Flexible Dynamic Information Flow Control in the Presence of Exceptions

Sequential LIO

Deian Stefan[1] Alejandro Russo[2] David Mazières[1] John C. Mitchell[1]
(1) Stanford University, Stanford, CA, USA
(2) Chalmers University of Technology, Gothenburg, Sweden
(*email:* `{deian,mitchell}@cs.stanford.edu, russo@chalmers.se`)

Abstract

We describe a language-based, dynamic information flow control (IFC) system called LIO. Our system presents a new design point for IFC, influenced by the challenge of implementing IFC as a Haskell library, as opposed to the more typical approach of modifying the language runtime system. In particular, we take a coarse-grained, floating-label approach, previously used by IFC Operating Systems, and associate a single, mutable label—the *current label*—with all the data in a computation's context. This label is always raised to reflect the reading of sensitive information and it is used to restrict the underlying computation's effects. To preserve the flexibility of fine-grained systems, LIO also provides programmers with a means for associating an explicit label with a piece of data. Interestingly, these labeled values can be used to encapsulate the results of sensitive computations which would otherwise lead to the creeping of the current label. Unlike other language-based systems, LIO also bounds the current label with a *current clearance*, providing a form of discretionary access control that LIO programs can use to deal with covert channels. Moreover, LIO provides programmers with mutable references and exceptions. The latter, exceptions, are used in LIO to encode and recover from monitor failures, all while preserving data confidentiality and integrity—this addresses a longstanding concern that dynamic IFC is inherently prone to information leakage due to monitor failure.

1 Introduction

Information flow control (IFC) tracks the flow of data through a system and prohibits code from operating on data in violation of a 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 runtime overhead, fewer runtime failures, and robustness against implicit flows (Denning & Denning, 1977). However, static analysis in difficult to use in certain scenarios, such as web apps, where for example, users can join (or leave) the system arbitrarily, and where the security policy may depend on data provided by users, at runtime. For such systems, dynamic enforcement techniques are a more natural fit; dynamic IFC systems address many of the shortcomings of static IFC systems while retaining permissiveness.

Dynamic IFC systems fall into roughly two categories: fine-grained and coarse-grained enforcement. Fine-grained approaches, typically employed by language-based systems, e.g., (Austin & Flanagan, 2009; Austin & Flanagan, 2010; Hedin & Sabelfeld, 2012; Hriţcu *et al.*, 2013), explicitly associate security policies—or *labels*—with every value. Such systems have the benefit of giving programmers the ability to associate a particular security policy with a particular value. Unfortunately, this also places the burden of understanding and specifying labels on values that are not relevant for certain tasks. Moreover, fine-grained IFC systems are typically implemented as new (or changes to) languages or runtimes, imposing a large start-up cost on programmers.

Coarse-grained approaches, typically employed by IFC Operating Systems (VanDeBogart *et al.*, 2007; Zeldovich *et al.*, 2006; Krohn *et al.*, 2007), associate a single label with every value in the context of a computation, usually a process. The advantage of such systems is simplicity: programmers do not need to clutter code with labels and can easily understand the security policy of any value—it is simply the label of the context. However, this is also a downside; programmers cannot associate a particular, and heterogeneous, security policy with a particular value. Moreover, incorporating sensitive data into a context usually amounts to "tainting" the whole context, which can lead to the *label creep* problem. Label creep occurs when the context label is tainted to a point where the computation cannot perform any useful side-effects.

In this work, we present LIO, a language-based dynamic IFC system, implemented as a Haskell library, that borrows ideas from both fine- and coarse-grained IFC systems. Like coarse-grained systems, LIO associates a label—the *current label*—with the current context. In particular, we define a monad, *LIO*, that restricts computations to a safe, IFC sublanguage of Haskell.[1] This monad keeps track of the current label, which is, in turn, used to permit restricted access to IO functionality by executing actions in the underlying *IO* monad. Like many Operating Systems (OSes), LIO is a *floating-label* system; the current label is raised to allow reading of sensitive data, thereby "floating above" the labels of all data observed by the current computation. Of course, raising a computation's label comes at the cost of restricting where the computation may subsequently write.

Like fine-grained systems, LIO allows code to associate explicit labels with particular values, thus allowing applications to handle differently-labeled data in the same context. Specifically, LIO provides a *Labeled* type and a value constructor *label* that wraps explicit labels around values. Typically, labels are created at run time and incorporate dynamic information such as usernames or email addresses. LIO safely allows the label of a *Labeled* value to always be inspected. The wrapped value, on the other hand, can be inspected only using *unlabel*, a monadic function that appropriately raises the current label before returning the value.

Explicit unlabeling trivially addresses the problem of implicit flows endemic to fine-grained IFC systems, where control flow constructs are (ab)used to leak sensitive information. In LIO, code cannot branch on a *labeled boolean value* without first calling *unlabel* on the value; this ensures that the code cannot leak information via control flow. However, label creep could still occur if code keeps unlabeling heterogeneously labeled

[1] Using *SafeHaskell* (Terei *et al.*, 2012) we ensure that untrusted code executes in this sublanguage.

data. To address this problem, we introduce a function called *toLabeled*. This primitive executes a computation (that may raise the current label) and restores the current label upon its termination—i.e., it provides a separate context in which to execute the sensitive computation. Importantly, however, the result of the computation is encapsulated as a *Labeled* value—only when the (outer) computation wishes to inspect the result will the current label be raised.

Our dynamic IFC approach makes LIO more permissive than previous static approaches for functional languages (e.g., (Pottier & Simonet, 2002; Li & Zdancewic, 2010; Russo *et al.*, 2008)), while still providing similar security guarantees (Sabelfeld & Russo, 2009). Intuitively, dynamic IFC monitors, such as LIO, are more permissive since they only reject the run of a program if the executed code is about to violate policy. Static IFC analysis, on the other hand, would reject a program, even if a single line of unreachable code is insecure. But, of course static IFC analysis does not incur runtime overheard. More importantly, static approaches also do not usually suffer from covert channel leaks, present in most dynamic language-based IFC systems because of the typical stop-the-world semantics (see (Myers & Liskov, 1997)). LIO addresses these limitations in several ways.

Unlike other language-based work, LIO limits the ability to leak information via covert channels by bounding the current label of a computation with a *current clearance*. The clearance of a region of code may be set to impose an upper bound on the floating current label within the region. Hence, clearance can be understood as a discretionary access control mechanism that restricts the data that a subcomputation can access. And, by limiting access to data on a "need to know" basis, it reduces opportunities for code to leak data through covert channels—after all, code that cannot access sensitive data cannot leak it.

LIO furthermore addresses two limitations common to most dynamic fine-grained systems: the lack of exception handling facilities and inability to recover from IFC monitor failures (and thus the reason for stop-the-world semantics). Laminar (Roy *et al.*, 2009), Breeze (Hriţcu *et al.*, 2013), and an early, unpublished, version of LIO (Stefan *et al.*, 2012b) are the notable exceptions, further discussed in Section 7. Our "mostly coarse-grained" dynamic IFC approach makes it easy to reason about leaks due to exceptional control flow. In particular, we need only reason about exception propagation across *toLabeled* blocks since the current label is only restored, or "lowered," at these points; by treating computations executed by *toLabeled* as being of a separate context, the solution becomes clear: exceptions should not propagate outside the *toLabeled* block.

Equipped with exception-handling facilities, LIO encodes all IFC violations as catchable exceptions. This has the important consequence of allowing untrusted code to recover from IFC violations; this is in contrast to most language-level systems, which consider monitor failures fatal and leave the program in a stuck state (which itself may leak a bit). And, in contrast to Laminar, which also supports recovery from monitor failure (albeit in limited form, when compared to LIO), our uniform treatment of exceptions has led to a more flexible and permissive system (see Section 7)—as with other exceptions, LIO code can *always* recover from monitor failures.

This paper extends an earlier conference version (Stefan *et al.*, 2011b) with *dynamic* exception-handling facilities, an implementation of a real-world conference review web application called λChair, and formal proofs mechanized in the Coq theorem prover. Moreover, this paper corrects the formalism of the sequential LIO calculus to match the Haskell implementation; the formal semantics given in the conference version of this paper (and our tech report presenting an alternative semantics for dynamic IFC exceptions (Stefan *et al.*, 2012b)) did not faithfully capture Haskell's evaluation strategy. The contributions of this paper are the design, formalization, and implementation of a flexible and practical language-level dynamic IFC system. Our main contributions are as follows.

- We propose a new, mostly coarse-grained, design point for dynamic language-level IFC in which most values in lexical scope are protected by a single, mutable, current label. This design has the simplicity of OS-style IFC systems—e.g., because it alleviates the need for developers to annotate the sensitivity of all objects in scope. Instead, in LIO, programmers only associate labels with values they care about by encapsulating them using the *Label* constructor. Such *Labeled* values are similar to labeled values in fine-grained programming languages IFC systems, but differ in a crucial way: our encapsulation is explicitly reflected by types in a way that prevents implicit flows. In a similar way, our calculus and Haskell implementation provides labeled mutable references. In contrast to the Laminar IFC system (Roy *et al.*, 2009), which proposed a similar mostly coarse-grained system, LIO's mutable current label leads to a simpler and more flexible design—since it requires fewer annotations.
- Unlike other language-based work, our IFC model provides a notion of clearance which is used to provide a form of discretionary access control on code, i.e., it provides a way for restricting code to only access data it "needs to know." This is particularly useful in eliminating the opportunity for code to leak sensitive data by exploiting covert channels.
- We present a simple dynamic, yet safe, exception-handling mechanism and encode IFC monitor failures using exceptions. Exceptions are crucial to making LIO a practical IFC system; real-world applications cannot "stop the world" on an IFC violation attempt. This has been longstanding problem with dynamic IFC monitors, as highlighted by Myers and Liskov: "the difficulty with runtime checks is exactly the fact that they can *fail*... failure (or its absence) can serve as a covert channel (Myers & Liskov, 1997)."
- We prove information flow, access control, and isolation security properties of our design. A large part of our formalization is encoded in Coq. We remark that while our formal description of LIO is Haskell-centric, this is not a fundamental restriction—our formalism can be generalized to other programming languages.
- We describe a Haskell implementation of the IFC calculus in Haskell. LIO can be implemented entirely as a library, demonstrating both the applicability and simplicity of the approach. This has the added benefit of not imposing the burden of learning a new programming language on developers—they simply need to understand a new API. Moreover, developers can use many existing compilers, tools, and libraries (e.g., roughly 12,500 Haskell modules on Hackage are safe to be used in LIO). Our library, applications built on top of it (including λChair), and Coq proofs are available at `http://labeled.io`.

This paper is organized as follows. Section 2 describes the core information flow control LIO calculus. In Sections 3–5, we extend the core with clearance, mutable references, and exception-handling facilities. The security guarantees of the full calculus are given in Section 6. Related work is described in Section 7. We conclude in Section 8.

2 Core dynamic information flow control LIO calculus

IFC systems track and restrict the propagation of information according to a security policy. The core policy enforced by LIO, and most other IFC systems, is *noninterference*. Noninterference guarantees confidentiality (Goguen & Meseguer, 1982), by preventing sensitive information from being leaked to public entities, and integrity (Biba, 1977), by preventing unreliable information from flowing into critical operations.

In this section, we detail the core design of LIO and discuss the design trade-offs of a library-driven, mostly coarse-grained approach using the λChair conference review system as a driving example. In λChair, authenticated users can read any paper and can normally read any review. This reflects the normal practice in conference reviewing where, for example, every member of the program committee can see submissions and their reviews, and can participate in related discussion. In λChair, users can be added dynamically and assigned to review specific papers. Importantly, we use IFC to ensure that only assigned reviewers can write reviews for any given paper and that committee members in conflict with a paper cannot access the related discussions.

We incrementally describe the semantics of LIO using an extended simply-typed λ-calculus. First, we describe a pure base calculus. This calculus is then extended with labels (Section 2.2), labeled computations (Section 2.3), and labeled values (Section 2.4). Further leveraging labeled values we extend the calculus with *toLabeled* blocks to address label creep (Section 2.5). Finally, we extend this core with other features such as clearance, references and exceptions (Sections 3–5).

2.1 Base calculus for pure terms

$$\begin{array}{l}\text{Values } v ::= \mathit{True} \mid \mathit{False} \mid () \mid \lambda x.t \\ \text{Terms } t ::= v \mid x \mid t_1\ t_2 \mid \textbf{fix}\ t \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \\ \text{Types } \tau ::= \mathit{Bool} \mid () \mid \tau_1 \to \tau_2\end{array}$$

Fig. 1. Formal syntax for values, terms, and types.

Our semantics build on a pure, base calculus. The formal syntax of this base calculus is given in Fig. 1. Syntactic categories v, t, and τ represent values, terms, and types, respectively. Values include primitives (Booleans *True*, *False*; and unit ()) and functions ($\lambda x.t$). Terms constitute values (v), variables (x), function applications ($t_1\ t_2$), the standard fixpoint operator **fix** t, and conditionals (**if** t_1 **then** t_2 **else** t_3). Types consist of *Bool*, unit (), and functions $\tau_1 \to \tau_2$.

Fig. 2 shows the reduction rules for these pure terms using structural operational semantics (Winskel, 1993). The relation $t_1 \rightsquigarrow t_2$ represents a single evaluation step of pure term t_1 to term t_2; we say that t_1 reduces to t_2 in one step. We write $\rightsquigarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$.

Substitution $\{t_2\,/\,x\}\,t_1$ is defined in the usual way, homomorphic on all operators, renaming bound names to avoid capture. The reduction rules for these terms are self-explanatory and very much the same as those of standard λ-calculus—we do not explain them further.

$$\overset{\textsc{appCtx}}{\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}} \qquad \overset{\textsc{app}}{\frac{}{(\lambda x.t_1)\ t_2 \rightsquigarrow \{t_2\,/\,x\}\ t_1}} \qquad \overset{\textsc{fixCtx}}{\frac{t \rightsquigarrow t'}{\mathbf{fix}\ t \rightsquigarrow \mathbf{fix}\ t'}} \qquad \overset{\textsc{fix}}{\frac{}{\mathbf{fix}\ (\lambda x.t) \rightsquigarrow \{\mathbf{fix}\ (\lambda x.t)\,/\,x\}\ t}}$$

$$\overset{\textsc{ifCtx}}{\frac{t_1 \rightsquigarrow t_1'}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3}} \qquad \overset{\textsc{ifTrue}}{\frac{}{\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow t_2}}$$

$$\overset{\textsc{ifFalse}}{\frac{}{\mathbf{if}\ \mathit{False}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow t_3}}$$

Fig. 2. Semantic rules for pure terms in the base calculus.

We solely remark that our semantics does not model the sharing in lazy evaluation, as implemented by Haskell; modeling full lazy evaluation is beyond the scope of this paper and has no impact on our termination- and timing-insensitive security guarantees.

LIO is implemented as a domain specific language embedded in Haskell. Hence, the typing judgements for our calculus are a subset of Haskell's and standard. We do not give any of the type judgements in this paper. (The interested reader can see our Coq formalization.) Rather, we remark that LIO relies on types only to distinguish terms that can be used to compose safe computations and those that cannot, as further discussed in Section 2.3. Indeed, LIO can be generalized to dynamically-typed languages, as shown in (Heule *et al.*, 2015).

2.2 Security lattice

To enforce security policies, like most modern dynamic IFC systems, LIO associates *labels* with objects. Labels encode confidentiality and integrity data policies which are propagated alongside the information they protect. In turn, the system mandatorily enforces these individual policies when objects are read or written.

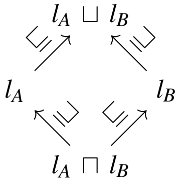

Fig. 3. Simple lattice.

Labels are elements of a set $\mathscr{L}$ that forms a security lattice $(\mathscr{L}, \sqsubseteq, \sqcap, \sqcup)$, with partial order $\sqsubseteq$ (pronounced "can flow to"), binary join $\sqcup$, and binary meet $\sqcap$ (Denning, 1976). The $\sqsubseteq$ relation is used by IFC systems when governing the allowed flows between differently labeled entities.[2] For example, LIO only allows data labeled $l_d \in \mathscr{L}$ to be written to a channel labeled $l_c \in \mathscr{L}$ if $l_d \sqsubseteq l_c$ holds true. The binary join is used to label computation results that depend on two objects by encoding the restrictions imposed by their labels, i.e., for labels $l_A, l_B \in \mathscr{L}$, the join $l_A \sqcup l_B$ is the smallest element such that $l_A \sqsubseteq l_A \sqcup l_B$ and $l_B \sqsubseteq l_A \sqcup l_B$. Dually, the binary meet $l_A \sqcap l_B$ encodes the intersection of the restrictions imposed

[2] Decentralized IFC (DIFC) extends IFC with the decentralized label model of Myers and Liskov (Myers & Liskov, 1997), in which computations execute with a set of *privileges*, that can be used to loosen the restrictions imposed by the $\sqsubseteq$ relation. LIO supports privileges and the DIFC model in full. But, since our formalisation is limited to the system without privileges, we omit this from the presentation and refer the interested reader to the library documentation.

by l_A and l_B; the meet is primarily used when labeling objects that we expect to be read by entities labeled l_A or l_B. Fig. 3 shows how information flows in a simple lattice.

In LIO, labels are typed values. But, unlike most existing IFC systems (Zeldovich *et al.*, 2006; Myers & Liskov, 2000; Krohn *et al.*, 2007; Hriţcu *et al.*, 2013), LIO is polymorphic in the label format. We solely require that the label type provide definitions for lattice relations $\sqsubseteq$, $\sqcup$, and $\sqcap$. In Haskell, this amounts to making the label type an instance of the typeclass *Label*; all LIO library functions are qualified by *Label*:

```
class Eq 𝓛 ⇒ Label 𝓛 where
  (⊑) :: 𝓛 → 𝓛 → Bool
  (⊔) :: 𝓛 → 𝓛 → 𝓛
  (⊓) :: 𝓛 → 𝓛 → 𝓛
```

As an example, consider the definition of the typical 2-point lattice $\mathscr{L}_2 = \{Public, Secret\}$, where $Public \sqsubseteq Secret$ and $Secret \not\sqsubseteq Public$:

```
data 𝓛₂ = Public | Secret deriving (Eq, Ord)
instance Label 𝓛₂ where
  x ⊑ y = x ⩽ y
  x ⊔ y  = max x y
  x ⊓ y  = min x y
```

Here, we simply use the *Ord* functions ($\leqslant$, *min*, and *max*), as defined by the compiler, to define the lattice operations. Of course, in real-world applications developers can define more complex label formats, such as the DLM (Myers & Liskov, 1997), HiStar (Zeldovich *et al.*, 2006), Flume (Krohn *et al.*, 2007), or DCLabels (Stefan *et al.*, 2011a). Since such label definitions are typically provided by trusted code, LIO simply assumes that labels form a lattice, i.e., we do not verify that labels form a partially ordered set with a well-defined least upper bound and greatest lower bound. However, in certain cases, static analysis (e.g., in the form of refinement types (Rondon *et al.*, 2008)) can be used to verify that provided definitions are well-defined.

To model labels, we extend our calculus to make labels first-class. Instead of modeling typeclasses, for simplicity, we assume that our calculus is polymorphic in the label type $\mathscr{L}$. With this in mind, we extend the syntactic categories of Fig. 1 as shown on the right (Fig. 4). Here, values are extended with labels—metavariables l and c span over such values; types are extended with the label type $\mathscr{L}$; and, terms are extended with label operations.

$$\begin{array}{lll}
\text{Values} & v ::= & \cdots \mid l \mid c \\
\text{Terms} & t ::= & \cdots \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_1 \sqsubseteq t_2 \\
\text{Types} & \tau ::= & \cdots \mid \mathscr{L}
\end{array}$$

Fig. 4. Formal syntax for labels and their operations.

The reduction rules for these label operations are straightforward and given in Fig. 5. The rules for the label operations $\sqcup$, $\sqcap$, and $\sqsubseteq$ rely on the label-specific implementation of these operators, as used in the premise of rule (LOP); we use the partial function $[\![\cdot]\!]_{\mathscr{L}}$, which maps terms to values, to denote this. For example, instantiating our calculus to $\mathscr{L}_2$, $[\![Public \sqcup Secret]\!]_{\mathscr{L}_2} = Secret$, $[\![Secret \sqsubseteq Public]\!]_{\mathscr{L}_2} = True$, etc. We highlight that our evaluation rules reduce the left operand first. Reducing the right operand first does

$$\textsc{lOpCtxL}\quad \frac{t_1 \rightsquigarrow t_1'}{t_1 \otimes t_2 \rightsquigarrow t_1' \otimes t_2} \qquad \textsc{lOpCtxR}\quad \frac{t_2 \rightsquigarrow t_2'}{l_1 \otimes t_2 \rightsquigarrow l_1 \otimes t_2'} \qquad \textsc{lOp}\quad \frac{v = [\![l_1 \otimes l_2]\!]_{\mathscr{L}}}{l_1 \otimes l_2 \rightsquigarrow v}$$

Fig. 5. Semantics for pure label operations, with binary operator $\otimes \in \{\sqcup, \sqcap, \sqsubseteq\}$. The precise definition of these operators depends on the underlying label mode $\mathscr{L}$.

not affect the semantics—we chose left-to-right evaluation solely because it matches the implementation of the labels used in λChair (see (Stefan *et al.*, 2011a)). In the rest of the paper, we sometimes use a more lax notation to describe label operations, e.g., $l_1 \sqsubseteq l_2$ in place of $l_1 \sqsubseteq l_2 \rightsquigarrow \mathit{True}$.

2.3 Restricting Haskell to safe IFC subset with the *LIO* monad

As previously mentioned, every object in an IFC system must be labeled. Importantly, this includes the *current execution context* whose label we call the *current label*.[3] The current label serves a role similar to the program counter (*pc*) in static IFC systems (Denning & Denning, 1977). Namely, it prevents the current computation from performing side-effects which might compromise confidentiality. For instance, if the current label is l_{cur}, LIO prevents the computation from writing to entities labeled l_e unless $l_{\text{cur}} \sqsubseteq l_e$.

To accomplish this, LIO provides a monad called *LIO*. The *LIO* monad encapsulates Haskell's *IO* monad as to allow for *LIO* computations to perform (restricted) I/O. The monad also encapsulates the current label l_{cur}, which is retrieved with the *getLabel* function. The relevant parts of the definition are given below. By convention, we use $\mathscr{L}$ for type variables that are expected to be instantiated by a label. The library is polymorphic over $\mathscr{L}$ for greater flexibility, but in any normal program, every occurrence of $\mathscr{L}$ will be instantiated by the same label type. Hence, it is more intuitive to think of $\mathscr{L}$ as representing a particular (though unspecified) label type. Below we give the interface for this monad. We omit the definitions for simplicity.

```
data LIO ℒ τ
instance Monad (LIO ℒ)
  return :: τ → LIO ℒ τ
   >>=   :: LIO ℒ τ₁ → (τ₁ → LIO ℒ τ₂) → LIO ℒ τ₂
getLabel :: Label ℒ ⇒ LIO ℒ ℒ
```

As usual, *return* lifts a value into the *LIO* $\mathscr{L}$ monad, while bind (>>=) is used to chain two actions by executing the first and binding the result to be used in the executing second. The definitions for the monadic *return* and bind (>>=) are straightforward—a reference to the current label is simply threaded through the computation. This label is exposed via *getLabel*; *getLabel* is a monadic action (in the *LIO* $\mathscr{L}$ monad), which, when executed, returns the current label (of type $\mathscr{L}$).

[3] More generally, every thread in the system is labeled. But, since we are focusing on a single-threaded system, we refer to the main thread context as the current execution context and its label as the current label.

RETURN

$$\frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{LIO}^{\text{TCB}}\ t\rangle}$$

BIND

$$\frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\text{TCB}}\ t'_1\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1 \gg\!= t_2\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t_2\ t'_1\rangle}$$

LIOPURE

$$\frac{t_1 \rightsquigarrow t'_1}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t'_1\rangle}$$

GETLABEL

$$\frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{getLabel}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ l_{\text{cur}}\rangle}$$

Fig. 7. Semantics for the *LIO* monad.

We remark that since *return* and bind are essentially the standard State monad combinators (Liang *et al.*, 1995), no security checks are performed internally by these combinators. Instead, LIO library functions (e.g., *readFile*) use the current label to perform security checks (so as to enforce IFC) before executing any underlying *IO* actions. Taking this approach, the LIO library provides a collection of *LIO* actions that are similar to the *IO* actions available in standard Haskell libraries—and, indeed, usually wrap them—but additionally enforce IFC. Henceforth, we assume that all computations are in the *LIO* monad.

To formally describe the behavior of the *LIO* monad, we extend the syntactic categories of our calculus as shown on the right (Fig. 6). Our extension simply adds monadic actions ($\mathit{LIO}^{\text{TCB}}\ t$) to values, monadic operations to terms, and a type for *LIO* computations. We note that the $\mathit{LIO}^{\text{TCB}}$ constructor is not part of the surface syntax, i.e., programs that use $\mathit{LIO}^{\text{TCB}}$ are not considered valid.[4]

$$\begin{array}{ll}
\text{Values} & v ::= \cdots \mid \mathit{LIO}^{\text{TCB}}\ t \\
\text{Terms} & t ::= \cdots \mid \mathit{return}\ t \mid t_1 \gg\!= t_2 \mid \mathit{getLabel} \\
\text{Types} & \tau ::= \cdots \mid \mathit{LIO}\ \mathscr{L}\ \tau \\
\mathit{Memories}\ m & \\
\mathit{Programs}\ k ::= \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle &
\end{array}$$

Fig. 6. Formal syntax for core *LIO*.

We explicitly distinguish pure-term evaluation from top-level monadic-term evaluation. Specifically, an LIO program is a *configuration*—spanned over by metavariable k—of the form $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle$, where l_{cur} is the current label, c_{cur} is the current clearance (explained in Section 3), m is the memory store (see Section 4), and t is the monadic term under evaluation. The reduction $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ represents a single evaluation step from term t, with current label l_{cur}, current clearance c_{cur}, and memory m, to term t', with current label l'_{cur}, current clearance c'_{cur}, and memory m'. For the moment, we ignore the clearance and memory in the configuration. Index n in the transition relation counts the number of executed *toLabeled* actions; this is an artifact of the proof technique and not relevant to the semantics. We write $\xrightarrow{n}{}^{*}$ for the reflexive and transitive closure

[4] For simplicity, we do not use additional syntactic categories to distinguish between values and terms that are part of the surface syntax from those that are not. In Section 6, we define a *safe* predicate for making such a distinction.

of $\xrightarrow{n}$. The reduction rules for the core LIO operations are given in Fig. 7. The rules for *return* and $(\gg=)$ are trivial and standard—all IFC checks are performed by the non-proper morphism of *LIO*. Similarly, the (LIOPURE) rule specifies that if we have a top level pure term, it should be evaluated to completion, i.e., until it reduces to a monadic term. Rule (GETLABEL) defines the *LIO* library function for retrieving the underlying current label, further discussed below.

2.3.1 Coarse-grained labeling with the current label

To soundly reason about IFC, every value *must* be labeled. However, and in contrast to other language-based systems (e.g., Jif (Myers & Liskov, 2000), FlowCaml (Simonet, 2003), Breeze (Hriţcu *et al.*, 2013) etc.) in which every value is explicitly labeled, the values in our calculus are not associated with explicit labels (see Fig. 1–6). This is a direct consequence of taking a library-based approach: we cannot explicitly label every Haskell value without modifying the language runtime. Instead, and like several IFC operating systems (Efstathopoulos *et al.*, 2005; Zeldovich *et al.*, 2006), we take a coarse-grained approach and use the current label to protect all values in scope, i.e., in LIO, the current label l_{cur} is the label on all "unlabeled" values in the current execution context. Since we use the current label to restrict the current computation from performing arbitrary side-effects, this also ensures that the confidentiality (and integrity) of all values in scope is preserved.

In addition to ensuring that every value is labeled, this coarse-grained labeling approach has two other interesting consequences. First, it does not force developers to explicitly label every piece of data. This eliminates the need to clutter code with labels, reason about the security implications of every value, or define a special *default* label (e.g., that would be used to label literals). Instead, developers only explicitly label data they care about, as detailed in Section 2.4.

Second, it eliminates the implicit flows problem by construction (Sabelfeld & Myers, 2003). As previously mentioned, this problem arises when information can be leaked through the program control flow. An example of an implicit flow is given in Fig. 8, written in a hypothetical alternative LIO language without explicit labels. Here, secret bit b is leaked into public reference x according to the program control flow, i.e., what code—which assignment (to public reference x)—is executed depends on the secret b.

$\textbf{if}\ b_{Secret}$
$\quad\textbf{then}\ x_{Public} := 1_{Public}$
$\quad\textbf{else}\ \ x_{Public} := 0_{Public}$

Fig. 8. Implicit flows problem.

To prevent such leaks, language-based approaches rely on the program counter label to reflect the sensitivity of the branch condition within each branch and, in turn, disallow such unsafe assignments. In Haskell, and thus LIO, branch conditions have type *Bool*—they are not explicitly labeled values. Rather, the branch condition is (conceptually) labeled by the current label, which is common across both branches. As a consequence, control flow cannot be used to leak sensitive information: regardless of the branch taken, the current label prevents writes to public entities. Consider implementing the attack in Fig. 8 with LIO. Since the branch condition b_{Secret} is not explicitly labeled, it is protected by l_{cur}. But since b_{Secret} is secret, we must have $l_{\text{cur}} = Secret$, meaning any subsequent writes (within the branches or after) to public references are disallowed since $l_{\text{cur}} \not\sqsubseteq \textit{Public}$. In Section 4, we give the precise semantics for mutable references in LIO.

2.3.2 A floating current label

The current label protects all data in scope by serving as an upper bound label on all values. To preserve this invariant, when reading sensitive data, we can either disallow reads from entities more sensitive or raise the current label to protect the newly read data. Like other coarse-grained systems we take the latter approach and raise the current label to "float" above the labels of all the entities from which data has been read.

Raising the current label allows computations to flexibly read data, at the cost of being more limited in where they can subsequently write. Concretely, a computation with current label l_{cur} can read data labeled l_d by raising its current label to $l'_{\text{cur}} = l_{\text{cur}} \sqcup l_d$, but can thereafter only write to entities labeled l_e if $l'_{\text{cur}} \sqsubseteq l_e$. For example, LIO allows a public computation to read secret data by raising l_{cur} from *Public* to *Secret*. Importantly, the new current label prevents the computation from subsequently writing to public entities. Some static IFC systems, such as Jif (Myers & Liskov, 2000), are even more permissive in allowing public writes after reading secret data if no secret data is actually being leaked. In Section 2.5, we present a method that can be used to safely restore the current label, making our dynamic IFC system equally permissive.

2.3.3 Ensuring all code executes in the LIO monad

To ensure security, all side-effecting computations must be encoded in *LIO*. LIO can only guarantee confidentiality and integrity for computations written using the LIO library; if an attacker can bind an arbitrary *IO* action within a larger *LIO* computation, IFC can trivially be violated. Hence, the visibility of the *LIO* value constructor, i.e., the constructor used to create values of type $\textit{LIO}\ \mathscr{L}$, must be limited to the LIO *trusted computing base* (TCB) so as to guarantee that "untrustworthy" (and potentially malicious) code cannot perform arbitrary I/O. In our formal mode, this amounts to not making $\textit{LIO}^{\text{TCB}}$ part of of the surface syntax.

To accomplish this, we use Safe Haskell (Terei *et al.*, 2012). Specifically, the module in which the *LIO* data type is defined is marked `Unsafe`, while the modules that expose IFC-enforcing *LIO* actions are marked—by us, the library providers—as `Trustworthy`. In doing so, Safe Haskell ensures that we can safely execute arbitrary, attacker-provided *LIO* actions by simply marking the top-level modules as `Safe`. Safe Haskell prevents `Safe` code from depending on `Unsafe` modules thus ensuring that the computation could only have been composed of `Trustworthy` LIO library functions or the subset of Haskell that is "safe," i.e., the part that does not contain the *LIO* value constructor or other unsafe features such as `unsafePerformIO` (Terei *et al.*, 2012).

2.4 Explicitly labeling values

While LIO ensures that everything in a context is protected by the current label, for many applications it is useful to be able to handle differently-labeled data in a single scope. To motivate this, let's consider an HTTP route (e.g., `/papers/index.html`) in λChair which lists all the papers submitted by the logged-in user.

In λChair, each submitted paper is associated with a label to ensure that the paper can only be read by users that have the appropriate role (e.g., is the author or committee member). When reading a paper from the database system, the label of the *HTTP request handler* for the route, or *controller*, which is an *LIO* action, is raised to reflect the fact that sensitive data is being incorporated into the context. In doing so, LIO can ensure that a response is only sent to the user's browser—which, itself, has a label corresponding to the authenticated user—when the controller label can flow to the browser label.

Suppose that the λChair database contains two papers, as shown in Fig. 9, submitted by Alice and Bob (neither of whom is part of the committee). When Alice wishes to see the index of all papers she submitted, the controller must read from the database only data whose labels can flow to the browser label l_{Alice}. Otherwise, the controller will reach a state in which the current label is above the browser label (e.g., $l_{\text{Alice}} \sqcup l_{\text{Bob}}$) and it will no longer be allowed to respond to the user. In language-based IFC systems (Myers *et al.*, 2001; Simonet, 2003), this is typically not a concern because values returned from the database can be individually and explicitly labeled. As a result, the controller would be able to compare the label of the value retrieved from the database and the browser label, only using the retrieved value if its label flows to the browser label. In LIO, reading both values into the context would taint the controller with both l_{Alice} and l_{Bob}, preventing the overtainted controller from replying to Alice.

Paper	**Label**
p_{Alice}	l_{Alice}
p_{Bob}	l_{Bob}

Fig. 9. Database containing two papers.

To avoid being overly restrictive, LIO provides *Labeled* values. A labeled value protects an arbitrary term with a strict, explicit label, irrespective of the current label. We define such values as follows.

$$\textbf{data}\ \mathit{Labeled}\ \mathscr{L}\ \tau$$

As before, we restrict the value constructor to the TCB. However, to allow non-TCB code to create and manipulate labeled values, we provide a safe, IFC-abiding, interface. This is particularly important since labeled values are protected by their explicit labels—untrusted code should not be allowed to bypass the label and arbitrarily inspect (or modify) the protected value. This interface for creating and inspecting labeled values is given below.

$$\begin{aligned}
&\mathit{label} && :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{Labeled}\ \mathscr{L}\ \tau)\\
&\mathit{unlabel} && :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau\\
&\mathit{labelOf} && :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \to \mathscr{L}
\end{aligned}$$

$$\begin{aligned}
&\text{Values } v && ::= \cdots \mid \mathit{Labeled}^{\mathsf{TCB}}\ v\ t\\
&\text{Terms } t && ::= \cdots \mid \mathit{label}\ t_1\ t_2 \mid \mathit{unlabel}\ t \mid \mathit{labelOf}\ t\\
&\text{Types } \tau && ::= \cdots \mid \mathit{Labeled}\ \mathscr{L}\ t
\end{aligned}$$

Fig. 11. Formal syntax for labeled values.

To describe the semantics of these functions, we extend the values, terms and types of our calculus as shown in Fig. 11. (As with $\mathit{LIO}^{\mathsf{TCB}}$, we do not consider the $\mathit{Labeled}^{\mathsf{TCB}}$ constructor part of the surface syntax.) The reduction rules for the new terms are given in Fig. 10; rule (LABELCTX), (UNLABELCTX), and (LABELOFCTX) reduce terms until they have appropriate structures to trigger rules (LABEL), (UNLABEL),

$$\textsc{labelCtx}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ t_1\ t_2 \rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ t_1'\ t_2 \rangle}$$

$$\textsc{label}\quad \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ l\ t \rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t) \rangle}$$

$$\textsc{unlabelCtx}\quad \frac{t \rightsquigarrow t'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ t \rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ t' \rangle}$$

$$\textsc{unlabel}\quad \frac{l_{\text{cur}} \sqcup l \rightsquigarrow l_{\text{cur}}' \qquad l_{\text{cur}}' \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t) \rangle \xrightarrow{0} \langle l_{\text{cur}}', c_{\text{cur}}, m \mid \mathit{return}\ t \rangle}$$

$$\textsc{labelOfCtx}\quad \frac{t \rightsquigarrow t'}{\mathit{labelOf}\ t \rightsquigarrow \mathit{labelOf}\ t'} \qquad\qquad \textsc{labelOf}\quad \frac{}{\mathit{labelOf}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t) \rightsquigarrow l}$$

Fig. 10. Semantics for labeled values.

and (LABELOF), respectively. We ignore parts of these rules that involve the current clearance c_{cur} until Section 3.

The *label* function is used to explicitly label terms. The function takes two arguments, a label and a term, and returns an *LIO* action, which, when executed, produces an explicitly labeled value. Rule (LABEL) gives the precise semantics: the function associates the supplied label l with term t by wrapping the term with the $\mathit{Labeled}^{\mathsf{TCB}}$ constructor. It first asserts that the new label (l) used to protect t is at least as restricting as the old label (the current label, l_{cur}), i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$.

We remark that if the premise does not hold the function throws an exception to indicate an IFC violation—our semantics do not employ stop-the-world semantics as a way to encode monitor failures. This is the case for all other rules in LIO in which a premise is not satisfied. Section 5 describes this in more detail and defines exception handling facilities that code can use to recover from such IFC violations.

The dual of *label*, *unlabel*, takes an explicitly labeled value and returns an *LIO* action which, when executed, returns the underlying wrapped value. As given by rule (UNLABEL), the function takes a labeled value $\mathit{Labeled}^{\mathsf{TCB}}\ l\ t$ and returns the wrapped term t. However, since the returned term is no longer protected by l and is, instead, protected by the current label, l_{cur} must be at least as restricting as l[5]. To ensure this, the current label is raised from l_{cur} to $l_{\text{cur}} \sqcup l$—this captures the fact that the remaining computation might depend on t. The current label always "floats" above the labels of the values observed by the current computation.

[5] The effects of *unlabel* are similar to those of *bind* in DCC (Abadi *et al.*, 1999): subsequent computations must be protected by the label of the recently observed value.

Finally, we provide the *labelOf* function as a way to inspect the label of a labeled value. As detailed by the (LABELOF) reduction rule, *labelOf* takes a labeled value $Labeled^{\mathsf{TCB}}\ l\ t$ and simply returns the label l protecting term t. Since the label of a *Labeled* value is strict, *labelOf* does not require an additional context rule for reducing the label. Unlike *unlabel*, *labelOf* also does not raise the current label—*labelOf* is part of the pure calculus. Indeed, this allows code to check the label of a labeled value before deciding to *unlabel* it (and thereby raise the current label). This design decision has an important consequence: regardless of the current label (and clearance) of the configuration, *labelOf* always succeeds. While this may seem like LIO labels are "public," they are in fact protected by a label—the current label—and thus cannot be used as a covert channel. Section 2.5 describes an alternative design in which labels are not public and shows how labels can be used to leak information when not properly protected.

Example 1 (*Fetching papers for reviewers*)
Turning to our λChair use case, we now consider some of the core functions that are used by the top-level request handler. In particular, we show how to fetch papers for a given reviewer using a simple underlying database system. The specific label type used by λChair is *DCLabel*. As defined in (Stefan *et al.*, 2011a), a *DCLabel* is a pair of formulae over principals (e.g., users) in conjunctive normal form, representing the principals that can read and write data labeled as such. We define a type alias for the *LIO* monad with the label instantiated to *DCLabel*:

$$\textbf{type}\ DC\ \tau = LIO\ DCLabel\ \tau$$

The λChair database system operates on *DCLabel*ed papers, in the *DC* monad. As defined below, a paper is simply a record with several fields, including the (unique) paper id (*paperId*), the paper itself (*pdf*), labeled *reviews*, etc.[6]

$$\textbf{data}\ Paper = Paper\ \{paperId :: Id, pdf :: PDF, reviews :: [LabeledReview], \ldots\}$$
$$\textbf{type}\ LabeledPaper = Labeled\ DCLabel\ Paper$$

Among other operations, the database system provides a *fetchPapers* function which is used to get the list of *all* such papers:

$$fetchPapers :: DC\ [LabeledPaper]$$

For simplicity, we omit the implementation details of *fetchPapers* and only remark that it relies on TCB code to wrap an underlying *IO*-based database system API and explicitly label the fetched papers.

While simple, the *fetchPapers* function is sufficient for fetching a given reviewer's papers. Note that if the controller simply unlabels the papers returned by *fetchPapers*, the current label may be raised to a point where the computation cannot respond back to the user, i.e., the current label may not flow to the browser label. This situation, for example, happens when the current user is not part of the committee and another author's paper is

[6] We elide the details of labeled reviews used in the actual λChair implementation and simplify some of the application details (e.g., the generic database system API). The interested reader is referred to the code documentation at `http://labeled.io` for more details.

unlabeled—λChair prevents such data from being sent (leaked) back to the user's browser. Hence, we need to make sure that the controller only reads data that the end-user can see.

To this end, we define *fetchPapersFor*:

```
fetchPapersFor :: User → DC [LabeledPaper]
fetchPapersFor user = do
  -- Get all labeled papers:
  lpapers ← fetchPapers
  -- Filter the papers the user is allowed to read:
  let browserLabel = userToLabel user
      lpapers′ = filter (λlpaper.labelOf lpaper ⊑ browserLabel) lpapers
  -- Unlabel and return all the papers this user can read:
  mapM unlabel lpapers′
```

This function fetches the papers, filters the ones the user is allowed to read by comparing the paper's label with the user's browser label—itself computed with function *userToLabel*—and unlabels them. At this point, the controller can compose the HTML page containing the paper information and safely respond to the user.

In addition to providing a simple illustration of how labeled values are used in LIO, this simple example serves to illustrate the importance of labeled values. Specifically, by providing labeled values in the language, we can implement core functionality such as *fetchPapersFor* in the untrusted LIO application code; without labeled values such functionality would otherwise have to be implemented in the trusted database layer or database system itself. Indeed, building on this observation, we can, for example, extend λChair to implement an in-memory database which solely uses the aforementioned database system as a persistence layer, i.e., it solely relies on the actual database system to keep the papers persistent.

2.5 Addressing label creep with *toLabeled*

In conference systems, it is often the case that some reviews are superseded by others, papers change titles, submissions are withdrawn, etc. Hence, the λChair database system provides functions for updating (or deleting) existing papers. For instance, *updatePaper* is used to update the paper with the supplied paper id with the new labeled paper. The type for this function is given below.

```
updatePaper :: Id → LabeledPaper → DC ()
```

Similar to *fetchPapers*, this function relies on TCB code to communicate with the actual database system; from a security stance, it is only interesting to note that the function always ensures that the current computation can overwrite the existing paper (by performing a $\sqsubseteq$-check with the current label, current clearance (see Section 3), and label protecting the existing paper).

Suppose we wish to implement a function that performs a partial update, i.e., an update wherein only part of the paper object is updated. This is useful, for example, when a user only updates the abstract of the paper and leaves other parts such as the underlying PDF intact. Indeed, sending a PDF file, which may be large, to simply perform a "full" update is not practical. An implementation of such a partial update function is given below.

```
partialUpdatePaper :: Id → PartialPaper → DC ()
partialUpdatePaper i new = do
  -- Get the existing paper according to its id:
  lold ← fetchPaperById i
  old ← unlabel lold
  -- Merge the new (partial) paper and existing paper:
  lnew ← label (labelOf lold) (merge new old)
  -- Perform actual update:
  updatePaper i lnew
```

Here, we assume that the type *PartialPaper* encodes a partial paper (e.g., by using a *Maybe* type for each of the fields in *Paper*) and function *merge* simply merges the content of the new partial paper and existing paper. The underlying *fetchPaperById* database function behaves as expected: it returns the labeled paper corresponding to the id.[7]

Unfortunately, this implementation has the drawback of always raising the current label to the label of the paper being updated. This can result in a scenario where actions that follow a partial update fail (e.g., writes to less sensitive entities), solely because the current label is overly restricting. Raising the current label to a point where the computation can no longer perform certain useful side-effects is known as *label creep* (Sabelfeld & Myers, 2003). Label creep does not compromise security, since the current label still protects all data in lexical scope. But, it hinders functionality. In the *partialUpdatePaper* example, label creep is particularly unappealing since *partialUpdatePaper* does not return any information about the existing paper—it simply writes back to the database. Ideally, we should be able to implement the *partialUpdatePaper* computation that operates on sensitive data, but avoid raising the current label and thus label creep.

In general, being able to perform computations on sensitive data without raising the current label is crucial to building practical applications. To this end, LIO provides the *toLabeled* function which can be used to execute an *LIO* action and subsequently restores the current context label. The type signature for this function is:

$$toLabeled :: Label\ \mathscr{L} \Rightarrow \mathscr{L} \rightarrow LIO\ \mathscr{L}\ \tau \rightarrow LIO\ \mathscr{L}\ (Labeled\ \mathscr{L}\ \tau)$$

The function takes a label l (the upper bound, describe below) and the *LIO* term t that computes on sensitive data. Intuitively, if the current label at the point where *toLabeled* l t gets executed is l_{cur}, *toLabeled* executes t and restores the current label to l_{cur}, i.e., *toLabeled* provides a separate context in which t is evaluated. Of course, returning the result of t directly would allow for trivial leaks of sensitive data. Hence, *toLabeled* labels the result of t with l. This design decision effectively states that the result of t is protected by label l, as opposed to the current label at the point t completed. Of course, *toLabeled* requires that the result of t not be more sensitive than l.

[7] Note that this has the implication that id's are effectively public. However, since the number of elements in the database is public (as revealed by the length of the list returned by *fetchPapers*), this is not surprising.

TOLABELEDCTX

$$\frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ t_1\ t_2 \rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ t_1'\ t_2 \rangle}$$

TOLABELED

$$\frac{\begin{array}{c} l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \\ \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle \overset{n\ *}{\longrightarrow} \langle l_{\text{cur}}', c_{\text{cur}}', m' \mid \mathit{LIO}^{\mathsf{TCB}}\ t' \rangle \qquad l_{\text{cur}}' \sqsubseteq l \rightsquigarrow \mathit{True} \end{array}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ l\ t \rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t') \rangle}$$

Fig. 12. Semantics for *toLabeled*.

To formally describe the semantics of *toLabeled*, we extend terms with the *toLabeled* primitive: $t ::= \cdots \mid \mathit{toLabeled}\ t_1\ t_2$ and give two new reduction rules in Fig. 12. In both rules, the current label and clearance are preserved. Rule (TOLABELEDCTX) simply reduces the label argument. Rule (TOLABELED) specifies the non-trivial case. As noted above, the label l is used to label the result of t. Hence, the rule first ensures that we are not trying to create a labeled value below the current label (or above the current clearance, see Section 3), i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$. The rule then completely reduces t to an *LIO* value.[8] If the current label l_{cur}' at the time of completion is below the provided upper bound l, then "transferring protection" of the result t' from l_{cur}' to l is safe and we thus simply return the result, labeled with l. Observe that if $l_{\text{cur}}' \sqsubseteq l \rightsquigarrow \mathit{False}$, then labeling the result t' with l might result in a leak, e.g., if t' actually contains information above l. In Section 5, we consider the cases where these conditions do not hold. We finally remark that the (TOLABELED) increments the index n to indicate that *toLabeled* was executed. This decoration is used to simplify the proof burden and is further explained in Section 6.

Example 2 (*Partially updating papers*)
Returning to our partial update λChair example, we can now use *toLabeled* in a straightforward way to implement *partialUpdatePaper*. This new implementation is given below.

```
partialUpdatePaper :: Id → PartialPaper → DC ()
partialUpdatePaper i new = do
  -- Get the existing paper according to its id:
  lold ← fetchPaperById i
  lnew ← toLabeled (labelof lold) (do
    old ← unlabel lold
    -- Merge the new (partial) paper and existing paper:
    return (merge new old))
  -- Perform actual update:
  updatePaper i lnew
```

This implementation is almost identical to the original one. It only differs in wrapping the part of the code that is computing on sensitive data with *toLabeled*. Specifically, it wraps the part of the code that unlabels the existing paper and performs the merge. (Since

[8] By using big-step semantics, we do not need to rely on the use of trusted functions that (save and) restore the current label and clearance.

toLabeled returns a labeled value, we no longer need to explicitly *label* the merged paper—we simply return it.) The current label within the *toLabeled* blocks is raised to the join of the current label and the label of the existing paper (*labelOf lold*) by function *unlabel*. Importantly, however, the current label before and after calling *partialUpdatePaper* remains the same.

2.5.1 An alternative semantics for toLabeled

Naturally, one may ask why *toLabeled* demands that we provide the label of the result as an argument, as opposed to simply using the final current label of the executed computation. Indeed, an early version of LIO had such an implementation. The reduction rule for this alternative function

$$\mathit{toLabeled}' :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIO}\ \mathscr{L}\ \tau \rightarrow \mathit{LIO}\ \mathscr{L}\ (\mathit{Labeled}\ \mathscr{L}\ \tau)$$

is given below.

TOLABELED’

$$\frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\ *} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}\ t'\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}'\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l'_{\text{cur}}\ t')\rangle}$$

But, different from the version of LIO as presented in this paper, inspecting the label of labeled values with *labelOf* must raise the current label to the join of the current label and label of the value. The semantics for this alternative function

$$\mathit{labelOf}' :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \rightarrow \mathit{LIO}\ \mathscr{L}\ \mathscr{L}$$

is given below.

LABELOF’

$$\frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{labelOf}'\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l'_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ l\rangle}$$

This difference is particularly important since information can otherwise be leaked by encoding it into the labels themselves (Russo & Sabelfeld, 2010; Buiras *et al.*, 2014). To illustrate this point, consider the 3-point lattice $\mathscr{L}_3 = \{\mathit{Public}, \mathit{Secret}, \mathit{TopSecret}\}$ and the following code that uses *toLabeled*′ and *labelOf* to leak the value of a secret Boolean.

```
leakBool :: Labeled ℒ3 Bool → LIO ℒ3 Bool
leakBool secretBool = do
  -- Current label is Public
  secretBool′ ← toLabeled′ (do
                   s ← unlabel secretBool   -- Raise current label to Secret
                   -- Raise label to TopSecret if s is True
                   when s (raiseLabel TopSecret))
  -- Current label is Public
  return (labelOf secretBool′ ≡ TopSecret)
    where raiseLabel l = label l () ≫= unlabel
```

The key distinction between the two designs is *what label is used to protect the label of a labeled value* (Buiras *et al.*, 2014). (Recall that in an IFC system every piece of data must be labeled—this include labels themselves.) In the early version of LIO (that with *toLabeled*$'$ and *labelOf*$'$) the label on the label of a value was the label itself. Hence, inspecting the label of a value required raising the current label. Importantly, however, *toLabeled*$'$ did not require programmers to supply an upper bound label for the labeled result. In contrast, the current version of LIO considers the current label l_{cur} as the label protecting the labels of labeled values. In this system, inspecting the label of a value does not require raising the current label, and *labelOf* is, in turn, pure. Of course, the trade-off is that the label on the result produced by *toLabeled* must be provided a-priori.

Our experience with building λChair and other larger-scale applications has shown that the ability to inspect labels outweighs the "burden" of specifying an upper bound for *toLabeled*. The interested reader is referred to (Giffin *et al.*, 2012) for a description of an example system built on top of LIO. In fairness, most of the systems and applications we built on top of LIO are web-centric and while we believe this experience to extend to other domains, evaluating this trade-off for other kinds of applications is an interesting direction for future work.

3 Addressing covert channels with clearance

IFC systems do not typically restrict what data code can read, rather—and as we have done thus far—they only restrict where the code can write to once it has read the data. Similarly, code can always write to channels or create objects with arbitrary labels, as long as doing so does not leak information, i.e., code can always write to and allocate entities more sensitive than the current label. But, in many cases it is useful to execute code with *least privilege* by limiting its access to the data/entities it needs to perform its task (Saltzer & Schroeder, 1975). This principle not only simplifies security auditing, but, as shown in this section, it also eliminates the opportunity for code to leak sensitive data by exploiting covert channels (Lampson, 1973). LIO introduces the notion of *clearance* to language-based IFC systems (Stefan *et al.*, 2011b), later adopted by Breeze (Hriţcu *et al.*, 2013), as a means for restricting access to certain labeled entities. Clearance in LIO can be seen as a particular discretionary access control mechanism (DAC) integrated into a IFC system, where DAC security checks are performed before their IFC counterparts (Stoughton, 1981).

3.1 Restricting data-access with clearance

The current clearance c_{cur} is a label tracked by the *LIO* monad alongside the current label l_{cur}; in our formalization, the clearance appears as the second component of a program configuration $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t\rangle$. LIO restricts access to certain labeled entities using the clearance in two different ways.

First, the clearance is used to restrict the reading of overly-sensitive data by enforcing that the current clearance always be an upper bound on the current label, i.e., for all valid program configurations $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t\rangle$, it is the case that $l_{\mathrm{cur}} \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True}$. This restriction is enforced by the LIO interface. For example, *unlabel* as given in rule (UNLABEL) of Fig. 10 only unlabels the labeled value if raising the current label l_{cur} will not result in a current label l'_{cur} that is above the current clearance, i.e., $l'_{\mathrm{cur}} \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True}$. In a similar way, before reading from a file or reference (see Section 4), we ensure that raising the current label will not violate this guarantee.

The use of clearance to restrict code from reading certain entities is a form of discretionary access control; we can prevent malicious code from exploiting covert channels to leak overly-sensitive information by ensuring that it cannot read such data. As an example, suppose that the partial update function in λChair is implemented by a third-party developer (e.g., to implement a better merging function). If the developer is malicious, they can use the partial update function to leak the contents of a competing author's paper through covert channels. Indeed, this is simple since the developer can create an account on the λChair platform and take on the role of an author to ensure that their malicious code is executed. A malicious version of *partialUpdatePaper* is given below.

```
leakyPartialUpdatePaper :: Id → PartialPaper → DC ()
leakyPartialUpdatePaper i new = do
  -- Get all existing papers:
  papers ← fetchPaperById i
  -- Leak information about some of the papers
  mapM maybeLeak papers
  -- Execute the normal partial update:
  partialUpdatePaper i new
    where maybeLeak lpaper = toLabeled (labelOf lpaper) (do
            paper ← unlabel lpaper
            -- If the paper has a specific author, leak it:
            when (paperAuthors paper ≡ ...) (leakToCovertChannel paper))
```

Here, we use function *leakToCovertChannel* to leak information about papers written by certain authors; otherwise the function behaves in the same way as the normal *partialUpdatePaper* code. The function *leakToCovertChannel* leaks (part of) the sensitive paper content through a covert channel. For instance, the code can leak information by diverging (or not) according to the paper content, i.e., one bit at a time through the termination covert channel (Askarov *et al.*, 2008); alternatively, it can leverage the external timing covert channel (Agat, 2000) to leak the information by delaying the response according to the content, etc. Using clearance, we can prevent such leaks by setting the clearance to the label of the browser—in this case, the *leakyPartialUpdate* will fail to *unlabel* papers which the requesting user, i.e., the attacker, is not allowed to read. Since the code running on behalf of one user does not have access to another user's data, it cannot leak it—the code can only leak data it can already read.

The second role of clearance is to restrict code from writing to and allocating entities labeled above the clearance. For example, *label* as given in rule (LABEL) of Fig. 10 only creates a *Labeled* value if the label of the value is bounded by the clearance. Similarly, *toLabeled* as given in rule (TOLABELED) of Fig. 12 requires the upper bound of the result to be below the clearance. In a similar way, before creating or writing to a file or reference (see Section 4), we ensure that their label is below the current clearance. As in (Zeldovich *et al.*, 2006), this addresses attacks in which malicious code duplicates sensitive data, e.g., by copying a file, only to read it later, when the system policy changes (e.g., in λChair, promoting a member to a co-chair and granting them the corresponding

$$\textsc{getClearance}\qquad \frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{getClearance}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ c_{\text{cur}}\rangle}$$

$$\textsc{lowerClearanceCtx}\qquad \frac{t \rightsquigarrow t'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ t'\rangle}$$

$$\textsc{lowerClearance}\qquad \frac{l_{\text{cur}} \sqsubseteq c'_{\text{cur}} \rightsquigarrow \mathit{True} \qquad c'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ c'_{\text{cur}}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c'_{\text{cur}}, m \mid \mathit{return}\ ()\rangle}$$

Fig. 13. Semantics for clearance related terms.

privileges). While, within a single run, LIO programs can use robust declassification as in (Zdancewic & Myers, 2001; Waye *et al.*, 2015) to reason about policy changes, without clearance, reasoning about the consequence of a system policy change across multiple program runs is more difficult. We refer the intersted reader to (Zeldovich *et al.*, 2006) for a more detailed consideration of this use case.

3.2 Making clearance first-class

To leverage clearance for isolation, as described above, we execute a term in a configuration that has initially set the desired clearance. Of course, in many applications it is useful to be able to "drop" privileges and continue executing with least privilege (Saltzer & Schroeder, 1975). For example, in λChair when authenticating user requests, the clearance must be high enough to read credentials, but once the authentication is complete, having access to such information is unnecessary and dangerous: a simple bug in the code that generates an HTML list of the user's papers could potentially leak the credentials. Hence, we provide a means for inspecting and manipulating the clearance. Specifically, we provide:

$$\begin{aligned}
&\mathit{getClearance} && :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIO}\ \mathscr{L}\ \mathscr{L}\\
&\mathit{lowerClearance} && :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \rightarrow \mathit{LIO}\ \mathscr{L}\ ()
\end{aligned}$$

The *getClearance* and *lowerClearance* functions are used to get and set the current clearance, respectively.

We add the primitives *getClearence* and *lowerClearance* to the syntactic category of terms $t ::= \cdots \mid \mathit{getClearance} \mid \mathit{lowerClearance}\ t$ and formally describe its semantics in Fig. 13. The rules are mostly self-explanatory. We solely highlight that the premise in rule (LOWERCLEARANCE) requires the new current clearance c'_{cur} to be below the current clearance c_{cur} and above the current label. By lowering the clearance, code can effectively run with least privilege. Of course, allowing code to arbitrarily raise the clearance would trivially prevent us from confining untrusted code—hence code can only decide to access fewer entities.

However, recall from rule (TOLABELED) that *toLabeled* restores the current label and clearance. Hence, combined with *toLabeled*, we can use *lowerClearance* to execute a term t, at a lower clearance, without lowering the current clearance:

$$\begin{array}{l}
\mathit{withClearance} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \mathit{LIO}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{Labeled}\ \mathscr{L}\ \tau)\\
\mathit{withClearance}\ c'_{\mathrm{cur}}\ t = \mathit{toLabeled}\ c'_{\mathrm{cur}}\ (\mathit{lowerClearnce}\ c'_{\mathrm{cur}} \gg t)
\end{array}$$

This use of *toLabeled* addresses the dual to the label creep described in Section 2.5: by lowering the current clearance a program can reach a state where $l_{\mathrm{cur}} = c_{\mathrm{cur}}$, at which point it cannot read or write to entities more-sensitive than l_{cur}. More interestingly, this enables powerful security patterns. For instance, it allows arbitrary untrusted code to treat code it depends on as untrustworthy. Indeed, this primitive can be used to address the poison pill attacks described in (Hriţcu *et al.*, 2013), wherein untrusted libraries carry out denial of service attacks via label creep. Additionally, *withClearance* can be used to structure programs in such a way that different components execute with least privilege and are isolated from one another. For example, in λChair, we can wrap request handlers with *withClearance* to isolate requests based on the user (browser) label. This is similarly done in the Hails web framework, when serving HTTP requests and accessing database tables, which themselves have a notion of clearance for the labels on stored data (Giffin *et al.*, 2012).

4 Mutable labeled references

Many practical applications rely on imperative data-structures, often implemented using mutable reference. In the context of λChair mutable references can, for example, be used to implement an efficient in-memory database. Indeed, by modeling each paper as a labeled reference, instead of a labeled immutable value, updating a paper becomes very cheap; it simply amounts to writing to a reference, as opposed to creating a large immutable data structure (that contains the rest of the papers).

Unsurprisingly, LIO provides labeled alternatives to Haskell's *IORef*s (Peyton Jones, 2001). The LIO reference API is given below.

$$\begin{array}{l}
\textbf{data}\ \mathit{LIORef}\ \mathscr{L}\ \tau\\
\mathit{newLIORef}\ \ :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{LIORef}\ \mathscr{L}\ \tau)\\
\mathit{readLIORef}\ \ :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIORef}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau\\
\mathit{writeLIORef} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIORef}\ \mathscr{L}\ \tau \to \tau \to \mathit{LIO}\ \mathscr{L}\ ()
\end{array}$$

While the implementation of secure references can vary, we simply wrap Haskell's *IORef*s. Intentionally, this API resembles the standard Haskell API for mutable references. The key difference is that the function for creating references takes an additional argument: the label of the reference.

To formally describe this API, we extend our calculus with references as shown in Fig. 14. Like $\mathit{Labeled}^{\mathsf{TCB}}$, the $\mathit{LIORef}^{\mathsf{TCB}}$ constructor is restricted to the TCB and is strict in its first argument. References are created with

$$\begin{array}{llcl}
\text{Values} & v & ::= & \cdots \mid \mathit{LIORef}^{\mathsf{TCB}}\ v\ a\\
\text{Terms} & t & ::= & \cdots \mid \mathit{newLIORef}\ t_1\ t_2 \mid \mathit{readLIORef}\ t\\
 & & & \mid \mathit{writeLIORef}\ t_1\ t_2\\
\text{Types} & \tau & ::= & \cdots \mid \mathit{LIORef}\ \mathscr{L}\ \tau
\end{array}$$

Fig. 14. Formal syntax for references.

$$\textsc{newLIORefCtx}\qquad \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{newLIORef}\ t_1\ t_2 \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{newLIORef}\ t_1'\ t_2 \rangle}$$

$$\textsc{newLIORef}\qquad \frac{l_{\mathrm{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True} \qquad \mathrm{fresh}(a) \qquad m' = m[a \mapsto \mathit{Labeled}^{\mathsf{TCB}}\ l\ t]}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{newLIORef}\ l\ t \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid \mathit{return}\ (\mathit{LIORef}^{\mathsf{TCB}}\ l\ a) \rangle}$$

$$\textsc{readLIORefCtx}\qquad \frac{t \rightsquigarrow t'}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{readLIORef}\ t \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{readLIORef}\ t' \rangle}$$

$$\textsc{readLIORef}\qquad \frac{v = m(a)}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{readLIORef}\ (\mathit{LIORef}^{\mathsf{TCB}}\ l\ a) \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{unlabel}\ v \rangle}$$

$$\textsc{writeLIORefCtx}\qquad \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{writeLIORef}\ t_1\ t_2 \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{writeLIORef}\ t_1'\ t_2 \rangle}$$

$$\textsc{writeLIORef}\qquad \frac{l_{\mathrm{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True} \qquad m' = m[a \mapsto \mathit{Labeled}^{\mathsf{TCB}}\ l\ t]}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{writeLIORef}\ (\mathit{LIORef}^{\mathsf{TCB}}\ l\ a)\ t \rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid \mathit{return}\ () \rangle}$$

$$\textsc{labelOfLIORef}\qquad \frac{}{\mathit{labelOf}\ (\mathit{LIORef}^{\mathsf{TCB}}\ l\ a) \rightsquigarrow l}$$

Fig. 15. Semantics for monadic LIO terms related to references.

newLIORef, read with function *readLIORef*, and modified with *writeLIORef*. We overload the *labelOf* function to allow code to inspect the label of a reference.[9]

The reference store—spanned over by metavariable m—is a map from addresses—spanned over by metavariable a—to labeled values.[10] Since we do not provide any mechanisms for explicit deallocation or address inspection/comparison in the LIO API we can model the store as an infinitely-large map, while allowing the implementation to safely garbage collect unused references as necessary.[11] In our formalization, this memory store appears as the third component of a program configuration $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle$.

The reduction rules for references are given in Fig.15. When creating a reference, as given by rule (NEWLIOREF), *newLIORef* $l\ t$ creates a labeled value that guards t with label

[9] In our implementation, we use a typeclass *LabelOf* to define the *labelOf* function. Both *LIORef* and *Labeled* are instances of this class.

[10] Since the label of a reference accompanies the address (both wrapped by the $\mathit{LIORef}^{\mathsf{TCB}}$ constructor), an alternative memory store that simply maps addresses to terms is sufficient—we chose the labeled-store approach to simplify the proof burden (see Section 6).

[11] Non-opaque pointers could potentially be used to leak information (e.g., by freeing a reference in a secret context only to allocate a reference and inspect its address in a public context). Adapting LIO to deal with non-opaque pointers can be done as in (Hedin & Sands, 2006).

l and stores it in the memory store at a new, fresh, address a. Subsequently, the function returns an *LIORef* value that contains the reference label and the address where the term is stored. (Like $\mathit{Labeled}^{\text{TCB}}$, the constructor $\mathit{LIORef}^{\text{TCB}}$ is not part of the surface syntax and thus cannot be abused by untrusted code.) Rule (READLIOREF) specifies the semantics for reading a labeled reference; reading the term stored at address a simply amounts to unlabeling the value $m(a)$ stored at the underlying address. Function *writeLIORef*, specified by rule (WRITELIOREF), updates the memory store with a new labeled term t for the reference at location a, leaving the label intact. Note that in the latter three rules, we impose the restriction that the label of the reference l must be bound by the current label and clearance, i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$ and $l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}$. This ensures that we both preserve the confidentiality of data in scope and avoid reading/modifying entities above the clearance. It is worth remarking that when one considers the current label l_{cur} as the dynamic version of the *pc*, our restriction that the label of the reference be above the current label ($l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$) when writing to the reference is similar to the one imposed by other IFC λ-calculi (Zdancewic, 2002; Austin & Flanagan, 2009). The rule *labelOf*, given by (LABELOFLIOREF), is self-explanatory and we do not discuss it further.

5 Exception handling

Like references, exceptional control flow is common in real-world applications. As already noted, LIO provides support for throwing and catching exceptions. Code can throw an exception using the *throwLIO* function and catch exceptions using *catchLIO*:

$$\begin{aligned}&\mathit{throwLIO} :: (\mathit{Exception}\ e, \mathit{Label}\ \mathscr{L}) \Rightarrow e \to \mathit{LIO}\ \mathscr{L}\ \tau\\&\mathit{catchLIO} :: (\mathit{Exception}\ e, \mathit{Label}\ \mathscr{L}) \Rightarrow \mathit{LIO}\ \mathscr{L}\ \tau \to (e \to \mathit{LIO}\ \mathscr{L}\ \tau) \to \mathit{LIO}\ \mathscr{L}\ \tau\end{aligned}$$

This API is identical to that of standard Haskell, except that it operates in the *LIO* monad. Moreover, the semantics for these functions are standard.[12] Nevertheless, we must consider the implication on security when they are used in concert with other LIO library functions—in particular, *toLabeled*.

In Fig. 16, we formally extend values with exceptions ξ and a new *LIO* constructor ($\mathit{LIO}_X^{\text{TCB}}$), terms with the exception handling functions (*throwLIO* and *catchLIO*), and types with *Exception*s. For simplicity, we only consider a single exception type.

$$\begin{array}{ll}\text{Values} & v ::= \cdots \mid \xi \mid \mathit{LIO}_X^{\text{TCB}}\ t\\ \text{Terms} & t\ ::= \cdots \mid \mathit{throwLIO}\ t \mid \mathit{catchLIO}\ t_1\ t_2\\ \text{Types} & \tau ::= \cdots \mid \mathit{Exception}\end{array}$$

Fig. 16. Formal syntax for exceptions.

Fig. 17 gives the exception-related reduction rules. Function *throwLIO*, as given by rule (THROWLIO), raises an exception by simply lifting the exception term t into the *LIO* monad with constructor $\mathit{LIO}_X^{\text{TCB}}$. Indeed, the role of the $\mathit{LIO}_X^{\text{TCB}}$ constructor is to distinguish between exceptional and non-exceptional monadic control flow. Building on this, we add

[12] This is in contrast with the original semantics of exceptions as presented in (Stefan *et al.*, 2012b), where an explicit label was associated with every thrown exception. In comparison to the treatment of exceptions in (Stefan *et al.*, 2012b) and (Hriţcu *et al.*, 2013), the approach of this paper is considerably simpler.

THROWLIO

$$\frac{}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{throwLIO}\ t\rangle \xrightarrow{0} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{LIO}_X^{\mathsf{TCB}}\ t\rangle}$$

BINDEX

$$\frac{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{LIO}_X^{\mathsf{TCB}}\ t'_1\rangle}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t_1 \gg\!\!= t_2\rangle \xrightarrow{n} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{throwLIO}\ t'_1\rangle}$$

CATCH

$$\frac{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}\ t'_1\rangle}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{catchLIO}\ t_1\ t_2\rangle \xrightarrow{n} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{return}\ t'_1\rangle}$$

CATCHEX

$$\frac{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{LIO}_X^{\mathsf{TCB}}\ t'_1\rangle}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{catchLIO}\ t_1\ t_2\rangle \xrightarrow{n} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t_2\ t'_1\rangle}$$

Fig. 17. Semantics for exceptions without *toLabeled*. The remaining changes are given in Fig. 18.

a new reduction rule for bind ($\gg\!\!=$) that propagates exceptions; as shown by the (BINDEX) rule, bind re-throws the exception if the term under evaluation reduced to an exceptional monadic term ($\mathit{LIO}_X^{\mathsf{TCB}}\ t$). (We explicitly define the (BINDEX) in terms of *throwLIO* to more closely match our Haskell implementation.) Otherwise, it behaves as before, according to rule (BIND).

The semantics for *catchLIO* is also straightforward. Since throwing an exception depends on the information present in the lexical scope, *catchLIO* must retain the current label to reflect this fact; observe that all the *catchLIO* reduction rules in Fig. 17 leave the context intact. Rule (CATCH) specifies the case where the term does not raise an exception and reduces to a "normal" *LIO* value. Here, the value is simply returned. Rule (CATCHEX) specifies the case where the term raises an exception. In this case, the exception handler t_2 is applied to the exception t_1. We note that our semantics are lazy in the exception value, much in the same way as Haskell; neither *throwLIO* nor *catchLIO* force the evaluation of the exception.

The reduction rules of Fig. 17 take the standard approach of *propagating exceptions up the call stack until the nearest enclosing catchLIO*. Though necessary, this is not sufficient; without modifying the semantics of *toLabeled*, exceptions can be used to leak information. Consider the following function:

```
condThrow :: Labeled ℒ2 Bool → LIO ℒ2 ()
condThrow secretBool = do
  s ← unlabel secretBool
  when s (throwLIO ξ)
```

Suppose that *condThrow* is invoked with the current label *Public* and *secretBool* has label *Secret*. Then, *throwLIO* raises exception ξ if the secret is *True*; if the secret is *False* *condThrow* simply returns (). This function alone cannot be used to leak the secret, since the current label at the end of *condThrow* is *Secret*. But, by wrapping *condThrow* with *toLabeled*, we can avoid raising the current label when the secret is *False* and thus leak the value into a public reference:

$$\textsc{toLabeledEx}\quad \frac{\begin{array}{c} l_{\mathrm{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True} \\ \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t\rangle \overset{n\ *}{\longrightarrow} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{X}\ t'\rangle \qquad l'_{\mathrm{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \end{array}}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{toLabeled}\ l\ t\rangle \overset{n+1}{\longrightarrow} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}_{X}\ l\ t')\rangle}$$

$$\textsc{unlabelEx}\quad \frac{l_{\mathrm{cur}} \sqcup l \rightsquigarrow l'_{\mathrm{cur}} \qquad l'_{\mathrm{cur}} \sqsubseteq c_{\mathrm{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}^{\mathsf{TCB}}_{X}\ l\ t)\rangle \overset{0}{\longrightarrow} \langle l'_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{throwLIO}\ t\rangle}$$

$$\textsc{labelOf2}\quad \frac{}{\mathit{labelOf}\ (\mathit{Labeled}^{\mathsf{TCB}}_{X}\ l\ t) \rightsquigarrow l}$$

Fig. 18. Semantics for terms affected by exceptions.

```
leakSecret :: Labeled ℒ2 Bool → LIO ℒ2 Bool
leakSecret secretBool = do
  -- Create public reference:
  publicRef ← newLIORef Public True
  toLabeled Secret (catchLIO (do
    toLabeled Secret (condThrow secretBool)
    writeLIORef publicRef False   -- Write only if no exception is thrown
    )(λ_ → return ()))
  -- Read direct leak of secret:
  readLIORef publicRef
```

Assume that this function is invoked with a *Public* current label. First, the function creates a public reference *publicRef* initialized to *True*. Then, if the secret is *True*, the exception thrown by *condThrow* escapes the innermost *toLabeled* block up to the *catchLIO*, which invokes the handler. At this point the current label is *Secret*, since *condThrow* raised the label to read the secret. However, the outer *toLabeled* restores the current label to *Public*. This allows us to read the *publicRef*, which is still *True*. By contrast, if the secret is *False*, *condThrow* simply returns (); the enclosing *toLabeled* ensures that the current label remains *Public*. At this point, we write *False* into the public reference. Finally, we again read and return the reference contents. In both cases the returned value corresponds to the secret boolean.

This code illustrates that the standard propagation of exceptions up the call stack until reaching the nearest enclosing *catchLIO* is not sufficient. LIO must only propagate exceptions up to the nearest *catchLIO or toLabeled*. Intuitively, the correct semantics for *toLabeled* are as before with the added requirement that all exceptions be caught by it: regardless of how the computation enclosed by *toLabeled* terminates—with an exception or value—a *Labeled* value must always be returned. In other words, we adapt the semantics of some LIO actions (including *toLabeled*) to secure the exception handling mechanism provided by *throwLIO* and *catchLIO*.

Formally, we extend values with another *Labeled* constructor $v ::= \cdots \mid \mathit{Labeled}^{\mathsf{TCB}}_{X}\ v\ t$, that encodes the fact that t is an exception. The additional rule for *toLabeled* is given by (TOLABELEDEX) in Fig. 18: if term t raises an exception (that is not caught) $\mathit{LIO}^{\mathsf{TCB}}_{X}\ t'$, we wrap the exception by the new *Labeled* constructor. When unlabeling such a labeled value, as given by (UNLABELEX), LIO simply propagates the exception. Of course, *unlabel* raises the current label, ensuring that information from the point of the throw cannot be leaked. Finally, (LABELOF2) gives the additional rule for *labelOf*, which allows programs to inspect the label of *Labeled* values wrapping exceptions. Note that we do not allow code to distinguish between $\mathit{Labeled}^{\text{TCB}}$ and $\mathit{Labeled}_X^{\text{TCB}}$; doing so would allow for trivial leaks.

With these modifications in place, we highlight that the actions in *leakSecret* following the *toLabeled* block will always be executed, even if an exception is raised inside *condThrow*. Intuitively, we close the leak due to exception propagation by simply assuring that the execution of (possibly public) actions following a *toLabeled* block does not depend on the abnormal termination of a computation wrapped by *toLabeled*. In a similar manner, but using concurrent threads, we can address leaks due to the timing and non-termination behavior of the enclosed computation (Stefan *et al.*, 2012a).

We remark that closing leaks due to exception propagation, as such, is not without cost. In particular, “delaying” exceptions raised within *toLabeled* blocks raises two challenges. First, developers need to handle exceptions at the point of *unlabel*ing data, even though the exception was potentially raised in a different part of the program. This imposes a somewhat nonstandard, asynchronous programming model which closely resembles promises (Friedman & Wise, 1976; Miller, 2006). We have found that, in general, debugging IFC programs is non-trivial for average developers (Giffin *et al.*, 2012).

To address this, our LIO implementation associates a stack-trace like data-structure with exceptions. Internally, LIO defines an annotation function which is used in the rest of the library:

$$\mathit{withContext} :: \mathit{String} \to \mathit{LIO}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau$$

This function takes a string message (typically the name of the function) and the action to execute, and returns an action that wraps the original action with *catchLIO*. The catch is used to interpose any thrown monitor failure exceptions as to add the annotation message before rethrowing it. Consider the following program:

$$\mathit{withClearance}\ \mathit{lAliceOrBob}\ (\mathit{label}\ \mathit{lAlice}\ 42)$$

Here, the program starts with an initial current label and clearance set to *lPublic*, where *lPublic* ⊑ *lAliceOfBob* ⊑ *lAlice*, but neither relations flow hold in the reverse direction. This program throws an exception because it attempts to create a labeled value above the current clearance (within the *withClearance* block). In particular, it produces the following error message:

```
LabelError {
  lerrContext = ["withClearance", "label"],
  lerrFailure = "guardAllocP",
  lerrCurLabel = lPublic,
  lerrCurClearance = lAliceOrBob,
  lerrPrivs = [ ],
  lerrLabels = [lAlice]
}
```

Note that the error message contains a lot of useful information:

- A stack-trace like context of the functions called before the program terminated.
- The actual point of failure; in this case an internal function within *label* called *guardAllocP*, which performs the actual $\sqsubseteq$-check before creating a labeled value.
- The current label and clearance when the exception was thrown.
- The privileges supplied to the action that threw the exception.
- The labels supplied to the action that thew the exception.

While an actual stack trace would be more useful, this information has proved very useful in practice when building our Hails web framework and applications on top of it;[13] particularly because developers can use *withContext* to annotate their own constructs. We remark that in an imperative language, debugging could be simplified even further.

The second issue with delaying exceptions is that it may lead to scenarios in which exceptions go unnoticed. Consider, for example, executing a sensitive computation with the sole interest of performing a side-effect (e.g., a write to the database). Since, the result of the computation is of no interest, we are likely to never *unlabel* the result and, as a result, overlook a failure—*toLabeled* catches all exceptions.

Concretely, suppose we attempt to update a paper stored in the database with a value of type *LabeledPaper*, which was produced as a result of a *toLabeled* computation. (Our *partialUpdatePaper* is an example of one such computation.) Further suppose that the *toLabeled* computation read data more sensitive than its bound, which should be the paper label. In such a case we would write an exceptional value to the database, which will only be observed by the user on a follow-up read. While this is not an issue from a security stance, it is likely not the desired or expected behavior; the computation should not delay the exception and instead reply to the user with an error.

While, in practice, users can also use *label* to create labeled values that contain pure exceptions (e.g., using Haskell's *throw*), an alternative strict label type (e.g., *StrictLabeled*) can ensure that such labeled values never contain exceptions. Given this, an alternative *toLabeled* definition could simply return a labeled variant (see Section 5.1), i.e., a value of type $\mathit{StrictLabeled}\ \mathscr{L}\ (\mathit{Either\ Exception}\ \tau)$. While this alternative API would not prevent code from ignoring the result (and thus, the errors), it would prevent developers from overlooking exceptions raised in a *toLabeled* blocks when they try to reuse the resultant values (e.g., to insert them into the database).

In practice, we found that using clearance to restrict what a computation can read and write within a *toLabeled* block and having to provide an upper bound label to *toLabeled* (and the fact that one can freely inspect labels) help with reasoning about and preventing IFC monitor failures a-priori. But, of course, other failures (e.g., network connection failures) are less predictable and in such cases we cannot avoid inspecting the return values to catch any delayed exceptions. In such cases, LIO's support for declassification, though not discussed in this paper, was used to "safely leak" the success/failure of a sensitive computation. In general, we did not find delayed exceptions to be a hindrance. However,

[13] In debugging mode, it is possible to get more accurate information by rewriting the Haskell source to wrap at every bind, and also add file and line number annotations. We do not do this in production because of performance.

our experience comes from building Hails (Giffin *et al.*, 2012) and applications on top of Hails, which build on the concurrent version of LIO that uses threads in place of *toLabeled*; in these applications, we mostly relied on *toLabeled*-like construct to execute code in which failure was easy to predict (e.g., transformers from strings to abstract data types). Lastly, we refer the reader to the work of (Hriţcu *et al.*, 2013) for a more exhaustive discussion on the various design points of delayed exceptions.

5.1 Recovering from monitor failures

Our reduction rules given thus far in Fig. 2–18 do not consider cases where label checks fail. Like for other dynamic IFC systems (e.g., (Askarov & Sabelfeld, 2009b; Sabelfeld & Russo, 2009; Austin & Flanagan, 2009; Austin & Flanagan, 2010; Devriese & Piessens, 2011)), this would imply aborting the program execution when a monitor failure occurs. For practical systems, this approach is not appropriate: we cannot halt the system when a λChair request handler is about to violate IFC. Moreover, it is not safe—this introduces a covert channel (Myers & Liskov, 1997).

As we previously mentioned, LIO and Breeze (Hriţcu *et al.*, 2013) differ from most other dynamic IFC systems in using exceptions to encode monitor failures. For example, when the security conditions in rule (UNLABEL) are not met, we throw an exception:

$$\textsc{unlabelFail}\quad \frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{False}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}^{\text{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{throwLIO}\ \xi_{IFC}\rangle}$$

Here, ξ_{IFC} is simply an exception containing information about the failure. In the same way, we provide reduction rules dual to those of Fig. 10–18 that simply throw exceptions when a security condition is not met. We do not discuss these rules further since they are straightforward. The only interesting case is a particular failure of *toLabeled*, given below.

$$\textsc{toLabeledFail}\quad \frac{\begin{array}{c} l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \\ \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\ *} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid v\rangle \qquad l'_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{False} \end{array}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ l\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}_X^{\text{TCB}}\ l\ \xi_{IFC})\rangle}$$

Here, the enclosed term t raises the current label l'_{cur} above the upper bound l. By simply throwing an exception we would potentially be leaking information about data more sensitive than l_{cur}. (Malicious code can "throw" an exception by raising the current label above the upper bound imposed by *toLabeled*, reintroducing the attack from the previous section.) As mentioned before, *toLabeled* must return a labeled value. Therefore, we return a labeled value that contains an exception that encodes the monitor failure; at the point of *unlabel*, this "delayed" exception is raised.

By encoding monitor failures with exceptions, as opposed to stopping the program, LIO allows untrusted code to catch exceptions and safely recover from attempted IFC violations. Consider, for instance, the following function that unlabels a *Labeled* value and returns an *Either* value to indicate the success or failure of the operation:

$\mathit{safeUnlabel} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \rightarrow \mathit{LIO}\ \mathscr{L}\ (\mathit{Either}\ \mathit{Exception}\ \tau)$
$\mathit{safeUnlabel}\ \mathit{lv} = \mathit{catchLIO}\ (\mathbf{do}\ v \leftarrow \mathit{unlabel}\ \mathit{lv}$

```
      return (Right v)
   ) (λe → return (Left e))
```

If the label of *lv* is above the current clearance or if the value is a labeled exception, the LIO *unlabel* throws an exception (raising the label in the latter case), which is handled by simply returning the exception wrapped with the *Left* constructor. If no exception is raised, the current label is raised and the unlabeled result is returned, wrapped by *Right*. As discussed in (Hriţcu *et al.*, 2013), this is generally a very useful feature since it treats code in an egalitarian fashion, and allows one to integrate untrusted code in an application without having to worry that the code will halt the system by causing a monitor failure.

We remark that, unlike our original treatment of exceptions (Stefan *et al.*, 2012b), the (toLabeledFail) rule treats normal and exceptional results of a failed *toLabeled* block the same. This means that if a computation within a *toLabeled* block raised its current label above the bound and terminated with an exception, the exception will be hidden. (Though, a non-exceptional value would be hidden too.) As for Breeze's $\lambda^{\langle\rangle}_{\mathit{throw}+\mathbf{D}}$ calculus, this means that delayed exceptions are isomorphic to labeled tagged variants, i.e., values of type *Labeled* $\mathscr{L}$ (*Either Exception* τ). The trade-off between these semantics and our original ones are explored in detail in (Hriţcu *et al.*, 2013). The downside of our current approach is clear: error message are hidden, thus making it more difficult to debug LIO programs.[14] However, this trade-off comes with a benefit: *all* exceptions, including delayed exceptions, can be caught. (After all, when unlabeled, delayed exceptions, are isomorphic to tagged variants.)

This is not necessarily true of our original calculus. To understand the difference, suppose an exception is raised in a *toLabeled* block with an upper bound set to l; further suppose that the current label when exception is raised is l', where $l' \not\sqsubseteq l$. Since exceptions are not hidden (in our original calculus), when *unlabel*ing such delayed exceptions, the *unlabel* primitive re-threw the exception, raising the current label l_{cur} to $l_{\text{cur}} \sqcup l \sqcup l'$. Unfortunately, wrapping *unlabel* with a *catchLIO* does not guaranteed that the exception will be caught—in particular, if the l' is not below clearance, *catchLIO* would simply propagate the exception. At a high level, this effectively means that code cannot unlabel values from an untrusted computation without risking a *poison pill attack* (Hriţcu *et al.*, 2013), i.e., attacks wherein untrusted code running in a *toLabeled* block render outer computations useless by raising the current label above the *expected* label of the labeled value. Of course, code can always use *withClearance* to avoid such attacks, but this approach is less usable.

6 Security guarantees

In this section, we show that programs written in LIO satisfy noninterference and a form of discretionary access control. Informally, noninterference states that secret values cannot be leaked by LIO programs, while DAC ensures that computations cannot bypass the restrictions imposed by clearance to access or create arbitrary data. Before delving into

[14] We remark that this can be improved by keeping track of the precise point within the *toLabeled* block that the current label was raised above the bound and adding this to the exception stack-trace discussed above.

the details of these security guarantees, we first highlight some notational difference with the previous sections and describe the extent of our mechanization in Coq.

Notation To allow for incremental introduction of concepts, in the previous section we used $\mathit{LIO}^{\mathsf{TCB}}$ and $\mathit{LIO}^{\mathsf{TCB}}_X$ constructors to respectively denote non-exceptional and exceptional monadic *LIO* terms that have been executed to the point of containing no more side effects. In this section, we use a single constructor that additionally takes a boolean argument to indicate whether the value is an exception or not: term $\mathit{LIO}^{\mathsf{TCB}}_b\ t$ corresponds to $\mathit{LIO}^{\mathsf{TCB}}\ t$ if $b = \mathbf{true}$ and $\mathit{LIO}^{\mathsf{TCB}}_X\ t$ if $b = \mathbf{false}$. Similarly, we use $\mathit{Labeled}^{\mathsf{TCB}}_b$, with $b \in \{\mathbf{true}, \mathbf{false}\}$, instead of the $\mathit{Labeled}^{\mathsf{TCB}}$ and $\mathit{Labeled}^{\mathsf{TCB}}_X$ constructors.

Mechanized proofs We formalized a large subset of the calculus, described in Section 2, using the Coq theorem prover. The mechanized subset omits references and the reduction rules corresponding to monitor failures described in Section 5.1. Moreover, the Coq implementation uses a concrete four-point lattice similar to that shown in Fig. 3. For this subset, we mechanized the propositions, lemmas, theorems, and proofs given below; we distinguish the non-mechanized parts of the proofs with the symbol ✎. We leave the extension to the full calculus with an abstract lattice to future work.

6.1 Noninterference

In this section, we prove that LIO satisfies noninterference using the *term erasure* technique from (Li & Zdancewic, 2010; Russo *et al.*, 2008). Intuitively, the term erasure technique allows us to show that a program satisfies noninterference by showing that the behavior of the program with all the sensitive data (classified above l) "erased" cannot be distinguished by an attacker (at observation level l) from the behavior of the original program.

To model such programs, we extend our calculus and reduction rules with erased terms, denoted by a new terminal $\bullet$, as follows:

$$t ::= \cdots \mid \bullet \qquad\qquad \vdash \bullet : \tau \qquad\qquad \overset{\textsc{hole}}{\bullet \rightsquigarrow \bullet} \qquad\qquad \overset{\textsc{hole}\mathrm{LIO}}{\langle \bullet, \bullet, \bullet \mid \bullet \rangle \xrightarrow{n} \langle \bullet, \bullet, \bullet \mid \bullet \rangle}$$

Intuitively, an erased term can have any type. Moreover, an erased term or configuration, the latter represented by $\langle \bullet, \bullet, \bullet \mid \bullet \rangle$, always reduces to itself. We use a meta-level *erasure function* $\varepsilon_l(\cdot)$ to replace all terms more sensitive than the attacker's observation level l with $\bullet$. To an attacker, terms and configurations above their observation level appear as $\bullet$; the new reduction rules also ensure that no information can be learned from the reduction of such terms (by effectively diverging).

Fig. 19 gives the definition of the erasure function for values, terms, memories, and configurations. For most values, the erasure function is simply the identity function, since most values are not heterogeneously labeled. Similarly, for most terms, the function is simply applied homomorphically (e.g., $\varepsilon_l(\ \mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3) = \mathbf{if}\ \mathit{True}\ \mathbf{then}\ \varepsilon_l(t_2)\ \mathbf{else}\ \varepsilon_l(t_3))$. There are only four interesting cases. First, when erasing a $\mathit{Labeled}^{\mathsf{TCB}}_b\ l_1\ t_2$ value, we erase the term t_2 protected by label l_1 to $\bullet$ when the label does not flow to l; otherwise we simply apply the function homomorphically. Second, we aggressively erase values that are about to be labeled with *label*. While the erasure function only erases values when the first

$$\varepsilon_l(\mathit{True}) = \mathit{True} \qquad \varepsilon_l(\mathit{False}) = \mathit{False} \qquad \varepsilon_l(()) = () \qquad \varepsilon_l(l_1) = l_1$$

$$\varepsilon_l(\mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ t) = \begin{cases} \mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ \varepsilon_l(t) & l_1 \sqsubseteq l \\ \mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\mathit{label}\ l_1\ t_2) = \begin{cases} \mathit{label}\ l_1\ \varepsilon_l(t_2) & l_1 \sqsubseteq l \\ \mathit{label}\ l_1\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle) = \begin{cases} \langle l_{\text{cur}}, c_{\text{cur}}, \varepsilon_l(m) \mid \varepsilon_l(t)\rangle & l_{\text{cur}} \sqsubseteq l \\ \langle \bullet, \bullet, \bullet \mid \bullet\rangle & \text{otherwise} \end{cases}$$

$$\varepsilon_l(m) = \{(a, \varepsilon_l(m(a)) : a \in \text{dom}(m) \text{ and } \mathit{labelOf}\ m(a) \sqsubseteq l\} \qquad \varepsilon_l(\bullet) = \bullet$$

Fig. 19. Erasure function for values, terms, configurations, and memory store. For all other terms, the erasure function is simply applied homomorphically..

argument to *label* is a value (and not a term), we define a new reduction relation that applies the erasure function at every step and thus ensure that values are erased as soon as possible. We note that such aggressive erasure would not be correct for *toLabeled*, which also returns a labeled value, since *toLabeled* takes a monadic *LIO* action that may produce side-effects observable to the attacker. Third, we erase a whole configuration to $\langle \bullet, \bullet, \bullet \mid \bullet\rangle$ when the current label is not below l; this ensures that the attacker cannot observe anything about sensitive configurations. Fourth, we erase all reference more sensitive than the attacker observation label, even those created in public contexts. This ensures the attacker cannot observe anything about the sensitive parts of the memory store.

The addition of $\bullet$ and corresponding reduction rules completes our calculus and semantics definition. We now prove several general properties for this calculus, followed by two key properties needed for the noninterference theorem: simulation and determinacy of our monadic reduction relation and a new relation that erases sensitive terms.

Our first lemma states that values are in normal form, i.e., values do not reduce.

Lemma 1 (*Values do not reduce*)

- For any value v, there is no term t such that $v \rightsquigarrow t$.
- For any $l_{\text{cur}}, c_{\text{cur}}, m, v, n$, there is no program configuration k such that $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid v\rangle \xrightarrow{n} k$.

Proof
The first case follows by induction on the pure term reduction relation. The second case follows by induction on the structure of v. □

Though straightforward, this lemma is helpful when distinguishing terms that terminate since, as in most sequential IFC calculi, our noninterference guarantee is termination insensitive, i.e., it only holds for terminating terms. And, recall that our calculus allows non-terminating terms with **fix**.

The next proposition show that the erasure function is homomorphic over substitution and idempotent over terms, memories and configurations.

Proposition 1 (*Idempotence and distribution properties of the erasure function*)

1. Idempotent over terms: $\varepsilon_l(t) = \varepsilon_l(\varepsilon_l(t))$

2. Idempotent over memory✎: $\varepsilon_l(m) = \varepsilon_l(\varepsilon_l(m))$
3. Idempotent over configurations: $\varepsilon_l(k) = \varepsilon_l(\varepsilon_l(k))$
4. Homomorphic over substitution: $\varepsilon_l(\{\,t_1\,/\,x\,\}\;t_2) = \{\,\varepsilon_l(t_1)\,/\,x\,\}\;\varepsilon_l(t_2)$

Intuitively, the first three properties respectively state that multiple application of the erasure function does not affect the term, memory, or configuration once it has been erased. In other words, the erasure function should completely erase sensitive data encoded in a term.

The erasure function additionally distributes over the pure reduction relation.

Proposition 2 (*Erasure function distributes over the pure-term reduction relation*)
For any label l, if $t \rightsquigarrow t'$ then $\varepsilon_l(t) \rightsquigarrow \varepsilon_l(t')$.

Proof
Straightforward induction on t, using Lemma 1, and Proposition 1. □

In other words, taking a step in the pure reduction and erasing the end term is the same as first erasing the term and taking a step. Intuitively this is stating that sensitive data does not affect the reduction of a pure term.

We now extend this intuition to simulation with a new reduction relation under which sensitive terms and configurations are erased. This new monadic-term reduction relation with erasure is defined as follows:

Definition 1 (*Reduction of pure and monadic terms with erasure*)

$$\frac{k \xrightarrow{n} k'}{k \xrightarrow{n}_l \varepsilon_l(k')}$$

Configurations under this relation 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. In this manner, the relation guarantees that confidential data, i.e., data above level l, is erased as soon as it is created.

To illustrate the need for this relation, consider two labels l_1 and l_2, such that $l_1 \sqsubseteq l_2$, and the following program $p = \langle l_1, l_2, \emptyset \mid (\lambda l.\mathit{label}\,l\,42)\;l_2\rangle$. Assuming an attacker at observation level l_1, program p contains the secret 42, which is placed inside a *label* expression when β-reducing. Observe that $\varepsilon_{l_1}(p)$ is not enough to capture what an attacker should see, since $\varepsilon_{l_1}(p) = \langle l_1, l_2, \emptyset \mid (\lambda l.\mathit{label}\,l\,42)\;l_2\rangle$, i.e., it still contains the secret! However, observe that $p \xrightarrow{n}_{l_1} \langle l_1, l_2, \emptyset \mid \mathit{label}\,l_2\,\bullet\rangle$ erases the secret (42) as soon as it is β-reduced—capturing the attacker observational power at every reduction step of the program.

$$\begin{tikzcd}
k \arrow[r, "n"] \arrow[d, dashed, "\varepsilon_l"] & k' \arrow[d, dashed, "\varepsilon_l"] \\
\varepsilon_l(k) \arrow[r, "n"', "l" below] & \varepsilon_l(k')
\end{tikzcd}$$

Fig. 20. Simulation between $\xrightarrow{n}$ and $\xrightarrow{n}_l$.

Fig. 20 highlights the intuition behind our simulation result: erasing all sensitive data, i.e., data whose label is not below l, and then taking a step in $\xrightarrow{n}_l$ is the same as taking a step in $\xrightarrow{n}$ and then erasing all the secret values in the resulting configuration. Observe that if configuration k leaks data labeled above l (such that it is observable at l), then erasing all sensitive data and taking a step in $\xrightarrow{n}_l$ might not be the same as taking steps in $\xrightarrow{n}$ and then erasing all the secret values in the resulting configuration—the data might have already been leaked. We remark that, while this simulation result and several statements below involve configurations that are initially erased, we rely on the more general reduction relation for determinacy and prove the more general statement where appropriate.

First, we show that the current label after taking a step is always at least as restricting as the current label before taking the step.

Proposition 3 (*Monotonicity of the current label*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ then $l_{\text{cur}} \sqsubseteq l'_{\text{cur}}$.

Proof
Straightforward induction on t, using the lattice-properties of labels (namely, reflexivity of $\sqsubseteq$ and definition of $\sqcup$). □

This proposition not only reduces the number of cases we need to consider, but also reinforces our intuition that none of the LIO terms can lower the current label once sensitive data is incorporated in the context (and thereby allow for such data to be leaked). We note that since *toLabeled* is defined using big-step semantics it does not actually restore the current label of the context; rather it executes a term in a separate context in a single step.

We now prove simulation of the monadic-term reduction relation. The proof follows by induction on the number of executed *toLabeled* blocks, i.e., index n on the $\xrightarrow{n}$ relation. These cases are further broken down into several simpler cases, according to the observational level of the attacker and current labels (before and after taking a step). To simplify presentation, these supporting statements are given in Appendix A.

Lemma 2 (*Single-step simulation without toLabeled*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{0} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ then $\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle) \xrightarrow{0}_l \varepsilon_l(\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle)$.

Proof
Straightforward case analysis on $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \sqsubseteq l$. All cases follow directly from supporting Propositions 10, 11, and 12 given in Appendix A. □

This base-case simulation corresponds to the scenario where no *toLabeled* blocks are executed. The single-step simulation lemma for arbitrary terms follows by induction, using this lemma for the base case.

Lemma 3 (*Single-step simulation*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ then $\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle)$.

Proof
Straightforward case analysis on $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \sqsubseteq l$ using Lemma 2 for the base case. The cases follow directly from the supporting propositions—Propositions 11, 12, and 14—given in Appendix A. □

This lemma shows a simulation between a term taking a step in the normal reduction relation and that same term, with all sensitive information erased, taking a step in the reduction relation with erasure. This is highlighted by Fig. 20. Unfortunately, the statement is overly restricting—it imposes the number of *toLabeled* blocks to be the n. (Indeed, we are only able to prove this lemma because the reduction rule (HOLELIO) is defined for any index.)

A more general statement would allow for the number of *toLabeled* blocks to differ. In particular, when considering erasure the number of *toLabeled* blocks executed is at most n, since the erasure collapses all sensitive paths (an erased configuration reduces to itself) and thus the number *toLabeled* blocks executed in a sensitive context need not be counted. This statement is given below:

Corollary 1 (*Single-step collapsed simulation*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ then $\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle) \xrightarrow{n'}_l \varepsilon_l(\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle)$ for some $n' \leqslant n$.

Proof
Directly from Lemma 3 using n as a witness. □

We remark that while we directly use Lemma 3, this is not necessary. Indeed, one can prove a more precise bound by showing that n' corresponds to the number of *toLabeled* blocks executed in attacker-observable contexts, i.e., contexts that have a current label below the attacker observation level.

Having established the simulation between the standard reduction relation and the relation with erasure, we now solely need to show that the latter relation is deterministic to prove noninterference.

First, we show that the pure-term reduction relation is deterministic.

Proposition 4 (*Determinacy of pure-term reduction*)
If $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$ then $t' = t''$.

Proof
By induction on the pure-term reduction relation, using Lemma 1. □

Since several reduction rules for the monadic-term reduction relation are given in using big-step semantics, we show that the big-step relation, i.e., relation wherein the end-terms are values, is deterministic:

Proposition 5 (*Determinacy of big-step monadic-term reduction*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{b'}\ t'\rangle$ and $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n'}{}^{*} \langle l''_{\text{cur}}, c''_{\text{cur}}, m'' \mid \mathit{LIO}^{\mathsf{TCB}}_{b''}\ t''\rangle$, then $l'_{\text{cur}} = l''_{\text{cur}}$, $c'_{\text{cur}} = c''_{\text{cur}}$, $m' = m''$, $n = n'$, $t' = t''$, and $b' = b''$.

Proof
By induction on t. Most cases follow by inversion of the first multi-step monadic-term reduction hypothesis. The *LIO*, *return*, and *throwLIO* cases further require the inversion of the second hypothesis. □

This proposition is crucial to the noninterference theorem. Indeed, it can serve as a first sanity-check when extending the library with new primitives: adding *LIO* actions that are non-deterministic, such as *getTimeOfDay* would trivially break this statement. And, extending the system to consider a non-deterministic reduction relation is non-trivial. Indeed, it may require changing even the security condition (Zdancewic & Myers, 2003; Sabelfeld & Myers, 2003).

We now use these two propositions to show that the single-step monadic-term reduction relation is deterministic.

$$\varsigma\ (\mathit{True}) = \mathbf{true} \qquad \varsigma\ (\mathit{False}) = \mathbf{true} \qquad \varsigma\ (()) = \mathbf{true} \qquad \varsigma\ (l_1) = \mathbf{true} \qquad \varsigma\ (\lambda x.t) = \varsigma\ (t)$$

$$\varsigma\ (\mathit{LIO}_b^{\mathsf{TCB}}\ t) = \mathbf{false} \qquad \varsigma\ (\mathit{Labeled}_b^{\mathsf{TCB}}\ l\ t) = \mathbf{false} \qquad \varsigma\ (\mathit{LIORef}^{\mathsf{TCB}}\ l\ t) = \mathbf{false}$$

$$\varsigma\ (\xi) = \mathbf{true} \qquad \varsigma\ (\bullet) = \mathbf{false} \qquad \varsigma\ (x) = \mathbf{true} \qquad \varsigma\ (m) = \bigwedge_{(a,\mathit{Labeled}_b^{\mathsf{TCB}}\ t) \in m} \varsigma\ (t)$$

$$\varsigma\ (\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle) = \varsigma\ (l_{\text{cur}}) \wedge \varsigma\ (c_{\text{cur}}) \wedge \varsigma\ (m) \wedge \varsigma\ (t)$$

Fig. 21. Safe function for values, memories, and configurations. The safe function for terms is defined homomorphically over the structure of the term.

Proposition 6 (*Determinacy of monadic-term reduction*)
If $k \xrightarrow{n} k'$ and $k \xrightarrow{n'} k''$ then $k' = k''$ and $n = n'$.

Proof
By induction on the monadic-term reduction relation, using Proposition 4 and Lemma 1. We use Proposition 5 for the (BIND), (BINDEX), (TOLABELED), (TOLABELEDEX), (CATCHLIO), and (CATCHLIOEX) cases. □

From this, the determinacy of the relation with erasure follows in a straightforward way:

Lemma 4 (*Determinacy of monadic-term reduction with erasure*)
For any label l, configurations k, k', and k'', and index numbers n and n', if $k \xrightarrow{n}_l k'$ and $k \xrightarrow{n'}_l k''$ then $k' = k''$ and $n = n'$.

Proof
By inversion of the hypotheses, using Proposition 6. □

Before stating the noninterference theorem, we first define a *safe* function ς to distinguish terms that are only composed of surface syntax. Fig. 21 gives the definition of this function for values, memories and configurations. For terms, we define ς as the conjunction of its application to all the term components. Since the definition of ς is straightforward, we only remark that our definition for memories is permissive in treating a non-empty memory m as safe when m only contains safe terms.

As in previous works on noninterference, we state noninterference as the preservation of l-equivalence, defined according to a syntactic equivalence relation $\approx_l$.[15] We define this l-equivalence relation as the equivalence kernel of the erasure function $\varepsilon_l(\cdot)$ for configurations. That is, $k \approx_l k'$ iff $\varepsilon_l(k) = \varepsilon_l(k')$. Note that this equivalence relation precisely captures the power of an attacker: to an attacker at observation level l, two terms that are l-equivalent cannot be distinguished.

Theorem 1 (*Noninterference*)
For any label l, index n_1, and two configuration k_1 and k_2, such that $\varsigma\ (k_1)$ and $\varsigma\ (k_2)$, there exists an index n_2, such that if $k_1 \approx_l k_2$, $k_1 \xrightarrow{n_1} k_1'$ and $k_2 \xrightarrow{n_2} k_2'$ then $k_1' \approx_l k_2'$.

Proof

[15] While considering syntactic l-equivalence is standard, a treatment of semantic l-equivalence would be an interesting research direction.

Applying Corollary 1 to the two hypotheses, we have: $\varepsilon_l(k_1) \xrightarrow{n_1'}_l \varepsilon_l(k_1')$, for $n_1' \leqslant n_1$ and $\varepsilon_l(k_2) \xrightarrow{n_2'}_l \varepsilon_l(k_2')$, for $n_2' \leqslant n_2$. From $k_1 \approx_l k_2$ and the definition of $\approx_l$ we have $\varepsilon_l(k_1) = \varepsilon_l(k_2)$. Then, by Lemma 4, we have $\varepsilon_l(k_1') = \varepsilon_l(k_2')$ and $n_1' = n_2'$. From the definition of l-equivalence, this is the same as $k_1' \approx_l k_2'$. Our Coq proof uses types to eliminate degenerate cases, but this is not fundamental to the proof and we thus elide this detail. □

The theorem states that if two configurations with possibly secret information, but indistinguishable to an attacker at level l, take a step, then the resulting configurations are also indistinguishable to the attacker. In other words, the attacker does not learn any sensitive information by observing configurations at lower sensitivity levels. Note, however, that the number of *toLabeled* actions executed in each step may differ according to data the attacker cannot observe—we assume that the attacker cannot observe the index counts.

This noninterference statement is stronger than that considered in the conference version of this paper (Stefan *et al.*, 2011b), which is stated in terms of a big-step. Specifically, this statement says that no information is leaked at any intermediate step, as opposed to solely stating that the result of two l-equivalent programs do not leak information. However, as in the conference version, this is a termination-insensitive result, i.e., we only make claims about the case where the configurations can each take a step and thus leaks due to non-termination are not captured. In (Stefan *et al.*, 2012a), we modify LIO to ensure that no information about the termination of sensitive subcomputation is visible to public contexts. For that, we force the execution of each *toLabeled* block to occur in a separate thread. The concurrent version of LIO satisfies a much stronger property—termination-sensitive noninterference—and is the library we use to implement both Hails and λChair.

6.2 Discretionary access control and isolation

In this section, we show that LIO programs cannot write or allocate entities below the current label or read, write or allocate entities above their current clearance.[16] Building on this, we then show how LIO can be used to isolate untrusted computations to ensure they can only access a particular part of memory and any faults are contained, i.e., faults in the untrusted code do not percolate into the outer context.

6.2.1 Discretionary access control

In the previous section, we showed that the current label after taking a step is always at least as restricting as the current label before taking the step. The dual holds for clearance; the current clearance after taking a step is always at most as restricting as the current clearance before taking the step.

Proposition 7 (*Monotonicity of the current clearance*)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$ then $c'_{\text{cur}} \sqsubseteq c_{\text{cur}}$.

[16] When considering privileges, in the style of the decentralized label model of Myers and Liskov (Myers & Liskov, 1997), these access restrictions give the code containing the privilege the discretion to access certain entities below the current label and above the current clearance.

Proof
By induction on t, using the lattice-properties of labels (namely, reflexivity of $\sqsubseteq$) and the fact that only (LOWERCLEARANCE) modifies the clearance (for which the statement holds trivially). □

This proposition states that the current clearance monotonically decreases within a context. In other words, the context can give up access to certain entities as it progresses, but not conversely. This statement is the clearance equivalent of Proposition 3, which states that once a computation reads confidential data, it cannot lower its current label to write to entities less sensitive.

Before delving into our access control guarantees, we first define two store modifiers:

$$l \preceq m = \{(a, Labeled_b^{\text{TCB}}\ l'\ t) : (a, Labeled_b^{\text{TCB}}\ l'\ t) \in m \text{ and } l \sqsubseteq l'\}$$

$$m \preceq l = \{(a, Labeled_b^{\text{TCB}}\ l'\ t) : (a, Labeled_b^{\text{TCB}}\ l'\ t) \in m \text{ and } l' \sqsubseteq l\}$$

$$l_1 \preceq m \preceq l_2 = l_1 \preceq m \cap m \preceq l_2$$

Symbol $l \preceq m$ denotes the subset of m containing all the references whose labels are above or equal to l. Similarly, $m \preceq l$ contains the references whose label is below or equal to l. Operator $l_1 \preceq m \preceq l_2$ encompasses the subset of m containing all the reference whose labels are between the labels l_1 and l_2. Finally, we introduce the complement of the described subsets as $\overline{l \preceq m}$, $\overline{m \preceq l}$, and $\overline{l_1 \preceq m \preceq l_2}$, respectively.

Lemma 5 (*No write-access below current label*✎)
Given a term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{\text{cur}})$, if the term reduces to a value according to $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \overset{n\ *}{\longrightarrow} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$, then $\overline{l_{\text{cur}} \preceq m} = \overline{l_{\text{cur}} \preceq m'}$.

Intuitively, this lemma states that the partitions, of the initial and final memory stores, that (may) contain references with labels below l_{cur} are identical, i.e., the computation could not have modified or created references below l_{cur}. Note, however, that the lemma does not state that term t cannot read from a reference below the current label. A corollary of this lemma states that any labeled values created by t are labeled above l_{cur}.

A similar, though slightly stronger, access control statement holds for clearance.

Lemma 6 (*No access above current clearance*✎)
Given term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{\text{cur}})$, if the term reduces to a value according to $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \overset{n\ *}{\longrightarrow} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$, then $\overline{m \preceq c_{\text{cur}}} = \overline{m' \preceq c_{\text{cur}}}$.

In other words, the partition of memory above the initial current clearance remains inaccessible throughout the program execution, i.e., the computation could not have modified or created references above c_{cur}. A corollary of this lemma states that any labeled values created by t are labeled below c_{cur}. As shown in Appendix A, computations also cannot read data above the clearance; this allows us to execute a term t with an alternative memory—one where references above the clearance are arbitrarily modified—without affecting its behavior.

From these two lemmas, we can further state that the current computation is restricted to modifying references whose labels are between the current label and clearance:

Proposition 8 (*Memory writes bounded by current label and clearance*✎)

Given term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{\mathrm{cur}})$, if the term reduces to a value according to $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t\rangle \overset{n}{\longrightarrow}{}^{*} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t\rangle$, then $\overline{l_{\mathrm{cur}} \preceq m \preceq c_{\mathrm{cur}}} = \overline{l_{\mathrm{cur}} \preceq m' \preceq c_{\mathrm{cur}}}$.

Proof
Directly from Lemma 5 and Lemma 6. □

6.2.2 Isolation

Using the above access control properties of LIO, we now show how terms can be executed in isolation. To this end, we first define an *isolate* function, similar to the *withClearance* of Section 3:

$$\begin{array}{l} \mathit{isolate} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \mathscr{L} \to \mathit{LIO}\ \mathscr{L}\ () \to \mathit{LIO}\ \mathscr{L}\ () \\ \mathit{isolate}\ l\ c\ t = \mathit{toLabeled}\ c\ (\mathit{lowerClearance}\ c \gg \mathit{raiseLabel}\ l \gg t) \gg \mathit{return}\ () \\ \quad \mathbf{where}\ \mathit{raiseLabel}\ l = \mathit{label}\ l\ () \gg\!= \mathit{unlabel} \end{array}$$

This function executes a term t in a context where the initial current label and clearance are l and c, respectively. While simple, this *isolation* function can be used to ensure that the untrusted term t can only modify a specific portion of memory and indeed, behave, as if it executes in a separate context:

Lemma 7 (*Single term isolation*✎)
If $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{isolate}\ l\ c\ t\rangle \overset{n+1}{\longrightarrow}{}^{*} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$, then $\overline{l \preceq m \preceq c} = \overline{l \preceq m' \preceq c}$, $m' = (l \preceq m' \preceq c) \cup (\overline{l \preceq m \preceq c})$, and $\langle l, c, m \mid t\rangle \overset{n}{\longrightarrow}{}^{*} \langle l', c', m' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$.

Here, the memory equations simply state that term t could only have modified the part of the memory store m that is between l and c. Regardless of whether t terminates by raising the current label, lowering the current clearance, and/or throwing an exception, the *isolate* function ensures that this "fault" is not propagated to the outer computation. Indeed, this can directly be used to address the poison pill attacks described in (Hriţcu *et al.*, 2013). Unfortunately, like the noninterference theorem, this lemma assumes that term t terminates.

By wrapping different terms with *isolate* and using disjoint labels for their corresponding current labels and clearances, we can guarantee that the terms will execute in isolation, on disjoint parts of the memory. Such a term isolation theorem, for two terms, is given below.

Theorem 2 (*Term isolation*✎)
Assume fresh$(\cdot)$ deterministically creates objects that are globally unique. Given safe terms t_1 and t_2, memory m, and labels l_1, c_1, l_2, and c_2, bounded by l_{cur} and c_{cur}, such that $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, and $c_2 \not\sqsubseteq c_1$, if $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid \mathit{isolate}\ l_1\ c_1\ t_1 \gg \mathit{isolate}\ l_2\ c_2\ t_2\rangle \overset{n}{\longrightarrow}{}^{*} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$ then $\langle l_1, c_1, m \mid t_1\rangle \overset{n_1}{\longrightarrow}{}^{*} \langle l'_1, c'_1, m_1 \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$, $\langle l_2, c_2, m \mid t_2\rangle \overset{n_2}{\longrightarrow}{}^{*} \langle l'_2, c'_2, m_2 \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$, $n = (n_1 + 1) + (n_2 + 1)$, and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$, $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$, $l_1 \preceq m' \preceq c_1 = l_1 \preceq m_1 \preceq c_1$, and $l_2 \preceq m' \preceq c_2 = l_2 \preceq m_2 \preceq c_2$.

Intuitively, the theorem states that the behavior of terms t_1 and t_2 (under the supplied context labels) is not affected by *isolate* function. Importantly, it also states that the two terms operate on disjoint parts of the memory—indeed the behavior of t_2 is the same as executing it with initial memory m, as opposed to m_1, the memory after term t_1 was executed. In the context of λChair, this isolation property is especially important since it allows us to ensure that requests running on behalf of different users run in isolation.

7 Related Work

Heintze and Riecke (Heintze & Riecke, 1998) consider security for lambda-calculus where lambda-terms are explicitly annotated with security labels, for a type-system that guarantees noninterference. One of the key ideas behind their work is to provide an operator that raises the security label of a term. Similarly, Zdancewic's PhD thesis (Zdancewic, 2002) introduces a security λ-calculus which raises the *pc* associated to a term when sensitive information gets obtained by reading references. Austin and Flanagan (Austin & Flanagan, 2009) design a λ-calculus which might temporary raise the *pc* when reducing function application. These features are similar to raising the current label when manipulating labeled values whose labels are above the current label. The notion of a floating current label dates back to the High-Water-Mark security model (Landwehr, 1981) of the ADEPT-50 in the late 1960s, which was later adopted by Asbestos (Efstathopoulos *et al.*, 2005), HiStar (Zeldovich *et al.*, 2006), and Flume (Krohn *et al.*, 2007) IFC Operating Systems.

Abadi et al. (Abadi *et al.*, 1999) develop the dependency core calculus (DCC) based on a hierarchy of monads to guarantee noninterference. In their calculus, they define a monadic type that protects the confidentiality of pure values at different security levels. Our *LIO* and *Labeled* types serve a similar role. However, since LIO has the guarantee that code cannot create labeled values below the current label and or above the current clearance, the *Labeled* type is not a monad—we must inspect the current label and clearance before a new labeled value can be created (e.g., by applying a function to the protected value). Nevertheless, we can use *unlabel* and *toLabeled* in the *LIO* monad to achieve the dynamic equivalent functionality of DCC's (non-standard) typing rules for the bind operator. Tse and Zdancewic (Tse & Zdancewic, 2004) translate DCC to System F and show that noninterference can be stated using parametricity. Unfortunately, like DCC, they rely on a non-standard typing rule for bind—they provide several definitions for this operator and rely on GHC's `UndecidableInstance` extension (which lifts type conditions of (Sulzmann *et al.*, 2007)) to resolve the correct bind. Crary et al. (Crary *et al.*, 2005) present a monadic calculus for noninterference for programs with mutable state. While inspired by these works, we do not take a domain-specific approach to extend the Haskell type system or modify the Haskell runtime; rather, we take a dynamic, label-polymorphic, and library approach to IFC. Importantly, our implementation does not rely on any non-standard constructs—this reduces the task of understanding IFC enforcement to understanding the LIO API.

Harrison and Hook show how to monadically encode abstract operating systems called *separation kernels* (Harrison, 2005). The idea behind this work is to first partition a program into multiple processes, each associated with a separate domain (label), running in isolation. Inter-process communication is allowed through a kernel that mediates the message exchange according to a security policy (e.g., noninterference). To formally reason about separation kernels, the authors use a monad-layering approach, modeling state with the *State* monad, concurrency with the *Resumption* monad, etc. This approach is orthogonal to our approach; we use monads in a trivial fashion and primarily as a way to implement the calculus semantics as a library. In (Stefan *et al.*, 2012a), we describe the concurrent version of LIO, which unlike (Harrison, 2005), considers termination-sensitive noninterference.

The seminal work by Li and Zdancewic (Li & Zdancewic, 2006) presents an implementation of information-flow security for Haskell. Instead of modifying the language runtime, they take a library-based approach by encoding IFC-constrained computations using arrows (Hughes, 2000) (a generalization of monads). This work was extended by Tsai et al. (Tsai *et al.*, 2007) to consider concurrency and side-effecting computations. Russo et al. (Russo *et al.*, 2008) show an alternative library-based approach that eliminates the need for arrows; they, instead, describe a monadic library that encodes static IFC. This library relies on monadic types to track information-flow in pure and side-effecting computations. Morgenstern and Licata (Morgenstern & Licata, 2010) extend this idea to implement an authorization- and IFC-aware programming language in Agda. However, and as is the case with many static systems (Sabelfeld & Russo, 2009), their library is less permissive. Nevertheless, this library is a closely related work. In particular, we note that the SecIO library (Russo *et al.*, 2008) has functions that serve the static counterpart of some of the core LIO functions (e.g., like *unlabel*, they provide a function that maps pure labeled values into monadic computations; like *toLabeled*, they provide a function that allows safely writing to public entities after reading secret data).

Another closely related work is that of (Devriese & Piessens, 2011); this work uses monad transformers and parametrized monads (Atkey, 2009) to enforce noninterference, both dynamically and statically. Different from our work, they focus on modularity (separating IFC enforcement from underlying user API), using typeclass-level tricks. Unfortunately, like the work on separation kernels, this requires programmers to first partition their code to fit the new programming model, whereas the usage of *LIO* strives to be very close to Haskell's existing *IO* libraries.

Laminar (Roy *et al.*, 2009) is a closely related system that combines OS- and PL-techniques to jointly provide application and OS end-to-end guarantees. Although our work does not extend to the OS, Laminar's OS-confinement could be unified with LIO, much as they unify the mechanism with their Java language-level system. More interestingly, at the language level, Laminar enforces IFC within certain code regions named *security regions*, where labeled data can be accessed. Security regions have a (secrecy and integrity) label associated with them and are superficially similar to our *toLabeled* blocks.[17] Unlike in LIO, however, security regions cannot change their current label; if code wishes to read data more sensitive than the region's label, it must create another region with the supplied label. Moreover, if code within a region violates a security check (e.g., attempts to write to less sensitive file), the Laminar runtime raises an exception. Each security region has a required catch block, which is executed when such an exception is raised. (Though, code within a region can terminate the process by exiting.) Catch blocks run with the same label as the security region and provides developers with a way for recovering from monitor failures. Importantly, the runtime suppresses exceptions raised within the security region's catch block and any exceptions not explicitly caught. Similar to our approach,

[17] Laminar also associates a set of capabilities as a means for declassification and endorsement, much like LIO's privileges. However, we do not discuss them further since we do not address such topics in this work.

this decision is done to avoid an exception raised in a sensitive context to suppress less sensitive subsequent actions. Despite this similarity, there are several differences between LIO and Laminar. First, LIO provides a single, flexible mechanism for handling exceptions; we do not treat monitor exceptions differently from other exceptions—thus reducing the abstractions developers must understand. Second, LIO does not suppress exceptions. Our *toLabeled* block delays exceptions which may be suppressed, but do not have to be—we can inspect the result of a *toLabeled* result, whether it is a failure or not. A result of these two points is that Laminar's secure regions can be implemented in LIO using *toLabeled* and *withClearance*. More importantly, we remark that LIO code does not have to be wrapped in *toLabeled* blocks—this is unlike Laminar, where code that handles labeled data must always be wrapped by a secure region.

The secure treatment of exception-handling has been studied by the mainstream IFC compilers Jif (Myers & Liskov, 2000) and FlowCaml (Simonet, 2003). These compilers' type-systems enforce the following rule for exceptions: if an exception *might* be raised in a sensitive context, no public side effects must follow either in the subsequent code in a try block or in the catch handler. On the other hand, LIO enforces that *once exceptions are thrown* in a sensitive context, no subsequent public side effects can be executed either inside the *toLabeled* block where the exception is raised (if any) or in the catch handler. In (Askarov & Sabelfeld, 2009a), the authors provide a more permissive static exception-handling mechanism by introducing exceptions that *cannot be caught*. This idea could be easily incorporated in LIO and we state it as an interesting direction for future work.

Hedin and Sabelfeld present a dynamic information-flow monitor for a core JavaScript with exceptions (Hedin & Sabelfeld, 2012). In their calculus, they associate a security level with every exception. This is similar to our initial approach, described below, in associating the current label with exceptions thrown by *throwLIO*. Their semantics diverge from standard JavaScript in disallowing public exceptions from being thrown in secret contexts and, to address this permissiveness issue, they provide a non-standard construct that can be used to upgrade the label of an exception. Unfortunately, IFC violations (which may arise when an upgrade is not performed) are fatal.

Our initial treatment of exceptions was presented in the unpublished manuscript (Stefan *et al.*, 2012b). While the semantics are mostly the same as those presented in this paper, there are some subtle differences. In particular, in the original work, exceptions had an associated explicit label—the current label at the time of a *throwLIO*. And, at the time of a *catchLIO*, the current label was raised to the join of the exception label and current label. Unfortunately, these semantics are unnecessarily complex due to the implementation. Specifically, the *LIO* monad was implemented as a *State* monad with *IO* as the base monad and the current label and clearance as the monad state. Since the monad state may change according to the computation control flow, it was necessary that exceptions carry the additional state information to ensure that the current label is not arbitrarily lowered. By removing this implementation consideration, we were able to simplify the semantics to those presented in this paper and also simplify the implementation—the key insight is that the current label and clearance are global to the computation and thus the *State* monad needs to only contain a reference to these labels. Indeed, this simplification reduced the complexity of exception handling to the interaction of exceptions and *toLabeled*.

In parallel with our initial work on exceptions, Hriţcu et al. presented the Breeze IFC language (Hriţcu *et al.*, 2013). Breeze explored the design space of IFC and exceptions. Not only do they consider various calculi with exceptions, but, like our work, also address the issue of treating IFC monitor failures as recoverable failures. We refer the interested reader to the Breeze paper for a very comprehensive comparison of Breeze and LIO, and a detailed analysis of different design trade-offs that arise due to exceptions. Here, we only remark that, like Breeze, we delay the propagation of exceptions raised in *toLabeled* blocks (in Breeze, these are called brackets). Indeed, our semantics for exceptions are very similar to their calculus $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$. Both of these calculi differ from our original presentation (Stefan *et al.*, 2012b) in hiding exceptions raised in a *toLabeled* block where the current label is above the supplied upper bound, see rule (TOLABELEDFAIL).

Different from most language-based IFC systems, LIO relies on the notion of clearance to restrict information leakage due to covert channels. Bell and La Padula (Bell & La Padula, 1976) formalized clearance as a bound on the current label of particular users' processes. In the 1980s, clearance became a requirement for high-assurance secure systems purchased by the US Department of Defense (Department of Defense, 1985). HiStar (Zeldovich *et al.*, 2006) 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.

8 Summary

We presented LIO, an IFC system that explores a new design point in language-based information flow security. LIO takes a mostly coarse-grained labeling approach, inspired by both IFC OSes and IFC programming languages. In particular, LIO only associates a single, mutable, label—the *current label* with all the values in context (lexical scope) and dictates how information flows to/from the context. Compared to typical language-based IFC systems, where labels are explicitly associated with values, this design approach is amenable to a fast, library implementation. But, to allow programmers to handle differently labeled data, LIO provides an abstract data type, *Labeled*, that encapsulates a term and its explicit label. (In a similar way we provide mutable labeled references.) *Labeled* values serve the dual purpose of addressing label creep—the raising of the current label as increasingly sensitive data is incorporated into the context—by encapsulating the result of sensitive sub-computation, as executed by *toLabeled*. Unlike other language-based work, our IFC system also implements *clearance* as a means for restricting the kinds of data a computation can read/write to; LIO relies on this form of discretionary access control to address covert channels: code cannot leak data it cannot read. Finally, LIO provides exception handling constructs which serve the dual purpose of encoding monitor failures, from which untrusted code can recover. This addresses a long standing problem with dynamic IFC enforcement—that monitor failures leak information.

We proved several security theorems for LIO. First, we showed that LIO programs, which may perform complex side-effects (e.g., mutate variables and throw exceptions), satisfy noninterference, i.e., LIO programs satisfy data confidentiality and integrity. Second, we showed that clearance is a form discretionary access control. And, finally, we showed that LIO can be used to execute terms in isolation, operating on disjoint parts of memory.

We implemented LIO as a Haskell library, using Safe Haskell to ensure that untrusted code executes in the *LIO* monad, i.e., our IFC sub-language. To illustrate the expressiveness of LIO, we described the core of a conference review system, λChair, that uses IFC to enforce high-level security policies. In addition to λChair, we (and others) have used LIO to implement several other web applications, some of which are in production use. We found the library-based approach to be very effective, both in terms of deployment (at the time of this writing, the library has thousands of downloads) and design (the interface matured as a result of several iterations).

Acknowledgments We thank the JFP reviewers for their constructive feedback and many insightful comments. We thank Pablo Buiras, Stefan Heule, Cătălin Hriţcu, Amit Levy, Benjamin C. Pierce, Alley Stoughton, and Edward Z. Yang for many fruitful discussions on the design of LIO. We thank the Haskell Symposium reviewers for useful feedback on the conference version of this paper. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, by a gift from The Mozilla Corporation, and by the Swedish research agencies VR and the Barbro Oshers Pro Suecia Foundation. Deian Stefan was supported by the DoD through the NDSEG Fellowship Program.

References

Abadi, M., Banerjee, A., Heintze, N., & Riecke, J. (1999). A core calculus of dependency. *Symposium on principles of programming languages*. ACM.

Agat, J. (2000). Transforming out timing leaks. *Symposium on principles of programming languages*. ACM.

Askarov, A., & Sabelfeld, A. (2009a). Catch me if you can: Permissive yet secure error handling. *Programming languages and analysis for security*. ACM.

Askarov, A., & Sabelfeld, A. (2009b). Tight enforcement of information-release policies for dynamic languages. *Computer security foundations symposium*. IEEE Computer Society.

Askarov, A., Hunt, S., Sabelfeld, A., & Sands, D. (2008). Termination-insensitive noninterference leaks more than just a bit. *European symposium on research in computer security*. Springer-Verlag.

Atkey, R. (2009). Parameterised notions of computation. *Journal of functional programming*, **19**(3-4).

Austin, T. H., & Flanagan, C. (2009). Efficient purely-dynamic information flow analysis. *Workshop on programming languages and analysis for security*. ACM.

Austin, T. H., & Flanagan, C. (2010). Permissive dynamic information flow analysis. *Workshop on programming languages and analysis for security*. ACM.

Bell, D. E., & La Padula, L. (1976). *Secure computer system: Unified exposition and multics interpretation*. Tech. rept. MTR-2997, Rev. 1. MITRE Corp.

Biba, K. J. 1977 (April). *Integrity considerations for secure computer systems*. Tech. rept. ESD-TR-76-372. MITRE Corp.

Buiras, P., Stefan, D., & Russo, A. (2014). On flow-sensitive floating-label systems. *Computer security foundations symposium*. IEEE Computer Society.

Crary, K., Kliger, A., & Pfenning, F. (2005). A monadic analysis of information flow security with mutable state. *Journal of functional programming*, **15**(2).

Denning, D. E. (1976). A lattice model of secure information flow. *Communications of the ACM*, **19**(5).

Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, **20**(7).

Department of Defense. (1985). *Trusted computer system evaluation criteria (orange book)*. DoD 5200.28-STD edn. Department of Defense.

Devriese, Dominique, & Piessens, Frank. (2011). Information flow enforcement in monadic libraries. *Workshop on types in language design and implementation*. ACM.

Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., & Morris, R. (2005). Labels and event processes in the Asbestos operating system. *Symposium on operating systems principles*. ACM.

Friedman, D. P., & Wise, D. S. (1976). The impact of applicative programming on multiprocessing. *International conference on parallel processing*.

Giffin, D. B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J., & Russo, A. (2012). Hails: Protecting data privacy in untrusted web applications. *Symposium on operating systems design and implementation*. USENIX.

Goguen, J.A., & Meseguer, J. (1982). Security policies and security models. *Symposium on security and privacy*. IEEE Computer Society.

Harrison, W. L. (2005). Achieving information flow security through precise control of effects. *Computer security foundations workshop*. IEEE Computer Society.

Hedin, D., & Sabelfeld, A. (2012). Information-flow security for a core of JavaScript. *Computer security foundations symposium*. IEEE Computer Society.

Hedin, D., & Sands, D. (2006). Noninterference in the presence of non-opaque pointers. *Computer security foundations workshop*. IEEE Computer Society.

Heintze, N., & Riecke, J. G. (1998). The SLam calculus: programming with secrecy and integrity. *Symposium on principles of programming languages*. ACM.

Heule, S., Stefan, D., Yang, E. Z., Mitchell, J. C., & Russo, A. (2015). IFC inside: Retrofitting languages with dynamic information flow control. *Conference on principles of security and trust*. Springer.

Hriţcu, C., Greenberg, M., Karel, B., Pierce, B. C., & Morrisett, G. (2013). All your IFC exceptions are belong to us. *Symposium on security and privacy*. IEEE Computer Society.

Hughes, J. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1–3), 67–111.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., & Morris, R. (2007). Information flow control for standard OS abstractions. *Symposium on operating systems principles*. ACM.

Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, **16**(10).

Landwehr, C. E. (1981). Formal models for computer security. *Computing survels*, **13**(3).

Li, P., & Zdancewic, S. (2006). Encoding information flow in Haskell. *Computer security foundations workshop*. IEEE Computer Society.

Li, P., & Zdancewic, S. (2010). Arrows for secure information flow. *Theoretical computer science*, **411**(19).

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. *Symposium on principles of programming languages*. ACM.

Miller, M. S. (2006). *Robust composition: Towards a unified approach to access control and concurrency control*. Ph.D. thesis, Johns Hopkins University.

Morgenstern, J., & Licata, D. R. (2010). Security-typed programming within dependently typed programming. *International conference on functional programming*. ACM.

Myers, A. C., & Liskov, B. (1997). A decentralized model for information flow control. *Symposium on operating systems principles*. ACM.

Myers, A. C., & Liskov, B. (2000). Protecting privacy using the decentralized label model. *ACM transactions on computer systems*, **9**(4).

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., & Nystrom, N. (2001). *Jif: Java information flow*. Software release. Located at http://www.cs.cornell.edu/jif.

Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, **180**, 47.

Pottier, F., & Simonet, V. (2002). Information flow inference for ML. *Symposium on principles of programming languages*. ACM.

Rondon, P. M, Kawaguci, M., & Jhala, R. (2008). Liquid types. *ACM SIGPLAN notices*, **43**(6).

Roy, Indrajit, Porter, Donald E., Bond, Michael D., McKinley, Kathryn S., & Witchel, Emmett. (2009). Laminar: Practical Fine-grained Decentralized Information Flow Control. *Proc. of the 30th ACM SIGPLAN conference on programming language design and implementation*. PLDI '09. ACM.

Russo, A., & Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. *Computer security foundations symposium*. IEEE Computer Society.

Russo, A., Claessen, K., & Hughes, J. (2008). A library for light-weight information-flow security in Haskell. *Symposium on Haskell*. ACM SIGPLAN.

Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. *IEEE journal on selected areas in communications*, **21**(1).

Sabelfeld, A., & Russo, A. (2009). From dynamic to static and back: Riding the roller coaster of information-flow control research. *Conference on perspectives of system informatics*. Springer.

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, **63**(9).

Simonet, V. (2003). *The Flow Caml system*. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`.

Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2011a). Disjunction category labels. *Nordic conference on secure IT systems*. Springer.

Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2011b). Flexible dynamic information flow control in Haskell. *Symposium on Haskell*. ACM SIGPLAN.

Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., & Mazières, D. (2012a). Addressing covert termination and timing channels in concurrent information flow systems. *International conference on functional programming*. ACM SIGPLAN.

Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2012b). Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arxiv:1207.1457*.

Stoughton, A. (1981). Access Flow: A protection model which integrates access control and information flow. *Symposium on security and privacy*. IEEE Computer Society.

Sulzmann, M., Duck, G. J., Peyton Jones, S., & Stuckey, P. J. (2007). Understanding functional dependencies via constraint handling rules. *Journal of functional programming*, **17**(1).

Terei, D., Marlow, S., Jones, S. Peyton, & Mazières, D. (2012). Safe Haskell. *Symposium on Haskell*. ACM SIGPLAN.

Tsai, T., Russo, A., & Hughes, J. (2007). A library for secure multi-threaded information flow in Haskell. *Computer security foundations symposium*. IEEE Computer Society.

Tse, S., & Zdancewic, S. (2004). Translating dependency into parametricity. *Proc. of the ninth ACM sigplan international conference on functional programming*. ACM.

VanDeBogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., & Mazières, D. (2007). Labels and event processes in the Asbestos operating system. *ACM transactions on computer systems*, **25**(4).

Waye, Lucas, Buiras, Pablo, King, Dan, Chong, Stephen, & Russo, Alejandro. (2015). It's My Privilege: Controlling Downgrading in DC-Labels. *Pages 203–219 of: Security and trust management - 11th international workshop, STM 2015, Vienna, Austria, September 21-22, 2015, proceedings*.

Winskel, G. (1993). *The formal semantics of programming languages: An introduction*. MIT Press.

Zdancewic, S., & Myers, A. C. (2003). Observational determinism for concurrent program security. *Computer security foundations workshop*. IEEE Computer Society.

Zdancewic, S. A. (2002). *Programming languages for information security*. Ph.D. thesis, Cornell University.

Zdancewic, Steve, & Myers, Andrew C. (2001). Robust declassification. *Page 15 of: csfw*. IEEE.

Zeldovich, N., Boyd-Wickizer, S., Kohler, E., & Mazières, D. (2006). Making information flow explicit in HiStar. *Symposium on operating systems design and implementation*.

A Detailed proofs

In this section, we provide expand the proof details for the results in Section 6.

Proposition 1 (*Idempotence and distribution properties of the erasure function*)

1. Idempotent over terms: $\varepsilon_l(t) = \varepsilon_l(\varepsilon_l(t))$
2. Idempotent over memory✎: $\varepsilon_l(m) = \varepsilon_l(\varepsilon_l(m))$
3. Idempotent over configurations: $\varepsilon_l(k) = \varepsilon_l(\varepsilon_l(k))$
4. Homomorphic over substitution: $\varepsilon_l(\{\, t_1 \,/\, x \,\}\; t_2) = \{\, \varepsilon_l(t_1) \,/\, x \,\}\; \varepsilon_l(t_2)$

Proof

The first property follows by induction on term t; all cases follow trivially from the inversion of the induction hypothesis. The second and third properties follow from the definition of the erasure function for memories and configurations and first property. The fourth property follows by induction on t_2; most cases follow directly from the induction hypothesis and definition of substitution. □

Since a number statements rely on several inversion and distribution properties for the erasure function, we give these below.

Proposition 9 (*Inversion properties of the erasure function*)

1. Labeled values:
   - If $l_1 \not\sqsubseteq l$ then $\mathit{Labeled}_b^{\mathsf{TCB}}\; l_1 \bullet = \varepsilon_l(\mathit{Labeled}_b^{\mathsf{TCB}}\; l_1\; t)$ for any t.
   - If $l_1 \sqsubseteq l$ then $\mathit{Labeled}_b^{\mathsf{TCB}}\; l_1\; \varepsilon_l(t) = \varepsilon_l(\mathit{Labeled}_b^{\mathsf{TCB}}\; l_1\; t)$.
2. Monadic values: $\mathit{LIO}_b^{\mathsf{TCB}}\; \varepsilon_l(t) = \varepsilon_l(\mathit{LIO}_b^{\mathsf{TCB}}\; t)$.
3. Configurations:
   - If $l_{\mathrm{cur}} \not\sqsubseteq l$ then $\langle \bullet, \bullet, \bullet \mid \bullet \rangle = \varepsilon_l(\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle)$ for any c_{cur}, m and t.
   - If $l_{\mathrm{cur}} \sqsubseteq l$ then $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, \varepsilon_l(m) \mid \varepsilon_l(t) \rangle = \varepsilon_l(\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle)$.

Proof

All properties follow directly from the definition of the erasure function. □

This proposition states that, in certain cases, we can invert the application of the erasure function to labeled values, *LIO* values, and configurations.[18]

Simulation

Our simulation lemma follows by induction on the number of executed *toLabeled* blocks. The two lemma, Lemma 2 and 3, rely on several supporting propositions. We give these below.

Our first base-case simulation proposition considers the case when both the starting and end configuration labels can flow to the attacker observation level. In other words, the current term t does not raise the current label (e.g., with *unlabel*) nor does it execute any *toLabeled* blocks.

Proposition 10
For any label l, such that $l_{\mathrm{cur}} \sqsubseteq l$ and $l'_{\mathrm{cur}} \sqsubseteq l$, if $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle \xrightarrow{0} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t' \rangle$ then $\varepsilon_l(\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle) \xrightarrow{0} \varepsilon_l(\langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t' \rangle)$

Proof
By induction on t. Most cases follow directly from inversion of the first $\xrightarrow{0}$ reduction hypothesis or Lemma 1. The $\gg\!=$ and *catchLIO* cases follow from the definition of the single- and multi-step relations, using Propositions 9 and a supporting proposition (not given here) whose statement is the multi-step version of this proposition. The terms for which there is a context reduction rule (e.g., *label*, *unlabel*, etc.), we further rely on Proposition 2. □

The next proposition considers the case when initial configuration cannot be observed by the attacker, i.e., the initial current label does not flow to the attacker label.

Proposition 11
For any label l, such that $l_{\mathrm{cur}} \not\sqsubseteq l$, if $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle \xrightarrow{n} \langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t' \rangle$ then it is also the case that $\varepsilon_l(\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t \rangle) \xrightarrow{n} \varepsilon_l(\langle l'_{\mathrm{cur}}, c'_{\mathrm{cur}}, m' \mid t' \rangle)$

Proof
We break the proof into two cases:

- Case $l'_{\mathrm{cur}} \sqsubseteq l$: follows from Proposition 3.
- Case $l'_{\mathrm{cur}} \not\sqsubseteq l$: follows trivially from the single-step reduction rule of an erased configuration, (HOLE).

□

Here, the simplicity of the proof allows us to consider the case where the number of executed *toLabeled* blocks is any natural n.

The more interesting case—when the current label is raised by the current term t—is given below. As shown below, only *unlabel* actually raises the current label, hence, we can directly consider the simulation for an arbitrary number n of executed *toLabeled* blocks. However, we must consider the case when *unlabel* is executed as part of a bigger action (e.g., in $\gg\!=$ or *catchLIO*).

[18] We note that, while we can prove inversion for all terms (and cases), we only need properties 1 and 2 to prove the more interesting simulation property.

Proposition 12
For any label l, such that $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \not\sqsubseteq l$, if $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t' \rangle$ then $\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t' \rangle)$

Proof
By induction on t. Most cases follow directly by inversion of the $\xrightarrow{n}$ reduction rule and $l_{\text{cur}} \sqsubseteq l$ hypothesis. The remaining cases are:

- Case *unlabel* t_1: Breaks down into the three reduction rules for *unlabel*:
  - — Case (UNLABELCTX): Follows directly from the definition of the $\xrightarrow{n}_l$ reduction rule and rule (UNLABELCTX), using Propositions 1 and 2.
  - — Case (UNLABEL): Both sub-cases (where the label of the value being unlabel can and cannot flow to l) follow directly from the definition of the $\xrightarrow{n}_l$ reduction rule and rule (UNLABEL), using Propositions 1 and 9.
  - — Case (UNLABELEX): Same as the (UNLABEL) case, but using the definition of (UNLABELEX) instead.
- Case $t_1 \gg\!= t_2$: Straight forward induction on t_1, using Propositions 1 and 9.
- Case *catchLIO* t_1 t_2: Straight forward induction on t_1.

□

These supporting statements are used to prove the base-case simulation, Lemma 2, where no *toLabeled* blocks are executed. However, all but one of the above supporting propositions consider the more general case, where any number of *toLabeled* blocks are executed. We need to extend Proposition 10 to arbitrary terms to prove the inductive case.

To do this, however, we must first show simulation for the big-step reduction relation holds (since *toLabeled* is defined in terms of a big-step), if the starting and end current labels can flow to the attacker label.

Proposition 13
For any label l, such that $l_{\text{cur}} \sqsubseteq l$, if $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle \overset{n\ *}{\longrightarrow} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid LIO_b^{\text{TCB}}\ t' \rangle$ then $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \varepsilon_l(t) \rangle \overset{n\ *}{\longrightarrow} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid LIO_b^{\text{TCB}}\ \varepsilon_l(t') \rangle$

Proof
By induction on t, most cases follow by inversion of the first $\overset{n\ *}{\longrightarrow}$ reduction hypothesis and the resulting $\xrightarrow{n}$ hypothesis. This leaves us with the terms that reduce to *LIO* values: *LIO*, *return*, and *throwLIO*. The first follows by inversion and Lemma 1. The latter two follow directly from the definition of the $\xrightarrow{n}$ and $\overset{n\ *}{\longrightarrow}$ reduction relations. □

Using this proposition, the general version of Proposition 10 follows:

Proposition 14
For any label l, such that $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \sqsubseteq l$, if $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t' \rangle$ then $\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t' \rangle)$

Proof
We break this into two cases:

- Case $n = 0$: Trivially from Lemma 2.

- Case $n = n' + 1$: By induction on t. Most cases follow trivially by inversio The remaining cases are:
  - Case $t_1 \gg\!= t_2$: By inversion we break this down into the two sub-cases corresponding to the reduction rules (BIND) and (BINDEX). Both cases follow directly from the definition of the $\xrightarrow{n}_l$ reduction rule, using Propositions 1, 9, and 13.
  - Case *toLabeled* t_1 t_2: By inversion we break this down into the three sub-cases corresponding to the reduction rules:
    - Case (TOLABELEDCTX): Trivially by inversion.
    - Case (TOLABELED): Both sub-cases (where the label of the result can and cannot flow to l) follow directly from definition of the $\xrightarrow{n}_l$ reduction rule and rule (TOLABELED), using Propositions 1, 9, and 13.
    - Case (TOLABELEDEX): Like the (TOLABELED) case, but using the definition of (TOLABELEDEX) instead.
  - Case *catchLIO* t_1 t_2: By inversion we have two cases corresponding to (CATCHLIO) and (CATCHLIOEX), both of which follow in the same way as the $\gg\!=$ case.

□

Directly, the single-step simulation lemma, Lemma 3, for arbitrary terms follows.

Discretionary access control and isolation

First, we give the proof for Lemma 5, which states that the current computation cannot write to references below the current label:

Lemma 5 (*No write-access below current label*✎)
Given a term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{\text{cur}})$, if the term reduces to a value according to $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\ *} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$, then $\overline{l_{\text{cur}} \preceq m} = \overline{l_{\text{cur}} \preceq m'}$.

Proof
Observe that t can only modify m by creating a new reference or writing to an existing reference with (NEWLIOREF) and (WRITELIOREF), respectively. Both of these rules require the label of the (potentially new) reference to be above l_{cur}. Hence we know that the memory below the current label will remain unchanged if t takes a single step. Using Proposition 3 we can directly extend this to an arbitrary number of steps. □

The somewhat dual statement, Lemma 6 states that the current computation cannot read or write to references above the current clearance (or create labeled values labeled as such):

Lemma 6 (*No access above current clearance*✎)
Given term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{\text{cur}})$, if the term reduces to a value according to $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\ *} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t'\rangle$, then $\overline{m \preceq c_{\text{cur}}} = \overline{m' \preceq c_{\text{cur}}}$.

Proof
Observe that t can only modify m by creating a new reference or writing to an existing reference with (NEWLIOREF) and (WRITELIOREF), respectively. Both of these rules require the label of the (potentially new) reference to be below c_{cur}. Hence we know that the memory above the current clearance will remain unchanged if t takes a single step. Using Proposition 3 we can directly extend this to an arbitrary number of steps. □

Indeed, since no memory above the current clearance can be accessed we can simply replace that part of the memory with arbitrary references:

Proposition 15 (*Reduction is independent of memory above clerance*✎)
If $\langle l_{\text{cur}}, c_{\text{cur}}, m_1 \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m'_1 \mid t'\rangle$, $\varsigma\ (t)$, and $m_1 \preceq c_{\text{cur}} = m_2 \preceq c_{\text{cur}}$, then $\langle l_{\text{cur}}, c_{\text{cur}}, m_2 \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m'_2 \mid t'\rangle$ and $m'_1 \preceq c_{\text{cur}} = m'_2 \preceq c_{\text{cur}}$.

Proof
Follows in the same way as the proof for Lemma 6. □

Before delving into the term isolation proof we first give two supporting propositions. First, a straightforward property for bind:

Proposition 16 (*Term evaluation is obvlivious to memory above clearance*✎)
The reductions $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ and $\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l''_{\text{cur}}, c''_{\text{cur}}, m'' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ hold iff $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1 \gg t_2\rangle \xrightarrow{n}{}^{*} \langle l''_{\text{cur}}, c''_{\text{cur}}, m'' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ holds and $n = n_1 + n_2$.

Proof
Directly from definition of bind. □

Second, we give simple memory equivalence when store modifiers are used:

Proposition 17 (*Equivalence of memory subsets*✎)
For labels l_1, c_1, l_2, and c_2, such that $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, and $c_2 \not\sqsubseteq c_1$, if $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m' \preceq c_1}$ then $l_2 \preceq m \preceq c_2 = l_2 \preceq m' \preceq c_2$.

Proof
Since the labels are incomparable, it is easy to show that $(l_2 \preceq m \preceq c_2) \subset (\overline{l_1 \preceq m \preceq c_1})$ and $(l_2 \preceq m' \preceq c_2) \subset (\overline{l_1 \preceq m' \preceq c_1})$, from which the statement trivially holds. □

From these, the term isolation theorem follows in a mostly straightforward way.

Theorem 2 (*Term isolation*✎)
Assume fresh$(\cdot)$ deterministically creates objects that are globally unique. Given safe terms t_1 and t_2, memory m, and labels l_1, c_1, l_2, and c_2, bounded by l_{cur} and c_{cur}, such that $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, and $c_2 \not\sqsubseteq c_1$, if $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{isolate}\ l_1\ c_1\ t_1 \gg \mathit{isolate}\ l_2\ c_2\ t_2\rangle \xrightarrow{n}{}^{*} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$ then $\langle l_1, c_1, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l'_1, c'_1, m_1 \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$, $\langle l_2, c_2, m \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l'_2, c'_2, m_2 \mid \mathit{LIO}^{\mathsf{TCB}}\ ()\rangle$, $n = (n_1 + 1) + (n_2 + 1)$, and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$, $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$, $l_1 \preceq m' \preceq c_1 = l_1 \preceq m_1 \preceq c_1$, and $l_2 \preceq m' \preceq c_2 = l_2 \preceq m_2 \preceq c_2$.

Proof
From Proposition 16, we have $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{isolate}\ l_1\ c_1\ t_1\rangle \xrightarrow{n_1+1}{}^{*} \langle l_{\text{cur}}, c_{\text{cur}}, m_1 \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ and $\langle l_{\text{cur}}, c_{\text{cur}}, m_1 \mid \mathit{isolate}\ l_2\ c_2\ t_2\rangle \xrightarrow{n_2+1}{}^{*} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$.
Applying Lemma 7 to the first reduction we have $m_1 = (l_1 \preceq m_1 \preceq c_1) \cup (\overline{l_1 \preceq m \preceq c_1})$, $\langle l_1, c_1, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l'_1, c'_1, m_1 \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$, and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$.
From Proposition 17 and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$ we have $l_2 \preceq m \preceq c_2 = l_2 \preceq m_1 \preceq c_2$.
Applying Lemma 7 to the second reduction we have $m' = (l_2 \preceq m' \preceq c_2) \cup (\overline{l_2 \preceq m_1 \preceq c_2})$, $\langle l_2, c_2, m_1 \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l'_2, c'_2, m' \mid \mathit{LIO}^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$, and $\overline{l_2 \preceq m_1 \preceq c_2} = \overline{l_2 \preceq m' \preceq c_2}$.
From Proposition 17 and $\overline{l_2 \preceq m_1 \preceq c_2} = \overline{l_2 \preceq m' \preceq c_2}$ we have $l_1 \preceq m_1 \preceq c_1 = l_1 \preceq m' \preceq c_1$.

Further applying Proposition 15 we have $\langle l_2, c_2, m \mid t_2 \rangle \xrightarrow{n_2}{}^{*} \langle l'_2, c'_2, m_2 \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ () \rangle$ and $m \preceq c_2 = m_2 \preceq c_2$. From Proposition 8 we have $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$. □