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 function 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, fewer 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.

Haskell'11, September 22, 2011, Tokyo, Japan.

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 review) 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

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 course-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, 32] but providing similar security guarantees [34]. 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`, `M`, and `H`, respectively denoting unclassified, secret and top secret levels, form the lattice $\texttt{L} \sqsubseteq \texttt{M} \sqsubseteq \texttt{H}$. An IFC system such as our LIO library prohibits a computation running with security level `M` from reading top secret data (labeled `H`) or writing to public channels (labeled `L`). Dual to such confidentially 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:

```
class (Eq l) ⇒ Label l where
  leq :: l → l → Bool -- Can flow to (⊑)
  lub :: l → l → l    -- Join (⊔)
  glb :: l → l → l    -- Meet (⊓)
```

For any two labels L_1 and L_2, the join has the property that $L_i \sqsubseteq (L_1 \sqcup L_2), i = 1, 2$, 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 λChair.

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 [33]). 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 λChair 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:

```
readReview R = do              --  L_C
  rv ← retrieveReview R        --  L_C ⊔ L_R
  printLabeledCh rv            --  L_C ⊔ L_R
```

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 `rv`, 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 rv`. The trusted `printLabeledCh` function returns an action that writes the review content `rv` 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 $L_C \sqcup L_R \sqsubseteq L_O$ 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 (`M`) clearance can never raise its label to read top secret data (labeled `H`). 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_P such that $L_R \not\sqsubseteq L_P$. When the action `retrieveReview` R attempts to raise the current label to $L_C \sqcup L_R$ to retrieve the review contents, the dynamic check will fail because $L_C \sqcup L_R \not\sqsubseteq L_P$. For flexibility, the output channel label can simply be $L_O = \top$, 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:

```
leakingRetriveReview 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-insensitive* [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 library, but confining the application to a dedicated portion of the file system where a label is stored along with each file.

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_{cur}, i.e., the computation's floating label, and a *current clearance* C_{cur}, which is an upper bound on L_{cur}, i.e., $L_{\text{cur}} \sqsubseteq C_{\text{cur}}$. Specifically, the (slightly simplified) `LIO` monad can be defined as:

```
newtype Label l ⇒ LIO l a = LIO (StateT (l, l) 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 (`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 `IO` computations by binding `IO` actions with `LIO` ones (to bypass IFC restrictions), and thus effectively limits `evalLIO` to trusted code. Additionally, using `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_{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 `Labeled` type for labeling values with a label other than L_{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 `retrieveReviewAlt`) 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 *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 straight forward 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 `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:

- `label :: Label l ⇒l → a → LIO l (Labeled l a)`
  Given a label l such that $L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and a value v, the action `label` l v returns a labeled value that protects v with l.
- `unlabel :: Label l ⇒Labeled l a →LIO l a`
  Conversely, the action `unlabel lv` raises the current label (clearance permitting) to the join of `lv`'s label and the current label, returning the value with the label removed. Note that the new current label is at least as high as `lv`'s label, thus protecting the confidentiality of the value.
- `toLabeled :: Label l ⇒l → LIO l a →LIO l (Labeled l a)`
  Given a label l such that $L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and an `LIO` action m, `toLabeled` l m executes m without raising L_{cur}. However, instead of returning the result directly, the function returns the result of m encapsulated in a `Labeled` with label l. To preserve confidentiality (see Section 4 for further details), action m must not read any values with a label above l. We can implement `toLabeled` as follows:

  ```
  toLabeled l m = do
    (L'cur, C'cur) ← get      -- Save context
    res ← m                   -- Execute action
    (Lcur, _) ← get           -- Get inner context
    unless (Lcur ⊑ l) fail    -- Check IFC violation
    put (L'cur, C'cur)        -- Restore context
    lRes ← label l res        -- Encapsulate result
    return lRes               -- Return result
  ```

  In monadic terms, `toLabeled` is an environment-oriented action that provides a different context for a temporary bind thread, while `unlabel` is a state-oriented action that affects the current bind thread.
- `labelOf :: Label l ⇒Labeled l a →l`
  If `lv` is a labeled value with label l and value v, `labelOf lv` returns l.

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 `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:

```
addReview R L_R rv = do
  r ← label L_R rv
  addToReviewList R r
```

where the `addToReviewList` simply adds the `Labeled` review to the internal list. The implementation of `retrieveReview` directly follows:

```
retrieveReview R = do          -- Lcur = L_C
  r ← getFromReviewList R      -- Lcur = L_C
  rv ← unlabel r               -- Lcur = L_C ⊔ L_R
  return rv                    -- Lcur = L_C ⊔ L_R
```

where the `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`:

```
retrieveReviewAlt R = do                    -- Lcur = LC
  r ← toLabeled (LC ⊔ LR) $ do              -- Lcur = LC
        rv ← retrieveReview R               -- Lcur = LC ⊔ LR
        return rv                           -- Lcur = LC ⊔ LR
  return r                                  -- Lcur = LC
```

Note that although the current label within the inner computation is raised, the outer 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:

```
readReviewAlt R = do                        --  Lcur = LC
   r ← retrieveReviewAlt R                  --  Lcur = LC
   -- Perform other computations            --  Lcur = L'C
   rv ← unlabel r                           --  Lcur = L'C ⊔ LR
   printLabeledCh rv                        --  Lcur = L'C ⊔ LR
```

Our library also provides labeled alternatives to `IORefs` and files. Specifically, we provide labeled references `Ref l a` that are created with `newRef`, read with `readRef`, and written to with `writeRef`. When creating or writing to a reference with label L_R, it must be the case that $L_{\text{cur}} \sqsubseteq L_R \sqsubseteq C_{\text{cur}}$, while reading raises L_{cur} to $L_{\text{cur}} \sqcup L_R \sqsubseteq C_{\text{cur}}$. The rules for file operations follow identically, however writing to a file also implies observation (since the write can fail) and so the current label is raised in both cases. Finally, though beyond the scope of this paper, the library provides support for *privileges*. Privileges allow `LIO` code to operate under a more permissive $\sqsubseteq$ relation, but still more restricting than simply allowing the execution of arbitrary `IO` actions.

3. λChair

To demonstrate the flexibility of our dynamic information flow library, we present λChair, 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, fourth, 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.

λChair'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 λChair, we first introduce the specific label format used in the implementation.

3.1 DC Labels

λChair 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 its owners; for example, the category owned by principals P_1 and P_2 is written as $[P_1 \vee P_2]$. 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 S 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 $\langle \{[P_1 \vee P_2] \wedge [P_2 \vee P_3]\}, \{[P_4]\} \rangle$ 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 uphold 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

$$\frac{\forall c_1 \in S_1 . \exists c_2 \in S_2 : c_2 \Rightarrow c_1 \qquad \forall c_2 \in I_2 . \exists c_1 \in I_1 : c_1 \Rightarrow c_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle},$$

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 $\langle \{[P_1 \vee P_2] \wedge [P_2 \vee P_3]\}, \{[P_4]\} \rangle \sqsubseteq \langle \{[P_1] \wedge [P_3]\}, \{[P_4 \vee P_6]\} \rangle$ since $P_1 \Rightarrow P_1 \vee P_2$, $P_3 \Rightarrow P_2 \vee P_3$, and $P_4 \Rightarrow P_4 \vee P_6$. 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:

$$L_1 \sqcup L_2 = \langle \mathsf{reduce}(S_1 \wedge S_2), \mathsf{reduce}(I_1 \vee I_2) \rangle$$
$$L_1 \sqcap L_2 = \langle \mathsf{reduce}(S_1 \vee S_2), \mathsf{reduce}(I_1 \wedge I_2) \rangle$$

Here, reduce removes any extraneous categories from a given set and $\wedge$ and $\vee$ denote the conjunction and disjunction of two cate-

gory 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 $[P_1 \vee P_2 \vee \cdots \vee P_n]$ can be interpreted as the (slightly modified) DLM label component/policy $\{P_1, P_2, \ldots, P_n : P_1, P_2, \ldots, P_n\}$, where principals $P_1, \ldots, P_n$ 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

```
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 ith-review entry is set to $\langle\{\}, \{[P_i]\}\rangle$. 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 $\langle\{[R_i]\}, \{[R_i]\}\rangle$. 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 $(R_i \vee C) \not\Rightarrow R_j$ for any C and $i \neq j$. Consider, for instance, a reviewer's code that has the current label set (by the trusted login procedure) to $\langle\{[R_i]\}, \{[R_i]\}\rangle$, i.e., in the process of reviewing paper P_i. If the code reads another review with label $L_j = \langle\{[R_j]\}, \{[R_j]\}\rangle$, the current label is then updated to $L = \langle\{[R_i] \wedge [R_j]\}, \{[R_i \vee R_j]\}\rangle$, which clearly implies that $L \not\sqsubseteq L_j$. 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:

```
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 $1, \ldots, n$ is executed with the current label initially set to $\langle\{\}, \{[R_1] \wedge \cdots \wedge [R_n]\}\rangle$, where R_i is the principal corresponding to review entry i. The current clearance is set to $\langle$`ALL`$, \{\}\rangle$. 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 write 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 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 $S_o = \{[R_1] \wedge \cdots \wedge [R_n] \wedge [R_{n+1} \vee$ `CONFLICT`$] \wedge \cdots \wedge [R_N \vee$ `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 $[R_i \vee$ `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 $L = \langle\{[R_i] \wedge \cdots\}, \{\cdots\}\rangle$, and subsequently when attempting to write to the output channel, it is the case that $L \not\sqsubseteq L_o$. For $L \sqsubseteq L_o$ 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 $[R_i \vee$ `CONFLICT`$]$ (and clearly $[R_i \vee$ `CONFLICT`$] \not\Rightarrow [R_i]$), 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 $[R_j]$ without principal `CONFLICT`.

3.3 Implementation

In this section we present the API provided by λChair. As the main goal of λChair 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 λChair 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 λChair administrator is provided with several functions that dynamically change the system state. Of these, we detail the most interesting cases below.

- `addPaper` :: `Paper` →`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` :: `Name` →`Password` →`RevLIO ()`
  Given a unique user name and password, it adds the new user.
- `addAssignment` ::`Name` →`Id` →`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` ::`Name` →`Id` →`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` :: `Name` →`RevLIO ()` →`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 3.2. 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` :: `String` →`RevLIO Id`
  Given a paper title, it returns its paper id, or fails if the paper is not found.
- `readPaper` :: `Id` →`RevLIO Paper`
  Given a paper id, the function returns an action which, when executed, returns the paper content.
- `readReview` ::`Id` →`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` ::`Id`→`Content`→`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 λChair 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 λChair primitives that may be used to implement a usable paper review system.

Figure 1 An example of code using λChair API.

```
module Admin where

import Alice
import Bob

main = evalRevLIO $ do
  -- Adding users to system
  addUser "Alice" "password"
  -- Adding papers to system
  p1 ← addPaper "Flexible Dynamic..."
  p2 ← addPaper "A Static..."
  -- Assign reviewers
  addAssignment "Alice" p1
  addAssignment "Alice" p2

  -- Executing Alice's code
  asUser "Alice" $ aliceCode

  -- Adding new users to system
  addUser "Bob" "password"
  -- Assign reviewers and conflicts
  addAssignment "Bob" p2
  addConflict "Bob"   p1

  -- Executing Bob's code
  asUser "Bob" $ bobCode


module AliceCode where

aliceCode = do
  p1 ← findPaper "Flexible Dynamic..."
  p2 ← findPaper "A Static..."
  readPaper p1
  appendToReview p1 "Interesting work!"
  readPaper p2
  readReview p2
  appendToReview p2 "What about adding new users?"
  return ()


module BobCode where

bobCode = do
  p1 ← findPaper "Flexible Dynamic..."
  p2 ← findPaper "A Static..."
  appendToReview p2 "Hmm, IFC.."
  readReview p1 -- IFC violation attempt
  return ()
```

4. Formal Semantics for LIO

This section formalizes our library for a simple call-by-name[1] λ-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 straight forward, as shown in [23].

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

Label:		l
Address:		a
Term:	$v ::=$	$\texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid a \mid x \mid \lambda x.e \mid (e, e)$
		$\mid \texttt{fix}\ e \mid \texttt{Lb}\ v\ e \mid (e)^{\texttt{LIO}} \mid \bullet$
Expression:	$e ::=$	$v \mid e\ e \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ e$
		$\mid \texttt{label}\ e\ e \mid \texttt{unlabel}\ e \mid \texttt{toLabeled}\ e\ e$
		$\mid \texttt{newRef}\ e\ e \mid \texttt{readRef}\ e \mid \texttt{writeRef}\ e\ e$
		$\mid \texttt{lowerClr}\ e \mid \texttt{getLabel} \mid \texttt{getClearance}$
		$\mid \texttt{labelOf}\ e \mid \texttt{labelOfRef}\ e$
Type:	$\tau ::=$	$\texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid (\tau, \tau)$
		$\mid \ell \mid \texttt{Labeled}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau \mid \texttt{Ref}\ \ell\ \tau$
Store:		ϕ :Address $\to \texttt{Labeled}\ \ell\ \tau$

the formal syntax of the considered language. Syntactic categories v, e, and τ 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 `()` represents the unit value. Symbol ℓ denotes security labels. Symbol a represent memory addresses in a given store. Terms include variables (x), functions ($\lambda x.e$), tuples (e, e), and recursive functions ($\texttt{fix}\ e$). Three special syntax nodes are added to this category: $\texttt{Lb}\ v\ e$, $(e)^{\texttt{LIO}}$, and $\bullet$. Node $\texttt{Lb}\ v\ e$ denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ 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 and they are merely introduced for technical reasons.

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

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

The typing judgments have standard form $\Gamma \vdash e : \tau$, such that expression e has type τ assuming the typing environment Γ; we use Γ 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, 32], we do not require to use any of the sophisticated features of Haskell's type-system, a direct consequence of our dynamic approach.

Figure 3 Typing rules for terms.

$$\vdash l : \ell \qquad \frac{\Gamma(a) = \texttt{Labeled}\ \ell\ \tau}{\Gamma \vdash a : \texttt{Ref}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Lb}\ e_1\ e_2 : \texttt{Labeled}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau} \qquad \Gamma \vdash \bullet : \tau$$

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 Σ denotes a run-time environment that contains the current label, written $\Sigma.\texttt{lbl}$, the current clearance, written $\Sigma.\texttt{clr}$, and store, written $\Sigma.\phi$. A run-time environment Σ and `LIO` computation form a *configuration* $\langle \Sigma, e \rangle$. Given a configuration $\langle \Sigma, e \rangle$, the current label, clearance, and store when starting evaluation e is given by $\Sigma.\texttt{lbl}$, $\Sigma.\texttt{clr}$, and $\Sigma.\phi$, respectively.

The relation $\langle \Sigma, e \rangle \longrightarrow \langle \Sigma', e' \rangle$ represents a single evaluation step from expression e, under the run-time environment Σ, to expression e' and run-time environment Σ'. 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 λ-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 `label`, `unlabel`, and `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 $\texttt{Lb}\ l\ e$ in `LIO` computations. When the content of a labeled value is "retrieved" and used in a `LIO` computation, the current label is raised ($\Sigma' = \Sigma[\texttt{lbl} \mapsto l']$, where $l' = \Sigma.\texttt{lbl} \sqcup l$), 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 $\texttt{toLabeled}\ l\ e$ is used to execute the provided `LIO` computation e until completion ($\langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma', (v)^{\texttt{LIO}} \rangle$) 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 ($\Sigma'.\texttt{lbl} \sqsubseteq l$). 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 `toLabeled` did not require an upper bound on the data to be observed within e, 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 4 Semantics for non-standard constructs.

$$
\begin{aligned}
E \quad ::= \; & [\cdot] \mid \texttt{Lb}\ E\ e \mid E\ e \mid \pi_i\ E \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{return}\ E \mid E\ \texttt{>>=}\ e \\
& \mid \texttt{label}\ E\ e \mid \texttt{unlabel}\ E \mid \texttt{toLabeled}\ E\ e \\
& \mid \texttt{newRef}\ E\ e \mid \texttt{readRef}\ E \mid \texttt{writeRef}\ E\ e \\
& \mid \texttt{lowerClr}\ E \mid \texttt{labelOf}\ E \mid \texttt{labelOfRef}\ E
\end{aligned}
$$

$$\langle \Sigma, E[\texttt{return}\ v] \rangle \longrightarrow \langle \Sigma, E[(v)^{\texttt{LIO}}] \rangle$$

$$\langle \Sigma, E[(v)^{\texttt{LIO}}\ \texttt{>>=}\ e_2] \rangle \longrightarrow \langle \Sigma, E[e_2\ v] \rangle$$

(LAB)
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}}{\langle \Sigma, E[\texttt{label}\ l\ e] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle}$$

(UNLAB)
$$\frac{l' = \Sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ e] \rangle}$$

(TOLAB)
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma', (v)^{\texttt{LIO}} \rangle \qquad \Sigma'.\texttt{lbl} \sqsubseteq l \qquad \Sigma'' = \Sigma'[\texttt{lbl} \mapsto \Sigma.\texttt{lbl}, \texttt{clr} \mapsto \Sigma.\texttt{clr}]}{\langle \Sigma, E[\texttt{toLabeled}\ l\ e] \rangle \longrightarrow \langle \Sigma'', E[\texttt{label}\ l\ v] \rangle}$$

(NREF)
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e]}{\langle \Sigma, E[\texttt{newRef}\ l\ e] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ a] \rangle}\ a\ \textit{fresh}$$

(RREF)
$$\frac{\Sigma.\phi(a) = \texttt{Lb}\ l\ e \qquad l' = \Sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, E[\texttt{readRef}\ a] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ e] \rangle}$$

(WREF)
$$\frac{\Sigma.\phi(a) = \texttt{Lb}\ l\ e \qquad \Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e']}{\langle \Sigma, E[\texttt{writeRef}\ a\ e'] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ ()] \rangle}$$

(LWCLR)
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\texttt{clr} \mapsto l]}{\langle \Sigma, E[\texttt{lowerClr}\ l] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ ()] \rangle}$$

(CLAB)
$$\frac{l = \Sigma.\texttt{lbl}}{\langle \Sigma, E[\texttt{getLabel}] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ l] \rangle}$$

(CCLR)
$$\frac{l = \Sigma.\texttt{clr}}{\langle \Sigma, E[\texttt{getClearance}] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ l] \rangle}$$

(GLAB)
$$\langle \Sigma, E[\texttt{labelOf}\ (\texttt{Lb}\ l\ e)] \rangle \longrightarrow \langle \Sigma, E[l] \rangle$$

(GLABR)
$$\frac{e = \Sigma.\phi(a)}{\langle \Sigma, E[\texttt{labelOfRef}\ a] \rangle \longrightarrow \langle \Sigma, E[\texttt{labelOf}\ e] \rangle}$$

(confidential) labeled values as arguments, with respective labels l_1 and l_2 such that $l_i \not\sqsubseteq l_0, i = 1, 2$.

```
leak lV1 lV2 = do
  lV3 ← toLabeled' $ do
    v1 ← unlabel lV1       -- L_cur = l1
    if v1 then return True
          else unlabel lV2 -- L_cur = l2
  return (labelOf lV3)
```

Therefore, if the returned value is l_1 or l_2 (remember that labels are public information), information is directly leaked! To close this 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` [31].

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

$$\varepsilon_L(l) = l \qquad \varepsilon_L(a) = a$$

$$\varepsilon_L(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \bullet & l \not\sqsubseteq L \\ \texttt{Lb}\ l\ \varepsilon_L(e) & \text{otherwise} \end{cases}$$

$$\frac{\varepsilon_L(\Sigma.\phi) = \{(x, \varepsilon_L(\Sigma.\phi(x)) : x \in \mathrm{dom}(\Sigma.\phi)\}}{\varepsilon_L(\Sigma) = \Sigma[\phi \mapsto \varepsilon_L(\Sigma.\phi)]}$$

$$\varepsilon_L(\langle \Sigma, e \rangle) = \begin{cases} \langle \varepsilon_L(\Sigma), \bullet \rangle & \Sigma.\texttt{lbl} \not\sqsubseteq L \\ \langle \varepsilon_L(\Sigma), \varepsilon_L(e) \rangle & \text{otherwise} \end{cases}$$

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 l (`Lb` $l\ e$) and stores it in the memory store ($\Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e]$). 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 `LIO` monad ($\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}$). The restriction $l \sqsubseteq \Sigma.\texttt{clr}$ is to assure 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 l' ($\Sigma' = \Sigma[\texttt{lbl} \mapsto l']$ where $l' = \Sigma.\texttt{lbl} \sqcup l$). As in the previous rule, (RREF) enforces that the result of reading a reference is below the clearance ($l' \sqsubseteq \Sigma.\texttt{clr}$). Rule (WREF) updates the memory store with a new value for the reference ($\Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e']$) as long as the label of the reference is above the current label and it does not exceed the clearance ($\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}$). If considering $\Sigma.\texttt{lbl}$ as a dynamic version of the `pc` the restriction that the label of the reference must be above the current label ($\Sigma.\texttt{lbl} \sqsubseteq l$) 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 `LIO` 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, 32], 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 ε_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., $\varepsilon_L(\texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e') = \texttt{if}\ \varepsilon_L(E)\ \texttt{then}\ \varepsilon_L(e)\ \texttt{else}\ \varepsilon_L(e')$). In the case of data constructors it is simply the identity function. The two interesting cases for this function are when ε_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 ε_L for several terms and configurations. A labeled value is erased if the label assigned to it is above L ($\varepsilon_L(\texttt{Lb}\ l\ e) = \texttt{Lb}\ l\ \bullet$, if $l \not\sqsubseteq L$). Similarly, the computation performed in a certain configuration is erased if the current label is above L ($\varepsilon_L(\langle\Sigma, e\rangle) = \langle\varepsilon_L(\Sigma), \bullet\rangle$ if $\Sigma.\texttt{lbl} \not\sqsubseteq L$).

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle\Sigma, e\rangle \longrightarrow \langle\Sigma', e'\rangle}{\langle\Sigma, e\rangle \longrightarrow_L \varepsilon_L(\langle\Sigma', e'\rangle)}$$

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., run-time environment and expression. In that manner, the relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$.

Most results that prove non-interference pursue the goal of establishing a relationship between $\longrightarrow^*$ and $\longrightarrow_L^*$ 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 $\longrightarrow_L$ is the same as taking steps in $\longrightarrow$ 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 $\longrightarrow_L$ might not be the same as taking steps in $\longrightarrow$ and then erasing all the secret values in the resulting configuration.

$$\begin{array}{ccc} \langle\Sigma, e\rangle & \longrightarrow^* & \langle\Sigma', e'\rangle \\ \downarrow \varepsilon_L & & \downarrow \varepsilon_L \\ \varepsilon_L(\langle\Sigma, e\rangle) & \longrightarrow_L^* & \varepsilon_L(\langle\Sigma', e'\rangle) \end{array}$$

Figure 1. Simulation between $\longrightarrow^*$ and $\longrightarrow_L^*$.

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 $\longrightarrow$ and $\longrightarrow_L$ 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 Σ and Σ'.

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

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

Proof. By induction on expressions and evaluation contexts. □

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

Proof. From Proposition 1 and definition of ε_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. $\varepsilon_L(E[e]) = \varepsilon_L(E)[\varepsilon_L(e)]$
2. $\varepsilon_L([e_2/x]e_1) = [\varepsilon_L(e_2)/x]\varepsilon_L(e_1)$
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. $\varepsilon_L(\varepsilon_L(\langle\Sigma, e\rangle)) = \varepsilon_L(\langle\Sigma, e\rangle)$

Proof. From the definition of ε_L and by induction on expressions and evaluation contexts. □

The next lemma establishes a simulation between $\longrightarrow$ and $\longrightarrow_L$ for expressions that do not execute `toLabeled`.

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

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 $\Gamma \vdash e : \tau$, $\langle\Sigma, e\rangle \longrightarrow^* \langle\Sigma', e'\rangle$ where there are no executions of `toLabeled`, then $\Gamma \vdash e' : \tau$ and $\varepsilon_L(\langle\Sigma, e\rangle) \longrightarrow_L^* \varepsilon_L(\langle\Sigma', e'\rangle)$.*

Proof. By induction on $\longrightarrow$ 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 $\longrightarrow^*$ and $\longrightarrow_L^*$ 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 $\langle\Sigma, e\rangle \longrightarrow^* \langle\Sigma', e'\rangle$ then $\varepsilon_L(\langle\Sigma, e\rangle) \longrightarrow_L^* \varepsilon_L(\langle\Sigma', e'\rangle)$.*

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

Figure 6 L-equivalence for expressions.

$$\frac{e \approx_L e' \qquad l \sqsubseteq L}{\texttt{Lb}\ l\ e \approx_L \texttt{Lb}\ l\ e'} \qquad\qquad \frac{l \not\sqsubseteq L}{\texttt{Lb}\ l\ e \approx_L \texttt{Lb}\ l\ e'}$$

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: $\texttt{L} \sqsubseteq \texttt{M} \sqsubseteq \texttt{H}$ and an attacker at level L, it holds that $\texttt{Lb H}\ 8 \approx_\texttt{L} \texttt{Lb H}\ 9$, but it does not hold that $\texttt{Lb L}\ 2 \approx_\texttt{L} \texttt{Lb L}\ 3$ or $\texttt{Lb H}\ 8 \approx_\texttt{L} \texttt{Lb M}\ 8$. 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., `true` $\approx_L$ `true`) or homomorphisms (e.g., `if` e `then` e_1 `else` $e_2 \approx_L$ `if` e' `then` e'_1 `else` e'_2 if $e \approx_L e'$, $e_1 \approx_L e'_1$, and $e_2 \approx_L e'_2$).

Since our language encompasses side-effecting expressions, it is also necessary to define 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)**.**

$$\frac{l \sqsubseteq L \vee l' \sqsubseteq L \qquad \forall a. \Sigma.\phi(a) = \texttt{Lb}\ l\ e \approx_L \Sigma'.\phi(a) = \texttt{Lb}\ l'\ e'}{\Sigma.\phi \approx_L \Sigma'.\phi}$$

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)**.**

$$\frac{e \approx_L e' \qquad \Sigma.\phi \approx_L \Sigma'.\phi \qquad \Sigma.\texttt{lbl} = \Sigma'.\texttt{lbl} \qquad \Sigma.\texttt{clr} = \Sigma'.\texttt{clr} \qquad \Sigma.\texttt{lbl} \sqsubseteq L}{\langle \Sigma, e\rangle \approx_L \langle \Sigma', e'\rangle}$$

$$\frac{\Sigma.\phi \approx_L \Sigma'.\phi \qquad \Sigma.\texttt{lbl} \not\sqsubseteq L \qquad \Sigma'.\texttt{lbl} \not\sqsubseteq L}{\langle \Sigma, e\rangle \approx_L \langle \Sigma', e'\rangle}$$

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 e and e' would be reduced to $\bullet$ when applying our simulation between $\longrightarrow^*$ and $\longrightarrow^*_L$ (recall Figure 5).

The next theorem shows the non-interference policy. 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 $\bullet$, $(\,)^{\texttt{LIO}}$, or `Lb`) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$, environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \emptyset$, security label l, an attacker at level L such that $l \sqsubseteq L$, then*

$$\begin{aligned}
&\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\ \ell\ \tau)_{i=1,2} \\
&\quad \wedge (e_i = \texttt{Lb}\ l\ e'_i)_{i=1,2} \wedge \langle \Sigma_1, e_1\rangle \approx_L \langle \Sigma_2, e_2\rangle \\
&\quad \wedge \langle \Sigma_1, e\ e_1\rangle \longrightarrow^* \langle \Sigma'_1, (\texttt{Lb}\ l_1\ e''_1)^{\texttt{LIO}}\rangle \\
&\quad \wedge \langle \Sigma_2, e\ e_2\rangle \longrightarrow^* \langle \Sigma'_2, (\texttt{Lb}\ l_2\ e''_2)^{\texttt{LIO}}\rangle \\
&\quad \Rightarrow \langle \Sigma'_1, \texttt{Lb}\ l_1\ e''_1\rangle \approx_L \langle \Sigma'_2, \texttt{Lb}\ l_2\ e''_2\rangle
\end{aligned}$$

Observe that even though we assume that the input labeled values e_1 and e_2 are observable by the attacker ($l \sqsubseteq L$), they might contain confidential data. For instance, e_1 could be of the form $\texttt{Lb}\ l\ (\texttt{Lb}\ l'\ \texttt{true})$ where $l' \not\sqsubseteq L$.

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

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 $\langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', e'\rangle$, then $\Sigma.\texttt{lbl} \sqsubseteq \Sigma'.\texttt{lbl}$.*

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 $\langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', e'\rangle$, then $\Sigma'.\texttt{clr} \sqsubseteq \Sigma.\texttt{clr}$.*

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:

$$\frac{(\Sigma.\phi)_{\downarrow l} = \Sigma.\phi \setminus \{(a, \texttt{Lb}\ l'\ e) : a \in \mathrm{dom}\,(\Sigma.\phi) \wedge l \sqsubseteq l'\}}{(\Sigma.\phi)_{\downarrow l}}$$

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 $\bullet$, $(\,)^{\texttt{LIO}}$, or `Lb`) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)$, environment $\Sigma[\texttt{lbl} \mapsto l, \texttt{clr} \mapsto l_c]$ such that $l \sqsubseteq l_c$, and $l \not\sqsubseteq l_v$.*

$$\begin{aligned}
&\Gamma \vdash e_1 : \texttt{Labeled}\ \ell\ \tau \\
&\wedge e_1 = \texttt{Lb}\ l_v\ e'_1 \wedge \langle \Sigma, e\ e_1\rangle \longrightarrow^* \langle \Sigma', (\texttt{Lb}\ l_v\ e''_1)^{\texttt{LIO}}\rangle \\
&\Rightarrow (\Sigma.\phi)_{\downarrow l} = (\Sigma'.\phi)_{\downarrow l} \wedge e'_1 = e''_1
\end{aligned}$$

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

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:

$$\frac{(\Sigma.\phi)_{\uparrow l} = \Sigma.\phi \setminus \{(a, \texttt{Lb}\ l'\ e) : a \in \mathrm{dom}\,(\Sigma.\phi) \wedge l' \sqsubseteq l\}}{(\Sigma.\phi)_{\uparrow l}}$$

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 $\bullet$, $(\,)^{\texttt{LIO}}$, or `Lb`) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)$, environment $\Sigma[\texttt{lbl} \mapsto l, \texttt{clr} \mapsto l_c]$ such that $l \sqsubseteq l_c$ and $l_v \not\sqsubseteq l_c$,*

$$\begin{aligned}
&\Gamma \vdash e_1 : \texttt{Labeled}\ \ell\ \tau \\
&\wedge e_1 = \texttt{Lb}\ l_v\ e'_1 \wedge \langle \Sigma, e\ e_1\rangle \longrightarrow^* \langle \Sigma', (\texttt{Lb}\ l_v\ e''_1)^{\texttt{LIO}}\rangle \\
&\Rightarrow (\Sigma.\phi)_{\uparrow l_c} = (\Sigma'.\phi)_{\uparrow l_c} \wedge e'_1 = e''_1
\end{aligned}$$

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

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 label 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 of `Labeled`, `unlabel`, 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 `-fallow-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 allow 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. [32] 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 `unlabel`, 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 parametrized 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 La Padula [5] formalized clearance as a bound on the current label of a particular users' 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 λChair 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

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the 13th European Symp. on Research in Computer Security*, pages 333–348. Springer-Verlag, 2008.

[3] R. Atkey. Parameterised notions of computation. In *Workshop on mathematically structured functional programming, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society*, pages 31–45, 2006.

[4] A. Banerjee and D. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(02):131–177, 2005.

[5] D. E. Bell and L. L. Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[6] K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.

[7] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell, July 2007.

[8] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15:249–291, March 2005.

[9] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[11] *Trusted Computer System Evaluation Criteria (Orange Book)*. Department of Defense, DoD 5200.28-STD edition, December 1985.

[12] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New York, NY, USA, 2011. ACM.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system, October 2005.

[14] M. Felleisen. The theory and practice of first-class prompts. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 180–190. ACM, 1988.

[15] J. Goguen and J. Meseguer. Security policies and security models, April 1982.

[16] W. L. Harrison. Achieving information flow security through precise control of effects. In *In 18th IEEE Computer Security Foundations Workshop*, pages 16–30. IEEE Computer Society, 2005.

[17] D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *Proc. of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[18] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[19] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[20] S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, POPL '06, pages 79–90. ACM, 2006.

[21] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions, October 2007.

[22] C. E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[23] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 144–154. ACM, 1993.

[24] X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 391–403. ACM, 1998.

[25] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[26] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[27] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10. ACM, 2010.

[28] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

[29] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[31] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

[32] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell, 2008.

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

[34] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, June 2009.

[35] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[36] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2004.

[37] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. on Computer Systems*, 25(4):11:1–43, December 2007. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005.

[38] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar, November 2006.

[40] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control, April 2008.