Flexible Dynamic Information Flow Control in Haskell

Deian Stefan\(^1\) Alejandro Russo\(^2\) John C. Mitchell\(^1\) David Mazières\(^1\)

(1) Stanford University, Stanford, CA, USA
(2) Chalmers University of Technology, Gothenburg, Sweden
{deian,mitchell}@cs.stanford.edu russo@chalmers.se

Abstract
We describe a new, dynamic, floating-label approach to language-based information flow control, and present an implementation in Haskell. A labeled IO monad, LIO, keeps track of a current label and permits restricted access to IO functionality, while ensuring that the current label exceeds the labels of all data observed and restricts what can be modified. Unlike other language-based work, LIO also bounds the current label with a current clearance that provides a form of discretionary access control. In addition, programs may encapsulate and pass around the results of computations with different labels. We give precise semantics and prove confidentiality and integrity properties of the system.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features—Modules, packages

General Terms Security, Languages, Design

Keywords Information flow control, Monad, Library

1. Introduction
Complex software systems are often composed of modules with different provenance, trustworthiness, and functional requirements. A central security design principle is the principle of least privilege, which says that each component should be given only the privileges it needs for its intended purpose. In particular, it is important to differentially regulate access to sensitive data in each section of code. This minimizes the trusted computing base for each overall functionality of the system and limits the downside risk if any component is either maliciously designed or compromised.

Information flow control (IFC) tracks the flow of sensitive data through a system and prohibits code from operating on data in violation of security policy. Significant research, development, and experimental effort has been devoted to static information flow mechanisms. Static analysis has a number of benefits, including reduced run-time overhead, lower run-time failures, and robustness against implicit flows [10]. However, static analysis does not work well in environments where new classes of users and new kinds of data are encountered at run-time. In order to address the needs of such systems, we describe a new, dynamic, floating-label approach to language-based information flow control and present an implementation in Haskell.

Our approach uses a Labeled type constructor to protect values by associating them with labels. However, the labels themselves are typed values manipulated at run-time, and can thus be created dynamically based on other data such as a username. Conceptually, at each point in the computation, the evaluation context has a current label. We use a labeled IO monad, LIO, to keep track of the current label and permit restricted access to IO functionality (such as a labeled file system), while ensuring that the current label accurately represents an upper bound on the labels of all data observed or modified. Unlike other language-based work, LIO also bounds the current label with a current clearance. The clearance of a region of code may be set in advance to impose an upper bound on the floating current label within that region. This restricts data access, limits the amount of code that could manipulate sensitive data, and reduces opportunities to exploit covert channels. Finally, we introduce an operator, toLabeled, that allows the result of a computation that would have raised the current label instead to be encapsulated within the Labeled type.

The main features of our system can be understood using the example of an online conference review system, called λChair. In this system, which we describe more fully later in the paper, authenticated users can read any paper and can normally read any review. This reflects the normal practice in conference reviewing, for example, where every member of the program committee can see submissions, their reviews, and participate in related discussion. Users can be added dynamically and assigned to review specific papers. In addition, as an illustration of the power of the labeling system, integrity labels are used to make sure that only assigned reviewers can write reviews for any given paper. Conversely, confidentiality labels are used to manage conflicts of interest. Users with a conflict of interest on a specific paper lack the privileges, represented by confidentiality labels, to read a review. As conflicts of interest are identified, confidentiality labels on the papers may change dynamically and become more restrictive. It is also possible to remove conflicts of interest dynamically, if desired. A subtlety that we have found advantageous is that reviewers with a conflict of interest can potentially refer to reviews (by having a name that is bound to a view) but cannot perform specific operations simply because they can refer to them. As we have structured our online conference review system, the actual display of a conflict-of-interest review is a prohibited operation.

The main contributions of this paper include:
- We propose a new design point for IFC systems in which most values in lexical scope are protected by a single, mutable, current label, yet one can also encapsulate and pass around the results of computations with different labels. Label encapsulation is explicitly reflected by types in a way that prevents implicit flows.
- We prove information flow and integrity properties of our design and describe LIO, an implementation of the new model in

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.

Copyright © 2011 ACM 978-1-4503-0860-1/11/09 ... $10.00
Haskell. LIO, which can be implemented entirely as a library (based on type safety), demonstrating both the applicability and simplicity of the approach.

Unlike other language-based work, our model provides a notion of clearance that imposes an upper bound on the program label, thus providing a form of discretionary access control on portions of the code.

IFC originated with military applications [5, 11] that label data and processes with sensitivity security levels. The associated label-checking algorithms then prevent a Trojan horse reading classified data, for example, from leaking the data into less classified files. In operating systems, IFC is generally enforced at the kernel boundary, allowing a small amount of trusted code to impose a flexible security policy on a much larger body of supporting software. Extending the core concepts of IFC to a broader range of situations involving mutually distrustful parties that mix their code and data, Myers and Liskov [28] subsequently introduced a decentralized label model (DLM) that has been the basis of much subsequent OS and language-based work. Unfortunately, despite its attractiveness, the DLM is not widely used to protect data in web applications, for example. In the operating systems domain, most of the past DLM-inspired work has relied exclusively on dynamic enforcement [21, 37, 39]. This is due to the dynamic nature of operating systems, which must support a changing set of users, evolving policies, and dynamically loaded code. But it is often inconvenient to establish security domains by arranging software according to coarse-grained kernel abstractions like processes and files. Moreover, adopting a new OS presents an even bigger barrier to deployment than adopting a new compiler. LIO uses the type system to enforce abstraction statically, but checks the values of labels dynamically. Thanks to the flexibility of dynamic checking, the library implements an IFC mechanism that is more permissive than previous static approaches [26, 30, 31] but providing similar security guarantees [33]. Though purely language-based, LIO explores a new design point centered on floating labels that draw on past OS work. Both the code and technical details omitted in this paper can be found at http://www.scs.stanford.edu/~deian/lio.

2. Security Library

In this section, we give an overview of the information flow control approach used in our dynamic enforcement library for Haskell.

2.1 Labels and IFC

The goal of information flow control is to track and control the propagation of information according to a security policy. A well-known policy addressed in almost every IFC system is non-interference: publicly-readable program results must not depend on secret inputs. This policy preserves confidentiality of sensitive data [15].

To enforce information flow restrictions corresponding to security policies such as non-interference, every piece of data is associated with a label, including the labels themselves. Labels form a lattice [9] with partial order \( \sqsubseteq \) (pronounced “can flow to”) governing the allowed flows. A lattice can be as simple as a few security levels. For instance, the three labels L, H, and T, respectively denoting unclassified, secret and top secret levels, form the lattice \( L \sqsubseteq H \sqsubseteq T \). An IFC system such as our LIO library prohibits a computation running with security level \( H \) from reading top secret data (labeled \( T \)) or writing to public channels (labeled \( L \)). Dual to such confidently secured policies are integrity policies [6], which use the partial order on labels to enforce restrictions on writes.

Our library is polymorphic in the label type, allowing different types of labels to be used. Custom label formats can be created by defining basic label operations: the can flow to label comparison (\( \sqsubseteq \)), a function computing the join of two labels (\( \sqcup \)), and a function computing the meet of two labels (\( \sqcap \)). Concretely, label types are instances of the Label type class:

```haskell
class (Eq l) ⇒ Label l where
  leq :: l → l → Bool -- Can flow to (\( \sqsubseteq \))
  lub :: l → l → l -- Join (\( \sqcup \))
  glb :: l → l → l -- Meet (\( \sqcap \))
```

For any two labels \( L_1 \) and \( L_2 \), the join has the property that \( L_1 \sqsubseteq (L_1 \sqcup L_2) \), \( i = 1, 2 \), while the meet has the property that \( (L_1 \sqcap L_2) \sqsubseteq L_i \), \( i = 1, 2 \). In this section we present examples using the simple three-point lattice introduced above, or a generic/abstract label format. Section 3 details DC labels, a new, practical label format used to implement \( \lambda \)Chair.

Compared to existing systems, LIO is a language-based floating-label system, inspired by IFC operating systems [39, 40]. In a floating-label system, the label of a computation can rise to accommodate reading sensitive data (similar to the program counter (pc) of more traditional language-based systems [34]). Specifically, in a floating-label system, a computation \( C \) with label \( L_C \) wishing to observe an object labeled \( L_R \) must raise its label to the join, \( L_C \sqcup L_R \), of the two labels. Consider, for instance, a simple \( \lambda \)Chair review system computation executing on behalf of a user, Clarice, with label \( L_C \) that retrieves and prints a review labeled \( L_R \), as identified by \( R \):

```haskell
readReview R = do
  \var lv ← retrieveReview R \n  \var pl vr = printLabeledCh \var lv \n
  \var lc = L_C
  \var lr = L_R

  \var lrv = printLabeledCh \var lv
```

The computation label (initially \( L_C \)) is shown in the comments, on the right. Internally, the `retrieveReview` function is used to retrieve the review contents \( \var lv \), raising the computation label to \( L_C \sqcup L_R \) to reflect the observation of confidential data. This directly highlights the notion of a “floating-label”: a computation’s label effectively “floats above” the labels of all objects it observes. Moreover, this implies that a computation cannot write below its label; doing so could potentially result in writing secret data to public channels.

To illustrate the way floating labels restrict data writes, consider the action following the review retrieval: `printLabeledCh \var rv`. The trusted `printLabeledCh` function returns an action that writes the review content \( \var lv \) to an output channel, permitting the output channel label \( L_O \). The output channel label \( L_O \) is dynamically set according to the user executing the computation. Specifically, \( L_O \) is carefully set as to allow for printing out all but the conflicting reviews. For example, if Clarice is in conflict with review \( R \) then \( L_O \) is set such that \( L_R \not\sqsubseteq L_O \). Since the computation label directly corresponds to the labels of the data it has observed, `printLabeledCh` simply checks that the computation label flows to the output channel. In the example above, the trusted function checks that \( L_C \sqcup L_R \not\subseteq L_O \) before printing to (standard) output channel \( O \).

As already mentioned, in contrast to other language-based systems, LIO also associates a clearance with each computation. This clearance sets an upper bound on the current floating label within some region of code. For example, code executing with a secret (\( H \)) clearance can never raise its label to read top secret data (labeled \( T \)). The notion of clearance can also prevent Clarice from retrieving (and not just printing) the contents of a conflicting review \( R \) by setting the computation’s clearance to \( L_O \) such that \( L_R \not\subseteq L_O \). When the action `retrieveReview R` attempts to raise the current label to \( L_O \sqcup L_R \) to retrieve the review contents, the dynamic check will fail because \( L_O \sqcup L_R \not\subseteq L_P \). For flexibility, the output channel label can simply be \( L_O = T \), allowing any information that can be retrieved to be written to the output channel.

Additionally, clearance can be used to prevent Clarice from using termination as a covert channel. For example, the following code can be used to leak conflicting-review information:
leakingRetrieveReview r = do
   rv ← retrieveReview r
   if rv == "Paper ...
   then forever (return rv)
   else return rv

However, using clearance, we prevent such leaks by setting the clearance and review labels in such manner that retrieveReview fails when raising the computation label to retrieve conflicting reviews.

2.2 Library Interface
LIO is a termination-insensitive [2] and flow-sensitive [20] IFC library that dynamically enforces information flow restrictions. At a high level, LIO defines a monad called LIO, intended to be used in place of IO. The library furthermore contains a collection of LIO actions, many of them similar to IO actions from standard Haskell libraries, except that they contain label checks that enforce IFC. For instance, LIO provides file operations like those of the standard libraries, except that they contain label checks that enforce IFC.

To implement the notion of floating label bounded by a clearance, our library provides LIO as a state monad that uses IO as the underlying base monad and it is parametrized by the type of labels. The state consists of a current label \(L_{\text{cur}}\), i.e., the computation's floating label, and a current clearance \(C_{\text{cur}}\), which is an upper bound on \(L_{\text{cur}}\), i.e., \(L_{\text{cur}} \subseteq C_{\text{cur}}\). Specifically, the (slightly simplified) LIO monad can be defined as:

\[
\text{newtype Label l a = LIO (StateT (1, 1) IO a)}
\]

where the state corresponds to the \((L_{\text{cur}}, C_{\text{cur}})\) pair. To allow for the execution of LIO actions, our library provides a function \(\text{evalLIO}()\) that takes an LIO action and returns an IO action which, when executed, will return the result of the IFC-secured computation. It is important to note that untrusted LIO code cannot execute LIO actions by binding IO actions with LIO ones (to bypass IFC restrictions), and thus effectively limits \(\text{evalLIO}\) to trusted code. Additionally, using \(\text{evalLIO}\), (trusted) programmers can easily, though cautiously, enforce IFC in parts of an otherwise IFC-unaware program.

The current label provides means for associating a label with every piece of data. Hence, rather than individually labeling definitions and bindings, all symbols in scope are protected by \(L_{\text{cur}}\) (when a single LIO action is executed). Moreover, the only way to read or modify differently labeled data is to execute actions that internally access restricted symbols and appropriately validate and adjust the current label (or clearance).

However, in many practical situations, it is essential to be able to manipulate differently-labeled data without monotonically increasing the current label. For this purpose, the library additionally provides a \text{Labeled} type for labeling values with a label other than \(L_{\text{cur}}\). A Labeled, polymorphic in the label type, protects an immutable value with a specified label (irrespective of the current label). This is particularly useful as it allows a computation to delay raising its current label until necessary. For example, an alternative approach to the above retrieveReview (called retrieveReviewAll) retrieves the review, encapsulates it as a Labeled value, and returns the Labeled review, leaving the current label unmodified. This approach delays the creeping of current label until the review content, as encapsulated by Labeled, is actually \textit{needed}, for instance, by printLabeledCh.

We note that LIO can be used to protect pure values in a similar fashion as Labeled. However, the protection provided by Labeled allows for serializing labeled values and straightforward inspection by trusted code (which should be allowed to ignore the protecting label). Unlike LIO, Labeled is not a monad. Otherwise, the monad instance would allow a computation to arbitrarily manipulate labeled values without any notion of the current label or clearance, and thus possibly violate the restriction that LIO computations should not handle values below their label and above their clearance. Moreover, such instance would require a definition for a default label necessary when lifting a value with \text{return}. Instead, our library provides several functions that allow for the creation and usage of labeled values within LIO. Specifically, we provide (among other) the following functions:

- \(\text{label} : \text{Label l} \Rightarrow a \rightarrow \text{LIO l (Labeled l a)}\)
- \(\text{unlabel} : \text{Label l} \Rightarrow \text{LIO l a} \rightarrow \text{LIO l a}\)
- \(\text{check} : \text{LIO l (Labeled l a)} \Rightarrow \text{return lRes}\)

The formal semantics of these functions are given in Section 4 (see Figure 4); in this section, we illustrate their functionality and use through examples.

Consider the previous example of \text{readReview}. The internal function retrieveReview takes a review identifier \(R\) and returns the review contents. This implies that, internally, retrieveReview has access to a list of reviews. These reviews are individually protected by a label, where the addition of a new review to the system can be implemented as:

\[
\text{addReview} R L_R rv = \text{do}
   r ← \text{label} L_R rv
   \text{addToReviewList} R r
\]

where the \(\text{addToReviewList}\) simply adds the Labeled review to the internal list. The implementation of retrieveReview directly follows:

\[
\text{retrieveReview} R = \text{do}
   \text{L_R = L_C}
   r ← \text{getFromReviewList} R
   \text{rv ← unlabel r}
   \text{return rv}
\]

where the \(\text{getFromReviewList}\) retrieves the Labeled review from the internal list and unlabel removes the protecting label, and raises the current label to reflect the read.
We previously alluded to an alternative implementation of retrieveReview which, instead, returns the labeled review content while keeping the current label the same. As getFromReviewList is a trusted function and not directly available to untrusted users, such as Clarice, retrieveReviewAlt can be implemented in terms of toLabeled and retrieveReview:

\[
\begin{align*}
\text{retrieveReviewAlt } R \text{ do} & \quad \text{Lout} = L_C \\
& \text{r } \leftarrow \text{toLabeled } (L_C \sqcup L_R) \text{ do} \quad \text{Lout} = L_C \\
& \quad \text{rv } \leftarrow \text{retrieveReview } R \\
& \quad \text{return rv} \\
\text{return r}
\end{align*}
\]

Note that although the current label within the inner computation is raised, the computation’s label does not change—instead the review content is protected by \((L_C \sqcup L_R)\). Hence, only when the review content is actually needed, unlabel can be used to retrieve the content and raise the computation’s label accordingly:

\[
\begin{align*}
\text{readReviewAlt } R \text{ do} & \quad \text{Lout} = L_C \\
& \text{r } \leftarrow \text{readReviewAlt } R \quad \text{Lout} = L_C \\
& \quad \text{Lout} = L'_C \\
& \quad \text{rv } \leftarrow \text{unlabel } r \\
\text{printLabeledCh rv} & \quad \text{Lout} = L'_C \sqcup L_R
\end{align*}
\]

Our library also provides labeled alternatives to IORef and files. Specifically, we provide labeled references to IORefs and files. We previously alluded to an alternative implementation of retrieveReview which, instead, returns the labeled review content while keeping the current label the same. As getFromReviewList is a trusted function and not directly available to untrusted users, such as Clarice, retrieveReviewAlt can be implemented in terms of toLabeled and retrieveReview:

\[
\begin{align*}
\text{retrieveReviewAlt } R \text{ do} & \quad \text{Lout} = L_C \\
& \text{r } \leftarrow \text{toLabeled } (L_C \sqcup L_R) \text{ do} \quad \text{Lout} = L_C \\
& \quad \text{rv } \leftarrow \text{retrieveReview } R \\
& \quad \text{return rv} \\
\text{return r}
\end{align*}
\]

Note that although the current label within the inner computation is raised, the computation’s label does not change—instead the review content is protected by \((L_C \sqcup L_R)\). Hence, only when the review content is actually needed, unlabel can be used to retrieve the content and raise the computation’s label accordingly:

\[
\begin{align*}
\text{readReviewAlt } R \text{ do} & \quad \text{Lout} = L_C \\
& \text{r } \leftarrow \text{readReviewAlt } R \quad \text{Lout} = L_C \\
& \quad \text{Lout} = L'_C \\
& \quad \text{rv } \leftarrow \text{unlabel } r \\
\text{printLabeledCh rv} & \quad \text{Lout} = L'_C \sqcup L_R
\end{align*}
\]

3. \(\lambda\text{Chair}\)

To demonstrate the flexibility of our dynamic information flow library, we present \(\lambda\text{Chair}\), a simple API (built on the examples of Section 2) for implementing secure conference reviewing systems. In general, a conference reviewing system should support various features (and security policies) that a program committee can use in the review process; minimally, it should support:

- **Paper submission**: ability to add new papers to the system.
- **User creation**: ability to dynamically add new reviewers.
- **User login**: a means for authenticating users.
- **Review delegation**: ability to assign reviewers to papers.
- **Paper reading**: means for reading papers.
- **Review writing**: means for writing reviews.
- **Review reading**: means for reading reviews.
- **Conflict establishment**: ability to restrict specific users from reading conflicting reviews.

Even for such a minimal system, a number of security concerns must be addressed. First, only users assigned to a paper may write the corresponding reviews. Second, information from the review of one paper should not leak into a different paper’s review. And, third, users should not receive any information on the reviews of the papers with which they are in conflict. We establish these four policies as non-interference policies for the confidentiality and integrity of reviews. We note that, although enforcing additional security properties is desirable, these four policies are sufficient when implementing a minimalistic and fair review system.

\(\lambda\text{Chair}\)’s API provides the aforementioned security policies by applying information flow control. Following the examples of Section 2, we take the approach of enforcing IFC when writing to output channels, and thus the security for the above policies correspond to that of non-interference, i.e., secret data is not leaked into less secret channels/reviews. We do, however, note that the alternative, clearance restricting approach of Section 2, can be implemented and thus enforce the security policies by confinement rather than non-interference (see Section 5). Before delving into the details of the \(\lambda\text{Chair}\), we first introduce the specific label format used in the implementation.

3.1 DC Labels

\(\lambda\text{Chair}\) is implemented using Disjunction-Category (DC) labels, a new label format especially suitable for systems with mutually distrusting parties. DC labels can be used to express a conjunction of restrictions on data, which allows for the construction of policies that reflect the concern of multiple parties. Such policies are expressed by leveraging the notions of principals and Disjunction Categories (henceforth just categories).

A principal is a string representation of a source of authority such as a user, a group, a role, etc. To ensure egalitarian protection mechanisms, any code is free to create principals.

A category is an information-flow restriction specifying the set of principals that *own* it. Each category is denoted as a disjunction of categories. For example, the DC label \(\{P_1, P_2\}\) written as \([P_1 \lor P_2]\). Additionally, categories are qualified to be secrecy or integrity categories. A secrecy category restricts who can read, receive, or propagate information; an integrity category specifies who can modify a piece of data.

A **DC label** \(L = \langle S, I \rangle\) is a set of secrecy categories and a set \(I\) of integrity categories. All categories must be satisfied in order to allow information to flow and thus we write each set as a conjunction of categories. For example, the DC label \(\langle [P_1 \lor P_2] \land [P_3 \lor P_4] \rangle\) has two secrecy categories and a single integrity category. Data with a DC label \(L_1\) can be propagated to an endpoint having a DC label \(L_2\) if the restrictions imposed by \(L_1\) are upheld by \(L_2\). We formalize this notion using the \(\sqsubseteq\)-relationship as follows.

**Definition 1** (Can flow to). Given any two DC labels \(L_1 = \langle S_1, I_1 \rangle\) and \(L_2 = \langle S_2, I_2 \rangle\), and interpreting each principal as a Boolean variable named according to the content of the string itself, we have

\[
\forall c_1 \in S_1, \exists c_2 \in S_2 : c_2 \Rightarrow c_1 \wedge \forall c_2 \in I_2, \exists c_1 \in I_1 : c_1 \Rightarrow c_2,
\]

where \(\Rightarrow\) denotes Boolean implication.

From now on, when we refer to a principal \(P\), it can be interpreted as a string or Boolean variable depending on the context. As an example of the use of \(\sqsubseteq\)-relationship, the DC label \(\langle [P_1 \lor P_2] \land [P_3 \lor P_4] \rangle\) is a set \([P_1 \land P_2] \lor [P_3 \land P_4]\) since \(P_1 \Rightarrow P_1 \lor P_2, P_3 \Rightarrow P_3 \lor P_4,\) and \(P_4 \Rightarrow P_4 \lor P_3\). Intuitively, the higher we move in the \(\sqsubseteq\)-relationship, the more restrictive the secrecy category becomes, while the integrity category, on the other hand, changes into a more permissive one. Additionally, we note that if a label contains a category that is implied by another, the latter is extraneous, as it has no effect on the value of the label, and can be safely removed.

The join and meet for labels \(L_1 = \langle S_1, I_1 \rangle\) and \(L_2 = \langle S_2, I_2 \rangle\) are respectively defined as follows:

\[
L_1 \sqcup L_2 = \langle \text{reduce}(S_1 \land S_2), \text{reduce}(I_1 \lor I_2) \rangle
\]

\[
L_1 \sqcap L_2 = \langle \text{reduce}(S_1 \lor S_2), \text{reduce}(I_1 \land I_2) \rangle
\]

Here, reduce removes any extraneous categories from a given set and \(\land\) and \(\lor\) denote the conjunction and disjunction of two cate-
category sets viewed as Boolean formulas of principals in conjunctive normal form.

In the context of the well-known DLM [28], a DC label secrecy category of the form $\{P_1 \lor P_2 \lor \cdots \lor P_n\}$ can be interpreted as the (slightly modified) DLM label component/policy $\{P_1, P_2, \ldots, P_n\}$, where principals $P_1, P_2, \ldots, P_n$ are both the owners and readers. Although a DLM component consists of a single owner, which does not need to be part of the reader list, a DC label component (category) consists of multiple owners which are also the (only) readers. Using this slightly modified notion of a label component, a DLM label (set of components) loosely corresponds to our notion of a label (conjunction of disjunctions).

Readers interested in the formal semantics of DC labels and the comparison with DLM can refer to http://www.scs.stanford.edu/~deian/dclabel for further details.

### 3.2 DC Labels in λChair

In this section we describe the data structures and the role of DC labels (from now on just labels) in λChair.

λChair provides an API to build review systems for the functionalities described in Section 3. Intuitively, the API just supplies administrators and reviewers with functions for querying review entries and modifying user accounts. Technically speaking, λChair runs over an underlying state monad that stores information regarding reviews and users.

#### Review entries

A review entry is defined as a record consisting of a paper id, a reference to the corresponding paper, and a reference to the shared review ‘notebook’. Specifically, a review entry is defined as

```plaintext
data ReviewEnt = ReviewEnt { paperId :: Id , paper :: DCRef Paper , review :: DCRef Review }
```

where DCRef is a labeled reference using DC labels. In other words, type DCRef a = Ref DCLabel a. Note that this differs from the examples of Section 2, in which the reviews were simply Labeled types.

#### Reading and writing papers

Upon logging in, users are allowed to read and print out any paper by providing the paper id. The label of the reference paper in the nth-review entry is set to $\{i \cdots \{P_i\}\}$. Observe that the secrecy categories is empty (we interpret the empty category as the true Boolean value), thus allowing any function (without other integrity categories in its label) to read the paper by reading the reference content, i.e., the paper. This label does, however, restrict the modification of the paper to code running in a process that owns the integrity category $P_i$ and can therefore run with the category $\{P_i\}$ in the integrity set of its current label. Only a trusted administrator is allowed to own such principals. Consequently, reviewers’ code cannot modify the paper because their current label (assigned by the trusted login procedure) never includes $P_i$ in their integrity set.

#### Reading and writing reviews

Similarly, reviewers’ code is allowed to access any reviews written to any reference review. However, once a review has been read, its contents must not be written to another paper’s review. We fulfill this requirement by identifying, using labels, when a given piece of code reads a certain review. More specifically, we label the reference review in the ith-review entry as $\{\{R_i\\} \cdots \{\{R_i\}\}\}$. As a consequence, when a function wishes to read the review for entry i, it must raise its current label as to include category $[R_i]$ in its secrecy and integrity sets (clearance permitted). Once a process has been tainted as such, it will not be able to modify the contents of another paper’s review since the integrity category $[R_i]$ will cause the current label’s integrity set to contain $R_i$ in every category and $(R_i \lor C) \nRightarrow R_j$ for any C and $i \neq j$. Consider, for instance, a reviewer’s code that has the current label (by the trusted login procedure) to $\{\{R_i\\} \cdots \{\{R_i\}\}\}$, i.e., in the process of reviewing paper $P_i$. If the code reads another review with label $L_j = \{\{R_j\}\}$, the current label is then updated to $L = \{\{R_i\} \land \{R_j\}\}$, which clearly implies that $L \nsubset L_i$. The integrity category $[R_i]$ restricts the modification of the review to processes that own $R_i$. In this case, however, the process running reviewers’ code, assigned to review paper $i$, contains, at least initially, category $[R_i]$ in the integrity set of its current label.

#### Users

A reviewer is defined as a record consisting of a unique user name, password (used for authentication), and two disjoint sets of paper ids (in our implementation these are simple lists). One set corresponds to the user’s conflicting papers, the second set corresponds to the papers the user has been assigned to review. Concretely, we define a user as follows:

```plaintext
data User = User { name :: Name , password :: Password , conflicts :: [Id] , assignments :: [Id] }
```

A user is authenticated using the name and password credentials. Upon logging in, the code of the reviewer assigned to papers $1 \cdots n$ is executed with the current label initially set to $\{\{\{R_i\} \land \cdots \land \{R_n\}\}\}$, where $R_i$ is the principal corresponding to review entry i. The current clearance is set to $\{\{\\}\}$. The special category set ALL (denoting the conjunction of all possible categories) in the clearance allows the executing code to (raise its current label and) read any data, while the integrity categories in the current label allow the process to only write to assigned reviews. Note, however, that in our case all reviewers append their review to the same review “notebook” and thus a write implies a read. Hence, to allow a reviewer to effectively perform a write-only operation, the process must execute the append function using toLabeled. We note the user is exposed to a function that appends to the review rather than directly writing to it, because multiple users are assigned to review the same paper and one should not be allowed to overwrite the work of another (using privileges a more elegant solution can easily be implemented).

#### Conflicts

Following the readReview examples of Section 2, we restrict the reading, or more specifically, printing of a review to those reviewers in conflict with the paper. Although every user is allowed to retrieve a review, they cannot observe the result unless they first read it to an output channel (in our simple example this corresponds to the standard output). Hence, code running on behalf of a user (determined after logging in) can only write to the output channel (using printLabelCh) if the current label $L_o$ can flow to the output channel label $L_o$. Using the set of conflicting paper ids, for every user, we dynamically assign the output channel label $L_o = \langle S_o, \{\\}\rangle$, where $S_o = \{[R_i] \land \cdots \land [R_n] \land [R_{n+1} \lor CONFLICT] \land \cdots \land [R_N \lor CONFLICT]\}$ and $R_i$, where $i = n + 1, \ldots, N$ are the principals corresponding to all the review entries in the system (at the point of the print) that the authenticated user conflicts with. Here, CONFLICT corresponds to a principal that none of the users own (similar to $P_i$ used in the labels of paper references). For each conflicting paper i, we create a category $[R_i \lor CONFLICT]$. To observe the properties of this label, consider the case when executing code reads a conflicting paper $R_i$. In this situation, the current label is raised to $L = \{\{R_i\} \land \cdots \land [\cdots]\}$, and subsequently when attempting to write to the output channel, it is the case that $L \not\subset L_o$. For $L \subset L_o$ to hold true, there must be a category in $L_o$ that implies $[R_i]$. However, due to the conflict, the only category containing $R_i$ in the channel label’s secrecy category is $[R_i \lor CONFLICT]$ (and clearly $[R_i \lor CONFLICT] \nRightarrow [R_i]$), which asserts that conflicting data cannot flow to the output channel. We further highlight that the channel label permits non-conflicting
reviews \( j \) to be observed by including the corresponding category \( \mathcal{R}_j \) without principal CONFLICT.

### 3.3 Implementation

In this section we present the API provided by \( \chi \). As the main goal of \( \chi \) is to demonstrate the flexibility and power of our dynamic information flow library, we do not extend our example to a full-fledged system; the API can, however, be used to build relatively complex review systems. Below, we present the details of the \( \chi \) functions, which return actions in the RevLIO monad. This monad is a State monad defined using the State monad transformer with LIO as the base monad, and a state consisting of the system users, review entries, and name of the user that the executing code is running on behalf of.

**System administrator interface** A \( \chi \) administrator is provided with several functions that dynamically change the system state. Of these, we detail the most interesting cases below.

- **addPaper** \( \text{::} \) Paper \( \rightarrow \) RevLIO Id
  
  Given a paper, it creates a new review entry for the paper and return the paper id. Internally, **addPaper** uses a function similar to addReview of Section 2.

- **addUser** \( \text{::} \) Name \( \rightarrow \) Password \( \rightarrow \) RevLIO ()
  
  Given a unique user name and password, it adds the new user.

- **addAssignment** \( \text{::} \) Name \( \rightarrow \) Id \( \rightarrow \) RevLIO ()
  
  Given a user name and paper id, it assigns the user to review the corresponding paper. The user must not already be in conflict with the paper.

- **addConflict** \( \text{::} \) Name \( \rightarrow \) Id \( \rightarrow \) RevLIO ()
  
  Given a user name and paper id, it marks the user as being in conflict with the paper. As above, it must be the case that the user is not already assigned to review the paper.

- **asUser** \( \text{::} \) Name \( \rightarrow \) RevLIO () \( \rightarrow \) RevLIO ()
  
  Given a user name, and user-constructed piece of code, it firsts authenticates the user and then executes the provided code with the current label and clearance of the user as described in Section 2. After the code is executed, the current label and clearance are restored and any information flow violations are reported.

**Reviewer interface** The reviewer, or user, composes an untrusted RevLIO computation (or action) that the trusted code executes using **asUser**. Such actions may be composed using the following interface:

- **findPaper** \( \text{::} \) String \( \rightarrow \) RevLIO Id
  
  Given a paper title, it returns its paper id, or fails if the paper is not found.

- **readPaper** \( \text{::} \) Id \( \rightarrow \) RevLIO Paper
  
  Given a paper id, the function returns an action which, when executed, returns the paper content.

- **readReview** \( \text{::} \) Id \( \rightarrow \) RevLIO ()
  
  Given a paper id, the function returns an action which, when executed, prints the review to the standard output. Its implementation is similar to the example of Section 2, except operating on references.

- **appendToReview** \( \text{::} \) Id \( \rightarrow \) Content \( \rightarrow \) RevLIO ()
  
  Given a paper id and a review content, the function returns an action which, when executed, appends the supplied content to the review entry. Since there is no direct observation of the current review content, and to avoid label creep, the function, internally, uses toLabeled.

An IFC violation results in an exception (not-catchable by untrusted code) being thrown (in the semantics presented in Section 4, the program gets “stuck”). Figure 1 shows a simple example using the \( \chi \) API. In this example, Alice is assigned to review two papers. She does so by reading each paper (for the second, she also reads the existing reviews) and appending to the shared review. Bob, on the other hand, is added to the system after Alice’s code is executed. Bob first writes a review for paper 2 and then attempts to violate IFC by trying to read (and write to the output channel) the reviews of paper 1. Though his review is appended to the correct paper, the review of the first paper is suppressed. We finally note that although the example is quite simple, it illustrates the use of the \( \chi \) primitives that may be used to implement a usable paper review system.

### 4. Formal Semantics for LIO

This section formalizes our library for a simple call-by-name\(^1\) \( \lambda \)-calculus extended with Booleans, unit values, pairs, recursion, references, and the monadic operations for LIO. Figure 2 provides

---

\(^1\) For clarity, we use a call-by-name instead of call-by-need calculus; extension to the latter is straightforward, as shown in [23].
the formal syntax of the considered language. Syntactic categories $v$, $e$, and $\tau$ represent terms, expressions, and types, respectively. Terms are side-effect free while expressions denote (possible) side-effecting computations.

In the syntax category $v$, symbol $true$ and $false$ represent Boolean values. Symbol $\bot$ represents the unit value. Symbol $\ell$ denotes security labels. Symbol $a$ represents memory addresses in a given store. Terms include variables ($x$), functions ($\lambda x. e$), tuples ($e_1, e_2$), and recursive functions ($fix e$). Three special syntax nodes are added to this category: $\mathbf{Lb} v e$, $e^{(e_1)}$, and $\bullet$. Node $\mathbf{Lb} v e$ denotes the run-time representation of a labeled value. Similarly, node $e^{(e_1)}$ denotes the run-time representation of a monadic LIO computation. Node $\bullet$ represents an erased term (explained in Section 5). None of these special nodes appear in programs written by users while in a $A B$ computation and thus guaranteeing containment properties (see Section 5). Examples of such terms appear in programs written by users and they are merely introduced for technical reasons.

Expressions are composed of values ($v$), function applications ($e_1 e_2$), pair projections ($\pi_i e, \pi_j e$), conditional branches ($if e \ then \ e \ else \ e$), and local definitions ($let x = e \ in \ e$). Additionally, expressions may involve operations related to monadic computations in the LIO monad. More precisely, $return e$ and $e >>= e$ represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the LIO monad are given by $\mathbf{label} e$, $\mathbf{unlabel} e$, and $\mathbf{toLabeled} e$. Expression $\mathbf{label} e_1 e_2$ creates a labeled value that guards $e_2$ with label $e_1$. Expression $\mathbf{unlabel} e$ acquires the contents of the labeled value $e$ while in a $A B$ computation. Expression $\mathbf{toLabeled} e_1 e_2$ creates a labeled value, with label $e_1$, of the result obtained by evaluating the LIO computation $e_2$. Non-proper morphisms related to creating, reading, and writing of references are respectively captured by expressions $\mathbf{newRef}, \mathbf{readRef}$, and $\mathbf{writeRef}$. Expression $\mathbf{lowerClr} e$ allows lowering of the current clearance to $e$. Expressions $\mathbf{getLabel}$ and $\mathbf{getClearance}$ return the current label and current clearance of an LIO computation. Finally, expressions $\mathbf{labelOf} e$ and $\mathbf{labelOfRef} e$ respectively obtain the security label of labeled values and references.

We consider standard types for Booleans ($\mathit{Bool}$), unit ($\bot$), pairs ($\tau, \tau$), and function ($\tau \to \tau$) types. Value $\ell$ describes security labels. Type $\mathbf{Labeled} \ \ell \ \tau$ describes labeled values of type $\tau$ where the label is of type $\ell$. Type $\mathbf{Lb} e$ represents monadic LIO computations, with a result type $\tau$ and the security labels of type $\ell$. Type $\mathbf{Ref} e \ \tau$ describes labeled references, with labels of type $\ell$, to values of type $\tau$.

The typing judgments have standard form $\Gamma \vdash e : \tau$, such that expression $e$ has type $\tau$ assuming the typing environment $\Gamma$: we use $\Gamma$ for both variable and store typings. The typing rules for several terms are shown in Figure 3; the typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We, however, note that, different from previous work [12, 31], we do not require to use any of the sophisticated features of Haskell’s type-system, a direct consequence of our dynamic approach.

The LIO monad presented in Section 2 is implemented as a State monad. To simplify the formalization and description of expressions, without loss of generality, we make the state of the monad part of a run-time environment. More precisely, for a given LIO computation, the symbol $\Sigma$ denotes a run-time environment that contains the current label, written $\Sigma_{lb, l}$, the current clearance, written $\Sigma_{clr}$, and store, written $\Sigma_{\phi}$. A run-time environment $\Sigma$ and LIO computation form a configuration $(\Sigma, e)$. Given a configuration $(\Sigma, e)$, the current label, clearance, and store when starting evaluation $e$ is given by $\Sigma_{lb, l}, \Sigma_{clr}$, and $\Sigma_{\phi}$, respectively.

The relation $(\Sigma, e) \longrightarrow (\Sigma', e')$ represents a single evaluation step from expression $e$, under the run-time environment $\Sigma$, to expression $e'$ and run-time environment $\Sigma'$. We say that $e$ reduces to $e'$ in one step. We define such relation in terms of a structured operational semantics via evaluation contexts [14].

The reduction rules for the core simply-typed $\lambda$-calculus are standard and therefore omitted. We note that substitution ($[[e_1/x] e_2]$) is defined in the usual way: homomorphic on all operators and renaming bound names to avoid captures. Figure 4 presents the evaluation contexts and non-standard reduction rules for our language. These rules guarantee that programs written using our approach fulfill non-interference, i.e., confidential information is not leaked, and confinement, i.e., a computation cannot access data above the clearance.

The main contribution of our language are the primitives $\mathbf{label}, \mathbf{unlabel}$, and $\mathbf{toLabeled}$. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation and thus guaranteeing containment properties (see Section 5). Rule (UNLAB) provides a method for accessing the content $e$ of a labeled value $\mathbf{Lb} l e$ in LIO computations. When the content of a labeled value is “retrieved” and used in a LIO computation, the current label is raised ($\Sigma' = \Sigma_{lb} l \rightarrow l'$), where $l' = \Sigma_{lb} \cup l$, capturing the fact that the remaining computation might depend on $e$. Rule (TOLAB) deserves some attention. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Expression $\mathbf{toLabeled} l e$ is used to execute the provided LIO computation $e$ until completion $(\Sigma, e) \longrightarrow^* (\Sigma', (v)\Sigma''))$ and wraps its result $v$ into a labeled value with label $l$. Observe that the label $l$ needs to be an upper bound on the current label for the evaluation of computation $e$ ($\Sigma'_{lb} l \subseteq l$). Specifying label $l$ is responsibility of the programmer. The reason for this is due to the fact that security labels are protected by the current label, effectively making them public information accessible to any computation within scope (see rules (GLAB) and (GLABR)). As a consequence, if $\mathbf{toLabeled}$ did not require an upper bound on the data to be observed within $l$, labels can be used to leak information. Recall that the current label and clearance of a given LIO computation can be changed dynamically. To illustrate this point, consider a computation whose current label is $l_0$, taking two

---

**Figure 2** Formal syntax for terms, expressions, and types.

<table>
<thead>
<tr>
<th>Label: $\ell$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: $a$</td>
</tr>
<tr>
<td>Term: $v ::= true</td>
</tr>
<tr>
<td>Expression: $e ::= v</td>
</tr>
<tr>
<td>Type: $\tau ::= \bot</td>
</tr>
<tr>
<td>Store: $\phi ::= \mathbf{Address} \rightarrow \mathbf{Labeled} \ell \tau</td>
</tr>
</tbody>
</table>

---

**Figure 3** Typing rules for terms.

| $\Gamma(a) = \mathbf{Labeled} \ell \tau$ |
| $\Gamma \vdash e : \tau$ |
| $\Gamma \vdash e_1 : \ell$ |
| $\Gamma \vdash e_2 : \tau$ |
| $\Gamma \vdash (e_1, e_2) : \mathbf{LIO} \ell \tau$ |
| $\Gamma \vdash \bullet : \tau$ |

| $\Gamma \vdash e : \tau$ |
| $\Gamma \vdash \ell : \ell' \rightarrow \tau$ |
| $\Gamma \vdash e_1 : \ell$ |
| $\Gamma \vdash (e_1, e_2) : \mathbf{LIO} \ell \tau$ |
| $\Gamma \vdash (e_1, e_2) : \mathbf{LIO} \ell \tau$ |

---

The syntax for and rules of inference for the LIO monad are defined in Figure 2 and Figure 3, respectively. The typing rules for several terms are shown in Figure 3; the typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We, however, note that, different from previous work [12, 31], we do not require to use any of the sophisticated features of Haskell’s type-system, a direct consequence of our dynamic approach.
Figure 4 Semantics for non-standard constructs.

\[ E ::= [\cdot] \mid Lb e \mid E e \mid \pi_i E \mid \text{if } E \text{ then } e \text{ else } e \mid \text{return } E \mid E \gg e \mid \text{label } E \mid \text{unlabel } E \mid \text{toLabeled } E \mid \text{newRef } e \mid \text{readRef } e \mid \text{writeRef } e \mid \text{lowerClr } E \mid \text{labelOf } E \mid \text{labelOfRef } E \]

\[ (\Sigma, E[\text{return } v]) \rightarrow (\Sigma, E[\text{v}]) \]
\[ (\Sigma, E[\text{e}] \gg e_2) \rightarrow (\Sigma, E[e_2]) \]

(LAB)

\[ \Sigma.lbl \subseteq l \subseteq \Sigma.clr \]
\[ (\Sigma, E[\text{label } l \ e]) \rightarrow (\Sigma, E[\text{return } (Lb \ l \ e)]) \]

(UNLAB)

\[ t' = \Sigma.lbl \cup l \quad t' \subseteq \Sigma.clr \quad \Sigma' = \Sigma[l \rightarrow t'] \]
\[ (\Sigma, E[\text{unlabel } (Lb \ l \ e)]) \rightarrow (\Sigma', E[\text{return } e]) \]

(TOLAB)

\[ \Sigma.lbl \subseteq l \subseteq \Sigma.clr \quad (\Sigma, e) \rightarrow (\Sigma', (v)) \quad \Sigma'.lcl \subseteq l \quad \Sigma' = \Sigma[l \rightarrow \Sigma'.lcl \rightarrow \Sigma.clr] \]
\[ (\Sigma, E[\text{toLabeled } l \ e]) \rightarrow (\Sigma', E[\text{return } e]) \]

(NEF)

\[ \Sigma.lbl \subseteq l \subseteq \Sigma.clr \quad \Sigma' = \Sigma.[\phi(a \rightarrow Lb \ l \ e)] \quad \text{a fresh} \]
\[ (\Sigma, E[\text{newRef } l \ e]) \rightarrow (\Sigma', E[\text{return } a]) \]

(RREF)

\[ \Sigma.lcl \subseteq l \subseteq \Sigma.clr \quad \Sigma' = \Sigma[l \rightarrow \Sigma.lcl] \]
\[ (\Sigma, E[\text{readRef } a]) \rightarrow (\Sigma', E[\text{return } e]) \]

(WREF)

\[ \Sigma.lcl \subseteq l \subseteq \Sigma.clr \quad \Sigma' = \Sigma.[\phi(a \rightarrow Lb \ l \ e')] \]
\[ (\Sigma, E[\text{writeRef } a \ e']) \rightarrow (\Sigma', E[\text{return } ()]) \]

(LWCLR)

\[ \Sigma.lcl \subseteq l \subseteq \Sigma.clr \quad \Sigma' = \Sigma[l \rightarrow \Sigma.lcl \rightarrow l] \]
\[ (\Sigma, E[\text{lowerClr } l]) \rightarrow (\Sigma', E[\text{return } ()]) \]

(CLAB)

\[ l = \Sigma.lcl \]
\[ (\Sigma, E[\text{getLabel}]) \rightarrow (\Sigma, E[\text{return } l]) \]

(CCLR)

\[ l = \Sigma.clr \]
\[ (\Sigma, E[\text{getClearance}]) \rightarrow (\Sigma, E[\text{return } l]) \]

(GLAB)

\[ (\Sigma, E[\text{labelOf } (Lb \ l \ e)]) \rightarrow (\Sigma, E[l]) \]
\[ (\GLAB) \]
\[ e = \Sigma.lcl \]
\[ (\Sigma, E[\text{labelOfRef } a]) \rightarrow (\Sigma, E[l]) \]

Figure 5 Erasure function for several terms, memory store and configurations.

\[ e_L(l) = l \]
\[ e_L(a) = a \]
\[ e_L(Lb \ l \ e) = \begin{cases} Lb \bullet & l \not\subseteq L \\ Lb \ l \ e \end{cases} \]
\[ e_L(\phi) = \begin{cases} \{x \mid e_L(\Sigma, \phi(x)) : x \in \text{dom}(\Sigma, \phi)\} \\ \Sigma \phi \rightarrow e_L(\Sigma, \phi) \end{cases} \]
\[ e_L((\Sigma, e)) = \begin{cases} (e_L(\Sigma), \bullet) & \Sigma.lcl \not\subseteq L \\ (e_L(\Sigma), e_L(e)) \end{cases} \]

channel, programmers should provide an upper bound of the current label obtained when \( e \) finishes computing. Since our approach is dynamic, flow-sensitive, and sound, this may require non-trivial, and possibly complicated, static analysis in order to automatically determine the label for each call of toLabeled [32].

By using big-step semantics instead of an evaluation context of the form toLabeled \( l \ E \), rule (TOLAB) does not need to rely on the use of trusted primitives or a stack for (saving and) restoring the current label and clearance when executing toLabeled.

When creating a reference, newRef \( l \ e \) produces a labeled value that guards \( e \) with label \( Lb \ l \ e \) and stores it in the memory store \( (\Sigma' = \Sigma.lcl \rightarrow (\Sigma.lcl \rightarrow l) \) ). The result of this operation is the memory address \( a \) (return \( a \)). Observe that references are created only if the reference's label \( l \) is between the current label and clearance label of the LB monad \( (\Sigma.lcl \subseteq l \subseteq \Sigma.clr) \). The restriction \( l \subseteq \Sigma.clr \) is to ensure that programs cannot manipulate or access data beyond their clearance (see Section 5). Rule (RREF) obtains the content \( e \) of a labeled value \( Lb \ l \ e \) stored in the address \( a \). This rule raises the current label to the security level \( t' = \Sigma.lcl \rightarrow l' \) where \( l' = \Sigma.lcl \rightarrow l \). As in the previous rule, (RREF) enforces that the result of reading a reference is below the clearance \( l' \subseteq \Sigma.clr \). Rule (WREF) updates the memory store with a new value for the reference \( (\Sigma' = \Sigma.lcl \rightarrow Lb \ l \ e') \) as long as the label of the reference is above the current label and it does not exceed the clearance \( \Sigma.lcl \subseteq l \subseteq \Sigma.clr \). If considering \( \Sigma.lcl \) as a dynamic version of the pc the restriction that the label of the reference must be above the current label \( (\Sigma.lcl \subseteq l) \) is similar to the one imposed by [30].

Rule (LWCLR) allows a computation to lower the current clearance to \( l \). This operation is particularly useful when trying to contain the access to some data as well as the effects produced by computations executed by toLabeled. Rules (CLAB) and (CCLR) obtain the current label and clearance from the run-time environment. Finally, rules (GLAB) and (GLABR) return the labels of labeled values and references. Observe that, regardless of the current label and clearance of the run-time environment, these two rules always succeed, effectively making labels public data.

5. Soundness

In this section we show that LB computations satisfy two security policies: non-interference and containment. Non-interference shows that secrets are not leaked, while containment establishes that certain piece of code cannot manipulate or have access to certain data. The latter policy is similar to the containment policies presented in [4, 24].

5.1 Non-interference

As in [26, 31], we prove the non-interference property by using the technique of term erasure. Intuitively, data at security levels where the attacker cannot observe information can be safely rewritten to the syntax node \( \bullet \). For the rest of the paper, we assume that the at-
tacker can observe data up to security level $L$. The syntactic term $\bullet$, denoting an erased expression, may be associated to any type (recall Figure 3). Function $\varepsilon_L$ is responsible for performing the rewriting for data at security level not lower than $L$. In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(\text{if } E \text{ then } e \text{ else } e') = \text{if } \varepsilon_L(E) \text{ then } \varepsilon_L(e) \text{ else } \varepsilon_L(e')$). In the case of data constructors it is simply the identity function. The two interesting cases for this function are when $\varepsilon_L$ is applied to a labeled value or a given configuration. In such cases, term erasing could indeed modify the behavior of the program. Figure 5 shows the definition of $\varepsilon_L$ for several terms and configurations. A labeled value is erased if the label assigned to it is above $L\varepsilon_L(L b l e v e l) = L b l e v e l$, if $l \not\subseteq L$.

Similarly, the computation performed in a certain configuration is erased if the current label is above $L\varepsilon_L((\Sigma, e)) = (\varepsilon_L(\Sigma), \bullet)$ if $\Sigma.1 b l e v e l \not\subseteq L$.

Following the definition of the erasure function, we introduce a new evaluation relation $\rightarrow_{\varepsilon_L}$ as follows:

$$\langle \Sigma, e \rangle \rightarrow_{\varepsilon_L} \langle \Sigma', e' \rangle$$

Expressions under this relationship are evaluated in the same way as before, with the exception that, after one evaluation step, the erasure function is applied to the resulting configuration, i.e., runtime environment and expression. In that manner, the relation $\rightarrow_{\varepsilon_L}$ guarantees that confidential data, i.e., data not below level $L$, is erased as soon as it is created. We write $\rightarrow_{\varepsilon_L}$ for the reflexive and transitive closure of $\rightarrow_{\varepsilon_L}$.

Most results that prove non-interference pursue the goal of establishing a relationship between $\rightarrow_{\varepsilon_L}$ and $\rightarrow_{\ast}$ through the erasure function, as highlighted in Figure 1. Informally, the diagram establishes that erasing all secret data, i.e., data not below $L$, and then taking evaluation steps in $\rightarrow_{\varepsilon_L}$ is the same as taking steps in $\rightarrow_{\ast}$ and then erasing all the secret values in the resulting configuration. Observe that if information from some level above $L$ is leaked by $e$, then erasing all secret data and then taking evaluation steps in $\rightarrow_{\varepsilon_L}$ might not be the same as taking steps in $\rightarrow_{\ast}$ and then erasing all the secret values in the resulting configuration.

For simplicity, we assume that the address space of the memory store is split into different security levels and that allocation is deterministic. In that manner, the address returned when creating a reference with level $l$ depends only on the references with level $l$ already in the store. These assumptions are valid in our language since, similar to traditional references in Haskell, we do not provide any mechanisms for deallocation or inspection of addresses in the API. However, when memory allocation is an observable channel, the library could be adapted in order to deal with non-opaque pointers [17].

We start by showing that the evaluation relationship $\rightarrow$ and $\rightarrow_{\varepsilon_L}$ are deterministic. We note that $e = e'$ means syntactic equality between expressions $e$ and $e'$. Equality between run-time environments, written $\Sigma = \Sigma'$, is defined as the point-wise equality between mappings $\Sigma$ and $\Sigma'$.

**Proposition 1** (Determinacy of $\rightarrow$).

$\triangleright$ For any expression $e$ and run-time environment $\Sigma$ such that $\langle \Sigma, e \rangle \rightarrow \langle \Sigma', e' \rangle$, there is a unique term $e'$ and unique evaluation context $E$ such that $e = E[e']$.

$\triangleright$ If $\langle \Sigma, e \rangle \rightarrow_{\varepsilon_L} \langle \Sigma', e' \rangle$ and $\langle \Sigma, e \rangle \rightarrow \langle \Sigma'', e'' \rangle$, then $e' = e''$ and $\Sigma' = \Sigma''$.

*Proof.* By induction on expressions and evaluation contexts.

**Proposition 2** (Determinacy of $\rightarrow_{\varepsilon_L}$). If $\langle \Sigma, e \rangle \rightarrow_{\varepsilon_L} \langle \Sigma', e' \rangle$ and $\langle \Sigma, e \rangle \rightarrow_{\varepsilon_L} \langle \Sigma'', e'' \rangle$, then $e' = e''$ and $\Sigma' = \Sigma''$.

*Proof.* From Proposition 1 and definition of $\varepsilon_L$.

The following proposition shows that the erasure function is homomorphic to the application of evaluation contexts and substitution as well as that it is idempotent.

**Proposition 3** (Properties of erasure function).

1. $\varepsilon_L(E[e]) = \varepsilon_L([e]_E)$
2. $\varepsilon_L(\frac{e}{x}x) = \varepsilon_L(e_x)$
3. $\varepsilon_L(\varepsilon_L(e)) = \varepsilon_L(e)$
4. $\varepsilon_L(\varepsilon_L(E)) = \varepsilon_L(E)$
5. $\varepsilon_L(\varepsilon_L(\Sigma)) = \varepsilon_L(\Sigma)$
6. $\varepsilon_L(\varepsilon_L((\Sigma, e))) = \varepsilon_L((\Sigma, e))$

*Proof.* From the definition of $\varepsilon_L$ and by induction on expressions and evaluation contexts.

The next lemma establishes a simulation between $\rightarrow$ and $\rightarrow_{\varepsilon_L}$ for expressions that do not execute toLabeled.

**Lemma 1** (Single-step simulation without toLabeled). If $\Gamma \vdash e : \tau$ and $\langle \Sigma, e \rangle \rightarrow \langle \Sigma', e' \rangle$ where toLabeled is not executed, then $\Gamma \vdash e : \tau$ and $\varepsilon_L(\langle \Sigma, e \rangle) \rightarrow_{\varepsilon_L} \varepsilon_L(\langle \Sigma', e' \rangle)$.

*Proof.* Subject reduction holds by showing that a reduction step does not change the types of references in the store $\Sigma.\phi$ and then applying induction on the typing derivations. The simulation holds by simple case analysis on $e$.

Using this lemma, we then show that the simulation is preserved when performing several evaluation steps.

**Lemma 2** (Simulation for expressions not executing toLabeled). If $\Gamma \vdash e : \tau$ and $\langle \Sigma, e \rangle \rightarrow_{\ast} \langle \Sigma', e' \rangle$ where there are no executions of toLabeled, then $\Gamma \vdash e : \tau$ and $\varepsilon_L(\langle \Sigma, e \rangle) \rightarrow_{\varepsilon_L} \varepsilon_L(\langle \Sigma', e' \rangle)$.

*Proof.* By induction on $\rightarrow_{\ast}$ and applying Lemma 1.

The reason for highlighting the distinction between expressions executing toLabeled and those not executing it is due to the fact that the evaluation of toLabeled involves big-step semantics (recall rule (TOLAB) in Figure 4). However, the next lemma shows the simulation between $\rightarrow_{\varepsilon_L}$ and $\rightarrow_{\varepsilon_L}$ for any expression $e$, and it is proved by simple induction on the number of executed toLabeled.

**Lemma 3** (Simulation). If $\Gamma \vdash e : \tau$ and $\langle \Sigma, e \rangle \rightarrow_{\varepsilon_L} \langle \Sigma', e' \rangle$ then $\varepsilon_L(\langle \Sigma, e \rangle) \rightarrow_{\varepsilon_L} \varepsilon_L(\langle \Sigma', e' \rangle)$.

*Proof.* By induction on the number of executed toLabeled and applying Lemma 2 for the base case.

**Figure 6** $L$-equivalence for expressions.

$$
\begin{array}{c|c|c}
\text{For any expression } e & \not\subseteq L & \subseteq L \\
\text{if } L b l e v e l & L b l e v e l & L b l e v e l \\
\text{if } L b l e v e l & L b l e v e l & L b l e v e l \\
\text{if } L b l e v e l & L b l e v e l & L b l e v e l \\
\end{array}
$$

We define $L$-equivalence between expressions. Intuitively, two expressions are $L$-equivalent if they are syntactically equal, modulo labeled values whose labels are above $L$. We use $\approx_L$ to represent $L$-equivalence for expressions. Figure 6 shows the definition for labeled values. Considering the simple lattice: $L \subseteq M \subseteq \infty$ and an attacker at level $L$, it holds that $L b L 8 \approx_L L b H 9$, but it does not hold that $L b L 2 \approx_{L b L 3} L b L 3$ or $L b L 8 \approx_{L b M 8} L b M 8$. Recall that labels are protected by the current label, and thus (usually) observable by an attacker — unlike the expressions they protect, labels must be
the same even if they are above $L$. The rest of $\approx_L$ is defined as syntactic equality between constants (e.g., $\texttt{true} \approx_L \texttt{true}$) or homomorphisms (e.g., $e_1 \texttt{ else } e_2 \approx_L e_1' \texttt{ else } e_2'$ if $e \approx_L e_1$, $e_1 \approx_L e_1'$, and $e_2 \approx_L e_2'$).

Since our language encompasses side-effecting expressions, it is also necessary to define $\approx_L$-equivalence between memory stores. Specifically, we say that two run-time environments are $L$-equivalent if an attacker at level $L$ cannot distinguish them:

**Definition 2** ($L$-equivalence for stores),

$$l \subseteq L \lor l' \subseteq L \quad \forall (\Sigma, \phi) \approx_L (\Sigma', \phi') \quad (\Sigma, \phi \approx_L Lb \quad \rightarrow \quad (\Sigma', \phi' \approx_L Lb'))$$

Note that the $L$-equivalence ignores the store references with labels above $L$. Similarly, we define $L$-equivalence for configurations.

**Definition 3** ($L$-equivalence for configurations),

$$e \approx_L e' \quad \Rightarrow \quad \Sigma.\phi \approx_L \Sigma'.\phi'$$

$$\Sigma.1b1 \equiv \Sigma'.1b1 \quad \Sigma.\text{clr} \equiv \Sigma'.\text{clr}$$

$$\Sigma.\phi \approx_L (\Sigma', \phi') \quad \Sigma.1b1 \subseteq L \quad \Sigma.1b1 \subseteq L$$

In the above definition, it is worth remarking that we do not require $\approx_L$ for expressions when the current label is not below $L$. This omission comes from the fact that $f$ and $e'$ would be reduced to $x$ by applying our simulation between $\rightarrow^*$ and $\rightarrow_L$ (recall Figure 5).

The next theorem shows the non-interference property. It essentially states that given two inputs with possibly secret information, the result of the computation is indistinguishable to an attacker. In other words, there is no information-flow from confidential data to outputs observable by the attacker.

**Theorem 1** (Non-interference). Given a computation $e$ (with no $\bullet$), $(\tau)_{\Sigma}^{\tau'}$, or $Lb$ where $\Gamma \vdash e : \textit{Labeled } \ell \rightarrow \textit{LIO } l \rightarrow (\textit{Labeled } \ell' \rightarrow)$, environments $\Sigma_1$ and $\Sigma_2$ where $\Sigma_1.\phi \approx \Sigma_2.\phi = \emptyset$, security label $l$, an attacker at level $L$ such that $l \subseteq L$, then

$$\forall e_1, e_2, (\Gamma \vdash e_1 : \textit{Labeled }\ell \rightarrow)_{i=1,2}$$

$$\land (e_1 \equiv Lb \land e_1')_{i=1,2} \land (\Sigma_1, e_1) \approx_L (\Sigma_2, e_2)$$

$$\land (\Sigma_1, e_1) \rightarrow^* (\Sigma_1', (Lb \ell_1 e_1')_{\Sigma_1})$$

$$\land (\Sigma_2, e_2) \rightarrow^* (\Sigma_2', (Lb \ell_2 e_2')_{\Sigma_2})$$

$$\Rightarrow (\Sigma_1', (Lb \ell_1 e_1')_{\Sigma_1}) \approx_L (\Sigma_2', (Lb \ell_2 e_2')_{\Sigma_2})$$

Observe that even though we assume that the input labeled values $e_1$ and $e_2$ are observable by the attacker ($l \subseteq L$), they might contain confidential data. For instance, $e_1$ could be of the form $Lb \ell (Lb \ell' true)$ where $l' \subseteq L$.

**Proof.** The $L$-equivalence (and, thus, the proof) directly follows by Lemma 3 and determinacy of $\rightarrow_L$.

5.2 Confinement

In this section we present the formal guarantees that LIO computations cannot modify data below their current label and manipulate information above their current clearance. These kind of properties are similar to the ones found in [4, 24].

We start by proving that the current label of a LIO computation does not decrease.

**Proposition 4** (Monotonicity of the current label). If $\Gamma \vdash e : \tau$ and $(\Sigma, e) \rightarrow^* (\Sigma', e')$, then $\Sigma.1b1 \subseteq (\Sigma').1b1$.

Similarly, we show that the current clearance of a LIO computation never increases.

**Proposition 5** (Monotonicity of the current clearance). If $\Gamma \vdash e : \tau$ and $(\Sigma, e) \rightarrow^* (\Sigma', e')$, then $\Sigma.\text{clr} \subseteq (\Sigma').\text{clr}$.

Proposition 4 and 5 are crucial to assert that once a LIO computation reads confidential data, it cannot lower its current label to leak it. Similarly, a computation should not be able to arbitrarily increase its clearance; doing so would allow it to read any data with no access restrictions.

Before delving into the containment theorems, we first define a store modifier that removes all store elements with a label above a given label $l$:

$$(\Sigma.\phi)_{\mu} = \{ (a, Lb l' e') : a \in \text{dom} (\Sigma.\phi) \land l \subseteq l' \}$$

In other words, this retains all the labeled references with a label below $l$, usually the current label.

The first theorem states that LIO computations cannot create labeled values, new locations or modify memory cells below their current label (no-write down).

**Theorem 2** (Containment imposed by the current label). Given labels $l, l_c$, and $l_v$, a computation $e$ (with no $\bullet$, $(\tau)_{\Sigma}^{\tau'}$, or $Lb$) where $\Gamma \vdash e : \textit{Labeled }\ell \tau \rightarrow \textit{LIO }l \rightarrow (\textit{Labeled }\ell' \rightarrow)$, environment $\Sigma[lb1 \mapsto l, clr \mapsto l_v]$ such that $l \subseteq l_c$, and $l_v \subseteq l_v$.

$$\Gamma \vdash e_1 : \textit{Labeled }\ell \tau$$

$$\land e_1 \equiv Lb \ell_1 e_1' \land (\Sigma, e_1) \rightarrow^* (\Sigma', (Lb \ell_1 e_1')_{\Sigma})$$

$$\Rightarrow (\Sigma.\phi)_{\mu} = (\Sigma'.\phi)_{\mu} \land e_1' = e_1'$$

**Proof.** The proof follows directly from Proposition 4, definition of the store modifier and induction on $\rightarrow^*$.}

The theorem simply states that the computation cannot allocate or modify the store below $l$. Moreover the computation should only be able to return a labeled value below its current label that was provided as input, or by capture.

Dual to Theorem 2, the next theorem captures the fact that LIO computations cannot compute on labeled values above their clearance. In other words, LIO computations cannot create, read, and write references or read and create contents for labeled values above the clearance (recall that references store labeled values). Again, we first define a store modifier; in this case, one that removes all store elements below a given clearance as follows:

$$(\Sigma.\phi)_{\mu l} = \Sigma.\phi \setminus \{ (a, Lb l' e) : a \in \text{dom} (\Sigma.\phi) \land l' \subseteq l \}$$

In other words, this retains all the labeled references with a label above $l$, usually the current clearance.

**Theorem 3** (Containment imposed by clearance). Given labels $l, l_c$, and $l_v$, a computation $e$ (with no $\bullet$, $(\tau)_{\Sigma}^{\tau'}$, or $Lb$) where $\Gamma \vdash e : \textit{Labeled }\ell \tau \rightarrow \textit{LIO }l \rightarrow (\textit{Labeled }\ell' \rightarrow)$, environment $\Sigma[lb1 \mapsto l, clr \mapsto l_c]$ such that $l \subseteq l_c$ and $l_v \subseteq l_c$.

$$\Gamma \vdash e_1 : \textit{Labeled }\ell \tau$$

$$\land e_1 \equiv Lb \ell_1 e_1' \land (\Sigma, e_1) \rightarrow^* (\Sigma', (Lb \ell_1 e_1')_{\Sigma})$$

$$\Rightarrow (\Sigma.\phi)_{\mu l_c} = (\Sigma'.\phi)_{\mu l_c} \land e_1' = e_1'$$

**Proof.** Directly from Proposition 5, definition of the store modifier and induction on $\rightarrow^*$.}

6. Related Work

Heintze and Riecke [18] consider security for lambda-calculus where lambda-terms are explicitly annotated with security labels, for a type-system that guarantees non-interference. One of the key aspects of their work consists of an operator which raises the security annotation of a term in a similar manner to our raise of the
current label when manipulating labeled values. Similar ideas of floating labels have been used by many operating systems, dating back to the High-Water-Mark security model [22] of the ADEPT-50 in the late 1960s. Asbestos [13] first combined floating labels with the Decentralized labeled model [28].

Abadi et al. [1] develop the dependency core calculus (DCC) based on a hierarchy of monads to guarantee non-interference. In their calculus, they define a monadic type that “protects” (the confidentiality of) side-effect-free values at different security levels. Though not a monad, our Labeled type similarly protects pure values at various security levels. To manipulate such values, DCC uses a non-standard typing rule for the bind operator; the essence of this operator, in a dynamic setting with side-effectful computations, is captured in our library through the interaction of Labeled, unlabeled, and LIO.

Tse and Zdancewic [36] translate DCC to System F and show that non-interference can be stated using the parametricity theorem for System F. The authors also provide a Haskell implementation for a two-point lattice. Their implementation encodes each security level as an abstract data type constructed from functions and binding operations to compose computations with permitted flows. Since they consider the same non-standard features for the bind operation as in DCC, they provide as many definitions for bind as different type of values produced by it. Moreover, their implementation needs to be compiled with the flag -fallows-undecidable-instances, in GHC. Our work, in contrast, defines only one bind operation for LIO, without the need for such compiler extensions.

Harrison and Hook show how to implement an abstract operating system called separation kernel [16]. Programs running under this multi-threading operating system satisfy non-interference. To achieve this, the authors rely on the state monad to represent threads, monad transformers to present parallel composition, and the resumption monad to achieve communication between threads. Non-interference is then enforced by the scheduler implementation, which only allows signaling threads at the same, or higher, security level as the thread that issued the signal. The authors use monads differently from us; their goal is to construct secure kernels rather than provide information-flow security as a library. Our library is simpler and more suitable for writing sequential programs in Haskell. Extending our library to include concurrency is stated as a future work.

Crary et al. [8] design a monadic calculus for non-interference for programs with mutable state. Similar to our work, their language distinguishes between term and expressions, where terms are pure and expressions are (possibly) effectful computations. Their calculus mainly tracks information flow by statically approximating the security levels of effects produced by expressions. Compared to their work, we only need to make approximations of the side-effects of a given computation when using toLabeled; the state of LIO keeps track of the dynamic security level upper bound of observed data. Overall, our dynamic approach is more flexible and permissive than their proposed type-system.

Pottier and Simonet [30, 35] designed FlowCaml, a compiler to enforce non-interference for OCaml programs. Rather than implementing a compiler from scratch, and more similar to our approach, the seminal work by Li and Zdancewic [25] presents an implementation of information-flow security as a library, in Haskell, using a generalization of monads called Arrows [19]. Extending their work, Tsai et al. [7] further consider side-effects and concurrency. Contributing to library-based approaches, Russo et al. [31] eliminate the need for Arrows by showing an IFC library based solely on monads. Their library defines monadic types to track information-flow in pure and side-effectful computations. Compared to our dynamic IFC library, Russo et al.’s library is slightly less permissive and leverages Haskell’s type-system to statically enforce non-interference. However, we note that our library has similar (though dynamic) functions provided by their SecIO library; similar to unlabeled, they provide a function that maps pure labeled values into side-effectful computations; similar to toLabeled, they provide a function that allows reading/writing secret files into computations related to public data.

Recently, Morgenstern et al. [27] encoded an authorization- and IFC-aware programming language in Agda. Their encoding, however, does not consider computations with side-effects. More closely related, Devriese and Piessens [12] used monad transformers and parameterized monads [3] to enforce non-interference, both dynamically and statically. However, their work focuses on modularity (separating IFC enforcement from underlying user API), using type-class level tricks that make it difficult to understand errors triggered by insecurities. Moreover, compared to our work, where programmers write standard Haskell code, their work requires one to firstly encode programs as values of a specific type.

Compared to other language-based works, LIO uses the notion of clearance. Bell and Na Patul [5] formalized clearance as a bound on the current label of a particular user’s processes. In the 1980s, clearance became a requirement for high-assurance secure systems purchased by the US Department of Defense [11]. More recently, HiStar [39] re-cast clearance as a bound on the label of any resource created by the process (where raising a process’s label is but one means of creating a something with a higher label). We adopt HiStar’s more stringent notion of clearance, which prevents software from copying data it cannot read and facilitates bounding the time during which possibly untrustworthy software can exploit covert channels.

7. Conclusion

We propose a new design point for IFC systems in which most values in lexical scope are protected by a single, mutable, current label, yet one can also encapsulate and pass around the results of computations with different labels. Unlike other language-based work, our model provides a notion of clearance that imposes an upper bound on the program label, thus providing a form of discretionary access control on portions of the code.

We prove information flow and integrity properties of our design and describe LIO, an implementation of the new model in Haskell. LIO, which can be implemented entirely as a library (based on type safety), demonstrates both the applicability and simplicity of the approach. Our non-interference theorem proves the conventional property that lower-level results do not depend on higher-level inputs – the label system prevents inappropriate flow of information. We also prove containment theorems that show the effect of clearance on the behavior of code. In effect, lowering the clearance imposes a discretionary form of access control by preventing subsequent code (within that scope) from accessing higher-level information.

As an illustration of the benefits and expressive power of this system, we describe a reviewing system that uses LIO labels to manage integrity and confidentiality in an environment where users and labels are added dynamically. Although we have use LIO for the XChair API and even built a relatively large web-framework that securely integrates untrusted third-party applications, we believe that changes in the constructs are likely to occur as the language matures. This further supports our library-based approach to language-based security.

An interesting future work consists on extending our library to handle concurrency. Enforcement mechanisms for sequential programs do not generalize naturally to multithreaded programs. In this light, it is hardly surprising that Jif [29], the mainstream IFC compiler, lack support for multithreading. Due to the monadic
structure of LIO programs, we believe it is possible to extend our library to consider concurrency that also addresses termination [2] and internal-timing leaks [38].

Acknowledgments

We thank Alex Aiken and the anonymous reviewers for their insightful comments. This work was funded by DARPA (CRASH and PROCEED), NSF (including a Cybertrust award and the TRUST Center), the Air Force Office of Scientific Research, the Office of Naval Research, and the Swedish research agency VR.

References


