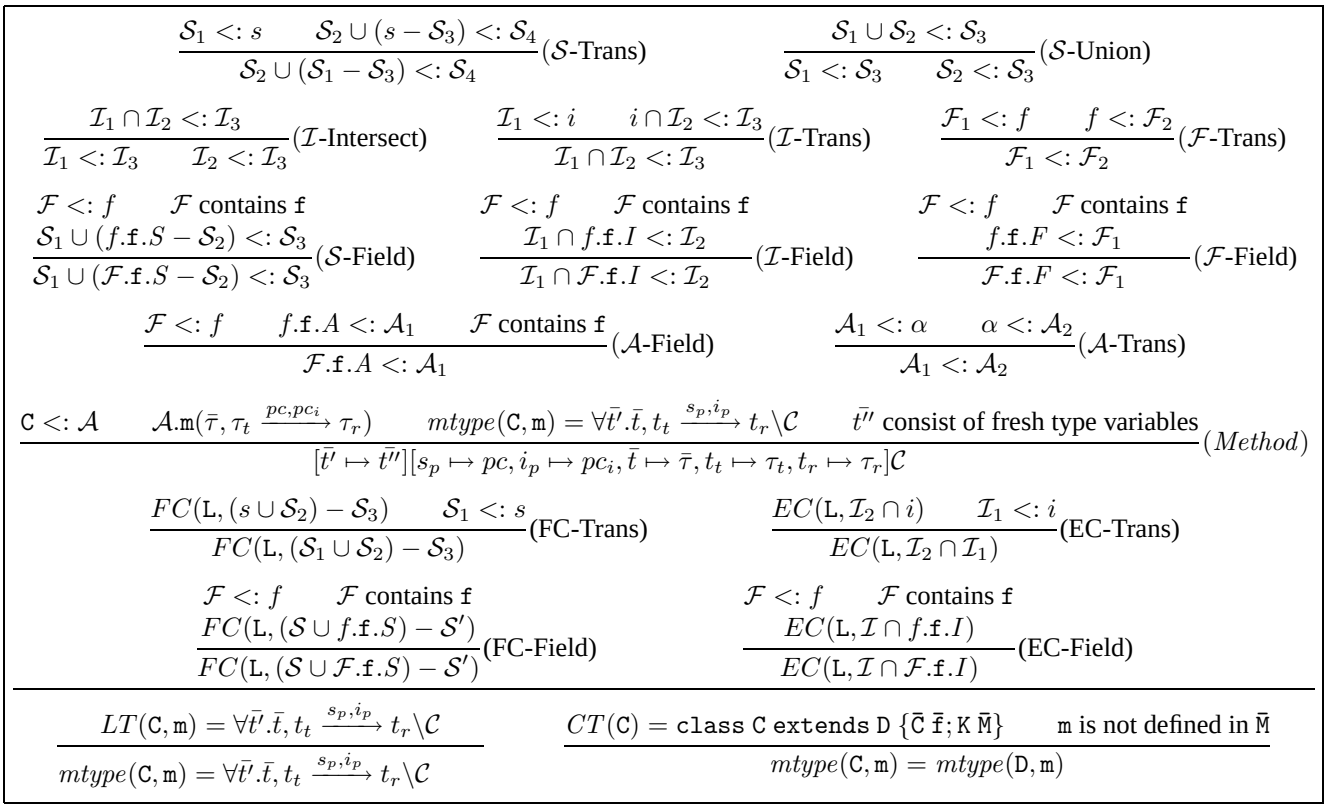

Figure 6. Label Closure Rules

label sets S and I, for secrecy and integrity, respectively. Labels in the label sets may appear due to direct or indirect flows. Final configurations, $(\!|\, v \,|\!)^S_I$, are values with an associated label set, or $CkFail$, denoting a failed label check.

Small-step reduction rules define the single-step reduction relation $\mathcal{E} \rightarrow \mathcal{E}'$, where $\mathcal{E}$ and $\mathcal{E}'$ are configurations. We omit several of the more obvious rules for brevity; these include reductions that take any configuration containing $CkFail$ to a final configuration of $CkFail$. Subterm computation rules push in a configuration to where the next computation will occur. This is necessary so the internal computations will have the proper labels. For example, the subterm computation rule for fields is $[\![\, \mathtt{e.f} \,]\!]^S_I \rightarrow [\![\, [\![\, \mathtt{e} \,]\!]^S_I \,.\mathtt{f} \,]\!]^S_I$. Rules for reduction under context are omitted; one example is assuming $\mathcal{E} \rightarrow \mathcal{E}'$, reduction under context for fields implies $[\![\, \mathcal{E}.\mathtt{f} \,]\!]^S_I \rightarrow [\![\, \mathcal{E}'.\mathtt{f} \,]\!]^S_I$.

Figure 8 give the main reduction rules. Secrecy and integrity labels are passed through the program as is expected based on the label type rules. (IfTrue-R) reduces a `true` conditional to $\mathtt{e}'$, with S and I in the new configuration to account for the indirect flow. $fields(\mathtt{C})$ is as given in Figure 5. (Invoke-R) uses $mbody$ (as in FJ[13]) to look up the method `m` in the class table. $CkFail$ is produced for values whose labels do not satisfy the check assertion.

4.3.1. Type Soundness and Noninterference

The type system along with our semantics allow us to prove the standard subject reduction lemma and type soundness results, whose proofs are omitted for brevity.

Since our operational semantics is based on configurations, we must add type rules for typing configurations. Since these rules are similar to the corresponding expression typing rules, we omit them here.

Initial typings and semantic reductions must start with the proper program counters. For secrecy, this is simply the empty set. For integrity, we use u, the top integrity label. This ensures integrity labels will not be destroyed by an empty program counter.

Lemma 4.3 (Subject Reduction) *If $\Gamma, \emptyset, u \vdash \mathcal{E} : \tau \backslash \mathcal{C}$, and $Closure(LT, \mathcal{C})$ is consistent, and $\mathcal{E} \rightarrow \mathcal{E}'$, then $\Gamma, \emptyset, u \vdash \mathcal{E}' : \tau \backslash \mathcal{C}'$, where $\mathcal{C}' \subseteq Closure(LT, \mathcal{C})$, and $Closure(LT, \mathcal{C}')$ is consistent.*

Theorem 4.4 (Type Soundness) *Assume a well-typed class table, CT and a corresponding label table, LT. If $\emptyset, u \vdash \mathtt{e} : \tau \backslash \mathcal{C}$, and $Closure(LT, \mathcal{C})$ is consistent, then $[\![\, \mathtt{e} \,]\!]^{\emptyset}_u \not\rightarrow CkFail$.*

The following Theorem 4.5 and Theorem 4.7 are statements of secrecy noninterference. These results are stated

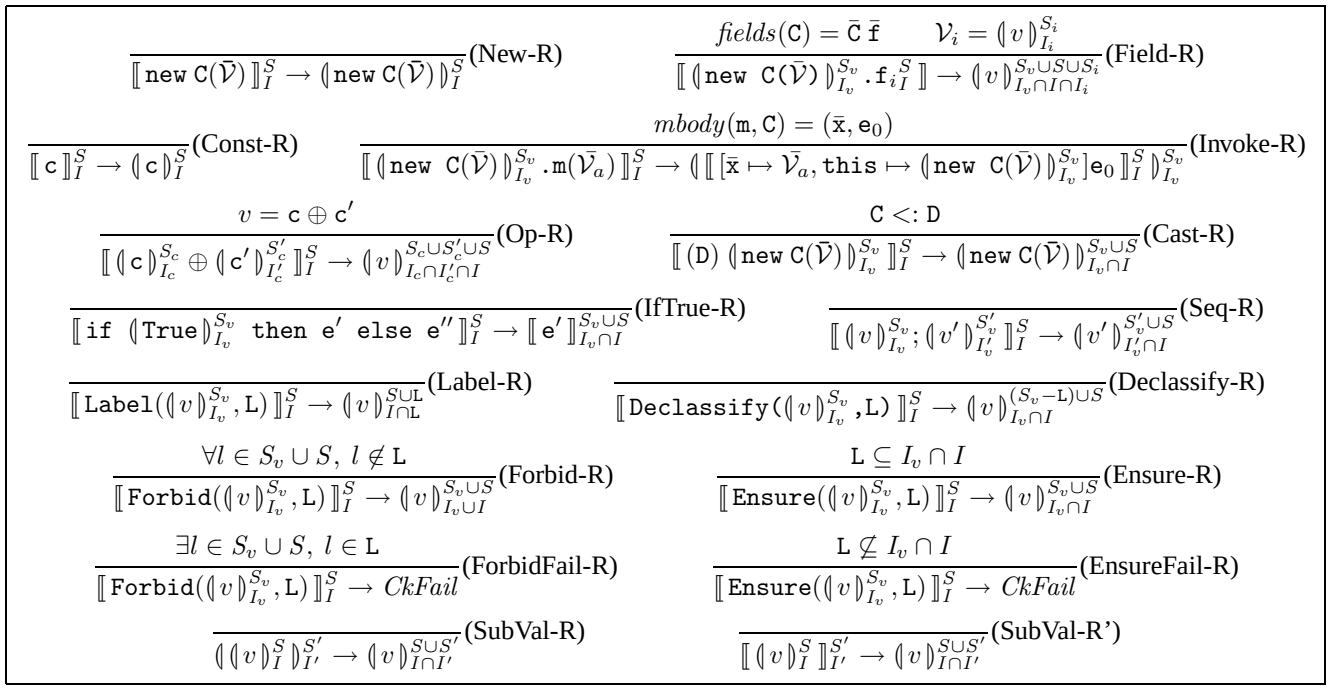

Figure 8. Operational Semantics Reduction Rules

Values:
$v ::= \texttt{C0} \mid \texttt{new C}(\bar{\mathcal{V}})$

Labels:
$S, I ::= \{\bar{\texttt{l}}\}$,
where $\bar{\texttt{l}}$ are unique label names

Final Configurations:
$\mathcal{V} ::= (\!| v |\!)^S_I \mid \mathit{CkFail}$

Expressions:
The definition of e remains the same except for the addition of $\mathcal{V}$ as a valid expression, which is necessary for computation.

Configurations:
$\mathcal{E} ::= \mathcal{V} \mid [\![\texttt{e}]\!]^S_I \mid (\!| \mathcal{E} |\!)^S_I \mid [\![\mathcal{E}.\texttt{f}]\!]^S_I \mid [\![\texttt{new C}(\bar{\mathcal{V}}, \mathcal{E}, \bar{\texttt{e}})^S_I]\!] \mid$
$[\![\mathcal{E}.\texttt{m}(\bar{\texttt{e}})]\!]^S_I \mid [\![(\!| \texttt{new C}(\bar{\mathcal{V}}) |\!)^{S_v}_{I_v}.\texttt{m}(\bar{\mathcal{V}}', \mathcal{E}, \bar{\texttt{e}})]\!]^S_I \mid$
$[\![\texttt{if } \mathcal{E} \texttt{ then e' else e''}]\!]^S_I \mid [\![\mathcal{E}; \texttt{e}]\!]^S_I \mid [\![\mathcal{V}; \mathcal{E}]\!]^S_I \mid$
$[\![\mathcal{E} \oplus \texttt{e}']\!]^S_I \mid [\![\mathcal{V} \oplus \mathcal{E}]\!]^S_I \mid [\![(\texttt{C})\, \mathcal{E}]\!]^S_I \mid [\![\texttt{Label}(\mathcal{E}, \texttt{L})]\!]^S_I \mid$
$[\![\texttt{Forbid}(\mathcal{E}, \texttt{L})]\!]^S_I \mid [\![\texttt{Ensure}(\mathcal{E}, \texttt{L})]\!]^S_I \mid$
$[\![\texttt{Declassify}(\mathcal{E}, \texttt{L})]\!]^S_I$

Figure 7. Operational Semantics Definitions

for a single label, high, and are generalizable to any set of labels. We leave off integrity labels, as they are inconsequential. We assume hereafter that there are no declassification statements since they obviously violate noninterference.

Theorem 4.5 (Semantic Noninterference) *Given a well-typed class table, CT, if* $[\![\texttt{e}]\!]^S \rightarrow^* (\!| \texttt{c} |\!)^{S_c}$ *where* high $\notin S_c$, *and for any* $(\!| \texttt{c}_1 |\!)^{S_1}$ *in* e *where* high $\in S_1$, *and for any* $(\!| \texttt{c}_2 |\!)^{S_2}$ *where* high $\in S_2$ *and* $[\![[(\!| \texttt{c}_1 |\!)^{S_1} \mapsto (\!| \texttt{c}_2 |\!)^{S_2}] \texttt{e}]\!]^S \rightarrow^* (\!| \texttt{c}' |\!)^{S'_c}$ *where* high $\notin S'_c$, *we must have* $\texttt{c} == \texttt{c}'$.

Our proof technique for semantic noninterference is similar in spirit to the proofs in [23]; we show that the *low* reductions occur identically in both derivations, while the *high* reductions may be different.

We define top-level secrecy types as follows in order to more cleanly state our noninterference result. Top-level secrecy types are the concrete secrecy labels for an expression as determined by the constraint set.

Definition 4.6 (Top-level Secrecy Types) *If* $\emptyset, u \vdash \texttt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, *then for a fixed class table* CT, *and fixed label table,* LT, $\emptyset, u \vdash_{Top} \texttt{e} : \langle \mathcal{S}_T, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$ *is a top-level secrecy type, where* $\mathcal{S}_T$ *is a set containing any concrete label,* l, *such that either* $\texttt{l} \in \mathcal{S}$, *or there exists an* $s \in \mathcal{S}$, *such that* $\texttt{l} <: s \in \mathit{Closure}(LT, \mathcal{C})$, *or there exists an* $f.\texttt{f}.S \in \mathcal{S}$, *such that* $\texttt{l} <: f.\texttt{f}.S \in \mathit{Closure}(LT, \mathcal{C})$.

We can then assert noninterference of typed programs.

Theorem 4.7 (Noninterference) *Assume* $\emptyset, u \vdash_{Top} \texttt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, *where* high $\notin \mathcal{S}$. *If* $[\![\texttt{e}]\!]^S \rightarrow^* (\!| \texttt{c} |\!)^{S_c}$, *for any* $\emptyset, u \vdash_{Top} (\!| \texttt{c}_1 |\!)^{S_1} : \langle \mathcal{S}_1, \mathcal{I}_1, \mathcal{F}_1, \mathcal{A}_1 \rangle \backslash \mathcal{C}_1$ *in* e *where* high $\in \mathcal{S}_1$, *and any* $\emptyset, u \vdash_{Top} (\!| \texttt{c}_2 |\!)^{S_2} : \langle \mathcal{S}_2, \mathcal{I}_2, \mathcal{F}_2, \mathcal{A}_1 \rangle \backslash \mathcal{C}_2$ *where* high $\in \mathcal{S}_2$ *and* $[\![[(\!| \texttt{c}_1 |\!)^{S_1} \mapsto (\!| \texttt{c}_2 |\!)^{S_2}] \texttt{e}]\!]^S \rightarrow^* (\!| \texttt{c}' |\!)^{S'_c}$, *then* $\texttt{c} == \texttt{c}'$.

Proof. This follows directly from Lemma 4.3 and Theorem 4.5. □

5. IO Analysis of Information Flows

In this section we describe in more detail how IO interfaces may be extracted from FJ programs. The motivations for this were covered in Section 2, and an example of the results it produces were presented in Section 3.

We extend the label type system to use it to extract information flows at IO points. Here, we focus only on `InputStream` and `OutputStream` classes, the parent classes for much of the IO functionality in Java. For input points, the extraction produces a summary of the labels assigned to data on each input channel. For each output point, the secrecy and integrity labels on the output data is given. The analysis also presents labels that were declassified on this flow path, so programmers may examine the declassification to ensure its accuracy. Finally, the extraction describes any checks that are enforced on the data before output.

5.1. Example Extraction

The following extractions from the password program illustrate our technique, whose formalism we describe in the following section.

$Input$: `SysFileIS.read`
$Returns$: `int`
$Labels$: {`Secret`, `Sys`}

This comes from reading the input from the password file. `passIn.readLine()` will trigger this, since it is reading from a `SysFileIS` through a `Reader` wrapper. This extraction shows that a `read` message with `int` return type was called on a `SysFileIS` object, and the labels placed on the input were {`Secret`,`Sys`}.

$Output$: `ScreenPS.println(String s)`
Secrecy: ∅ *Integrity:* ∅
Declassifications: {`Secret`, `Sys`, `User`}
Checks: `Forbid(s,` {`Secret`, `Sys`, `User`}`)`

This comes from the `screen.println()` statements. The extraction shows that a `println` method with a `String` argument was invoked on a `ScreenPS` object. The output data had no secrecy or integrity labels, but the labels {`Secret`, `Sys`, `User`} were at some point declassified. A `Forbid` check on the `String` argument `s` was performed before output.

Notice the same labels are declassified as those that occur in the check. This may alert the programmer to a possible leak in the security of the program, and the accuracy of the declassification should be confirmed.

5.2. Formalism for Extraction of IO Points

We enrich our type system to aid in extraction; types are now $\Gamma \vdash \mathtt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{D}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$. The $\mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A}$ are as before. $\mathcal{D}$ accumulates labels that were declassified at some point on this path; and $Input$ and $Output$ constraints, as defined below, may now be included in $\mathcal{C}$. We omit the enriched type and closure rules for space reasons, and instead summarize the key additions in words.

In the enriched (Declassify) type rule, the declassified labels are added to the $\mathcal{D}$ type. Only the labels *actually* being removed from the type are added, which may be less than the labels in the `Declassify` statement. This is done by intersecting the declassified labels with the secrecy label set.

The enriched type system also includes two new (Method) rules that apply only to method calls on `InputStream` or `OutputStream` objects. Input points $Input(\mathcal{A}, \mathtt{m}, \tau)$ are extracted whenever a method is called on an object that is a subclass of `InputStream`. The extraction has the actual type of the object, $\mathcal{A}$, the name of the method, `m`, and the label type on the return value of this particular method call, τ. Output points $Output(\mathcal{A}, \mathtt{m}, \{\bar{\tau}\}, C_k)$ are extracted whenever a method is called on an object that is a subclass of `OutputStream`. The extraction consists of the actual type of the object, $\mathcal{A}$, the name of the method, `m`, and the label types of every argument to the method $\{\bar{\tau}\}$. Checks on output points, C_k are compiled after the constraint closure has been computed, as discussed below.

Note that we are extracting α types instead of the declared Java types in order to track the actual classes of the objects being called, and not the declared one, which may be some supertype of the actual type.

The extracted IO points shown to the user are those which contain only concrete types and labels. $Output(\mathtt{C}, \mathtt{m}, \langle s, i, d, f, \alpha \rangle, \emptyset)$ is useless to a programmer due to the presence of unresolved variables, and will not be shown in the final analysis.

After all of the IO points have been extracted, and we have sorted out only those with concrete types and labels, the checks on the output points are extracted by looking at the method in the label table, and pulling out all the FC and EC constraints from the constraint set. The labels being checked are matched with the arguments, and the extraction informs the user as to which arguments to the method were checked. The final output is seen in the extraction example of section 5.1.

5.3. Top-level Policies

In this section, we present a policy system for declaring class-based policies at the top level of a program, meaning the policy will not be buried in the code. This also pro-

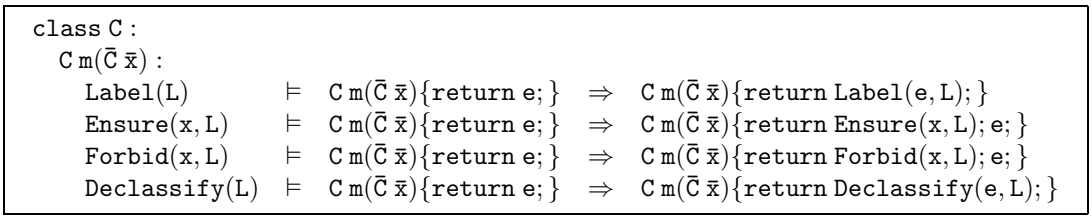

Figure 9. Top-level Policy Translation

vides a simpler means of adding information flow controls to programs, since the underlying programs will not need to include any explicit flow annotations and so there is no need to define a new language syntax for an information flow extension.

We use a simple translation-based approach for these top-level policies. Given a valid program and a top-level policy, the translation produces a new program with `Label`, `Ensure`, `Forbid`, and `Declassify` statements inserted so as to enforce the policy. Policies are declared at the per method level in a class. Each policy statement for a class `C` produces a translation, where method `M` is translated to `M′`, which includes the information flow statement. Figure 9 gives the translation rules.

Policies consist of four types of statements. `Forbid` and `Ensure` statements insert checks that immediately check the labels on arguments when they are passed to a method, before executing the body of the method. This is useful for checking the labels on values before output. `Label` statements are used to label data that is returned from a method, which is useful for labeling data on an input stream. `Declassify` statements specify what labels will be declassified from the method's return value. Note that although we provide the ability to specify declassification policies at the top-level, declassification of data requires knowledge of the underlying code to be sure the data is truly diluted enough to warrant declassification, so it must be used with care.

5.3.1. Example Top-level Policies

The following is a top-level policy for the program for changing passwords in section 3.

```
class SysFileIS
 read():  Label({Secret,Sys})
class UserBufferedIS
 read():  Label({Secret,User})
class PwdFileOS
 write(int v):  Ensure(v,{Secret})
class ScreenPS
 println(String s):
  Forbid(s,{User,Sys,Secret})
class PwdFile
  ChangePwd(String uname,oldpwd,newpwd):
   Declassify({User,Sys,Secret})
```

Supposing the program had no explicit information flow labels, checks, or declassifications; if this policy were applied to that program, we would obtain a program identical to the one presented in Section 3. Even though programs may contain no explicit information flow policy information, it still may be necessary to rewrite parts of a program for purposes of adding an information flow policy: a unique subclass needs to be defined for each different IO security policy. We believe this is independently a good thing, because it gives an object-oriented information flow policy.

6. Related Work

Static analysis of information flow control systems is a well-studied area [12, 31, 32, 4, 1, 23]; Sabelfeld and Myers present a survey in [26]. Much of the literature focuses on proving formal results for small programming languages, although there has been some focus on working systems [24, 19]. None of the previous work focuses on IO boundaries or explicit checks.

Several information flow extensions to languages have been implemented. Flow Caml [24] is an information flow extension to Core ML. The JFlow/Jif system [19, 20], provides information flow control for full Java. Both systems provide some control of information flow at IO points by wrapping raw IO channels with information flow labels. Channels are then treated as variables and given labels based on those principals who are presumed to have access to the channels. This methodology means checks on IO channels are intermixed with the multitude of other internal checks within a program (e.g. on function application, or assignment). Our system is designed to reduce the number of checks to IO points only.

Jif is unique as an information flow system since it covers essentially the full Java language, but it lacks a formal analysis. Jif provides parametric polymorphism and some inference of labels. Programs must be annotated with security labels, including label parameters for polymorphic classes. This creates a backward compatibility issue, where

all code must be re-coded to introduce the proper annotations. Additionally, method overloading requires subclass types to conform to the types of the superclass.

Our type system in contrast infers all label types and all parametric polymorphism, removing the need for additional program annotations. Our label types are inferred across existing code, meaning libraries can be used as is, provided the proper labels and checks are placed on the IO points in the program.

We incorporate a concrete class analysis [3, 22, 33] that tracks the concrete classes of objects through the program, allowing us to statically determine a conservative approximation of the runtime object. This means overridden methods in the subclass can have different types from the superclass, and the type system will correctly distinguish the information flow controls on the different objects statically.

Banerjee and Naumann [4, 5] prove noninterference for an information flow type system for a Java-like language using a denotational semantics. They provide an inference extension for libraries that are parameterized by security levels [29]. This is a similar form of polymorphism to Jif, requiring annotations in the form of label parameters. They also require polymorphic types for methods must be satisfied by all overriding methods. As mentioned above, we employ a more implicit polymorphism that requires no program modifications, and we prove soundness and noninterference using an extensible operational approach.

Flow Caml provides label type inference and parametric polymorphism for an information flow extension to Core ML. They prove soundness of type inference and a noninterference property. Our form of polymorphism and type inference also differs from Flow Caml in that Flow Caml is also not based on concrete class analysis. In addition, our type system is significantly different, since it is based on an object-oriented language, which presents unique issues, (i.e. inheritance) that do not arise in a functional language.

Several recent works have developed policies for downgrading data. Li and Zdancewic [15] and Chong and Myers [8] describe systems which provide downgrading policies that specify the conditions under which data can be declassified. Thus, the data labels contain policies which describe when it is safe to declassify the data, whether after a certain method call, operation, or some other property. In comparison, our policies for downgrading are attached to the methods. The method policies describe what labels will be downgraded for data passed to the method. This mechanism follows the object-oriented philosophy, allowing downgrading at the class and method level, and showing it in the API.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] French Security Incident Response Team Advisories. Veritas backup exec and netbackup remote file access vulnerability. `http://www.frsirt.com/english/advisories/2005/1387`, August 2005.

[3] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *LNCS*. SV, 1995.

[4] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, 2002.

[5] A. Banerjee and D. Naumann. Using access control for secure information flow in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003., 2003.

[6] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, 1975.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, Massachusetts, April 1977.

[8] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[9] Symantec Corp. Symantec brightmail antispam static database password. `http://securityresponse.symantec.com/avcenter/security/Content/2005.05.31a.html`, June 2005.

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

[11] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[12] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[13] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[14] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST)*, Sep. 2003.

[15] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[16] Peng Li and Steve Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Foundations of Computer Security Workshop (FCS)*, 2005.

[17] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[18] Yu David Liu and Scott F. Smith. Modules with interfaces for dynamic linking and communication. In *ECOOP'04*, 2004.

[19] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[20] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999.

[21] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[22] John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–340, 1994.

[23] Franois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 46–57, Montral, Canada, 2000.

[24] Franois Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[25] Paul Roberts. Cisco warns of wireless security hole. *Computerworld*, April 2004. `http://www.computerworld.com/securitytopics/security/holes/story/0,10801,92015,00.html`.

[26] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Jounal on Selected Areas in Communications*, 21(1), January 2003.

[27] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.

[28] Scott Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Verlag, March 2000.

[29] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proc. of the Eleventh International Static Analysis Symposium (SAS)*, volume 3148, pages 84–99. Lecture Notes in Computer Science, Springer-Verlag, August 2004.

[30] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

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

[32] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[33] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *European Conference on Object-Oriented Programming(ECOOP'01)*, Budapest, Hungary, June 2001.

[34] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, page 5, Washington, DC, USA, 2001. IEEE Computer Society.