2014 IEEE 27th Computer Security Foundations Symposium

# On Dynamic Flow-sensitive Floating-Label Systems

Pablo Buiras
*Chalmers*
*buiras@chalmers.se*

Deian Stefan
*Stanford*
*deian@cs.stanford.edu*

Alejandro Russo
*Chalmers*
*russo@chalmers.se*

***Abstract*—Flow-sensitive analysis for information-flow control (IFC) allows data structures to have mutable security labels, i.e., labels that can change over the course of the computation. This feature is often used to boost the permissiveness of the IFC monitor, by rejecting fewer programs, and to reduce the burden of explicit label annotations. However, when added naively, in a purely dynamic setting, mutable labels can expose a high bandwidth covert channel. In this work, we present an extension for LIO—a language-based floating-label system—that safely handles flow-sensitive references. The key insight to safely manipulating the label of a reference is to not only consider the label on the data stored in the reference, i.e., the reference label, but also the *label on the reference label* itself. Taking this into consideration, we provide an *upgrade* primitive that can be used to change the label of a reference in a safe manner. To eliminate the burden of determining when a reference should be upgraded, we additionally provide a mechanism for automatic upgrades. Our approach naturally extends to a concurrent setting, not previously considered by dynamic flow-sensitive systems. For both our sequential and concurrent calculi, we prove non-interference by embedding the flow-sensitive system into the flow-insensitive LIO calculus, a surprising result on its own.**

## I. Introduction

Modern software systems are composed of many complex components that handle sensitive data. In many cases (e.g., mobile and web—both client- and server-side—applications) these disparate components are provided by different authors, of varying trustworthiness. This combination of untrusted code and sensitive data increases the risk of data theft or corruption.

Information-flow control (IFC) is a promising approach to security that provides data confidentiality and integrity in the presence of untrusted code. In its simplest form, IFC tracks and controls the flow of information through a system according to a security policy, usually *non-interference* [14]. Non-interference states that public events should not depend on sensitive data and dually, trusted data should not be affected by untrusted events. Hence, a program, which may be composed of untrusted components, is guaranteed to preserve data confidentiality and integrity if it satisfies non-interference. Indeed, this appealing guarantee of IFC has recently led to significant research and development efforts to secure web applications (e.g., [8, 13, 17, 41]) and mobile platforms (e.g., [10, 20]), where untrusted code is the norm.

In particular, language-based systems that employ dynamic execution monitors to enforce IFC have become popular [15]. This is due, in part, to the permissiveness of dynamic techniques (when compared to static approaches [34]), and the ability to handle complex language features like dynamic code evaluation—a feature common to many scripting languages.

To ensure data confidentiality and integrity, these dynamic IFC systems associate *security labels* with data (e.g., by mapping variables to labels) and monitor where such data can flow [25, 38]. In this paper, we use the labels **H** and **L**, to respectively denote secret and public data, and ensure that information cannot flow form a secret entity into a public one, i.e., the labels are ordered such that $\mathbf{L} \sqsubseteq \mathbf{H}$ and $\mathbf{H} \not\sqsubseteq \mathbf{L}$. in general, the partial order $\sqsubseteq$ (label check) is used when governing the allowed flows. We remark that our main results use a generalized lattice that may also express integrity concerns [25, 38], we only use the two-point lattice for simplicity of exposition.

One of the facets of IFC analysis lies in how such labels, when associated with objects, are treated [19]. Specifically, some IFC systems (e.g., [7, 9, 18, 21, 36, 37, 44]) treat labels on objects as *immutable* and do not allow for changes over the lifetime of the program, i.e., labels of objects are *flow-insensitive*. In contrast, other systems (e.g., [2, 3, 43]) are *flow-sensitive*, i.e., they allow object labels to change, in certain conditions, according to the sensitivity of the data that is stored in the object. In general, these flow-sensitive systems are more permissive, i.e., they allow programs that flow-insensitive monitors would reject.

Consider, for instance, a web application that writes to a labeled log while servicing user requests. If the label of the log is **L**, a flow-insensitive IFC monitor would disallow writing any sensitive data (e.g., error messages containing user-supplied data) to the log, since this would constitute a leak. However, in a flow-sensitive system, the label of the log can change (to **H**), as to accommodate the kinds of data being written to the log. For many applications, allowing labels to change in such a way is very desirable—it alleviates the burden of having to, a priori, determine the precise labels of objects (e.g., the log).

Unfortunately, naively introducing flow-sensitive objects to a purely dynamic IFC system can turn label changes into a covert channel [31]. Consider the code fragment of Figure 1


DOI 10.1109/CSF.2014.13

where references $l$ and $h$ are respectively labeled **L** and **H**. By naively allowing arbitrary label changes—even if the new label is more sensitive—we can leak the contents of $h$ into $l$. In particular, suppose that the temporary variable $tmp$ is initially labeled **L**. If the value stored in $h$ is $True$, then in the first conditional we assign $True$ into $tmp$ and raise its label to **H**, reflecting the fact that the branch condition depends on sensitive data. Since the $tmp$ is $True$, the branch condition for the second conditional is $False$ and thus the value and label of $l$ are left intact, i.e., $True$ at **L**. However, if $h$ is $False$, then the value and label of $tmp$ remains untouched—the code does not perform the first assignment. Instead, the second assignment, setting $l$ to $False$, is performed; importantly, however, the label of $l$ remains **L**, since the label of the branch condition is also **L**. Note that in both cases the label of $l$ stays **L**, but the value of $l$ is the same as the secret $h$. In systems such as LIO and Breeze, which allow labels to be inspected, this attack can be further simplified by simply checking the label of $tmp$ after the first assignment—if the secret is true then the label will be **H**, otherwise it will be **L** (hence why the *label change* is considered a covert channel).

$$
\begin{array}{l}
l \;\; := \mathit{True} \\
\mathit{tmp} := \mathit{False} \\
\textbf{if } h \textbf{ then } \mathit{tmp} := \mathit{True} \\
\textbf{if } \neg\ \mathit{tmp} \textbf{ then } l := \mathit{False}
\end{array}
$$

Figure 1: Flow-sensitive attack

This attack is not new, and, to ensure that the covert channel is not introduced when adding flow-sensitive references in such a way, several solutions have already been proposed. These solutions fall into roughly three categories. First, the IFC monitor can incorporate static information to ensure that such leaks are disallowed [31]. Second, the IFC monitor can forbid certain label changes, depending on the context (e.g., the program counter (*pc*) label [33]). For instance, the *no-sensitive upgrades* policy disallows raising the label of a public reference in a sensitive context (e.g., when a branch condition is **H**) [2, 43]. And, third, the monitor can disallow branches that depend on certain variables, for which the label was mutated, as done by the *permissive upgrades* policy [3].

In this paper, we take a fresh perspective on flow-sensitivity in the context of coarse-grained floating-label systems, in particular, the LIO IFC system [36, 37]. LIO brings ideas from IFC Operating Systems—notably, HiStar [44], Asbestos [9], and Flume [21]—into a language-based setting. In particular, LIO takes an OS-like coarse-grained approach by associating a single "current" floating-label with a computation (and everything in scope), instead of heterogeneously labeling every variable, as typically done by language-based systems (e.g., [27, 35]). This floating-label is raised (e.g., from **L** to **H**) to accommodate reading sensitive data and thus serves as a form of "taint" reflecting the sensitive of data in context, i.e., LIO is flow-sensitive in the current label. (This can be seen as raising the *pc* in more traditional language-based systems.) In turn, the LIO monitor uses the "current" floating-label to restrict where the computation can write (e.g., once the current label is raised to **H** it can no longer write to references labeled **L**). However, and like other similar IFC systems, LIO is *flow-insensitive* in object labels.

This work extends the LIO IFC system, both the sequential and concurrent versions, to incorporate flow-sensitive references. A key insight of this work is to consider labels of references as being composed of two elements: the reference label describing the confidentiality of the stored value, and another label, called *the label on the label*, which describes the confidentiality of the reference label itself. Our monitor, then only forbids changing a label of a reference if *the label on the label is below the floating-label*. Inspired by [17], we add a primitive for *upgrading* labels, when permitted by our monitor. This boosts the permissiveness of LIO, and, for instance, allows programs, such at the logging web application described above, which would otherwise be rejected by the IFC monitor.

To reduce the programmer's burden of introducing upgrade annotations, our calculus provides a means for automatically upgrading references for which the computation is about to "lose" write access, i.e., before tainting the computation by raising the current label, we first upgrade all the references that are less sensitive than this label. While secure, this feature facilitates a form of *label creep*, wherein all flow-sensitive references might end up with labels that are "too high." To further address this, we propose a block-structured primitive which only upgrades the labels of declared flow-sensitive references, while disallowing access to undeclared ones.

By taking a fresh perspective on flow-sensitivity, we also show that the no-sensitive-upgrade policy and our upgrade operation can be encoded using existing flow-insensitive constructs—the key insight is to explicitly model labels on labels with *nested labeled references*. In the context of LIO, this encoding has the added benefit of allowing us to prove non-interference by simply invoking previous results. More interestingly, our sequential semantics for LIO with flow-sensitive references directly extend to the concurrent setting.

The contributions of this paper are as follows:

- We extend LIO to incorporate flow-sensitive objects, with a focus on references. This extension not only increases LIO's permissiveness, but also provides a means for safely combining flow-insensitive and flow-sensitive references.
- We present a uniform mechanism for treating flow-insensitive and flow-sensitive references in both sequential and concurrent settings. To the best of our knowledge, we are the first to analyze the challenges of purely dynamic monitors with flow-sensitive references in the presence of concurrency.
- A non-interference proof for the different calculi that

leverages the encoding of flow-sensitive references using flow-insensitive constructs.

We remark that while our development focuses on LIO, we believe that our results generalize to other sequential and concurrent floating-label systems (e.g., [9, 18, 21, 44]).

The rest of the paper is organized as follows. Section II provides an introduction to LIO and its formalization. Section III presents our flow-sensitivity extensions and enforcement mechanism. Section IV extends this approach to the concurrent setting. Section V presents the embedding of our enforcement using flow-insensitive constructs, from which our formal security guarantees follow. We discuss related work in Section VI and conclude in Section VII.

## II. Introduction to LIO

LIO is a language-level IFC system, implemented as a library in Haskell. The library provides a new *monad*, *LIO*, atop which programmers implement computations, which may, in turn, use the LIO API to perform side effects (e.g., mutate a reference or write to a file).

The *LIO* monad implements a purely dynamic execution monitor. To this end, *LIO* encapsulates the state necessary to enforce IFC for the computation under evaluation. Part of this state is the *current (floating) label*. Intuitively, the current label serves a role similar to the program counter (*pc*) of more-traditional IFC systems (e.g., [35]). More specifically, the current label is used to restrict the current computation from performing side-effects that may compromise the confidentiality or integrity of data (e.g., by restricting where the current computation may write).

To soundly reason about IFC, every piece of data *must* be labeled, including literals, terms, and labels themselves. However and in contrast to language-based systems (e.g., [18, 26, 35]) where every value is explicitly labeled, LIO takes a coarse-grained approach and uses the current label to protect all values in scope. As in IFC operating systems [9, 44], in LIO, the current label $l_{\text{cur}}$ is the label on all the non-explicitly labeled values in the context of a computation.

Further borrowing from the OS community, LIO raises the current label to protect newly read data. That is, the current label is raised to "float" above the labels of all the objects read by the current computation. 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 $l_{\text{cur}}$ 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 $l'_{\text{cur}} \sqsubseteq l_e$. Hence, for example, a public LIO computation can read secret data by first raising $l_{\text{cur}}$ from $\mathbf{L}$ to $\mathbf{H}$. Importantly, however, the new current label prevents the computation from subsequently writing to public entities.

$$
\begin{array}{lll}
\text{Values } v & ::= & \mathit{True} \mid \mathit{False} \mid () \mid \lambda x.t \mid \ell \mid \mathit{LIO}^{\text{TCB}}\ t \\
\text{Terms } t & ::= & v \mid x \mid t\ t \mid \mathbf{fix}\ t \mid \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t \\
 & & \mid t \otimes t \mid \mathbf{return}\ t \mid t \gg\!= t \mid \mathbf{getLabel} \\
\text{Types } \tau & ::= & \mathit{Bool} \mid () \mid \tau \to \tau \mid \ell \mid \mathit{LIO}\ \tau \\
\text{Ops}_\ell\ \otimes & ::= & \sqcup \mid \sqcap \mid \sqsubseteq
\end{array}
$$

Figure 2: Syntactic categories for base $\lambda_\ell^{\text{LIO}}$.

### A. $\lambda_\ell^{\text{LIO}}$: A coarse-grained IFC calculus

We give the precise semantics for LIO by extending the simply-typed, call-by-name $\lambda$-calculus; we call this extended IFC calculus $\lambda_\ell^{\text{LIO}}$. The formal syntax of the core $\lambda_\ell^{\text{LIO}}$ calculus, parametric in the label type $\ell$, is given in Fig. 2. Syntactic categories $v$, $t$, and $\tau$ represent values, terms, and types, respectively. Values include standard primitives (Booleans, unit, and $\lambda$-abstractions) and terminals corresponding to labels ($\ell$) and monadic values ($\mathit{LIO}^{\text{TCB}}\ t$).[1] We note that values of the form $\mathit{LIO}^{\text{TCB}}\ t$ denote computations subject to security checks. (In fact, security checks are only applied to such values.) Terms are composed of standard constructs (values, variables $x$, function application, the **fix** operator, and conditionals), terminals corresponding to label operations ($t \otimes t$, where $\sqcup$ is the join, $\sqcap$ is the meet, and $\sqsubseteq$ is the partial-order on labels), standard monadic operators ($\mathbf{return}\ t$ and $t \gg\!= t$), and **getLabel**, to be explained below. Terms annotated with $\cdot^{\text{TCB}}$ are not part of the surface syntax, i.e., such syntax nodes are not made available to programmers and are solely used internally in our semantic description. Types consist of Booleans, unit, function types, labels, and *LIO* computations; since the $\lambda_\ell^{\text{LIO}}$ type system is standard, we do not discuss it further.

We include monadic terms in our calculus since (in Haskell) monads dictate the evaluation order of a program and encapsulate all side-effects, including I/O [24, 40]; LIO leverages monads to precisely control what (side-effecting) operations the programmer is allowed to perform at any given time. In particular, an LIO program is simply a computation in the *LIO* monad, composed from simpler monadic terms using *return* and *bind*. Term $\mathbf{return}\ t$ produces a computation which simply returns the value denoted by $t$. Term $\gg\!=$, called *bind*, is used to sequence LIO computations. Specifically, term $t \gg\!= (\lambda x.t')$ takes the result produced by term $t$ and applies function $\lambda x.t'$ to it. (This operator allows computation $t'$ to depend on the value produced by $t$.) We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the term $t \gg\!= \lambda x.\mathbf{return}\ (x+1)$, which simply adds 1 to the value produced by the term $t$, can be written using **do**-notation as shown in Figure 3.

$$
\begin{array}{l}
\mathbf{do}\ x \leftarrow t \\
\quad \mathbf{return}\ (x+1)
\end{array}
$$

Figure 3: **do**-notation

[1] We restrict our formalization to computations implemented in the *LIO* monad and only consider Haskell features relevant to IFC, similar to the presentation of LIO in [36, 37].

$$E ::= E\ t \mid \mathbf{fix}\ E \mid \mathbf{if}\ E\ \mathbf{then}\ t\ \mathbf{else}\ t \mid E \otimes t \mid v \otimes E$$
$$\mathbf{E} ::= [\,] \mid E \mid \mathbf{E} \gg\!= t$$

GETLABEL
$$\frac{\Sigma = (l_{\text{cur}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\mathbf{getLabel}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ l_{\text{cur}}] \rangle}$$

Figure 4: Evaluation contexts and **getLabel** reduction rule.

A top-level $\lambda_\ell^{\text{LIO}}$ computation is a *configuration* of the form $\langle \Sigma | t \rangle$, where $t$ is the monadic term and $\Sigma$ is the state associated with the term. As in [36, 37], we take an imperative approach to modeling the LIO state as a separate component of the configuration (as opposed to it being part of the term). We partially specify the state of $\lambda_\ell^{\text{LIO}}$to at least contain the current label $l_{\text{cur}}$, i.e., $\Sigma = (l_{\text{cur}}, \ldots)$, where ... denotes other parts of the state not relevant at this point. Under this definition, a top-level well-typed $\lambda_\ell^{\text{LIO}}$ term has the form $\Delta, \Gamma \vdash t : LIO\ \tau$, where $\Delta$ is the store typing, and $\Gamma$ is the usual type environment.

We use evaluation contexts in the style of Felleisen and Hieb to specify the reduction rules for $\lambda_\ell^{\text{LIO}}$ [11]. Figure 4 defines the evaluation contexts for pure terms ($E$) and monadic ($\mathbf{E}$) terms for the base $\lambda_\ell^{\text{LIO}}$. The definitions are standard; we solely highlight that monadic terms are evaluated only at the outermost use of bind ($\mathbf{E} \gg\!= t$), as in Haskell. For the base $\lambda_\ell^{\text{LIO}}$, we also give the reduction rule for the monadic term **getLabel**, which simply retrieves the current label. As shown later, it is precisely this label that is used to restrict the reads/writes performed by the current computation. The rest of the reduction rules for the base calculus are straight forward and given Appendix A.

### B. Labeled values

Using $l_{\text{cur}}$ as the label on all terms in scope makes it trivial to deal with implicit flows. Branch conditions, which are simply values of type $Bool$, are already implicitly labeled with $l_{\text{cur}}$. Consequently, all the subsequent writes cannot leak this bit—the current label restricts all the possible writes (even those in a branch). However, this coarse-grained labeling approach suffers from a severe restriction: a piece of code cannot, for example, write the public constant 42 to a public channel labeled $\mathbf{L}$ after observing secret data, even if this constant is independent from the secret—once secret data is read, the current label is raised to $\mathbf{H}$ thereby "over tainting" the public data in scope.

To address this limitation, LIO provides *Labeled* values. A *Labeled* value is a term that is explicitly protected by a label other than the current label. Figure 5 shows the extension of the base $\lambda_\ell^{\text{LIO}}$ with *Labeled* values.

The **label** terminal (**label** $l\ t$) is used to explicitly label a term. As rule (LABEL) shows, the function associates the supplied label $l$ with term $t$ by wrapping the term with the $Lb^{\text{TCB}}$ constructor. Importantly, it first asserts that the new

$$\begin{aligned} v &::= \cdots \mid Lb^{\text{TCB}}\ l\ t \\ t &::= \cdots \mid \mathbf{label}\ t\ t \mid \mathbf{unlabel}\ t \mid \mathbf{labelOf}\ t \mid \mathbf{upgrade}\ t\ t \\ \tau &::= \cdots \mid Labeled\ \tau \\ E &::= \cdots \mid \mathbf{label}\ E\ t \mid \mathbf{unlabel}\ E \mid \mathbf{labelOf}\ E \\ &\quad \mid \mathbf{upgrade}\ E\ t \mid \mathbf{upgrade}\ v\ E \end{aligned}$$

LABEL
$$\frac{\Sigma = (l_{\text{cur}}, \ldots) \qquad l_{\text{cur}} \sqsubseteq l}{\langle \Sigma | \mathbf{E}\ [\mathbf{label}\ l\ t] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ (Lb^{\text{TCB}}\ l\ t)] \rangle}$$

UNLABEL
$$\frac{\Sigma = (l_{\text{cur}}, \ldots) \qquad l'_{\text{cur}} = l_{\text{cur}} \sqcup l \qquad \Sigma' = (l'_{\text{cur}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\mathbf{unlabel}\ (Lb^{\text{TCB}}\ l\ t)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ t] \rangle}$$

LABELOF
$$\frac{}{E\ [\mathbf{labelOf}\ (Lb^{\text{TCB}}\ l\ t)] \longrightarrow E\ [l]}$$

UPGRADE
$$\frac{\Sigma = (l_{\text{cur}}, \ldots) \qquad l_u = l_{\text{cur}} \sqcup l \sqcup l'}{\langle \Sigma | \mathbf{E}\ [\mathbf{upgrade}\ (Lb^{\text{TCB}}\ l\ t)\ l'] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{label}\ l_u\ t] \rangle}$$

Figure 5: Extending $\lambda_\ell^{\text{LIO}}$ with labeled values.

label ($l$), which will used to protect $t$, is at least as restricting as the current label, i.e., $l_{\text{cur}} \sqsubseteq l$.

Dually, terminal **unlabel** unwraps explicitly labeled values. As defined in rule (UNLABEL), given a labeled value $Lb^{\text{TCB}}\ l\ t$, the function returns the wrapped term $t$. Since the returned term is no longer explicitly labeled by $l$, and is instead protected by the current label, $l_{\text{cur}}$ must be at least as restricting as $l$. To ensure this, the current label is raised from $l_{\text{cur}}$ to $l_{\text{cur}} \sqcup l$, capturing the fact that the remaining computation might depend on $t$. Moreover, the rule highlights the fact that the current label always "floats" above the labels of the values observed by the current computation.

The **labelOf** function provides a means for inspecting the label of a labeled value. As detailed by reduction rule (LABELOF), given a labeled value $Lb^{\text{TCB}}\ l\ t$, the function returns the label $l$ protecting term $t$. This allows code to check the label of a labeled value before deciding to unlabel it, and thereby raising the current label. It it worth noting that regardless of the current label in the configuration, the label of a labeled value can be inspected—hence labels are effectively "public."[2]

Finally, **upgrade** allows a piece of code to raise the label of a labeled value. Since labeled values are immutable, this function, in effect, produces another labeled value that is protected by a label that is more restrictive than the current one. Since LIO has the invariant that a computation can only create or write to entities above the current label, we

[2]Since labeled values can be nested, this only applies to the labels of top-level labeled values. Indeed, even these labels are not public—they are protected by the current label. However, since code can always observe objects labeled at the current label, this is akin to being public.

use $l_{\text{cur}}$ in addition to the supplied label ($l'$) when upgrading a labeled value.

Intuitively, we can try to define **upgrade** in terms of existing terminals by first **unlabel**ing the labeled value and then **label**ing the result with the join of the current label, existing label and new label. Unfortunately, using **unlabel** would cause the current label to be raised, since we now have (potentially) sensitive data in scope, thus preventing the computation from performing subsequent side effects on less sensitive locations. The raising of the current label to a point where the computation can no longer perform useful tasks is known as *label creep* [33]. Despite not compromising security, label creep can make LIO overly restricting when building practical applications.

To avoid label creep, LIO provides the **toLabeled** function which allows the current label to be *temporarily* raised during the execution of a given computation. We extend the terms and the pure evaluation context with $t ::= \cdots \mid \textbf{toLabeled}\ t\ t$ and $E ::= \cdots \mid \textbf{toLabeled}\ E\ t$, respectively, and give the precise semantics of **toLabeled** as follows:

TOLABELED

$$\frac{\Sigma = (l_{\text{cur}}, ...) \quad l_{\text{cur}} \sqsubseteq l \quad \langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | LIO^{\text{TCB}}\ t' \rangle \quad \Sigma' = (l'_{\text{cur}}, ...) \quad l'_{\text{cur}} \sqsubseteq l \quad \Sigma'' = \Sigma \ltimes \Sigma'}{\langle \Sigma | \mathbf{E}\ [\textbf{toLabeled}\ l\ t] \rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [\textbf{label}\ l\ t'] \rangle}$$

If the current label at the point of executing **toLabeled** $l\ t$ is $l_{\text{cur}}$, **toLabeled** evaluates $t$ to completion ($\langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | LIO^{\text{TCB}}\ t' \rangle$) and restores the current label to $l_{\text{cur}}$, i.e., **toLabeled** provides a separate context in which $t$ is evaluated. (Here, the state merge function $\ltimes$ is defined as: $\Sigma \ltimes \Sigma' \triangleq \Sigma$, in the next section we present an alternative definition.) We note that returning the result of evaluating $t$ directly (e.g., as $\langle \Sigma | \mathbf{E}\ [\textbf{toLabeled}\ l\ t] \rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [t'] \rangle$) would allow for trivial leaks; thus, **toLabeled** labels $t'$ with $l$ ($\langle \Sigma'' | \mathbf{E}\ [\textbf{label}\ l\ t'] \rangle$). This effectively states that the result of $t$ is protected by label $l$, as opposed to the current label ($l'_{\text{cur}}$) at the point $t$ completed. Importantly, this requires that the result not be more sensitive than $l$, i.e., $l'_{\text{cur}} \sqsubseteq l$.

### C. Labeled references

To complete the description of LIO, we extend our $\lambda^{\text{LIO}}_{\ell}$ calculus with mutable, flow-insensitive references. Conceptually, flow-insensitive references are simply mutable *Labeled* values. Like labeled values, the label of a reference is immutable and serves to protect the underlying term. (Recall that **upgrade** for labeled values create another immutable value with the new label.) The immutable label makes the semantics straightforward: writing a term to a reference amounts to ensuring that the reference label is as restrictive as the current label, i.e., the reference label must be above the current label; reading from a reference taints the current label with the reference label.

The syntactic extensions to our calculus are shown in Figure 6. We use meta-variable $s$ to distinguish flow-insensitive (FI) and flow-sensitive (FS) productions (the latter are described in Section III). We also extend configurations to contain a reference (memory) store $\mu_{\text{FI}}$: $\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, ...)$; $\mu_{\text{FI}}$ maps memory addresses—spanned over by metavariable $a$—to *Labeled* values.

$$\begin{aligned} v &::= \cdots \mid Ref^{\text{TCB}}_{\text{FI}}\ l\ a \\ t &::= \cdots \mid \textbf{newRef}_s\ t\ t \mid \textbf{writeRef}_s\ t\ t \mid \textbf{readRef}_s\ t \\ &\quad \mid \textbf{labelOf}_s\ t \\ \tau &::= \cdots \mid Ref_s\ \tau \\ E &::= \cdots \mid \textbf{newRef}_s\ E\ t \mid \textbf{writeRef}_s\ E\ t \mid \textbf{readRef}_s\ E \\ &\quad \mid \textbf{labelOf}_s\ E \end{aligned}$$

NEWREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, ...) \quad l_{\text{cur}} \sqsubseteq l \quad \mu'_{\text{FI}} = \mu_{\text{FI}}\ [a \mapsto Lb^{\text{TCB}}\ l\ t] \quad \Sigma' = (l_{\text{cur}}, \mu'_{\text{FI}}, ...)}{\langle \Sigma | \mathbf{E}\ [\textbf{newRef}_{\text{FI}}\ l\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ (Ref^{\text{TCB}}_{\text{FI}}\ l\ a)] \rangle}\ \text{fresh}(a)$$

READREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, ...)}{\langle \Sigma | \mathbf{E}\ [\textbf{readRef}_{\text{FI}}\ (Ref^{\text{TCB}}_{\text{FI}}\ l\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ \mu_{\text{FI}}\ (a)] \rangle}$$

WRITEREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, ...) \quad l_{\text{cur}} \sqsubseteq l \quad \mu'_{\text{FI}} = \mu_{\text{FI}}\ [a \mapsto Lb^{\text{TCB}}\ l\ t] \quad \Sigma' = (l_{\text{cur}}, \mu'_{\text{FI}}, ...)}{\langle \Sigma | \mathbf{E}\ [\textbf{writeRef}_{\text{FI}}\ (Ref^{\text{TCB}}_{\text{FI}}\ l\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ ()] \rangle}$$

LABELOF-FI

$$\frac{}{E\ [\textbf{labelOf}_{\text{FI}}\ (Ref^{\text{TCB}}_{\text{FI}}\ l\ a)] \longrightarrow E\ [l]}$$

Figure 6: Extending $\lambda^{\text{LIO}}_{\ell}$ with references.

When creating a flow-insensitive reference, $\textbf{newRef}_{\text{FI}}\ l\ t$ creates a labeled value that guards $t$ with label $l$ ($Lb^{\text{TCB}}\ l\ t$) and stores it in the memory store at a fresh address $a$ ($\mu_{\text{FI}}\ [a \mapsto Lb^{\text{TCB}}\ l\ t]$). Subsequently, the function returns a value of the form $Ref^{\text{TCB}}_{\text{FI}}\ l\ a$ which simply encapsulates the reference label and address where the term is stored. We remark that since any references created within a **toLabeled** block may outlive the **toLabeled** block computation, the merge function used in rule (TOLABELED) must also account for this, i.e., $(l_{\text{cur}}, \mu_{\text{FI}}, ...) \ltimes (l'_{\text{cur}}, \mu'_{\text{FI}}, ...) = (l_{\text{cur}}, \mu'_{\text{FI}}, ...)$.

Rule (READREF-FI) gives the semantics for reading a labeled reference; reading the term stored at address $a$ simply amounts to unlabeling the value $\mu\ (a)$ stored at the underlying address ($\textbf{unlabel}\ \mu_{\text{FI}}\ (a)$).

Terminal $\textbf{writeRef}_{\text{FI}}$ is used to update the memory store with a new labeled term $t$ for the reference at location $a$. Note that $\textbf{writeRef}_{\text{FI}}$ *leaves the label of the reference intact*, i.e., the label of a flow-insensitive reference is never changed, but, in turn, requires the current label to be below the reference label when performin the write ($l_{\text{cur}} \sqsubseteq l$). In addition to keeping the semantics simple, this has the additional benefit of allowing code to always inspect the label of a reference (with $\textbf{labelOf}_{\text{FI}}$).

```
leakRef :: Ref^TCB Bool → LIO Bool
leakRef href = do
  tmp ← newRef L ()
  toLabeled H $ do h ← readRef href
                   when h $ writeRef tmp ()
  return $ labelOf tmp ≡ H
```

Figure 7: Attack on LIO with naive flow-sensitive reference extension. We omit subscripts for clarity.

## III. Flow-sensitivity extensions

Unfortunately, the flow-insensitive references described in the previous section are somewhat inflexible. Consider, for example, an application that uses a reference as a log. Since the log may contain sensitive information, it is important that the reference be labeled. Equally important is to be able to read the log at any point in the program to, for instance, save it to a file. Although labeling the reference with the top element in the security lattice ($\top$) would always allow writes to the log, and **toLabeled** can be used to read the log and then write it to a file, this is unsatisfactory: it assumes the existence of a top element, which in some practical IFC systems, including HiStar [44] and Hails [13], does not exist. Moreover, it almost always over-approximates the sensitivity of the log. Hence, for example, a computation that never reads sensitive data, yet wishes to read the log content as to send error message to a user over the network (e.g., as done in a web application) cannot do so—LIO prevents the computation from reading the log, thereby tainting itself $\top$, and subsequently writing to the network.[3] It is clear that even for such a simple use case, having references with labels that vary according to the sensitivity of what is stored in the reference is useful.

However, naively implementing flow-sensitive references can effectively introduce label changes as a covert channel. Suppose that we allow for the label of a reference to be raised to the current label at the time of the **writeRef**. So, for example, if the label of our log reference is **L** and the computation has read sensitive data (such that the current label is **H**), subsequently writing to the log will raise the label of the reference to **H**. Unfortunately, while this may appear safe, as previously shown in [2, 3, 31], the approach is unsound.

The code fragment in Figure 7 defined a function, $leakRef$, that can be used to leak the contents of a reference by leveraging the newly introduced covert channel: the label of references. (In this and future examples we use function **when** to denote an **if** statement without the **else** branch and ($) as lightweight notation for function application, i.e., $f\,\$\,x$ is the same as $f\ (x)$.) To illustrate an attack, suppose that the current label is public (**L**) and $leakRef$ is called with a secret (**H**) reference ($href$). $leakRef$ first creates a public reference $tmp$ and, then, within the **toLabeled** block—which is used to ensure that the current label remains **L**—the label of this reference is changed to **H** if the secret stored in $href$ is $True$, and left intact (**L**) if the secret is $False$. Finally, by simply inspecting the label of the reference the value stored in $href$ is revealed.[4]

Fundamentally, the label protecting the *label* of an object, such as a reference or labeled value, is the current label $l_{\text{cur}}$ at the time of creation. Hence, to modify the label of the object within some context (e.g., **toLabeled** block) wherein the current label is $l'_{\text{cur}}$, *it must be the case that* $l'_{\text{cur}} \sqsubseteq l_{\text{cur}}$, i.e., we must be able to write data at sensitivity level $l'_{\text{cur}}$ into an entity—the label of the object—labeled $l_{\text{cur}}$. This restriction is especially important if $l_{\text{cur}} \sqsubset l'_{\text{cur}}$ and we can restore the current label from $l'_{\text{cur}}$ to $l_{\text{cur}}$, since a leak would then be observable within the program itself. In the case where the label of the object is immutable, as is the case for flow-insensitive references (and labeled values), this is not a concern: even if the current label is raised to $l'_{\text{cur}}$ and then restored to $l_{\text{cur}}$, we do not learn any information more sensitive than $l_{\text{cur}}$—the label of the label at the time of creation—by inspecting the label of the reference (or value): the label has not changed!

Thus, to extend LIO with flow-sensitive references, we must account for the label on the label of the reference at the time of creation, $l_{\text{cur}}$. (This label is, however, immutable.) In turn, when changing the label of the reference, we must ensure that no data from the context at the time of the change, whose label is $l'_{\text{cur}}$, is leaked into the label of the reference by ensuring that $l'_{\text{cur}} \sqsubseteq l_{\text{cur}}$, i.e., we can write data labeled $l'_{\text{cur}}$ into the label that is labeled $l_{\text{cur}}$.

Formally, we extend the $\lambda^{\text{LIO}}_{\ell}$ syntax and reduction rules as shown in Figure 8; we call this calculus $\lambda^{\text{LIO}}_{\ell,\text{FS}}$. When creating a flow-sensitive reference, $\textbf{newRef}_{\text{FS}}\ l\ t$ creates a labeled value that guards $t$ with label $l$ ($Lb^{\text{TCB}}\ l\ t$). However, since we wish to allow programmers to modify the label $l$ of the reference, we additionally store the label on $l$, i.e., $l_{\text{cur}}$, by simply labeling the already-guarded term ($\mu'_{\text{FS}} = \mu_{\text{FS}}\ [a \mapsto Lb^{\text{TCB}}\ l_{\text{cur}}\ (Lb^{\text{TCB}}\ l\ t)]$), as shown in rule (NEWREF-FS) . Primitive $\textbf{newRef}_{\text{FS}}$ returns a $Ref^{\text{TCB}}_{\text{FS}}\ a$ which simply encapsulates the fresh reference address where the doubly-labeled term is stored. Different from the constructor $Ref^{\text{TCB}}_{\text{FI}}$, the constructor $Ref^{\text{TCB}}_{\text{FS}}$ does not encapsulate the label of the reference. This is precisely because the label of a flow-sensitive reference is mutable and must be looked up in the store. As given by rule (LABELOF-FS), $\textbf{labelOf}_{\text{FS}}$ returns the label of the reference after raising the current label (with **unlabel**) to account for the fact that the label of the reference $l'$ is a value at sensitivity level $l$, i.e., we raise the current label to the join of the current label and the label on the label.

[3] Here, as in most IFC systems, we assume the network is public.

[4] The use of **labelOf** is not fundamental to this attack and in Appendix B we show an alternative attack that does not rely on such label inspection.

$$
\begin{array}{rcl}
v & ::= & \cdots \mid Ref_{\text{FS}}^{\text{TCB}}\ t \\
t & ::= & \cdots \mid \textbf{upgrade}_{\text{FS}}\ t\ t \mid \Uparrow \\
E & ::= & \cdots \mid \textbf{upgrade}_{\text{FS}}\ E\ t \mid \textbf{upgrade}_{\text{FS}}\ v\ E
\end{array}
$$

NEWREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad l_{\text{cur}} \sqsubseteq l \qquad \text{fresh}(a) \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\ [a \mapsto Lb^{\text{TCB}}\ l_{\text{cur}}\ (Lb^{\text{TCB}}\ l\ t)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\ [\textbf{newRef}_{\text{FS}}\ l\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle}$$

READREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t) \qquad l'' = l \sqcup l'}{\langle \Sigma | \mathbf{E}\ [\textbf{readRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ (Lb^{\text{TCB}}\ l''\ t)] \rangle}$$

WRITEREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}\ \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t') \qquad l_{\text{cur}} \sqsubseteq (l \sqcup l') \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\ [a \mapsto Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\ [\textbf{writeRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ ()] \rangle}$$

WRITEREF-FS-FAIL

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}\ \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t') \qquad l_{\text{cur}} \not\sqsubseteq (l \sqcup l')}{\langle \Sigma | \mathbf{E}\ [\textbf{writeRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{unlabel}\ (Lb^{\text{TCB}}\ l \Uparrow)] \rangle}$$

LABELOF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t)}{\langle \Sigma | \mathbf{E}\ [\textbf{labelOf}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ (Lb^{\text{TCB}}\ l\ l')] \rangle}$$

UPGRADEREF

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ v \qquad l_{\text{cur}} \sqsubseteq l \qquad \langle \Sigma | \textbf{upgrade}\ v\ l' \rangle \longrightarrow^{*} \langle \Sigma | LIO^{\text{TCB}}\ v' \rangle \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\ [a \mapsto Lb^{\text{TCB}}\ l\ v'] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\ [\textbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l'] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ ()] \rangle}$$

Figure 8: $\lambda^{\text{LIO}}_{\ell,\text{FS}}$: $\lambda^{\text{LIO}}_{\ell}$ with flow-sensitive references.

The rule for reading flow-sensitive references is standard. As given by (READREF-FS), **readRef**$_{\text{FS}}$ simply raises the current label to the join of the reference label and label on the reference label ($l \sqcup l'$) and returns the protected value. This reflects the fact that the computation is observing both data at level $l$ (the label on the reference) and $l'$ (the actual term).

The rule for writing flow-sensitive references deserves more attention. First, **writeRef**$_{\text{FS}}$ ensures that the current computation can write to the reference by checking that $l_{\text{cur}} \sqsubseteq (l \sqcup l')$. We impose this condition instead of the two conditions $l_{\text{cur}} \sqsubseteq l$ and $l_{\text{cur}} \sqsubseteq l'$—which respectively check that the current computation can modify both, the label of the reference, and the reference itself—since it is more permissive, yet still safe. When imposing the two conditions independently, certain programs, such as the one given in Figure 9, would fail. In this program, we first create a flow-sensitive reference labeled **H** when the current label is **L** (and thus the label on **H** is **L**). Then, we raise the label by reading from the reference. Finally, we attempt to write the to the reference. Under our semantics, this program behaves as expected; however, when imposing the two conditions independently, the write fails—the current label does not flow to the label on the label of the reference.

$$\begin{array}{l}
\textbf{do}\ r \leftarrow \textbf{newRef}_{\text{FS}}\ \mathbf{H}\ () \\
\quad \textbf{readRef}_{\text{FS}}\ r \\
\quad \textbf{writeRef}_{\text{FS}}\ r\ ()
\end{array}$$

Figure 9: Permissiveness test.

Another peculiarity with **writeRef**$_{\text{FS}}$ is the case when the current label does not flow to the join of the reference label, i.e., $l_{\text{cur}} \not\sqsubseteq l \sqcup l'$, and thus the write should not be allowed. If the semantics simply got stuck, the current label (at the point of the stuck term) would not reflect the fact that the success of applying such rule depends on the label $l'$, which is itself protected by $l$. Indeed, this might lead to information leaks and we thus we provide an explicit rule, (WRITEREF-FS-FAIL), for this failure case that first raises the current label (through **unlabel**) to $l$ and then diverges; in the rule, $\Uparrow$ represents a divergent term for which we do not provide a reduction rule.

Finally, we note that **writeRef**$_{\text{FS}}$ does not modify the label of the reference. This is, in part, because we wish to keep the difference between flow-insensitive and flow-sensitive references as small as possible. Instead, we provide **upgrade**$_{\text{FS}}$ precisely for this purpose; this primitive is used to raise the label of the reference. Rule (UPGRADEREF) is straight forward—it simply ensures that the current computation can modify the label of the reference by checking that the current label flows to the label on the label ($l_{\text{cur}} \sqsubseteq l$).

UPGRADESTORE

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}} = \{a_1 \mapsto v_1, ..., a_n \mapsto v_n\} \qquad t_i = \textbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a_i)\ l, i = 1, ..., n}{\langle \Sigma | \mathbf{E}\ [\textbf{upgradeStore}_{\text{FS}}\ l] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [t_1 \gg ... \gg t_n] \rangle}$$

UNLABEL-AU

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad l'_{\text{cur}} = l_{\text{cur}} \sqcup l \qquad \langle \Sigma | \textbf{upgradeStore}_{\text{FS}}\ l'_{\text{cur}} \rangle \longrightarrow^{*} \langle l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}} | LIO^{\text{TCB}}\ () \rangle \qquad \Sigma' = (l'_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ (Lb^{\text{TCB}}\ l\ t)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ t] \rangle}$$

Figure 10: $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$: Extending $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ with auto-upgrades.

### A. Automatic upgrades

We can use $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ to implement various applications that rely on flow-sensitive references, even those that rely on policies such as the popular no-sensitive upgrades [2]. (In Section VI, we describe the encoding of a policy that is similar to, but more permissive than, no-sensitive upgrades.) Using $\lambda^{\text{LIO}}_{\ell,\text{FS}}$, we can also safely implement our logging application using a flow-sensitive reference. Unfortunately, our system (and others like it) requires that we insert **upgrade**s

$$
\begin{array}{rcl}
v & ::= & \cdots \mid \overline{v, \ldots} \mid \epsilon_{\overline{\tau,\ldots}} \\
t & ::= & \cdots \mid \overline{t, \ldots} \mid \textbf{withRefs}_{\text{FS}}\ t\ t \\
\tau & ::= & \cdots \mid \overline{\tau, \ldots} \\
E & ::= & \cdots \mid \overline{E, t, \ldots} \mid \overline{v, E, t, \ldots} \mid \textbf{withRefs}_{\text{FS}}\ E\ t
\end{array}
$$

$$
\begin{array}{l}
addrs(\epsilon_{\overline{\tau,\ldots}}) \triangleq \emptyset \\
addrs(\overline{Ref_{\text{FS}}^{\text{TCB}}\ a_1, Ref_{\text{FS}}^{\text{TCB}}\ a_2, \ldots}) \triangleq \{a_1, a_2, \ldots\}
\end{array}
$$

WITHREFS-CTX

$$
\frac{\begin{array}{c}\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \\ \mu'_{\text{FS}} = \{a \mapsto \mu_{\text{FS}}(a) \mid a \in \text{dom}\ \mu_{\text{FS}} \cap (addrs(v))\} \\ \langle l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}} | \mathbf{E}\ [t]\rangle \longrightarrow \langle l'_{\text{cur}}, \mu'_{\text{FI}}, \mu''_{\text{FS}} | \mathbf{E}\ [t']\rangle \\ \Sigma'' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu''_{\text{FS}} \ltimes \mu_{\text{FS}}) \qquad v' = addrs^{-1}(\text{dom}\ \mu''_{\text{FS}})\end{array}}{\langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v\ t]\rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v'\ t']\rangle}
$$

WITHREFS-DONE

$$
\frac{}{\langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v\ v']\rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [v']\rangle}
$$

TYPE-WITHREF

$$
\frac{\begin{array}{c}\Delta' = \{a \mapsto \Delta(a) \mid a \in \text{dom}\ \Delta \cap (addrs(v))\} \\ \Delta', \Gamma \vdash v : \overline{Ref_{\text{FS}}\ \tau_1, \ldots} \qquad \Delta', \Gamma \vdash t : LIO\ \tau\end{array}}{\Delta, \Gamma \vdash \textbf{withRefs}_{\text{FS}}\ v\ t : LIO\ \tau}
$$

Figure 11: Extending $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ with **withRefs**$_{\text{FS}}$.

before we raise the current label so that it is possible to write references in a more-sensitive context, e.g., to modify a public reference (with label on the label **L**) after reading a secret. In the case of the logging example, we would need to upgrade the label before reading any sensitive data, if we later wish to write to the log.

Inspired by [17], we thus provide an extension to $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ that can be used to automatically upgrade references. This extension, called $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, is given in Figure 10. Intuitively, whenever the current label is about to be raised, we first upgrade all the references in the $\mu_{\text{FS}}$ store and then raise the current label. Rule (UPGRADESTORE) upgrades every reference in the flow-sensitive store $\mu_{\text{FS}}$ by executing $t_1 \gg t_2 \gg \cdots \gg t_n$, where $t_i = \textbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a_i)\ l$. Term $t \gg t'$ is similar to bind except that it discards the result produced by $t$. Since **unlabel** is the only function that raises the current label, we augment the (UNLABEL) rule with (UNLABEL-AU), given in Figure 10. This ensures that as the computation progresses it does not "lose" write access to its references. Returning to our logging example, with auto-upgrades, the reference used as the log never needs to be explicitly upgraded and can always be written to—an interface expected of a log.

Recall, however, that **toLabeled** is used to avoid label creep by allowing code to only temporarily raise the current label. Unfortunately, with auto-upgrades, when the current label gets raised within a **toLabeled** block, the upgrades of the flow-sensitive references remain even after the current label is restored. Thus, reading from any flow-sensitive reference after the **toLabeled** block will raise the current label to (at least) the current label at the end of the **toLabeled** block (since all references are upgraded every time that the current label gets raised). This can be used to carry out a *poison pill*-like attack [18], wherein the (usually untrusted) computation executing within the **toLabeled** block will render the outer computation useless by imposing label creep. (We note that this attack is possible in $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ without the auto-upgrade, but requires the attacker to manually insert all the upgrades.)

To address this issue, we extend $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ (and thus $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$) with **withRefs**$_{\text{FS}}$ $v$ $t$ which takes a bag (strict heterogeneous list) $v$ of references and a computation $t$, and executes $t$ in a configuration where the flow-sensitive reference store only contains the subset of references $v$. This extension and type rule (TYPE-WITHREF), which ensures that a term cannot access a reference outside its store, are shown in Figure 11. A bag is either empty $\epsilon_{\overline{\tau,\ldots}}$, or it may contain a set of references of (potentially) distinct types $\overline{v, \ldots}$. Rules (WITHREFS-CTX) and (WITHREFS-DONE) precisely define the semantics of this new primitive, where meta function $addrs(\cdot)$ converts a bag of references to a set of their corresponding addresses, $addrs^{-1}(\cdot)$ performs the inverse conversion, and $\ltimes$ is used to merge the stores, giving preferences to the left-hand-side store, i.e., when there is a discrepancy on a stored value between both stores, it chooses the one appearing on the left-hand-side. We note that (WITHREFS-CTX) is triggered until the term under evaluation is reduced to a value, at which point (WITHREFS-DONE) is triggered, returning said value; we specify this big-step rule in terms of small-steps to facilitate the formalization of our concurrent calculus (see Section IV). Aside from the modeling of bags, the **withRefs**$_{\text{FS}}$ primitive is straightforward and mostly standard; indeed, the programming paradigm is similar to that already present in some mainstream languages (e.g., C++'s lambda closures require the programmer to specify the captured references). Lastly, we note that the poison pill attack can now be addressed by simply wrapping **toLabeled** with **withRefs**$_{\text{FS}}$, which prevents (untrusted) code within the **toLabeled** block from upgrading arbitrary references.

## IV. CONCURRENCY

In this section, we consider flow-sensitive references in the presence of concurrency (e.g., a web application in which request-handling threads share a common log). Concretely, we extend our sequential $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ calculi with threads and a new terminal, **forkLIO**, for dynamically creating new threads, as in the concurrent version of LIO [37]. Intuitively, this concurrent calculus $\lambda^{\parallel\text{-LIO}}_{\ell}$ simply defines a scheduler over sequential threads, such that taking a step in the concurrent calculus amounts to taking a step in a sequential thread and context switching to a different one. For brevity, we restrict our discussion in this section to the case where the underlying sequential calculus is $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, since this calculus extends $\lambda^{\text{LIO}}_{\ell,\text{FS}}$.

Figure 12 shows our extended concurrent calculus, $\lambda^{\parallel\text{-LIO}}_{\ell}$. A concurrent program configuration has the form

$t ::= \cdots \mid \textbf{forkLIO}\ t \mid \textbf{toLabeled}\ \ell\ t$

FORKLIO

$$\frac{}{\langle \Sigma | \mathbf{E}\ [\textbf{forkLIO}\ t]\rangle \xrightarrow{fork(t)} \langle \Sigma | \mathbf{E}\ [\textbf{return}\ ()]\rangle}$$

WITHREFS-OPT

$$\frac{v = addrs^{-1}((addrs(v_1)) \cap (addrs(v_2))) \quad \langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v\ t]\rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [t']\rangle}{\langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v_1\ (\textbf{withRefs}_{\text{FS}}\ v_2\ t)]\rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [t']\rangle}$$

T-STEP

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \quad \langle \Sigma | \textbf{withRefs}_{\text{FS}}\ v\ t\rangle \longrightarrow \langle \Sigma' | t'\rangle \quad \Sigma' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu'_{\text{FS}}) \quad v' = addrs^{-1}(\text{dom}\ \mu'_{\text{FS}})}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, t\rangle, k_2, ...\} \longrightarrow \{\mu'_{\text{FI}}, \mu'_{\text{FS}} | k_2, ..., \langle l'_{\text{cur}}, v', t'\rangle\}}$$

T-STUCK

$$\frac{}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, \Uparrow\rangle, k_2, ...\} \longrightarrow \{\mu_{\text{FI}}, \mu_{\text{FS}} | k_2, ...\}}$$

T-DONE

$$\frac{}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, v'\rangle, k_2, ...\} \longrightarrow \{\mu_{\text{FI}}, \mu_{\text{FS}} | k_2, ...\}}$$

T-FORK

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \quad \langle \Sigma | \textbf{withRefs}_{\text{FS}}\ v\ t\rangle \xrightarrow{fork(t')} \langle \Sigma' | t''\rangle \quad \Sigma' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu'_{\text{FS}}) \quad v' = addrs^{-1}(\text{dom}\ \mu'_{\text{FS}}) \quad k_{\text{new}} = \langle l'_{\text{cur}}, v', t'\rangle}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, t\rangle, k_2, ...\} \longrightarrow \{\mu'_{\text{FI}}, \mu'_{\text{FS}} | k_2, ..., \langle l'_{\text{cur}}, v', t''\rangle, k_{\text{new}}\}}$$

Figure 12: Semantics for $\lambda_\ell^{\parallel\text{-LIO}}$, parametric in the flow-sensitivity policy, i.e., with and without auto-upgrade.

$\{\mu_{\text{FI}}, \mu_{\text{FS}} | k_1, k_2, ...\}$, where $\mu_{\text{FI}}$ and $\mu_{\text{FS}}$ are respectively the flow-insensitive and flow-sensitive stores shared by all the threads $k_1, k_2, ...$ in the program. Since the memory stores are global, a thread $k$ is simply a tuple encapsulating the current label of the thread $l_{\text{cur}}$, the term under evaluation $t$, and a bag of references $v$ the thread may access, i.e., $k = \langle l_{\text{cur}}, v, t\rangle$.

The reduction rules for concurrent programs are mostly standard. Rule (T-STEP) specifies that if the first thread in the thread pool takes a step in $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, the whole concurrent program takes a step, moving the thread to the end of the pool. We note that the thread term $t$ executed with its stored current label $l_{\text{cur}}$, and a subset of the flow-sensitive memory store, by wrapping it in **withRefs**$_{\text{FS}}$. While the use of **withRefs**$_{\text{FS}}$ makes the extension straightforward, one peculiarity arises: since (T-STEP) always wraps the thread term $t$ with **withRefs**$_{\text{FS}}$, if $t$ does not reduce in one step to a value, and instead reduces to a term $t'$, the next time the thread is scheduled, we will superfluously wrap **withRefs**$_{\text{FS}}$ $t'$ with yet another **withRefs**$_{\text{FS}}$—thus preventing the thread from making progress! To address this problem, we extend the calculus with rule (WITHREFS-OPT) that collapses nested **withRefs**$_{\text{FS}}$ blocks.[5]

```
leakRef :: Ref^TCB Bool → LIO Bool
leakRef href = do
  tmp ← newRef L ()
  forkLIO $ do h ← readRef href
               when h $ writeRef tmp ()
  delay
  return $ labelOf tmp ≡ H
```

Figure 13: Attack on concurrent LIO with naive flow-sensitive reference extension.

Rules (T-DONE) and (T-STUCK) specify that once a thread term has reduced to a value or got stuck, which is represented by $\Uparrow$, the scheduler removes it from the thread pool and schedules the next thread.

As shown in Figure 12, to allow for dynamic thread creation, we extend $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$'s terms with **forkLIO**, and add a new reduction rule that sends an event to the scheduler, specifying the term to execute in a new thread.[6] Rule (T-FORK) describes the corresponding scheduler rule, triggered when a *fork* $(t')$ event is received. Here, we create a new thread $k_{\text{new}}$ whose current label $l'_{\text{cur}}$ and partition of the store, i.e., bag of references $v'$, is the same as that of the parent thread; the term evaluated in the newly created thread is provided in the event: $t'$. Subsequently, we add the new thread to the thread pool.

The final modification in extending $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ to $\lambda^{\parallel\text{-LIO}}_\ell$ given in Figure 12 is the removal of **toLabeled** from the underlying calculus. As described in [37], we must remove **toLabeled** to guarantee termination-sensitive non-interference. Importantly, however, **forkLIO** with synchronization primitives (e.g., flow-insensitive labeled MVars, as discussed in [37]) can be used to provide functionality equivalent to that of **toLabeled**. Due to space constraints we omit synchronization primitives from $\lambda^{\parallel\text{-LIO}}_\ell$; we only remark that extending $\lambda^{\parallel\text{-LIO}}_\ell$ to provide flow-sensitive labeled MVars follows in a straightforward way.

Since the flow-sensitive attack in Figure 7 relied on **toLabeled** to restore the current label, a natural question, given that we remove **toLabeled**, is whether we can use the naive flow-sensitive reference semantics of Section III for concurrent LIO. As shown by the attack code in Figure 13, in which we use **forkLIO** instead of **toLabeled** to address a potential label creep, the fundamental problem remains: the label on the reference label is not protected! This precisely motivated our principled approach of extending $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ to a concurrent setting as opposed to extending concurrent LIO with flow-sensitive references.

[5]This change also requires modifying (WITHREFS-CTX) to not be triggered when the term being evaluated is a **withRefs**$_{\text{FS}}$ term.

[6]In fact, the reduction rule for $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ must be changed to account for events. However, since *fork* is the only event in our system, we treat $\longrightarrow$ as implicitly carrying an empty event.

## V. Formal results

In this section, we show that our flow-sensitive enforcement can be embedded into the flow-insensitive version of LIO. Additionally, we provide security guarantees in terms of non-interference definitions by reusing previous results about LIO.

### A. Embedding into $\lambda_\ell^{\mathsf{LIO}}$

We now show that we can implement our flow-sensitive semantics in terms of flow-insensitive references. More concretely, every flow-sensitive reference with label $l_d$ created in a context where the current label is $l_o$ (and thus stored in $\mu_{\text{FS}}$ as $Lb^{\mathsf{TCB}}\ l_o\ (Lb^{\mathsf{TCB}}\ l_d\ t)$), can be represented by a flow-insensitive reference with label $l_o$, whose contents are another flow-insensitive reference containing $t$ and labeled $l_d$.

Figure 14 shows our implementation of the flow-sensitive reference operations in this setting. For a given store $\Sigma$, we define the $[\![-]\!]^{\Sigma}_{\text{FI}}$ function, which given a term $t$ in $\lambda^{\mathsf{LIO}}_{\ell,\text{FS}}$, produces a term $[\![t]\!]^{\Sigma}_{\text{FI}}$ in $\lambda^{\mathsf{LIO}}_{\ell}$, which is obtained by expanding the definitions of flow-sensitive operations in terms of flow-insensitive ones. This function is applied homomorphically in all other cases. We use the *WrapRef* to mark the flow-insensitive references that are being used to represent flow-sensitive ones, so as to distinguish them from ordinary flow-insensitive references. The functions *wrap* and *unwrap* are used to add and remove this boundary encoding. In the embedding of $\mathbf{writeRef}_{\text{FS}}$, we use **toLabeled** to make any changes to the current label (possibly caused by reading the outer reference) local to this operation. Inside **toLabeled**, the code fetches the inner reference ($\mathbf{readRef}_{\text{FI}}$), and then performs the actual write of the new value. If this fails, the computation diverges, but, importantly, the current label was raised (with $\mathbf{readRef}_{\text{FI}}$) to reflect the fact that the label on the label of the reference was observed. The embedding of $\mathbf{upgrade}_{\text{FS}}$ follows similarly, but further relies on $\mathbf{upgrade}_{\text{FI}}$ which, like the upgrade function for labeled values, creates a *new* reference with the same contents as the supplied reference $i$, but whose label is upgraded. For the sake of brevity, we do not explain the mapping in further detail, but remark that it is directly based on the rules in Figure 8.

We extend this definition naturally to convert $\lambda^{\mathsf{LIO}}_{\ell,\text{FS}}$ environments into $\lambda^{\mathsf{LIO}}_{\ell}$ environments, by having $[\![(l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}})]\!]_{\text{FI}} \triangleq (l_{\text{cur}}, \mu'_{\text{FI}})$ where $\mu'_{\text{FI}} = \mu_{\text{FI}}\,[a_1 \mapsto v_1, ..., a_n \mapsto v_n]$ for each binding of the form $a_i \mapsto Lb^{\mathsf{TCB}}\ l_i\ v_i$ in $\mu_{\text{FS}}$. Note that the domains of $\mu_{\text{FI}}$ and $\mu_{\text{FS}}$ are disjoint because the fresh$(\cdot)$ predicate that we use in the semantics is assumed to produce globally unique addresses.

In order to prove that our implementation is correct with respect to the semantics, we show that, if we take a program with flow-sensitive operations, and expand those operations, replacing them by the code in Figure 14, then its behavior corresponds with the flow-sensitive semantics.

$$
\begin{aligned}
&\mathit{wrap}\ r \triangleq \mathit{WrapRef}\ r\\
&\mathit{unwrap}\ (\mathit{WrapRef}\ r) \triangleq r\\
&[\![Ref^{\mathsf{TCB}}_{\text{FS}}\ r]\!]^{(l_{\text{cur}},\mu_{\text{FI}},\mu_{\text{FS}})}_{\text{FI}} \triangleq \mathit{wrap}\ (Ref^{\mathsf{TCB}}_{\text{FI}}\ (\mathbf{labelOf}_{\text{FI}}\ \mu_{\text{FS}}\ (r))\ r)\\
&[\![\mathbf{newRef}_{\text{FS}}]\!]^{\Sigma}_{\text{FI}} \triangleq \lambda l\ t.\mathbf{do}\\
&\quad i \leftarrow \mathbf{newRef}_{\text{FI}}\ l\ t\\
&\quad l_{\text{cur}} \leftarrow \mathbf{getLabel}\\
&\quad o \leftarrow \mathbf{newRef}_{\text{FI}}\ l_{\text{cur}}\ i\\
&\quad \mathbf{return}\ (\mathit{wrap}\ o)\\
&[\![\mathbf{readRef}_{\text{FS}}]\!]^{\Sigma}_{\text{FI}} \triangleq \lambda r.\mathbf{readRef}_{\text{FI}}\ (\mathit{unwrap}\ r) \gg\!= \mathbf{readRef}_{\text{FI}}\\
&[\![\mathbf{writeRef}_{\text{FS}}]\!]^{\Sigma}_{\text{FI}} \triangleq \lambda r\ t.\mathbf{let}\ o = \mathit{unwrap}\ r\ \mathbf{in}\ \mathbf{do}\\
&\quad l_{\text{cur}} \leftarrow \mathbf{getLabel}\\
&\quad \mathbf{toLabeled}\ (l_{\text{cur}} \sqcup (\mathbf{labelOf}\ o))\ \$\ \mathbf{do}\\
&\quad\quad i \leftarrow \mathbf{readRef}_{\text{FI}}\ o\\
&\quad\quad \mathbf{writeRef}_{\text{FI}}\ i\ t\\
&[\![\mathbf{labelOf}_{\text{FS}}]\!]^{\Sigma}_{\text{FI}} \triangleq \lambda r.\\
&\quad \mathbf{readRef}_{\text{FI}}\ (\mathit{unwrap}\ r) \gg\!= \mathbf{return}.\mathbf{labelOf}_{\text{FI}}\\
&[\![\mathbf{upgrade}_{\text{FS}}]\!]^{\Sigma}_{\text{FI}} \triangleq \lambda r\ l.\mathbf{let}\ o = \mathit{unwrap}\ r\\
&\qquad\qquad\qquad\qquad\qquad l' = \mathbf{labelOf}\ o\ \mathbf{in}\ \mathbf{do}\\
&\quad l_{\text{cur}} \leftarrow \mathbf{getLabel}\\
&\quad \mathbf{if}\ (l_{\text{cur}} \not\sqsubseteq l)\ \mathbf{then}\ \Uparrow\ \mathbf{else}\\
&\quad\quad \mathbf{toLabeled}\ l'\ \$\ \mathbf{do}\\
&\quad\quad\quad i \leftarrow \mathbf{readRef}_{\text{FI}}\ o\\
&\quad\quad\quad i' \leftarrow \mathbf{upgrade}_{\text{FI}}\ i\ l\\
&\quad\quad\quad \mathbf{writeRef}_{\text{FI}}\ o\ i'\\
&[\![\mathbf{withRefs}_{\text{FS}}\ v\ t]\!]^{(l_{\text{cur}},\mu_{\text{FI}},\mu_{\text{FS}})}_{\text{FI}} \triangleq [\![t]\!]^{(l_{\text{cur}},\mu_{\text{FI}},\mu'_{\text{FS}})}_{\text{FI}}\\
&\quad \mathbf{where}\\
&\quad\quad \mu'_{\text{FS}} = \{\, a \mapsto \mu_{\text{FS}}\ (a) \mid a \in \text{dom}\ \mu_{\text{FS}} \cap (\mathit{addrs}(v))\}
\end{aligned}
$$

Figure 14: Implementation mapping for flow-sensitive references. For all other terms, the function is applied homomorphically.

*Theorem 1 (Embedding $\lambda^{\mathsf{LIO}}_{\ell,\text{FS}}$ in $\lambda^{\mathsf{LIO}}_{\ell}$):* Let $t$ be a well-typed term in $\lambda^{\mathsf{LIO}}_{\ell,\text{FS}}$. Then if $\langle\Sigma|t\rangle \longrightarrow^* \langle\Sigma'|v\rangle$, we have $\langle[\![\Sigma]\!]_{\text{FI}}|[\![t]\!]^{\Sigma}_{\text{FI}}\rangle \longrightarrow^* \langle[\![\Sigma']\!]_{\text{FI}}|[\![v]\!]^{\Sigma}_{\text{FI}}\rangle$, and if $\langle\Sigma|t\rangle \longrightarrow^* \langle\Sigma'|\Uparrow\rangle$, then $\langle[\![\Sigma]\!]_{\text{FI}}|[\![t]\!]^{\Sigma}_{\text{FI}}\rangle \longrightarrow^* \langle[\![\Sigma']\!]_{\text{FI}}|\Uparrow\rangle$.

While straight forward, this theorem highlights an important result: in floating label systems, flow-sensitive references can be encoded in a calculus with flow-insensitive references and explicitly labeled values.

### B. Security guarantees for $\lambda^{\mathsf{LIO}}_{\ell,\text{FS}}$, $\lambda^{\mathsf{LIO}}_{\ell,\text{FS+AU}}$ and $\lambda^{\parallel\text{-}\mathsf{LIO}}_{\ell}$

From previous results, we know that LIO satisfies termination-insensitive non-interference (TINI) in the sequential setting, and termination-sensitive non-interference (TSNI) in the concurrent setting. By using the embedding theorem we can extend these results for LIO with flow-sensitive references.

For completeness, we now present our non-interference theorems, as straightforward applications of the theorems in previous work. Our security results rely on the notion of $l$-equivalence for terms and configurations, which captures the idea of terms that cannot be distinguished by an attacker which can observe data at level $l$. A pair of terms $t_1, t_2$ is said to be $l$-equivalent (written $t_1 \approx_l t_2$) if, after erasing

all the information more sensitive than $l$ from $t_1$ and $t_2$, we obtain syntactically equivalent terms. This definition extends naturally to configurations.

Intuitively, non-interference means that an attacker at level $l$ cannot distinguish among different runs of a program with $l$-equivalent initial configurations.

*Theorem 2 (TINI for $\lambda^{LIO}_{\ell,\text{FS}}$):* Consider two well-typed terms $t_1$ and $t_2$ in $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ which do not contain any $\cdot^{\text{TCB}}$ syntax nodes, such that $t_1 \approx_l t_2$, where $l$ is the attacker observation level. Let $\Sigma$ be an initial environment, and let

$$\langle \Sigma | t_1 \rangle \longrightarrow^* \langle \Sigma_1 | v_1 \rangle \text{ and } \langle \Sigma | t_2 \rangle \longrightarrow^* \langle \Sigma_2 | v_2 \rangle$$

Then, we have that $\langle \Sigma_1 | v_1 \rangle \approx_l \langle \Sigma_2 | v_2 \rangle$.

*Proof:* By expanding all the flow-sensitive operations in $t_1$ and $t_2$ using their definition given in Figure 14, we get terms in $\lambda^{\text{LIO}}_{\ell}$, which by Theorem 1 has equivalent semantics. Therefore, the result follows from the $\lambda^{\text{LIO}}_{\ell}$ TINI result of [36]. ■

*Corollary 1 (TINI for $\lambda^{LIO}_{\ell,\text{FS+AU}}$):* The previous non-interference result can be easily extended to $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$. In $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, the **unlabel** operation triggers the automatic upgrades mechanism, which performs the **upgrade** operation for every flow-sensitive reference in scope before actually raising the current label. Regardless of how **unlabel** is used, we note that the resulting term (after inserting the necessary **upgrade**s), is just an $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ term. Therefore, the main TINI result for $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ applies.

For the concurrent result, we need a supporting lemma which states that the current label is always at least as sensitive as the label of every reference in scope. Formally,

*Lemma 1:* Let $t$ be a well-typed term in $\lambda^{\parallel\text{-LIO}}_{\ell}$, $\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}})$ an initial environment and $a$ the address of a flow-sensitive reference $r$ in $\Sigma$, where $\mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l_o\ (Lb^{\text{TCB}}\ l_d\ v)$. Then, if $\langle \Sigma | t \rangle \longrightarrow^* \langle l'_{\text{cur}}, \mu'_{\text{FI}}, \mu'_{\text{FS}} | t' \rangle$, we have that $l_o \sqsubseteq l'_{\text{cur}}$.

*Proof:* Note that the result holds immediately after creating $r$, since the current label is the label on the label of $r$, i.e., $l_o = l_{\text{cur}}$. It is easy to show that $l_o$ is immutable, since there are no reduction rules that modify it. Moreover, given that the current label is monotonic, the only way in which $l_o \sqsubseteq l_{\text{cur}}$ can cease to hold is if $r$ is accessed from a different thread. But in order to pass $r$ to a different thread, a labeled object must be used as intermediary, and the label of such object would have to be at least $l_{\text{cur}}$, the current label in the thread that created $r$. As a result, if we were to pass $r$ to another thread in this way, then the target thread would also have to be tainted by $l_{\text{cur}}$, and the result would still hold. ■

We can now prove our non-interference theorem for $\lambda^{\parallel\text{-LIO}}_{\ell}$. This result is slightly stronger than TINI, since it implies that there can be no termination or internal timing leaks.

*Theorem 3 (TSNI for $\lambda^{\parallel\text{-LIO}}_{\ell}$):* Consider two well-typed terms $t_1$ and $t_2$ in $\lambda^{\parallel\text{-LIO}}_{\ell}$ which do not contain any $\cdot^{\text{TCB}}$ syntax nodes, such that $t_1 \approx_l t_2$, where $l$ is the attacker observation level. Let $\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}})$ be an initial environment, and let

$$\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, addrs^{-1}(\text{dom } \mu_{\text{FS}}), t_1 \rangle\} \longrightarrow^* M_1$$

Then, there exists some configuration $M_2$ such that $\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, addrs^{-1}(\text{dom } \mu_{\text{FS}}), t_2 \rangle\} \longrightarrow^* M_2$ and $M_1 \approx_l M_2$.

*Proof:* Because of Lemma 1, and looking at the embedding of **writeRef**$_{\text{FS}}$ and **upgrade**$_{\text{FS}}$, we note that the first **readRef**$_{\text{FI}}$ operation in each **toLabeled** block will be trying to raise the current label to $l$. However, since $l \sqsubseteq l_{\text{cur}}$, these operations will never effectively raise the current label. This means that using **toLabeled** is not necessary to preserve the semantics, because there is no need to restore the current label afterwards. As a result, and after removing **toLabeled** in these two cases, we note that the embedding produces valid concurrent $\lambda^{\text{LIO}}_{\ell}$ terms (which does not have **toLabeled**).

Finally, by expanding all the flow-sensitive operations in $t_1$ and $t_2$ using their definition given in Figure 14, we get terms in concurrent $\lambda^{\text{LIO}}_{\ell}$. Therefore, the result follows from the termination-sensitive non-interference of concurrent $\lambda^{\text{LIO}}_{\ell}$ [37]. We remark, however, that our embedding includes no synchronization to ensure atomicity of the flow-sensitive operations, so certain interleavings that break semantic equivalence are possible. Importantly, this does not affect security. ■

The detailed proofs for the results in this section can be found in the extended version of the paper [6].

### C. Permissiveness

In Section VI we compare the permissiveness of our system with previous flow-sensitive IFC systems. Here, we solely remark that the above results imply that our flow-sensitive calculus is as permissive as flow-insensitive LIO. In particular, any flow-insensitive LIO program can be trivially converted to a flow-sensitive LIO program (without auto-upgrades) by using flow-sensitive references instead of flow-insensitive ones. Since these references would never be upgraded, they will behave just like their flow-insensitive counterparts. This means that all existing LIO programs can be run in our flow-sensitive monitor. This includes Hails [13], a web framework using LIO, on top of which a number of applications have been built (e.g., GitStar[7], a code-hosting web platform, LearnByHacking[8], a blog/tutorial platform similar to School of Haskell, and LambdaChair [39], an EasyChair-like conference review system).

## VI. Related work

The *label on the label* could be seen as a fixed label that dictates which principals can read or modify the policy (inner label) of a flow-sensitive entity. In a different setting,

[7] www.gitstar.com
[8] www.learnbyhacking.org

trust management frameworks have explored this idea [4], where role-based rules are labeled to restrict the view on policies—the mere presence of certain policies could become inappropriate conduits of information.

Several authors propose an *existence security label* to remove leaks due to the termination covert channel [29, 30] or certain behaviors in dynamic nested data structures [16, 32]. While the existence security label and the label on the label are structurally isomorphic, they are used for different purposes and in different scenarios, e.g., it is not allowed to query labels in [16, 29, 30, 32].

Hunt and Sands [19] show the equivalence (modulo code transformation) between flow-sensitive and flow-insensitive type-systems. In a dynamic setting, Russo and Sabelfeld [31] formally pin down the menace of mutable labels for purely dynamic monitors. They prove that monitors require static analysis in order to be more permissive than traditional flow-sensitive type-systems. Targeting purely dynamic monitors, Austin and Flanagan [2, 3] describes the label-change policies *no-sensitive-upgrade* (originally proposed by Zdancewic [43]) and *permissive-upgrade*, where the latter is provably more permissive than the former, i.e., it rejects fewer programs. The no-sensitive-upgrade discipline stops execution on any attempt to change the label of a public variable inside a secret context. In contrast, permissive-upgrade allows such changes, marking the altered variables so that the program cannot subsequently branch on them. The marking consists in replacing the security label of the variables with **P**, where $\mathbf{L} \sqsubseteq \mathbf{H} \sqsubseteq \mathbf{P}$. Austin and Flanagan propose a *privatization* operation to boost the permissiveness of permissive-upgrade. It is not clear how this technique generalizes to arbitrary lattices. Moreover, the privatization operation can only enforce non-interference when outputs are suppressed after branching on a marked flow-sensitive reference. Unfortunately, none of the mentioned work consider flow-sensitive in the presence of concurrency. In fact, the notion of permissive-upgrade does not easily generalize to the concurrent setting, as this would require tracking occurrences of branches across threads.

The flow-sensitive extension for LIO is capable of encoding a generalized version of no-sensitive-upgrade by using an extra level of indirection. To illustrate this, we consider all the references as three-level nested labeled values—easily expressible in $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$. Specifically, we use flow-sensitive references of type $Ref_{\mathrm{FS}}\ (Labeled\ l\ a)$, where $\mu_{\mathrm{FS}}\ (a) = Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l'\ t))$ for a given flow-sensitive reference at address $a$. Observe that the first and second label of the nested structure are the same—this is effectively like having collapsed the first and second levels of the structure. For creating a reference, we execute $r \leftarrow \mathbf{newRef}_{\mathrm{FS}}\ l_{\mathrm{cur}}\ lv$, where $l_{\mathrm{cur}}$ is the current label and $lv$ is just a labeled value of the form $Lb^{\mathsf{TCB}}\ l''\ t$ for an arbitrary label $l''$. The newly created reference is of the form $Lb^{\mathsf{TCB}}\ l_{\mathrm{cur}}\ (Lb^{\mathsf{TCB}}\ l_{\mathrm{cur}}\ lv)$. As in the no-sensitive-upgrade case, writing a (labeled) value into r using $\mathbf{writeRef}_{\mathrm{FS}}\ r\ lv'$, requires that the current label flow to $l_{\mathrm{cur}}$, i.e., the label on the label at the time of creating $r$. However, the no-sensitive-upgrade rule prevents a program from modifying a reference $r$, created in a public environment (**L**), from being modified in a secret context (**H**). However, $\mathbf{writeRef}_{\mathrm{FS}}\ r\ lv'$ allows changing the initially stored labeled value $lv$ with another arbitrary-labeled value $lv'$. Such "label changes" (of the innermost labeled value) make our encoding more permissive than no-sensitive-upgrade, which only allows changing the label of a reference if the new label is above the initial label of the reference.

Recently, Hritcu et al. [18] propose a floating-label system called Breeze. Like LIO, Breeze allows changes in the current context label (i.e., $pc$) and only considered values with flow-insensitive labels. Given the design similarities with LIO [36], we believe that our results could be easily adapted to Breeze.

Hedin et al. [17] recently developed JSFlow, an IFC flow-sensitive monitor for JavaScript. The monitor uses the no-sensitive-upgrade label changing policy. To overcome some of the restrictions imposed by this discipline, the primitive **upgrade** is introduced to explicitly change labels. Our upgrade operation resembles that proposed by Hedin et al. Moreover, the extension to **unlabel** as described Section III can be seen as an automatic application of **upgrade** every time that the current label gets raised. Using testing, Birgisson et al. [5] automatically insert **upgrade** instructions to boost the permissiveness of no-sensitive-upgrade. We further extend this concept of (automatic) **upgrade**s to a concurrent setting.

The Operating System IFC community has also treated the mutable label problem in the presence of purely dynamic monitors. Specifically, modern IFC OSes such aas Asbestos [9], HiStar [44], and Flume [21] distinguish between subjects (processes), and objects ( files, sockets, etc.) such that the security labels for objects are immutable, while subject labels change according to the sensitivity of data being read. As in language-based IFC systems, changing the label of subjects and object can become a covert channel, if not handled appropriated. Hence, HiStar and Flume require that the label of a subject be done done explicitly by the subject code. Asbestos, on the other hand, allowed (unsafe) changes to labels as the result of receiving messages under specific and safe conditions. Our work extends on these concepts to allow for changes in object labels.

Coarse-grained IFC enforcements, similar to the ones found in IFC OS work, have been applied to web browsers. BFlow [42] tracks the flow of information at the granularity of protection zones, i.e., compartments composed of iframes. As in LIO, a zone's label, i.e., a subject's label, must be explicitly updated. Different from LIO, BFlow does not have finer-grained labeled object; hence the flow-sensitivity result is only applicable at the protection zone level. Naturally, by

taking a more fine grained approach, the DOM-tree could be thought of as being composed of flow-sensitive objects, whose security labels change according to the dynamic behavior of the web page [32].

Hoare-like logics for IFC are often flow-sensitive [e.g. 1, 28]. Different from dynamic approaches, these logics have the ability to observe all the execution paths and safely approximate label changes. As a result, no leaks due to label changes are present in provably secure programs. Le Guernic et al. [22, 23] combine dynamic and static checks in a flow-sensitive execution monitor. For a flow-sensitive type-system, Foster et al. [12] propose a **restrict** primitive that limits the use of variables' aliases in a block of code. Our **withRefs**$_{\text{FS}}$ is similar to **restrict** in being used to increase the permissiveness of the analysis.

## VII. Conclusions

We presented an extension of LIO with flow-sensitive references. As in previous flow-sensitive work, our approach allows secure label changes using an **upgrade** operation, as a way to boost the permissiveness of the IFC system, i.e., **upgrade** can be used to allow for the encoding of programs that would otherwise be rejected by the IFC monitor. Since manually inserting **upgrade** operations can be cumbersome, we extend the calculus to automatically insert upgrades whenever the current label is raised, while still giving programmers fine-grained control over which references untrusted code can upgrade. Importantly, our approach extends to a concurrent setting. To the best of our knowledge, this is the first work to address the problem of flow-sensitive label changes for a concurrent, purely dynamic IFC language. A further insight of this work was to show that, by leveraging nested labeled objects, both the sequential and concurrent calculi with flow-sensitive references can be encoded using only flow-insensitive constructs. As a consequence, our soundness proof can be reduced to an invocation of previous results for LIO.

## Acknowledgments

We thank our colleagues in the ProSec group at Chalmers, Stefan Heule, David Mazières, and Edward Z. Yang for the useful discussions. We thank the anonymous reviewers for constructive feedback on an earlier version of this work. This work was funded by DARPA CRASH under contract #N66001-10-2-4088 and the Swedish research agency VR. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

## References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.

[2] T. H. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[3] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10. ACM, 2010.

[4] S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, June 2008.

[5] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Proc. European Symp. on Research in Computer Security*, 2012.

[6] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating label systems: Extended version. http://www.cse.chalmers.se/~buiras/fslio.html, 2014.

[7] W. Cheng, D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[8] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12. ACM, 2012.

[9] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles*, SOSP '05. ACM, 2005.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10. USENIX Association, 2010.

[11] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

[12] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02. ACM, 2002.

[13] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and*

*Implementation*, October 2012.

[14] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[15] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[16] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, 2012.

[17] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, Mar. 2014.

[18] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All Your IFCException Are Belong to Us. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.

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

[20] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on Android (extended abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*. Springer, Sept. 2013.

[21] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007.

[22] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

[23] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06. Springer-Verlag, 2006.

[24] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

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

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

[27] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[28] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11. IEEE Computer Society, 2011.

[29] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 33–48. IEEE, 2013.

[30] W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing Interactive Programs. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, 2012.

[31] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

[32] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09. Springer-Verlag, 2009.

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

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

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

[36] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[37] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012.

[38] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, NordSec'11. Springer-Verlag, 2012.

[39] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[40] P. Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, August 1992.

[41] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *The 14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX, 2013.

[42] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *Proc. of the 4th ACM European Conference on Computer Systems*, EuroSys '09. ACM, 2009.

[43] S. Zdancewic. PhD thesis: Programming languages for information security. Technical report, Cornell University, 2002.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

## APPENDIX A. SEMANTICS FOR THE BASE CALCULUS

The reduction rules for pure and monadic terms are given in Figure 15. We define substitution $\{ t_2 / x \}\ t_1$ in the usual way: homomorphic on all operators and renaming bound names to avoid captures. Our label operations $\sqcup$, $\sqcap$, and $\sqsubseteq$ rely on the label-specific implementation of these lattice operators, as used in the premise of rule (LABELOP); we use the meta-level partial function $[\![\cdot]\!]_\ell$, which maps terms to values, to precisely capture this implementation detail.

The reduction rules for pure terms are standard. For instance, in rule (IFTRUE), when the branch has a true condition, i.e., $E\ [\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3]$, it reduces to the then branch ($E\ [t_2]$). The rest are self-explanatory and we do not discuss them any further.

Since all the IFC checks are performed by individual LIO terms, the definition for **return** and $(\gg\!=)$ are trivial. The former simply reduces to a monadic value by wrapping the term with the $\mathit{LIO}^{\mathsf{TCB}}$ constructor, while the latter evaluates the left-hand term and supplies the result to the right-hand term, as usual.

## APPENDIX B. ATTACK ON NAIVE FLOW-SENSITIVE REFERENCES

As in the attack of Figure 7, the *leakRef* of Figure 16 can be used to leak the value stored in a **H** reference *href*, while keeping the current label **L**, without using **labelOf**. Internally, the value is leaked into public reference *lref* by leveraging the fact that, based on a secret value, the label of a public reference (*tmp*) can be changed (or not). In the first **toLabeled** block, if $h \equiv \mathit{True}$, then the label of *tmp* is raised to **H** and its value is set to *True*. In the second **toLabeled** block, we read *tmp*, which may raise the current label to **H** if the secret is *True* (and thus *tmp* was upgraded). Indeed, if the secret is *True* (and thus $t \equiv \mathit{True}$) we leave the public reference intact: *True*. However, if the secret is *False*, the *tmp* reference is not modified in the first **toLabeled** block and thus when reading it in the second **toLabeled** block, the current label remains **L**, and since $t \equiv \mathit{False}$, we write *False* into the public reference. In both cases the value stored in *lref* corresponds to that of *href*, yet leaving the current label and the label of *lref* intact (**L**).

APP
$$\frac{}{E\ [(\lambda x.t_1)\ t_2] \longrightarrow E\ [\{ t_2 / x \}\ t_1]}$$

FIX
$$\frac{}{E\ [\mathbf{fix}\ (\lambda x.t)] \longrightarrow E\ [\{\mathbf{fix}\ (\lambda x.t) / x\}\ t]}$$

IFTRUE
$$\frac{}{E\ [\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3] \longrightarrow E\ [t_2]}$$

IFFALSE
$$\frac{}{E\ [\mathbf{if}\ \mathit{False}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3] \longrightarrow E\ [t_3]}$$

LABELOP
$$\frac{v = [\![l_1\ \otimes\ l_2]\!]_\ell}{E\ [l_1\ \otimes\ l_2] \longrightarrow E\ [v]}$$

RETURN
$$\frac{}{\langle \Sigma | \mathbf{E}\ [\mathbf{return}\ t] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathit{LIO}^{\mathsf{TCB}}\ t] \rangle}$$

BIND
$$\frac{}{\langle \Sigma | \mathbf{E}\ [(\mathit{LIO}^{\mathsf{TCB}}\ t_1) \gg\!= t_2] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [t_2\ t_1] \rangle}$$

Figure 15: Reduction rules for standard $\lambda_\ell^{\mathsf{LIO}}$ terms.

```
leakRef :: Ref^TCB Bool → LIO Bool
leakRef href = do
  lref ← newRef L True
  tmp ← newRef L False
  toLabeled H $ do h ← readRef href
                   when h $ writeRef tmp True
  toLabeled H $ do t ← readRef tmp
                   when (¬ t) $ writeRef lref False
  readRef lref
```

Figure 16: An attack in LIO with naive flow-sensitive reference extension without **labelOf**.