Improving Usability of Information Flow Security in Java

Scott F. Smith and Mark Thober

Department of Computer Science
Johns Hopkins University
{scott,mthober}@cs.jhu.edu

Abstract

This paper focuses on improving the usability of information flow type systems. We present a static information flow type inference system for Middleweight Java (MJ) which automatically infers information flow labels, thus avoiding the need for a multitude of program annotations. Additionally, policies need only be specified on IO channels, the critical flow boundary. Our type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to properly distinguish the security policies of different IO channels.

We prove a noninterference property for programs that interactively input and output data. We then describe a mechanism that allows users to define top-level policies, which automatically inserts the security policies at the proper points in the program. This provides the further benefit that whomever is defining the policy does not necessarily need intimate knowledge of the program source.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Constraints, Frameworks, Input/output; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Security, Languages, Design, Theory

Keywords Information flow security, type inference, security policies, Java, declassification

1. Introduction

While the foundations of static information flow systems are solid, their usability could still be improved. The overhead for adding information flow security to programs is potentially large, since existing systems often require security annotations to be added to the code. 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. Input/output is another important practical concern which has also not been fully integrated into static information flow systems.

Information flow research [2, 10, 17, 14, 22] has shown how type systems can be defined to statically guarantee that high security data will not affect low security data. A *noninterference* [8] property is usually shown for well-typed programs: low security outputs are not affected by any high security inputs. The majority of these works assume a batch model of IO, although O'Neill *et. al.* recently described a technique for enforcing information flow security for interactive IO, using a simple imperative language and basic type system [16].

Our primary goal is to provide practical data secrecy and integrity protection to aid programmers in securing programs they write. To this end, we present a provably correct static information flow type inference system for a core subset of Java (namely, Middleweight Java) that automatically infers information flow labels, thus avoiding the need for a large amount of program annotations. Policies need only be specified on IO channels, which we will argue to be the only real flow boundary that must be considered. The type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to distinguish policies of different IO channels.

Our work places the focus on input and output points as *the* important boundaries for securing data. Thus, we are only indirectly concerned about internal flows, in how they ultimately will relate to the inputs and outputs. In general, we should speak of securing the *component interface* [21], since runtimes may be composed of multiple independent components with distinct security policies; here we focus on just the IO boundary for simplicity.

As is common practice in information flow type systems, we associate a flow label with each program value. Labels are explicitly placed on input data and checking policies explicitly declared at output points; for points in between, the type system automatically infers the labels and so programmers do not need to add declarations. Input statements are of the form $\mathtt{read}_{(\mathtt{L_s},\mathtt{L_i})}(\mathtt{fd})$, where $\mathtt{L_s}$ and $\mathtt{L_i}$ are the declared security level policy for secrecy and integrity of the channel, respectively, and `fd` is the file descriptor that names the channel. Similarly, output statements are of the form $\mathtt{write}_{(\mathtt{L_s},\mathtt{L_i})}(\mathtt{e},\mathtt{fd})$. For practicality we also support the ability to downgrade (*declassify*) secrecy labels, and upgrade (*endorse*) integrity labels when deemed safe to do so.

The type inference system provides an expressive form of parametric polymorphism. Polymorphism is crucial for modeling information flows with fine enough granularity. Different objects of the same class (e.g. two completely different `HashSet` objects) may be used in different security contexts, which must be differentiated in the analysis. Otherwise, secure programs may be rejected by a type system that unnecessarily merges flows. In our system, security policies on IO channels are defined at the level of Java `Stream` classes. This allows a `LowOutputStream` class to have a different security requirement than a `HighOutputStream` class. As described in Section 2, our fine-grained polymorphic type inference algorithm is essential for providing a fine enough distinction on IO channels. To demonstrate the correctness of our system, we prove a type soundness result, and we also show a noninterference property, extended to account for interactive inputs and outputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS'07 June 14, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-711-7/07/0006. . . $5.00.

One weakness of Java and other programming languages is how the IO points can get buried in the code through subclassing, method calls, *etc*. This in turn makes it difficult to observe the policies on the use of IO channels 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 knowledge of the code details in order to understand what information flows occur through IO. We describe a simple mechanism that allows users to define concise top-level policies which are then automatically applied to the proper IO points in the program. This reduces the burden on both the programmer as well as the policy validator – the security policy for the whole program is now defined in one place.

The result of our strong type inference system and IO policy declarations is a system for a real language, where programmers need only specify the security policy of IO channels, and the type system ensures the program does not violate the policy.

2. System Overview

Our syntax is based on Middleweight Java (MJ) [4], extended with labeled input and output operations, declassifying and endorse syntax as well as other minor additions. Input and output statements are $\texttt{read}_{(\texttt{L},\texttt{L}')}(\texttt{fd})$ and $\texttt{write}_{(\texttt{L},\texttt{L}')}(\texttt{e},\texttt{fd})$, where `fd` is the file descriptor of the IO channel, `e` is what is written to the output channel, and `L` and $\texttt{L}'$ are sets of labels specifying the secrecy and integrity levels of the channel, respectively. For convenience, we use labels sets and the usual set relations as our security lattice [7].

At the point of a read operation, the returned value is tagged with the security labels of the channel. Further, checks are performed to ensure it is safe to read in the current security context. For example, a low read must not occur under a high guard. Otherwise, an attacker would notice that the amount of data read from a low stream would differ if the high guard differed. For example, one execution may read from a low stream three times, while another execution with a different high guard may read from the stream seven times, indirectly leaking information in the three vs. seven number. At each write, the labels on the value to be put to the channel are checked against the channel policy, to ensure that high secrecy data is not output to a low secrecy channel (and, dually that low integrity data does not flow into a high integrity channel).

Integrity is an important dimension of information flow security, and is generally recognized as a dual to secrecy [3]. To simplify the presentation, we here omit integrity tracking from the formal system, though we include it in examples and discussion, when relevant. Hence, in $\texttt{read}_{\texttt{L}_s}(\texttt{fd})$, the integrity label is omitted. A full treatment of integrity can be found in the technical report [19].

We provide a `Declassify(e,L)` statement, which removes secrecy labels `L` from `e`. This serves to declassify data in infrequent, explicitly allowable instances [15, 24]. 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, because they may reveal too much information and compromise security. Although omitted from the formal system in this paper, the integrity dual, `Endorse(e,L)`, increases the integrity label of the argument, specifying increased confidence in the data.

We define a static constraint-based type inference system, with a form of automatic label polymorphism inference that is related to CPA-style concrete class analyses [1, 23]. The need for label polymorphism inference will become evident when we study the example program of Section 2.2.

2.1 Program Constants and Default Policies

The use of security-critical constants directly in the program text 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 [12, 6], where hard-coded passwords and cryptographic keys resulted in security problems.

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.

Establishing default policies for input and output channels is a closely related problem. This is important for establishing security for programs where not all IO channels have been given a security policy, and in describing policies for the standard input and output streams (`System.in`, `System.out` and `System.err` in Java). The default policy for an input channel is established as low secrecy and low integrity. This means the data is considered public and unreliable, which is a natural default for an unknown channel. The default policy for an output channel is also low secrecy and low integrity. This means the channel is considered observable to public users, and does not require any degree of confidence in the integrity of the data being output.

2.2 An Example Java Program

In this section we elaborate on how information flow is controlled at IO points in our system, by the study of a simple example. In the following subsection we then give an overview of our parametric polymorphism and label inference system.

IO channels in Java are created through subclassing, creating classes such as `FileInputStream`, `DataOutputStream`, `SocketInputStream`, etc. We build on this approach by defining different information flow policies via subclassing the core IO classes. In particular, a different subclass is created for each distinct security category of IO. 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. This example is somewhat oversimplified but is short enough to illustrate the key concepts. 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`, and `BR` is `BufferedReader`. Other obvious abbreviations have been made, and some code is omitted for lack of space.

```
class SysFileIS extends FileIS {
  int read() { return read({high,sys},{high,sys})(fd); }}

class UserIS extends IS {
  int read() { return Endorse(super.read(),{high});}}
```

```
class PwdFileOS extends FileOS {
 void write(int v) { write_({high,sys},{high})(v,fd); }}

class PwdFile extends Object {
 String fileName; String tempName;

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

 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 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,{high,sys});
 }
} // end class PwdFile

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

 //read uname,oldpwd,newpwd from a UserIS.

 bool succ = pf.ChangePwd(uname,oldpwd,newpwd);
 if (succ) { System.out.println("Success"); }
 else { System.out.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 three new IO policies, in the classes `SysFileIS` and `UserIS` (for input), and `PwdFileOS` (for output). For `SysFileIS`, the `read` method labels input values with `high` and `sys` for both secrecy and integrity. The `write` method of `PwdFileOS` allows secrecy labels `high` and `sys`, and requires the integrity label `high`, thereby enforcing the policy that only certain data may be written to the password file. The `UserIS` class is defined with an `Endorse` operation, expressing confidence in the integrity of the data on the channel. (Note that IO can occur with other methods such as file `rename`, but we are simplifying a bit in this example). There is a declassification of secrecy labels at the end of the `ChangePwd` method, necessary to allow the success or failure of the program to be output to the screen.

This program shows how code is written in the language; no explicit parametric type declarations are needed, and no label type declarations need to be placed on variables – type parametricity and variable information flow labels are both inferred automatically. So, the underlying Java program only needs to be changed to declare the appropriate IO channels and policies, and to add any needed downgrades and upgrades. The underlying program structure remains largely unchanged, *e.g.* a `SysFileIS` object `sysin` is still accessed via `sysin.read()`, with no need for annotation.

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 the type system would merge the information flows. 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 `UserIS`, and so it may look very difficult to type these methods distinctly. Our solution is to use a polymorphic form of concrete class analysis [1]: 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 [1, 23], and allows the methodology of placing different security policies in different subclasses to be sound yet expressive. The most obvious forms of polymorphic type inference, based on treating each class or interface as polymorphic and not each method and message send, are too weak to properly treat examples such as the `InputStream` mentioned above.

2.3 Polymorphism

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 `top`. In the `main` portion, two different calls are made to `ChangePwd`, one with a `PwdFileOS`, as before, and one to a `TopFileOS`.

```
class TopFileOS extends FileOS {
 void write(int v) { write_({top,high,sys},{top,high})(v,fd);}}

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 directly support this new `ChangePwd` method. 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 `top`.

In addition to the need for polymorphism for discriminating IO streams, we also need polymorphism for code re-use. Code should be reusable in multiple contexts, which may have different information flow policies. This means concretely that library classes and methods must be allowed to be instantiated at multiple security contexts, and the type system must not merge all of the flows.

We illustrate this with the following example of different `HashSet` objects: one holding high data, and the other holding low data.

```
class HighFileIS extends FileIS {
  int read() { return read({high},{high})(fd); }}

class LowFileIS extends FileIS {
  int read() { return read(∅,∅)(fd); }}

class LowFileOS extends FileOS {
  void write(int v) { write(∅,∅)(v,fd); }}

void main() {
  HashSet highSet = new HashSet();
  FileIS hin = new HighFileIS("high_infile");
  int i; int j;
  while(i = hin.read()) { highSet.add(i); }
  HashSet lowSet = new HashSet();
  FileIS lin = new LowFileIS("low_infile");
  while(j = lin.read()) { lowSet.add(i); }
  Iterator lowIt = lowSet.iterator();
  FileOS lowout = new LowFileOS("low_outfile");
  lowout.write(lowIt.next()); }
```

We define two input stream classes, one for reading in high data, and one for low data, and an output stream class for writing low data. The program reads from both high and low streams into separate `HashSet` objects. A value is then taken from the `HashSet` containing low data, and written to the low output channel.

This clearly shows the need for polymorphism over security levels. If the types for these two `HashSet` objects were merged, the program would be rejected, because high data would appear to flow out a low channel. Our system views `HashSet` as polymorphic and the `highSet` and `lowSet` are typed distinctly, so the program typechecks.

3. Types for Data Tracking and Checking

We now present the formal type inference system. 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 MJ type system for normal MJ typechecking not related to information flow. Our label type system is strong enough to handle any valid MJ 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 MJ type system and the label type system.

$$
\begin{array}{lll l}
\mathtt{P} & ::= & \bar{\mathtt{CL}};\bar{\mathtt{s}} & \textit{program}\\
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} & \textit{class}\\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{super}(\bar{\mathtt{e}});\bar{\mathtt{s}}\} & \textit{constructor}\\
\mathtt{M} & ::= & \mathtt{RT}\ \mathtt{m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\bar{\mathtt{s}}\} & \textit{method}\\
\mathtt{RT} & ::= & \mathtt{C} \mid \mathtt{void} & \textit{return type}\\
\mathtt{L} & ::= & \{\bar{\mathtt{l}}\}, \text{where } \mathtt{l} \text{ are unique labels.} & \textit{label}\\
\mathtt{CO} & ::= & \mathtt{c} \mid \mathtt{b} \mid \mathtt{str} \mid \mathtt{null} \mid \mathtt{fd} & \textit{constant}\\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this} \mid \mathtt{CO} \mid \mathtt{e.f} \mid (\mathtt{C})\ \mathtt{e} \mid & \textit{expression}\\
 & & \mathtt{e} \oplus \mathtt{e} \mid \mathtt{pe} \mid \mathtt{Declassify}(\mathtt{e},\mathtt{L}) \mid & \\
 & & \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) & \\
\mathtt{pe} & ::= & \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid & \textit{promotable exp.}\\
\mathtt{s} & ::= & \mathtt{pe};\ \mid \mathtt{if\ e\ then}\ \{\bar{\mathtt{s}}\}\ \mathtt{else}\ \{\bar{\mathtt{s}}\} \mid & \textit{statement}\\
 & & ;\ \mid \{\bar{\mathtt{s}}\} \mid \mathtt{e.f} := \mathtt{e};\ \mid \mathtt{return\ e};\ \mid & \\
 & & \mathtt{write}_{\mathtt{L}}(\mathtt{e},\mathtt{fd}) &
\end{array}
$$

Figure 1. Grammar

3.1 The Language

Our language is an extension of Middleweight Java (MJ) [4]. MJ contains the basic object constructs of Java, including state; it omits some of the more complex features of Java, which allows formal properties to be established. We eliminate local variables, which complexify the operational semantics and proofs, although their typings are a straight forward extension of object fields. We add *constants* (int, bool, string, file descriptor), *operators* (`+`,`-` etc.), in order to better reason about information flows in real programs. We also add low level read and write operations to the language, of the form $\mathtt{read}_{\mathtt{L}}(\mathtt{fd})$ and $\mathtt{write}_{\mathtt{L}}(\mathtt{e},\mathtt{fd})$, where `fd` is the file descriptor of the IO channel, `e` is what is written to the output channel, and `L` is a set of labels specifying the secrecy level of the channel.

We also add a `Declassify(e,L)` construct, which removes the secrecy labels in `L` from those on `e`. The grammar for our Extended MJ (EMJ) language is given in Figure 1.

We assume some familiarity with MJ, and do not reproduce its typing or semantic definitions; see [4] for the details. Note that EMJ follows MJ and types expressions with respect to a global class table, CT, that contains the types of all classes. At the top level a sequence of statement $\bar{\mathtt{s}}$ corresponding to the `main` method is typechecked with respect to this table. In addition to the standard type rules for MJ, we add the type rules corresponding to the EMJ extensions; they are mostly straightforward, and are omitted for lack of space. $\mathtt{read}_{\mathtt{L}}(\mathtt{fd})$ is typed to input an integer and $\mathtt{write}_{\mathtt{L}}(\mathtt{e},\mathtt{fd})$ outputs an integer (`e` has an integer type), while `fd` is of type `FileDescriptor`. For `Declassify(e, L)` , the resulting type of the expression is the same type as `e`, since the label tracking is only handled in the label typing rules.

3.2 Label Types

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

$$
\begin{array}{lll l}
\tau & ::= & \langle\,\mathcal{S},\mathcal{F},\mathcal{A}\,\rangle|t & \textit{types}\\
\mathcal{S} & ::= & \{\bar{\mathtt{l}}\}|s^{\sigma}|\mathcal{S}\cup\mathcal{S}|\mathcal{S}-\mathcal{S}|\mathcal{F}.\mathtt{f}.\mathsf{S}|\emptyset & \textit{secrecy types}\\
\mathcal{F} & ::= & \{\bar{\mathtt{f}}\mapsto\bar{\tau}\}|f^{\sigma}|\mathcal{F}.\mathtt{f}.\mathsf{F}|\emptyset & \textit{field types}\\
\mathcal{A} & ::= & \mathtt{C}|\alpha^{\sigma}|\mathcal{F}.\mathtt{f}.\mathsf{A} & \textit{alpha types}\\
\sigma & ::= & \mathtt{C},\mathtt{m},\bar{\mathcal{A}},\mathcal{A}_t,\mathcal{A}_r \mid \epsilon & \textit{contours}\\
\multicolumn{3}{l}{s^{\sigma},f^{\sigma},\alpha^{\sigma},s_p} & \textit{label variables}\\
t & ::= & \langle\,s^{\sigma},f^{\sigma},\alpha^{\sigma}\,\rangle & \textit{type variables}\\
\kappa & ::= & \bar{t},t_t\xrightarrow{s_p}t_r\backslash\mathcal{C} & \textit{method types}\\
 & & \forall\bar{t'}.\bar{t},t_t\xrightarrow{s_p}t_r\backslash\mathcal{C} & \\
c & ::= & \mathcal{S}<:\mathcal{S}|\mathcal{F}<:\mathcal{F}|\mathcal{A}<:\mathcal{A} & \textit{constraints}\\
 & & |\mathcal{A}.\mathtt{m}(\bar{\tau},\tau_t\xrightarrow{pc}\tau_r) & \\
 & & |\tau<:\mathbf{get}\ \tau|\tau<:\mathbf{set}\ \tau|SC(\mathtt{L},\mathcal{S}) & \\
\mathcal{C} & ::= & \{c\}|\mathcal{C}\cup\mathcal{C}|\emptyset & \textit{constraint sets}\\
pc & ::= & \mathcal{S} & \textit{prog. counter}\\
\multicolumn{4}{l}{\tau<:\tau' \text{ is short for } \mathcal{S}<:\mathcal{S}',\mathcal{F}<:\mathcal{F}',\mathcal{A}<:\mathcal{A}'}
\end{array}
$$

Figure 3. Type Definitions

An object's fields has its own labels, represented by the field type $\mathcal{F}$, which is a mapping of field names to types, $\{\mathtt{f}_1\mapsto\tau_1,\ldots,\mathtt{f}_n\mapsto\tau_n\}$. The individual labels may be accessed by a dot notation: $\mathcal{F}.\mathtt{f}.\mathsf{S}$ is the secrecy label on the `f` field of the object. Primitive constants 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. 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 a different policy, or returns

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

$$\frac{\Gamma, pc \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C} \qquad t \text{ consists of fresh type variables.}}{\Gamma, pc \vdash \mathtt{e.f} : t \backslash \{\langle\, \mathcal{S} \cup \mathcal{F}.\mathtt{f}.\mathsf{S}, \mathcal{F}.\mathtt{f}.\mathsf{F}, \mathcal{F}.\mathtt{f}.\mathsf{A} \,\rangle <: \textbf{get}\, t\} \cup \mathcal{C}}\text{(Field)} \qquad \frac{\Gamma, pc \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc \vdash (\mathtt{C})\ \mathtt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}\text{(Cast)}$$

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

$$\frac{\Gamma, pc \vdash \bar{\mathtt{e}} : \bar{\tau} \backslash \bar{\mathcal{C}} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\,\bar{\mathtt{f}} \qquad \bar{t}, t_r \text{ consist of fresh type variables}}{\Gamma, pc \vdash \mathtt{new\ C}(\bar{\mathtt{e}}) : \langle\, pc, \{\bar{\mathtt{f}} : \bar{t}\}, \mathtt{C} \,\rangle \backslash \bar{\mathcal{C}} \cup \{\mathtt{C.K}(\bar{\tau}, \langle\, pc, \{\bar{\mathtt{f}} : \bar{t}\}, \mathtt{C} \,\rangle \xrightarrow{pc} t_r)\}}\text{(New)}$$

$$\frac{\Gamma, pc \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc \vdash \mathtt{Declassify}(\mathtt{e}, \mathtt{L}) : \langle\, pc \cup (\mathcal{S} - \mathtt{L}), \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}\text{(Declassify)} \qquad \frac{\Gamma, pc \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{e}) : \langle\, \mathcal{S} \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle \backslash SC(\mathtt{L}, \mathcal{S})}\text{(Input)}$$

Figure 2. Label Type Rules for Expressions

different labels. As discussed in Section 2, we need a more expressive form of polymorphism. Our analysis is closely related to Data-polymorphic CPA [23], a variant of CPA [1]. This ensures creation of distinct *contours* (polyinstantiations) when needed to give the type expressivity required for our system, while on the other hand merging enough contours to make sure the analysis terminates.

We use $\mathtt{l}$ to represent a concrete label, and s for label variables in the secrecy domain. Notation $\mathtt{L}$ refers to a set of concrete labels $\{\bar{\mathtt{l}}\}$, and label sets $\mathcal{S}$ 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 abstract fields of an object, 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 $\mathtt{C}$. σ defines the contours necessary for polymorphic method typing, and type variables are extended to allow a contour superscript, (e.g. s^σ) and ϵ represents no superscript. For convenience, we generally omit the superscript on variables when it is unimportant. t denotes a full three-tuple of label types, and is simply short-hand.

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, $\mathcal{S} \cup \mathcal{S}' = \{\bar{\mathtt{l}}, \bar{\mathtt{l}'}\}$ (without repeats). $\mathcal{S} - \mathcal{S}'$ is also equivalent to the obvious set difference when both are concrete label sets. For field access, $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{S}$ is equivalent to $\mathcal{S}_i$, where $\mathtt{f}_i \mapsto \langle\, \mathcal{S}_i, \mathcal{F}_i, \mathcal{A}_i \,\rangle$. A similar equivalence analogously holds for any $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{F}$, or $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{A}$.

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 MJ 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, as discussed in Section 3.2.4.

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

The constraint set, $\mathcal{C}$, contains normal subtyping constraints $<:$ for secrecy, field, and α-types. In addition, check constraints of the form $SC(\mathtt{L}, \mathcal{S})$, for secrecy checks, are placed in $\mathcal{C}$ and the closure process will need to verify their correctness. Method constraints $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc} \tau_r)$ 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, $\forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}$, so they may vary parametrically. This allows distinct contours to be formed for each combination of argument type and call site. We detail this analysis when discussing the constraint closure in section 3.2.5. We proceed by discussing specific elements of the type inference system separately.

3.2.1 Expression Typing

The label type inference rules for expressions are given in Figure 2. Here are a few highlights of the rules. (Const) types constants as label types containing only pc for secrecy, reflecting our view that constants should by default have no secrecy as discussed in section 2.1.

In (Field), we use a **get** constraint to obtain the type of a field access. These constraints are discussed further in section 3.2.3. The secrecy type includes the labels on the field within the object, along with the labels the object itself carries.

In (Invoke), the constraint $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau \xrightarrow{pc \cup \mathcal{S}} t_r)$ is added to the constraint set. $\mathcal{S}$ is added to the program counter, since the execution of method $\mathtt{m}$ depends on the object to which the method is being passed. The method type eventually needs to be looked up in the global label table LT. However, since $\mathcal{A}$ may at this point be of unknown 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 $\mathtt{C}$ are looked up using *fields*. We cannot simply add the types of each argument to the field types, since the constructor may not have this behavior. Thus, fresh type variables are created for each field, and the $\mathcal{F}$ element of the type contains these variables. A constraint is added to capture the call to the constructor, which is similar to a method call. The α-type is given the concrete class name of the object being created. pc is the secrecy label on the new object. Like constants, objects are assumed to have no secrecy by default.

As expected, `Declassify(e,L)` removes `L` from the secrecy labels of `e` in (Declassify).

The type of a $\mathtt{read}_{\mathtt{L}}(\mathtt{e})$ expression contains the security levels of the statement combined with the labels on the file descriptor argument. A secrecy checking constraint is also added to the constraint set. There are two reasons for this. Firstly, the this ensures that low reads are not happening under high guards; as discussed previously, this may cause an information leak (note the type of any sub-expression implicitly contains the types of the program counters, a fact easily shown by structural induction on

$$\frac{\Gamma, pc \vdash \mathtt{e} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, pc \cup \mathcal{S} \vdash \bar{\mathtt{s}}_1 : \langle \mathcal{S}_1, \mathcal{F}_1, \mathcal{A}_1 \rangle \backslash \mathcal{C}' \qquad \Gamma, pc \cup \mathcal{S} \vdash \bar{\mathtt{s}}_2 : \langle \mathcal{S}_2, \mathcal{F}_2, \mathcal{A}_2 \rangle \backslash \mathcal{C}''}{\Gamma, pc \vdash \mathtt{if\ e\ then}\ \{\bar{\mathtt{s}}_1\}\ \mathtt{else}\ \{\bar{\mathtt{s}}_2\} : \langle \mathcal{S} \cup \mathcal{S}_1 \cup \mathcal{S}_2, \emptyset, \mathtt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}''}\text{(If)}$$

$$\frac{}{\Gamma, pc \vdash ; : \langle pc, \emptyset, \mathtt{void} \rangle \backslash \emptyset}\text{(No-op)} \qquad \frac{\Gamma, pc \vdash \mathtt{e} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, pc \vdash \mathtt{e}' : \langle \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle \backslash \mathcal{C}'}{\Gamma, pc \vdash \mathtt{e.f} := \mathtt{e}'; : \langle \mathcal{S} \cup \mathcal{S}', \emptyset, \mathtt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup \{\mathcal{F}.\mathtt{f} <: \textbf{set}\ \langle \mathcal{S} \cup \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle\}}\text{(F-Assign)}$$

$$\frac{\Gamma, pc \vdash \mathtt{s} : \tau \backslash \mathcal{C} \qquad \Gamma, pc \vdash \bar{\mathtt{s}} : \tau' \backslash \mathcal{C}' \qquad \mathtt{s} \neq \mathtt{C\ x}}{\Gamma, pc \vdash \mathtt{s}; \bar{\mathtt{s}} : \tau' \backslash \mathcal{C} \cup \mathcal{C}'}\text{(Seq)} \qquad \frac{\Gamma, pc \vdash \bar{\mathtt{s}} : \tau \backslash \mathcal{C}}{\Gamma, pc \vdash \{\bar{\mathtt{s}}\} : \tau \backslash \mathcal{C}}\text{(Block)} \qquad \frac{\Gamma, pc \vdash \mathtt{e} : \tau \backslash \mathcal{C}}{\Gamma, pc \vdash \mathtt{return\ e}; : \tau \backslash \mathcal{C}}\text{(Return)}$$

$$\frac{\Gamma, pc \vdash \mathtt{e} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, pc \vdash \mathtt{e}' : \langle \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle \backslash \mathcal{C}'}{\Gamma, pc \vdash \mathtt{write_L}(\mathtt{e}, \mathtt{e}') : \langle \mathcal{S} \cup \mathcal{S}', \emptyset, \mathtt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup SC(\mathtt{L}, \mathcal{S} \cup \mathcal{S}')}\text{(Output)} \qquad \frac{\Gamma, pc \vdash \mathtt{e} : \tau \backslash \mathcal{C}}{\Gamma, pc \vdash \mathtt{e}; : \tau \backslash \mathcal{C}}\text{(PE)}$$

Figure 4. Label Type Rules for Statements

e, observing the base cases all add pc to the types). Secondly, if the file descriptor value has a higher label than the channel policy, performing the read may result in a security leak (*e.g.*, two executions that differ only in high inputs may read from different low channels, since the file descriptor for the channel differs).

3.2.2 Statement Typing

The type rules for statements are given in Figure 4. In rule (If), the secrecy type of the condition are added to the respective program counters when typing each branch. (F-Assign) adds a **set** constraint to the constraint set to set the flow of labels into an object field. These constraints are described in section 3.2.3. Typing a $\mathtt{write_L}(\mathtt{e}, \mathtt{e}')$ statement produces a secrecy check constraint to ensure the type of the output aligns with the policy of the channel. The type of the file descriptor is also checked against the policy for the same reasons as `read`, discussed earlier. The remaining rules are straightforward.

3.2.3 Get and Set Constraints

We use **get** constraints when typing fields in (Field), and **set** constraints for field assignment in (F-Assign). Constraint closure rules (Get) and (Set) ensures that values assigned to a field flow to any read-point of the field, while ensuring that no backward-flows occur in the types [23]. For example,

$\mathtt{x} := \mathtt{read}_{\{\mathtt{low}\}}(\mathtt{fd}); \mathtt{z} := \mathtt{x}; \mathtt{z} := \mathtt{read}_{\{\mathtt{high}\}}(\mathtt{fd});$
will not result in `x` having the secrecy type $\{\mathtt{high}\}$.

3.2.4 Class and Program Typing

Type inference rules for typing programs, classes, and methods are found in Figure 5. Typing of a whole program first requires each of the classes to be typed, and these types placed in a label table, LT; then the statements corresponding to `main` are typed using this label table.

Methods and constructors require the type variables to be set in an initial label table in order to support recursive class definitions and mutually recursive methods; hence, the first three rules in Figure 5 create the initial label table with unique label variables for each constructor and method of each class.

The first constructor rule is for classes that pass arguments to the parent's constructor; these classes are not direct subclasses of `Object`, and the second rule is for those that are. Note that the former rule includes a constraint denoting the call to the parent's constructor. The body of each constructor and method is typed with respect to the label variables as found in the initial label table. As previously noted, methods and constructors are given $\forall$ types so that they may vary polymorphically, and these types are instantiated when computing the constraint closure. For this reason, any free type variables occurring in the typing are found and placed in the $\forall$ type as $\bar{t}'$; these are type variables that are local to the method (or constructor) body, and therefore must be properly instantiated during the constraint closure, so the typing will not mix flows for different calls to the same method. Method typing then fills in the constraint types in the full label table, and the constraint $\{\tau <: t_r\}$ is added, since t_r appears in the label table as an abstract indication of the return type of the method, which must be bound by the type of the method body, τ.

Programs are therefore typed by typing each class definition, which types each method definition. $\bar{\mathtt{s}}$, representing `main` is also typed. The *fields* function returns the list of fields for a class, used in the (New) type rule.

3.2.5 Label Closure

The key closure rules for label constraint sets are given in Figure 6, along with some necessary definitions. Rules that add new constraints based on transitivity, obvious set propagations, and field labels have been omitted for space. 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 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 as defined by the function θ.

The manner in which local label variables are replaced defines the *contours* of a concrete class analysis. In other words, different instantiations of the $\forall$ type create unique types that distinguish different method invocations. Our definition of θ creates a new contour for each distinct receiver type `C`, method name `f`, argument type $\bar{A}$ ($\mathcal{A}_t$ is the type of `this`), and return type $\mathcal{A}_r$. This allows calls to be distinguished based on receiver and argument types, as in CPA [1], and additionally distinguishes call-sites based on unique program points. Since the (Invoke) type rule creates fresh variables for each method invocation, this serves as a unique marker of the call-site in the program; thus, $\mathcal{A}_r$ is the call-site of the method. Since constructor calls during (New) are similar to method invocations, the analysis can distinguish most object instances via call-sites and constructor arguments. Consider the following example.

$\mathtt{x} = \mathtt{new\ C}(); \mathtt{y} = \mathtt{new\ C}();$
$\mathtt{x.put}(\mathtt{read}_{\{\mathtt{low}\}}(\mathtt{fd})); \mathtt{y.put}(\mathtt{read}_{\{\mathtt{high}\}}(\mathtt{fd}'))$
$\mathtt{x.get}();$

Here, our analysis produces separate contours for the creation of `x` and `y`, where CPA merges them into one. Even though the `put` calls have different contours, since the types of `x` and `y` are not distinguished, the CPA analysis cannot determine that `x.get()` is low. We obtain more precision, so we can correctly identify the flows of data into and out-of abstract objects on the heap.

This precision is similar to that obtained in data-polymorphic CPA analysis [23]; although DCPA includes many optimizations

Initial Label Table:

$$\frac{\bar{t}, t_t, t_r, s_p \text{ consist of fresh variables. Each use creates distinct variables.}}{\mathit{InitMeth}() = \bar{t}, t_t \xrightarrow{s_p} t_r \backslash \emptyset} \qquad \frac{\bar{t}, t_t, t_r, s_p \text{ consist of fresh variables. Each use creates distinct variables.}}{\mathit{InitCon}() = \bar{t}, t_t \xrightarrow{s_p} t_r \backslash \emptyset}$$

$$\frac{\kappa_i = \mathit{InitCon}() \qquad \bar{\kappa}_i = \mathit{InitMeth}()}{\mathit{InitialLT} = LT[(\mathtt{C}_0, \mathtt{K}) : \kappa_0, (\mathtt{C}_0, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, \ldots]}$$

Constructor Typing:

$$\frac{\begin{array}{c}\mathtt{C}_0 = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{K} = \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{super}(\bar{\mathtt{e}}); \bar{\mathtt{s}}\} \qquad \mathtt{D} \neq \mathtt{Object} \qquad \mathit{InitialLT}(\mathtt{C}_0, \mathtt{K}) : \bar{t}, t_t \xrightarrow{s_p} t_r \backslash \emptyset \\ \Gamma[\bar{\mathtt{x}} : \bar{t}, \mathtt{this} : t_t], s_p \vdash \bar{\mathtt{e}} : \bar{\tau} \backslash \bar{\mathcal{C}} \qquad \Gamma[\bar{\mathtt{x}} : \bar{t}, \mathtt{this} : t_t], s_p \vdash \bar{\mathtt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \mathcal{C}' \cup \{\mathtt{D.K}(\bar{\tau}, t_t \xrightarrow{s_p} t_r)\})\end{array}}{\mathit{InitialLT} \vdash_M (\mathtt{C}_0, \mathtt{K}) : \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \bar{\mathcal{C}} \cup \{\mathtt{D.K}(\bar{\tau}, t_t \xrightarrow{s_p} t_r)\} \cup \{\tau <: t_r\}}$$

$$\frac{\begin{array}{c}\mathtt{C}_0 = \mathtt{class\ C\ extends\ Object}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{K} = \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{super}(); \bar{\mathtt{s}}\} \\ \mathit{InitialLT}(\mathtt{C}_0, \mathtt{K}) : \bar{t}, t_t \xrightarrow{s_p} t_r \backslash \emptyset \qquad \Gamma[\bar{\mathtt{x}} : \bar{t}, \mathtt{this} : t_t], s_p \vdash \bar{\mathtt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C})\end{array}}{\mathit{InitialLT} \vdash_M (\mathtt{C}_0, \mathtt{K}) : \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Method Typing:

$$\frac{\begin{array}{c}\mathtt{C}_0 = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{M} = \mathtt{RT}\ \mathtt{m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\bar{\mathtt{s}}\} \\ \mathit{InitialLT}(\mathtt{C}_0, \mathtt{M}) : \bar{t}, t_t \xrightarrow{s_p} t_r \backslash \emptyset \qquad \Gamma[\bar{\mathtt{x}} : \bar{t}, \mathtt{this} : t_t], s_p \vdash \bar{\mathtt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\})\end{array}}{\mathit{InitialLT} \vdash_M (\mathtt{C}_0, \mathtt{M}) : \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Class Typing:

$$\frac{\mathit{InitialLT} \vdash_M (\mathtt{C}_0, \mathtt{K}) : \kappa_0 \qquad \mathit{InitialLT} \vdash_M (\mathtt{C}_0, \bar{\mathtt{M}}_0) : \bar{\kappa}_0 \qquad \ldots}{\vdash_C LT[(\mathtt{C}_0, \mathtt{K}) : \kappa_0, (\mathtt{C}_0, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, \ldots]}$$

Program Typing:

$$\frac{\begin{array}{c}\vdash_C LT[(\mathtt{C}_0, \mathtt{K}) : \kappa_0, (\mathtt{C}_0, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, \ldots] \qquad \emptyset, \emptyset, u \vdash \bar{\mathtt{s}} : \tau \backslash \mathcal{C} \\ \mathit{Closure}(LT[(\mathtt{C}_0, \bar{\mathtt{M}}_0) : \bar{\kappa}_0, \ldots], \mathcal{C}) \text{ is consistent}\end{array}}{\vdash_P \{\mathtt{C}_0, \mathtt{C}_1, \ldots\};\ \bar{\mathtt{s}} : \tau \backslash \mathcal{C}}$$

Fields:

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

Figure 5. Label Type Rules for Classes and Programs

to combine contours whenever possible, while still supporting data polymorphism. The *flatten* function is necessary to merge contours for recursive calls and to ensure the analysis terminates. We discuss the termination of this algorithm in section 3.2.8.

We define a constraint closure as follows.

DEFINITION 3.1 (Constraint Closure). *$\mathit{Closure}(LT, \mathcal{C})$ is defined as 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.

3.2.6 Inconsistent Constraints

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

DEFINITION 3.2 (Inconsistent Constraints). *An inconsistent constraint is any constraint $SC(\mathtt{L}_s, \mathtt{L}'_s)$, where $\mathtt{L}'_s \not\subseteq \mathtt{L}_s$.*

Note that constraint consistency is defined only on concrete constraint sets, which are formed during the closure after all transitive flows into type variables have been considered. If $\mathit{Closure}(LT, \mathcal{C})$ contains an inconsistent constraint, then the closure is inconsistent, and type inference fails. SC constraints enforce secrecy policies. In the constraint $SC(\mathtt{L}_s, \mathtt{L}'_s)$, $\mathtt{L}_s$ is the secrecy policy of the IO channel, and $\mathtt{L}'_s$ is the set of labels on the data at that point. Proper enforcement of the policy requires the labels on the data to be a subset of the labels on the IO channel. For example, the constraint $SC(\{\mathtt{high}, \mathtt{low}\}, \{\mathtt{low}\})$ is consistent, with low data flowing to a high channel; $SC(\{\mathtt{low}\}, \{\mathtt{high}\})$ is inconsistent, since high data is flowing to a low channel.

3.2.7 Example Typing

To illustrate our type inference algorithm, consider the following code segment, where `hin`, `lin`, and `lowout` are as defined in the `HashSet` example in section 2.3; `C` is a class with a field `x`.

```
lowC = new C(0); highC = new C(0);
lowC.x = lin.read(); highC.x = hin.read();
lowout.write(lowC.x);
```

The relevant constraints in the label table are as follows.

$(\mathtt{HiFIS}, \mathtt{read}) : \forall t, t_t \xrightarrow{s_p} t_r \backslash \{\langle \ldots \{\mathtt{high}\}, \emptyset, \mathtt{int} \rangle <: t_r\} \ldots$

$(\mathtt{LowFIS}, \mathtt{read}) : \forall t', t'_t \xrightarrow{s'_p} t'_r \backslash \{\langle \ldots \emptyset, \emptyset, \mathtt{int} \rangle <: t'_r\} \ldots$

$(\mathtt{LowFOS}, \mathtt{write}) : \forall t_v, t''_t \xrightarrow{s''_p} t''_r \backslash SC(\emptyset, \cdots \cup s_v) \ldots$

When each `new C(0)` expression is typed, (New) gives them each distinct types for the field; the type of `lowC` is $\langle \emptyset, \{\mathtt{x} : t_l\}, \mathtt{C} \rangle$ and `highC`'s type is $\langle \emptyset, \{\mathtt{x} : t_h\}, \mathtt{C} \rangle$.

Using (Invoke), the typing of `hin.read()` creates a fresh set of variables for the return value, t_{rh}, and generates a method constraint that closure rule (Method), using the above defined label table, instantiates to $\langle \ldots \{\mathtt{high}\}, \emptyset, \mathtt{int} \rangle <: t_{rh}$.

By (F-Assign) and the relevant closure rules, we have $\{\mathtt{x} : t_h\}.\mathtt{x} <: \textbf{set}\ t_{rh}$, which by (Set) entails $t_{rh} <: \{\mathtt{x} : t_h\}.\mathtt{x}$, so by transitivity and field access, we get $\{\mathtt{high}\} <: s_{rh}$ and so $\{\mathtt{high}\} <: s_h$ (recall each t is a triple of type variables, $\langle s, f, \alpha \rangle$).

Note that a new type variable is used for each method invocation, so s_{rh} is unique to the call `hin.read()`, so the `high` label does *not* pollute the low call `lin.read()`; therefore, `high` does *not* flow into s_l.

Closure Rules:

$$\frac{\tau <: \textbf{get}\ t}{\tau <: t}\text{(Get)} \qquad \frac{\tau <: \textbf{set}\ \tau'}{\tau' <: \tau}\text{(Set)} \qquad \frac{\begin{array}{c}\mathtt{C} <: \mathcal{A} \quad \mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc} \tau_r) \quad mtype(\mathtt{C},\mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \\ \bar{\tau} = \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \quad \tau_t = \langle \mathcal{S}_t, \mathcal{F}_t, \mathcal{A}_t \rangle \quad \tau_r = \langle \mathcal{S}_r, \mathcal{F}_r, \mathcal{A}_r \rangle \\ \bar{t''} = \theta(\bar{t'}, \mathtt{C}, \mathtt{m}, \bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r)\end{array}}{[\bar{t'} \mapsto \bar{t''}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau_t, t_r \mapsto \tau_r]\mathcal{C}}(Method)$$

Auxiliary Definitions:

$$\frac{LT(\mathtt{C},\mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}}{mtype(\mathtt{C},\mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}} \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \bar{\mathtt{M}}}{mtype(\mathtt{C},\mathtt{m}) = mtype(\mathtt{D},\mathtt{m})}$$

$$\theta(t, \mathtt{C}, \mathtt{m}, \bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r) = t^{\mathtt{C},\mathtt{m},flatten(\bar{\mathcal{A}},\mathcal{A}_t,\mathcal{A}_r)} \qquad flatten(\mathcal{A}^{\sigma}) = \mathcal{A} \qquad flatten(x, y, \dots) = flatten(x), flatten(y), \dots$$

Figure 6. Label Closure Rules and Definitions

`lowC.x` produces a constraint $s_l <: s_f$, where s_f is created during (Field), and closure rule (Get) introduces this constraint. So, `lowout.write(lowC.x)`; produces a method type, which when instantiated with $(Method)$ gives $SC(\emptyset, \emptyset \cup s_f)$, since the type of the program counter, the object `lowout`, and the field `fd` of `lowout` are all $\emptyset$, and s_f was substituted for s_v. Now, we have $\emptyset <: s_l$ from `lowC.x` $=$ `lin.read()`, so $\emptyset <: s_f$ by transitivity, which leads to the constraint $SC(\emptyset, \emptyset)$, which is consistent.

Suppose we added the statement `lowout.write(highC.x)`; to the program. Then the constraint $s_h <: s'_f$ is produced by (Field) and (Get). So, `lowout.write(highC.x)`; yields a method type, which when instantiated with $(Method)$ gives $SC(\emptyset, \emptyset \cup s'_f)$. As noted above, we have $\{\mathtt{high}\} <: s_h$, so $\{\mathtt{high}\} <: s'_f$ by transitivity, which leads to the constraint $SC(\emptyset, \{\mathtt{high}\})$, which is inconsistent.

3.2.8 Typing Complexity and Termination

A potential pitfall of this form of type inference algorithm is non-termination, if contours are continually created for recursive method invocations.Our analysis merges contours for recursive calls, ensuring termination. We now address the complexity of type inference and constraint closure computation.

Inferring types completes in linear time. Closing the constraint set can be exponential in the worst case. This is evident from the definition of θ. t inputs to θ are all flat (i.e. have no superscript), since they are the free type variables that occur when typing method `m` of class `C`. Superscripted variables are only added during the closure. This means t is bounded by n, the size of the program. Since $\bar{\mathcal{A}}, \mathcal{A}_t$, and $\mathcal{A}_r$ are all flattened, the number of possibilities for these values is bounded by the number of concrete classes and the number of fresh variables created in the program, which are each less than n. Thus, in the worst case, we may create up to n^{n^5} contours (accounting also for `C` and `m`). This is a large exponential, but nevertheless terminates. Many optimizations (*e.g.* combining contours and constraint garbage collection) can be performed to make this practical, as shown in [23] and elsewhere; this is out of the scope of the current work.

The type inference system provides separate compilation of classes, since type inference can be done separately, and the final global constraint set must be closed and checked for inconsistencies. Classes and methods are analyzed only once, and their types and constraints built into the label table, which may be re-used for any number of programs.

4. Soundness and Noninterference

We now state the soundness and noninterference properties for our system. Soundness means that well-typed programs will not produce any run-time check failures. Due to space restrictions, we omit the semantic definitions and formal proofs of soundness and noninterference, which may be found in the companion technical report [19]. In proving noninterference, we assume expressions and statements do not contain any `Declassify(e', L)` subexpressions, which would violate the property that *high* inputs do not affect *low* outputs. The formal statement of noninterference follows.

THEOREM 4.1 (Noninterference). *Given class table CT and label table LT, if $\emptyset, \emptyset \vdash \bar{\mathtt{s}} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, and $Closure(LT, \mathcal{C})$ is consistent, and $\iota_1 \simeq_{\mathtt{Low}} \iota'_1$, and $\omega_1 \simeq_{\mathtt{Low}} \omega'_1$, and $\bar{\mathtt{s}}, \emptyset, \iota_1, \omega_1 \rightarrow^* \mathtt{c}, H_2, \iota_2, \omega_2$ and $\bar{\mathtt{s}}, \emptyset, \iota'_1, \omega'_1 \rightarrow^* \mathtt{c}', H'_2, \iota'_2, \omega'_2$, then $\iota_2 \simeq_{\mathtt{Low}} \iota'_2$ and $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$, and $\mathtt{c} = \mathtt{c}'$.*

Theorem 4.1 states that for a typeable program, any two runs of the program differing only in high input streams will produce equivalent low input and output streams (and equivalent termination values of the executions). The low input streams must be equal since the size of the low input streams after computation may convey secret information, as discussed in section 2. ι and ω define sets of input and output streams, respectively; the relation $\simeq_{\mathtt{Low}}$ describes the low equivalency of the sets of streams. The empty sets before the turnstyle indicate that the typing environment and secrecy program counter are empty in the original typing. We specify that the values must be integers for this theorem, as it intuitively doesn't make sense to input or output heap locations (pointers). Since our system is termination-insensitive, both runs of the program are assumed to terminate normally.

The proof [19] uses a labeled operational semantics, which maps the labels from the typing onto each sub-expression. We then prove, via bisimulation, that all *low* execution steps are identical for both executions, while *high* steps may differ. We then show how reductions in the labeled semantics are isomorphic to reductions in an unlabeled semantics.

5. Top-level Policies

In this section, we present a system for declaring class-based policies at the top level of a program, meaning the policy will not be buried in the code, but can be seen in the API. This also provides 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, as shown in Figure 7. Given a valid program and a top-level policy, the translation produces a new program with security levels on `read` and `write` expressions of `InputStream` and `OutputStream` subclasses, and `Declassify` statements on method return values, when downgrading is warranted. 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.

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{class C} : (S, I) \quad \text{where } \texttt{C} <: \texttt{InputStream} \text{ and } \texttt{FileDescriptor fd} \text{ is a (private) field of } \texttt{C}.} \\
\quad \texttt{int read}()\ \{\bar{\texttt{s}}\} & \Rightarrow & \texttt{int read}()\ \{\texttt{return read}_{(S,I)}(\texttt{fd});\}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{class C} : (S, I) \quad \text{where } \texttt{C} <: \texttt{OutputStream} \text{ and } \texttt{FileDescriptor fd} \text{ is a (private) field of } \texttt{C}.} \\
\quad \texttt{void write}(\texttt{e})\ \{\bar{\texttt{s}}\} & \Rightarrow & \texttt{void write}(\texttt{e})\ \{\texttt{write}_{(S,I)}(\texttt{e}, \texttt{fd});\}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{class C, method RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}}) : \texttt{Declassify(L)} \quad \text{where } \texttt{RT} \neq \texttt{void}} \\
\quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return e};\} & \Rightarrow & \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return Declassify}(\texttt{e}, \texttt{L});\}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{class C, method RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}}) : \texttt{Endorse(L)} \quad \text{where } \texttt{RT} \neq \texttt{void}} \\
\quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return e};\} & \Rightarrow & \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return Endorse}(\texttt{e}, \texttt{L});\}
\end{array}$$

Figure 7. Top-level Policy Translation

read policies declare the sets of security labels for an input channel using the Java representation of an InputStream subclass, C. Hence, the read method of C is re-written to perform a low level read operation with the security labels given by the policy. In a similar manner, write policies declare the sets of security labels for an output channel using the Java representation of an OutputStream subclass. The write method is re-written to perform a low level write operation with the security labels given by the policy. Notice we require both the InputStream and OutputStream subclasses to have a file descriptor as a (private) field. While the abstract classes InputStream and OutputStream do not have such a requirement, usable stream classes do, such as FileInputStream. In the Java implementation, low-level reads and writes are actually native methods. It is these low-level methods that we are re-defining. (In actuality, there is some variation in the Java implementations of various Stream classes. For example, FileInputStream uses an additional private native method readBytes for low-level reads of multiple bytes. For full Java, we would need to define additional translations to satisfy these inconsistencies, though the policy format would be the same.)

Any sub-classes of InputStream and OutputStream that do not have a defined policy receive the default policy, described earlier. Hence all unspecified input streams are low secrecy and low integrity; the default policy for an output stream is also low secrecy and low integrity.

Declassify statements specify what labels will be declassified from a 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. Declassify statements can only be applied to methods with non-void return types, since it is the value that is returned from the method that is declassified. Endorse statements are defined analogously for integrity upgrading. Note that MJ requires that methods only have one return statement, at the end of the method body. Generalizing the language to other return statements requires the translation to be applied to any return statement within a method body.

Since these top-level policies automatically insert the relevant security portions into the code, they may be used for defining policies at deployment time. This allows the users deploying an application to tailor it to fit their own security requirements. However, declassification (and endorsement) is a delicate issue that usually requires some inspection of the code, so automatic insertion of declassify is generally a bad idea. Security requirements should remain part of the software development process, from design through deployment. We plan to continue exploring these software engineering challenges in future work.

5.1 Example Top-level Policies

The following is a top-level policy for the program for changing passwords in section 2.2.

```
class SysFileIS: ({high,sys},{high,sys})
class UserIS
 read(): Endorse({high})
class PwdFileOS: ({high,sys},{high})
class PwdFile
 ChangePwd(String uname,oldpwd,newpwd):
  Declassify({high,sys})
```

Supposing the program of Section 2.2 had all of the explicit information flow labels, checks, and declassifications removed, to give a regular Java program; if the above policy were then applied to that stripped program, we would obtain exactly the program presented in Section 2.2 again. 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 a fine-grained information flow policy: a unique subclass needs to be defined for each different IO security policy. This can be viewed as a good step, because it leads to a more object-oriented information flow policy.

6. Related Work

Static analysis of information flow control systems is a well-studied area [10, 22, 2]; Sabelfeld and Myers present a survey in [18]. Much of the literature focuses on proving formal results for small programming languages, though there has been some effort to define working systems. Flow Caml [17] is an information flow extension to Core ML. The Jif system provides information flow control for full Java [14].

O'Neill *et. al.* describe an information flow security model for interactive IO using a simple imperative language [16]. They demonstrate that a simple type system can be used to obtain noninterference in an interactive setting involving user strategies, then expand the model to incorporate nondeterministic choice. In comparison, our system provides security for Middleweight Java, a much larger language. A smaller language allows their type system to be much simpler, and they do not describe an IO-based inference mechanism, as we do, and polymorphism is not a concern since their language does not allow methods. To our knowledge, this is the only other information flow type system that formally models interactive IO.

Jif [14] is unique as an information flow system since it covers essentially the full Java language, but it lacks a formal analysis. Checks on IO channels are intermixed with the many 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 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.

In contrast, our type system infers all label types and parametric types, removing the need for additional program annotations. Our label types are inferred for existing code, meaning libraries can be used as is, provided the proper labels and checks are placed on the IO points in the program. Our concrete class analysis [1, 23] 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 [2] prove a batch-model noninterference property 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 [20]. This form of polymorphism resembles Jif's, 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 interactive noninterference using an extensible operational approach.

Flow Caml [17] provides label type inference and parametric polymorphism for an information flow extension to Core ML. They prove soundness of type inference and a batch-model noninterference property. Our type system is significantly different, since it is based on an object-oriented language, which presents unique issues, (i.e. inheritance and dynamic dispatch) that do not arise in a functional language.

Hammer *et. al.* describe how program dependence graphs (PDGs) may be used for information flow control [9]. Although the technical methods used are significantly different than our approach, the expressive power is roughly similar: our polymorphic types are approximately matched by the context-sensitivity in their analysis for example. Their system incorporates flow-sensitivity and we do not. The advantage of a type-based approach is that it provides a much more succinct definition; for example, our formal system is completely defined in this short paper and they do not completely define a formal system in their paper.

Several works have developed policies for downgrading data. One approach is for the labels to contain downgrading policies which describe when it is safe to declassify the data, whether after a certain method call, operation, or some other property [13, 5]. In comparison, our policies for downgrading (and upgrading) are attached to the methods, similar to Hicks *et. al.*'s notion of declassifiers [11]. 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.

7. Conclusion

We have presented a static information flow type inference system for Middleweight Java and formally proved its correctness. Our type system provides a high level of polymorphism to promote IO-based policies and code re-use in multiple security contexts. We provide a top-level policy description, which automatically inserts information flow controls in a program and clarifies the policy in the API. Changes to Java programs are therefore minor, as only the underlying IO operations change. Type inference and easily identifiable policies are a necessary step towards a more usable information flow system.

References

[1] Ole Agesen. The cartesian product algorithm. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[2] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2002.

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

[4] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[5] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.

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

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

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

[9] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[10] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *the 25th ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[11] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: high-level policy for a security-typed language. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[12] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, 2004.

[13] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[14] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, 1999.

[15] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles (SOSP)*, 1997.

[16] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *the 19th IEEE Workshop on Computer Security Foundations (CSFW)*, 2006.

[17] François Pottier and Vincent Simonet. Information flow inference for ML. In *the 29th ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

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

[19] Scott F. Smith and Mark Thober. Improving usability of information flow security in Java. Technical report, Johns Hopkins University, March 2007. `http://www.cs.jhu.edu/~mthober/`.

[20] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *the Eleventh International Static Analysis Symposium (SAS)*, 2004.

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

[22] 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.

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

[24] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *the 14th IEEE Workshop on Computer Security Foundations (CSFW)*, 2001.