# Dynamic Enforcement of Dynamic Policies

Pablo Buiras    Bart van Delft
Chalmers University of Technology, Sweden

## Abstract

This paper presents SLIO, an information-flow control mechanism enforcing *dynamic policies*: security policies which change the relation between security levels while the system is running. SLIO builds on LIO, a floating-label information-flow control system embedded in Haskell that uses a runtime monitor to enforce security. We identify an implicit flow arising from the decision to change the policy based on sensitive information and introduce a corresponding check in the enforcement mechanism. We provide a formal security guarantee for SLIO, presented as a knowledge-based property, which specifies that observers can only learn information in accordance with the level ordering. Like LIO, SLIO is a generic enforcement mechanism, parametrised on the concrete instantiation of security labels and their policy change mechanism. To illustrate the applicability of our results, we implement well-known label models such as DLM, the Flowlocks framework, and DC labels in SLIO.

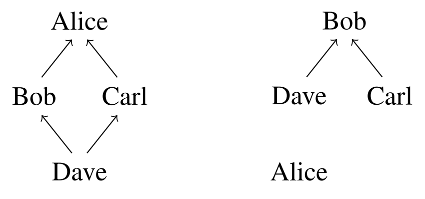
***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection – Information flow controls

***Keywords*** Information Flow Control, Dynamic Policies, LIO

## 1. Introduction

Many computing systems, such as personal computers, mobile phones and web pages, allow for the installation or inclusion of third-party code. This introduces the risk that untrusted code, either by intention or programming errors, leaks confidential information or violates the integrity of data. Information-flow control (IFC) mechanisms aim to ensure that the information flows in a system abide by the desired security policy.

*PLAS '15*, July 06, 2015, Prague, Czech Republic.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3661-1/15/07...$15.00.
http://dx.doi.org/10.1145/2786558.2786563



(a) Before.    (b) After.

Figure 1: Company's hierarchy before and after Alice leaves.

As a running example we consider an application responsible for combining and sharing employee files, where each file is labeled with the employee's name as security level. The application requires read and write access to all files, but the security policy dictates that information may only flow in accordance to the company's hierarchy. That is, an employee's file can only contain information from that employee, whereas information from files of all employees in a division may be written to the files of the divison manager. [1]

Many IFC mechanisms assume a *static* partial ordering ($\sqsubseteq$) between security levels (e.g. [20, 28, 30]). Figure 1a shows such an ordering for a company where Alice is the chief executive, Bob and Carl are managers of two different divisions, and Dave is an employee in both divisions. A commonly enforced security poperty is the notion of *noninterference* [15]: sensitive inputs may not influence outputs to less-sensitive locations. However, some mechanisms allow for occasional exceptions to this ordering in the form of declassification [29]. For example, a file of Alice's security level may be sent to the division managers in redacted form.

In this paper, we argue that the ordering between security levels can be much more *dynamic*, as others have argued before [1, 7, 21]. That is, rather than an occasional exception to the static ordering, the ordering might change much more drastically and permanently. As an example, we consider that Alice has accepted a position in a different company and leaves. Consequently, Bob and Dave are promoted

[1] One could argue that this particular situation could instead be modelled using roles for manager, employee etc. rather than employee names. We use this model for the sake of simplicity in our examples.

and Alice's documents have become isolated from the rest of the company, as show in Figure 1b. To enforce such dynamic policies, we need mechanisms that can account for the addition, as well as the removal of allowed information flows *during* program execution.

Similar changes in the information flow policy might occur in subscription-based services (such as music streaming) where information is only available for the time that the user paid for. Other examples are applications where users can change the policy themselves (e.g. in a smart phone system), and situations where the inherent value of information changes over time (e.g. revealing cards at the end of a poker game [3]). These examples show that it is natural for the value of information to change over time, motivating the need to support dynamic information-flow policies.

Enforcing dynamic policies brings new challenges when compared to IFC for static policies.

- Under the assumption of the static nature of a partial ordering it possible to approximate the security level of information. For example, a file containing information of both Bob and Carl could, in Figure 1a, be approximated with the security level Alice. Such approximations are incorrect when we need to account for the possibility that the ordering might change.
- With static orderings, IFC typically enforces noninterference [11, 15]. In the dynamic setting, the ordering between levels might be different at the moment of (each) output, so a different security condition is necessary.
- As the decision to modify the label ordering might be affected by sensitive inputs to the system, this creates a new potential flow of information that needs to be controlled.

This paper investigates the field of dynamic policies on both a theoretical and practical level in the context of dynamic enforcement, i.e. enforcement that checks information flows at runtime. As a starting point we use LIO [33], a dynamic information-flow control library for Haskell implemented as an embedded language augmented with runtime security checks. In its most general form, LIO is parametric on the policy specification language and label ordering to be used in the program; this ordering must be defined in advance and cannot change at runtime. The library provides a noninterference guarantee with respect to this ordering. LIO's genericity makes it a suitable framework for our exploration of dynamic policies and for implementing our results.

***Contributions*** Concretely, our contributions are:

- We present a new enforcement mechanism for dynamic policies, called SLIO. SLIO is a strict generalisation of LIO which, by providing a modifiable state component, allows for the generic enforcement of dynamic policies. We preserve LIO's genericity and abstract away from a concrete choice for both policy specification language and ordering change mechanism. (§ 3)
- We identify a clear constraint that needs to be checked for each ordering change to prevent information flow leaks via sensitive modifications to the label ordering. (§ 4)
- The original noninterference guarantee from LIO does not generalise to a setting with dynamic policies. For this purpose we introduce a new knowledge-based security condition. Though based on existing work ([1]), we introduce a novel extension to allow for the persistent relabeling (or: declassification) of information. (§ 5, § 6)
- To demonstrate the practicality of SLIO, we implement multiple instances for various policy specification languages, including the Decentralized Label Model (DLM) [27], Disjunction Category Labels [32] and Paralocks [7]. (§ 7)

Before turning to the details of these contributions, we summarise the essentials of LIO.

## 2. LIO

Labeled IO, or LIO [33], is a Haskell library that dynamically enforces information flow control, providing *termination-insensitive* non-interference [2] for sequential programs[2].

LIO leverages Haskell's monadic encapsulation of side-effects to provide security. A *monad* [23] is an abstract data type that can be used to structure effectful computations in purely functional languages. Monads specify how to build and bind computations together in sequence, and typically provide distinguished operations that model specific side-effects. Different monads are often used to model different kinds of effectful computations. In Haskell, monads are used to express all effects, including exception handling, nondeterminism, and input/output.

LIO leverages monads to precisely control what (side-effecting) operations the programmer is allowed to perform at any given time. An LIO program is a computation in the $LIO$ monad, composed from simpler monadic terms using the fundamental monadic combinators $\mathbf{return}$ and $(\gg=)$ (read as "bind").

The operation $\mathbf{return}\ x$ produces a computation which returns the value denoted by $x$. Function $(\gg=)$ is used to sequence LIO computations. Specifically, $t \gg= (\lambda x.t')$ takes the result produced by $t$ 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 $\mathbf{do}$-notation to write such monadic computations. For example, the program $t \gg= \lambda x.\mathbf{return}\ (x+1)$, which adds 1 to the value produced by $t$, can be written as shown below.

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

[2] A concurrent version of LIO exists [31], but this work is largely orthogonal to the work presented here.

In Haskell, input/output operations are provided by the $IO$ monad. That is, all computations that want to perform I/O operations have to be of the type $IO\ a$, where $a$ is the type of the returned value of the computation. The $LIO$ monad provided by the LIO library is intended to be used as a replacement for this type. It provides a collection of operations similar to $IO$, but enriched with security checks that prevent unwanted information flows. The noninterference guarantees provided by LIO only hold *within* this monad: an attacker is a piece of untrusted, potentially malicious LIO code that is run in the same context as the trusted code.

The LIO library employs the *floating-label* approach to dynamic information-flow control, which borrows ideas from the operating systems security research community [36] and brings them into the field of language-based security. Assuming a security lattice of labels (with operations $\sqcup$, $\sqcap$ and $\sqsubseteq$ defined on them), the $LIO$ monad uses its state to keep track of a *current label*, $l_{\text{cur}}$. This label represents the least upper bound over the labels on which the current computation depends. All the (I/O) operations provided by LIO take care to appropriately validate and adjust this label.

For example, when an LIO computation with current label $l_{\text{cur}}$ observes an entity with label $l_A$, its current label must change (possibly rise) to the least upper bound of the two labels, written $l_{\text{cur}} \sqcup l_A$. As it were, the current label *floats up* in the security lattice, to maintain its position as an upper bound on the security levels of the information that is currently in scope. Similarly, before performing a side-effect visible to label $l$, LIO first checks that the current label flows to $l$ ($l_{\text{cur}} \sqsubseteq l$) before allowing the operation to take place.

LIO is parameterised by a *label format*, i.e. the type of the labels is not fixed. Instead, actions work on a generic label type. The library leverages Haskell *type classes*, an overloading mechanism, to implement this: LIO actions have the context $(Label\ \alpha) \Rightarrow$ in their type signature, which restricts $\alpha$ to types that are instances of the $Label$ type class. This class requires the label type to implement security lattice operations such as $\sqsubseteq$, $\sqcup$, and $\sqcap$, making the behaviour of these operators depend on the particular label type of the labels to which they are applied.

***Labeled values*** Since LIO protects all values in scope with $l_{\text{cur}}$, the library also provides a way to manipulate differently-labeled data without monotonically increasing the current label, $l_{\text{cur}}$. For this purpose, there is a data type called $Labeled$, which represents explicit references to labeled, immutable data. It is still possible to bind a variable of, say, type $Labeled$ L $Int$, which contains an $Int$ protected by a label (of type L) different from $l_{\text{cur}}$.

The two most important functions that work on labeled values are **label** and **unlabel**. The action **label** $l\ v$ creates a $Labeled$ value with label $l$ and contents $v$, provided that $l_{\text{cur}} \sqsubseteq l$. That is, the label $l$ has to reflect that the value $v$ can depend on data of label $l_{\text{cur}}$. Dually, the action **unlabel** $lv$ raises $l_{\text{cur}}$ to $l_{\text{cur}} \sqcup l$, where $l$ is the label on $lv$, and returns $v$.

***Labeled references*** LIO also provides mutable state in the form of references. In Haskell, references are of type $IORef\ a$, where $a$ is the type of the contents of the reference. LIO introduces *labeled references*, typed $LIORef$ L $a$, where L is the type of the labels and $a$ is the type of the contents of the reference. The primitive operations on $LIORef$s are **newLIORef**, **readLIORef** and **writeLIORef**.

The action **newLIORef** $l\ v$ creates an $LIORef$ with label $l$ and contents $v$, provided that $l_{\text{cur}} \sqsubseteq l$. Given an $LIORef$ $r$ with label $lr$, the action **readLIORef** $r$ returns the value of $r$, and raises $l_{\text{cur}}$ to $l_{\text{cur}} \sqcup lr$. The action **writeLIORef** $r\ v$ replaces the contents of $r$ with $v$, provided that $l_{\text{cur}} \sqsubseteq lr$. Note that the operations on $LIORef$s and $Labeled$ values interact with the current label in analogous ways.

In floating-label systems, any computation on sensitive data raises the current label, even though the result of the computation may never be observable on a lower security level. As a result, it is possible for programs to inadvertently raise their current label to a point where they can no longer perform any useful side-effects, a situation known as *label creep*. For example, after a program reads from a file of a high security level, it is no longer allowed to write to a file of a lower security level, even when the information written does not depend in any way on the information read.

In order to mitigate this label creep, LIO provides the **toLabeled** operation. Given a computation $m$ that would raise $l_{\text{cur}}$ to $l'_{\text{cur}}$, **toLabeled** $l\ m$ executes $m$ without raising $l_{\text{cur}}$, and instead encapsulates $m$'s result in a $Labeled$ value protected by label $l$ – provided that $l'_{\text{cur}} \sqsubseteq l$. This allows for sub-computations that work on data above $l_{\text{cur}}$ without causing the main computation to raise its current label.

***Example*** Figure 2 shows an LIO program working with the lattice given in Figure 1a, of type $User$. The code defines a function $report$, which takes three labeled values as arguments. Information from these values is written into the files of certain principals, the $LIORef$s $aliceReport$ and $bobReport$. These files are assumed to be in scope, with labels Alice and Bob respectively. Firstly, Bob's data is unlabeled with **unlabel**, which raises the current label to Bob and binds $b$ to the contents of the labeled value. Then, a **toLabeled** computation is started, which unlabels data from Alice (raising the current label to Alice), binding it to $a$. Depending on the value of $b$, the block returns either $a + b$ or just $a$, which is bound to the main code block as a labeled value with label Alice. Note the use of **toLabeled** to delimit the scope of Alice's data and demarcate the block of code where her data might influence control flow: after the **toLabeled** block is finished, the current label is restored to Bob and the binding $a$ is no longer accessible.

Afterwards, the function unlabels data from Dave, combines it with Bob's data, and writes it to the reference $bobReport$. These operations would be potentially forbidden if we had not used **toLabeled**, since unlabeling $aliceData$ would have permanently raised the current label to Alice. Fi-

```
report :: Labeled User Int → Labeled User Int
        → Labeled User Int → LIO User ()
report bobData daveData aliceData =
  do b  ← unlabel bobData
     lv ← (toLabeled Alice
          (do a ← unlabel aliceData
              if b > 10
                then return (a + b)
                else return a))
     d ← unlabel daveData
     writeLIORef bobReport (combine d b)
     v ← unlabel lv
     writeLIORef aliceReport v
```

Figure 2: LIO code example

nally, the code unlabels the value returned from **toLabeled**, which raises the current label to Alice, and writes its contents to the reference *aliceReport*. Using **toLabeled** made it possible to perform side-effects at the level of Bob before the final write to *aliceReport*. As an illustration of the coarse-grainedness of the approach, note that in the label checks that are performed upon execution of **writeLIORef**, the particular values written to the references are irrelevant; when we write to *aliceReport*, the current label must flow to Alice, even if what we are attempting to write did not originate from an entity with label Alice.

## 3. Stateful LIO

In this section we introduce Stateful LIO, or SLIO for short: an extension of LIO with support for dynamic policies.

The key aspect of dynamic policies is that the ordering between labels can vary during execution. We therefore parametrise SLIO not only on the label format, but also on a data type representing the policy-relevant state of the application necessary to derive the relationship between labels. That is, the label type class now takes the form $Label\ \alpha\ \beta$, where $\alpha$ is the label format as before and $\beta$ is the type of the structure representing the policy-relevant state.

As exemplified in Figure 1b, the ordering between labels does not have to form a lattice per se in a dynamic setting, and the *Label* type class therefore no longer requires instances to implement lattice operations such as $\sqcup$ and $\sqcap$. The only operation that a *Label* instance is required to provide is the (reflexive and transitive) relation between labels in a given state, which we denote by $\sqsubseteq$. That is, $\sqsubseteq$ is of the type $S \rightarrow \mathrm{L} \rightarrow \mathrm{L} \rightarrow Bool$ and $\sqsubseteq\ s\ l_1\ l_2$, denoted $l_1 \sqsubseteq_s l_2$, returns *True* iff $l_1$ is less restrictive than $l_2$ in state $s$.

The LIO library only allows its own operations to interact with the current label. That is, only operations such as *readFile* and **unlabel** are allowed to read and modify $l_{\text{cur}}$. Similarly, SLIO's state contains both the current label and the current policy state $st$. The SLIO library exports operations that allow computations to read and modify $st$. Further encapsulations of the SLIO library may decide to only provide a limited interface to these operations, so as to better control the policy changes.

***The floating label*** SLIO only requires label formats to provide the $\sqsubseteq$ relation, but this clashes with the original floating label approach of LIO. LIO tracks $l$-labeled information entering the computation by computing $l_{\text{cur}} \sqcup l$. With varying policy states, the join-operator $\sqcup$ gives a different result in a different state – at times an upper bound may not even exist, as is the case for Alice $\sqcup$ Bob in Figure 1b. We address this in a manner inspired by the theoretical enforcement mechanisms suggested in [1]. We define *lset* to be a *set* of labels, to be used instead of the current label $l_{\text{cur}}$, and representing all labeled information that is present in the computation. Recording that $l$-labeled information has become accessible is then done by letting *lset* 'float up' to $lset \cup \{l\}$. Thus, *lset* behaves as the floating label in the powerset lattice of labels.

The original checks of the form $l_{\text{cur}} \sqsubseteq l_r$ that occurred e.g. when writing to a reference are replaced with a series of checks $\forall l \in lset.l \sqsubseteq_s l_r$ – where all checks need to hold in order for the flow to be allowed. That is, if all information that has entered the computation is allowed to flow to the label $l_r$ (according to the current policy), we allow the program to write to a file with that label. We abbreviate this check as $lset \sqsubseteq_s l_r$.

The **toLabeled** operation requires the programmer to explicitly specify the label to be placed on the result of the provided computation. Its operation becomes stateful as well, now checking that $lset' \sqsubseteq_{st'} l$ where $lset'$ and $st'$ are the current label resp. current policy state after executing the computation and $l$ is the provided label.

### 3.1 Exploring SLIO

The principal function of SLIO is to provide off-the-shelf enforcement for encodings of dynamic policy languages such as Paralocks [7] and DCLabels [32]. Before discussing this use of SLIO in more detail in § 7, we introduce the basic behaviour of SLIO programs using simple instantiations.

***Static Policies*** SLIO is a strict generalisation of LIO. More concretely, if an instance of *Label* does not use the policy state component in $\sqsubseteq_s$, SLIO behaves exactly like LIO. We demonstrate this for the static lattice shown in Figure 1a by using the unit type () for policy state.

```
data User = Alice | Bob | Carl | Dave
instance Label User () where
  l₁ ⊑ₛ Alice = True
  Dave ⊑ₛ l₂  = True
  l₁ ⊑ₛ l₂    = False
```

Instantiating SLIO with this *Label* format effectively enforces noninterference. To demonstrate this and later flows,

```
relabel l lv = toLabeled l (unlabel lv)
declassify l lv = do
  setState True
  result ← relabel l lv
  setState False
  return result
```

Figure 3: Declassification.

we introduce a function *copy* which copies information from one reference into another. As is common in LIO we perform this operation in a **toLabeled** computation, to avoid tainting the current label unnecessarily.

```
copy :: LIORef User String → LIORef User String
     → SLIO () User ()
copy from to = toLabeled (labelOf from) (do
  info ← readLIORef from
  writeLIORef to info)
```

SLIO detects a violation of noninterference when data from Carl is copied to Bob.

```
nonInterfering :: SLIO () User ()
nonInterfering = do
  dataAlice ← newLIORef Alice "Alice's data"
  dataBob   ← newLIORef Bob   "Bob's data"
  dataCarl  ← newLIORef Carl  "Carl's data"
  copy dataCarl dataAlice   -- Allowed flow.
  copy dataCarl dataBob     -- Violation detected.
```

***Dynamic Policies*** The last information flow to Bob would have been allowed if Bob had been promoted according to the policy depicted in Figure 1b. To enforce this dynamic policy using SLIO we need to incorporate the state component. The most direct way to encode the dynamic nature of the policy is to store the set of allowed flows in the state. In Haskell notation, this is a list of *User* pairs: $[(User, User)]$.

```
instance Label User [(User, User)] where
  l1 ⊑s l2 = l1 ≡ l2 ∨ (l1, l2) ∈ transClosure s
type LIOCompany = SLIO [(User, User)] User
```

We define $\sqsubseteq_s$ such that we can minimise the set of flow relations in the state, using *transClosure* to ensure that the relation is transitive. For brevity we introduce the type synonym *LIOCompany* for this kind of SLIO computations. The following function initialises the policy state to the situation shown in Figure 1a:

```
setInitState :: LIOCompany ()
setInitState = putState [(Dave, Bob), (Dave, Carl)
                        , (Bob, Alice), (Carl, Alice)]
```

The function *aliceLeaves* implements the event where Alice leaves the company, changing the label ordering from Figure 1a to Figure 1b.

```
aliceLeaves :: LIOCompany ()
aliceLeaves = do
  s ← getState
  putState (s ++ [(Carl, Bob)]) \\
           [(Bob, Alice), (Carl, Alice), (Dave, Carl)]
```

Assuming the references from the previous example, the dynamic nature of the information-flow policy can be manifested as follows:

```
dynamic :: LIOCompany ()
dynamic = do
  setInitState
  copyFile dataCarl dataAlice   -- Allowed flow.
  aliceLeaves
  copyFile dataCarl dataBob     -- Allowed flow.
  copyFile dataCarl dataAlice   -- Violation detected.
```

***Relabeling*** Figure 3 shows the function *relabel* which relabels a labeled value *lv* with the label *l*. This function can be used to perform declassification [29] using the following technique, described in [8]. Assume two security levels *Low* and *High*. The policy state is of type *Bool* and information can only flow from *High* to *Low* when the state is *True*. All other flows are allowed in either state. Figure 3 displays how this allows us to write *declassify* by temporarily changing the state and calling *relabel*. We revisit this pattern in § 6 where we construct a security condition which explicitly allows for persistent relabelings of information.

## 4. Conditional Change in Label Ordering

Allowing programs to freely change the policy state results in uncontrolled information flows, previously not present in LIO. This section establishes a condition on state change which ensures the absence of such flows. We identify this condition as a separate contribution of this paper, since it can also be applied on other enforcement mechanisms with dynamic policies (e.g. Paragon [8]).

We demonstrate the type of flow via a minimal example, assuming levels *Low* and *High* and a boolean state as in the relabeling example discussed above. Figure 4 displays a program which creates a *Low* reference $r$. When the *High* information provided equals 0 the computation changes the state to allow this information to flow to $r$. The result of the **toLabeled** computation gets labeled *High* but is ignored by the rest of the computation.

We assume the computation starts with $lset = \emptyset$ and policy state *False*. If the value of *highData* is 0, $r$ is updated while $l_{\text{cur}} = \{High\}$, which is allowed since $l_{\text{cur}} \sqsubseteq_{True} Low$. *lset* is set back to $\emptyset$ after the **toLabeled** computation,

```
$leak\ highData = \mathbf{do}$
  $r \leftarrow$ newLIORef $Low\ 1$
  $\_ \leftarrow$ toLabeled $High$ ($\mathbf{do}$
    $h \leftarrow$ unlabel $highData$
    when $(h \equiv 0)$ ($\mathbf{do}$
      setState $True$
      writeLIORef $r\ 0$))
  $v \leftarrow$ readLIORef $r$
  return $v$
```

Figure 4: Information leaks via conditional state change.

and the policy state is again $False$. Thus after reading $r$ and returning its value, $lset = \{Low\}$.

Although it might appear as if information only flows from $High$ to $Low$ when the policy state is $True$, this is not the case. In particular, when $highData \neq 0$, we learn this by observing that the value in $r$ did not change. Thus information flows from $High$ to $Low$ even though the policy state is never set to $True$ in the entire computation. Clearly the program should not be considered secure.

The computation's decision to allow the flow to $Low$ is based on information which, at the moment of decision, is *not* allowed to flow to $Low$. We identify this as the root of the problem. If instead the policy state changes to $True$ just *before* the **when** condition, the program would semantically be secure as it would *allow the conditional flow*, rather than to *conditionally allow the flow*.[3] This could be interpreted as the need to preserve the monotonicity property of the original LIO: information in scope can only become more confidential. That is, if at some point during the computation information can no longer flow to some label $l$, nothing can flow to $l$ in the rest of this (**toLabeled**) computation either.

In general, whenever the policy state changes from $s_1$ to $s_2$, we need to ensure that $s_2$ does not allow flows from labels in $lset$ which were previously disallowed in $s_1$. In other words, *the upper closure of lset should not increase by changing the ordering from* $\sqsubseteq_{s_1}$ *to* $\sqsubseteq_{s_2}$. To enforce this we require each instance of $Label$ to define an operation $incUpperSet :: S \rightarrow S \rightarrow \mathrm{L} \rightarrow Bool$, where $incUpperSet\ s_1\ s_2\ l$ returns $True$ if the upper set for label $l$ increases; that is, if there exists an $l'$ such that $l \not\sqsubseteq_{s_1} l'$ and $l \sqsubseteq_{s_2} l'$. The SLIO library then checks whether $\forall l \in lset\ .\ \neg\ (incUpperSet\ s_1\ s_2\ l)$.

In the example displayed in Figure 4, the call **setState** causes the SLIO library to check $incUpperSet\ False\ True\ \{High\}$. This should under a correct implementation return $True$, since the change in ordering increases the upper closure of $lset = \{High\}$ from $\{High\}$ to $\{Low, High\}$.

[3] Since the floating label approach does not distinguish explicit from implicit flows, the **setState** operation should, in practice, be placed before the unlabeling of $highData$.

$$
\begin{array}{lrl}
Values & v ::= & True \mid False \mid () \mid \lambda x.e \mid \ell \mid SLIO\ e \mid Lb\ l\ e \\
Expr. & e ::= & v \mid x \mid e\ e \mid \mathbf{fix}\ e \mid \mathbf{if}\ t\ \mathbf{then}\ e\ \mathbf{else}\ e \\
 & \mid & \mathbf{return}\ e \mid e \gg\!= e \mid \mathbf{getLabel} \\
 & \mid & \mathbf{toLabeled}\ l\ e \mid \mathbf{toLabeledRet}\ ls\ s\ l\ e \\
 & \mid & \mathbf{label}\ e\ e \mid \mathbf{unlabel}\ e \\
 & \mid & \mathbf{labelOf}\ e \mid \sqsubseteq \mid \mathbf{newLIORef}\ e\ e \\
 & \mid & \mathbf{writeLIORef}\ e\ e \mid \mathbf{readLIORef}\ e \\
 & \mid & \mathbf{setState}\ e \mid \mathbf{getState}
\end{array}
$$

Figure 5: $\lambda_{\text{SLIO}}$ syntax.

The requirement that $\sqsubseteq_s$ is a transitive relation is especially relevant here, since this check aims to control the yet unknown remainder of the execution, where flows might happen in a transitive manner. That this check enforces the monotone property of SLIO and prevents the information flows arising from policy state change is an essential step in the proof for our security condition (§ 6).

## 5. Semantics

In this section we formalise SLIO as a simply-typed, call-by-name $\lambda$-calculus, which we call $\lambda_{\text{SLIO}}$. Figure 5 gives the formal syntax of $\lambda_{\text{SLIO}}$, parametric in the label type $\ell$. Syntactic categories $v$ and $e$ represent values and expressions, respectively. Expressions of the form $SLIO\ e$, $Lb\ l\ e$ and **toLabeledRet** $ls\ s\ l\ e$ are not part of the surface syntax, i.e., they are not made available to programmers and are solely used internally to provide semantics to the other expressions. Values include standard primitives (Booleans, unit, and $\lambda$-abstractions) and terminals corresponding to labels ($\ell$) and monadic values ($SLIO\ e$). The latter denote effectful computations subject to security checks. Expressions consist of standard constructs (values, variables $x$, function application, the **fix** operator, and conditionals), a terminal corresponding to $\sqsubseteq$ (the partial order on labels), standard monadic operators ($\mathbf{return}\ e$ and $e \gg\!= e$), $\mathbf{getLabel}$, **toLabeled**, operations on labeled values and references, and operations for setting and getting the policy state (**setState** and **getState**). Even though the full LIO library can handle several other kinds of entities, such as files, we focus on labeled values and references since they accurately represent the security mechanisms of LIO; the security checks and effects on other kinds of labeled entities are analogous. For brevity, we do not describe the $\lambda_{\text{SLIO}}$ type system since it is standard and is not relevant for security checks. In what follows, we assume that all expressions involved are well-typed.

A top-level $\lambda_{\text{SLIO}}$ computation is a *configuration* of the form $\langle \Sigma | e \rangle$, where $e$ is the monadic expression and $\Sigma$ is the state associated with the expression. The state $\Sigma$ contains the current label set $lset$, the current policy state $st$, and the store $\phi$ (for references). We give a small-step operational semantics for $\lambda_{\text{SLIO}}$ in the form of a reduction relation $\longrightarrow$. Fig-

$$\textsc{getLabel}\quad \frac{\Sigma = (\mathit{lset}, \mathit{st}, \phi)}{\langle \Sigma | \mathbf{E}\,[\mathbf{getLabel}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathbf{return}\ \mathit{lset}] \rangle}$$

$$\textsc{label}\quad \frac{\Sigma = (\mathit{lset}, \mathit{st}, \phi) \qquad \mathit{lset}\ \sqsubseteq_{st}\ l}{\langle \Sigma | \mathbf{E}\,[\mathbf{label}\ l\ e] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathbf{return}\ (\mathit{Lb}\ l\ e)] \rangle}$$

$$\textsc{unlabel}\quad \frac{\begin{array}{c}\Sigma = (\mathit{lset}, \mathit{st}, \phi)\\ \mathit{lset}' = \mathit{lset}\ \cup\ \{l\} \qquad \Sigma' = (\mathit{lset}', \mathit{st}, \phi)\end{array}}{\langle \Sigma | \mathbf{E}\,[\mathbf{unlabel}\ (\mathit{Lb}\ l\ e)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ e] \rangle}$$

$$\textsc{labelOf}\quad \frac{}{E\ [\mathbf{labelOf}\ (\mathit{Lb}\ l\ e)] \longrightarrow E\ [l]}$$

$$\textsc{newLIORef}\quad \frac{\begin{array}{c}\Sigma = (\mathit{lset}, \mathit{st}, \phi)\\ \mathit{lset}\ \sqsubseteq_{st}\ l \qquad \Sigma' = (\mathit{lset}, \mathit{st}, \phi\ [x \to \mathit{Lb}\ l\ e]) \qquad \text{fresh}(x)\end{array}}{\langle \Sigma | \mathbf{E}\,[\mathbf{newLIORef}\ l\ e] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ x] \rangle}$$

$$\textsc{writeLIORef}\quad \frac{\begin{array}{c}\Sigma = (\mathit{lset}, \mathit{st}, \phi) \qquad \mathit{Lb}\ l\ v = \phi\ (x)\\ \mathit{lset}\ \sqsubseteq_{st}\ l \qquad \Sigma' = (\mathit{lset}, \mathit{st}, \phi\ [x \to \mathit{Lb}\ l\ e])\end{array}}{\langle \Sigma | \mathbf{E}\,[\mathbf{writeLIORef}\ x\ e] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ ()] \rangle}$$

$$\textsc{readLIORef}\quad \frac{\begin{array}{c}\Sigma = (\mathit{lset}, \mathit{st}, \phi) \qquad \mathit{Lb}\ l\ e = \phi\ (x)\\ \mathit{lset}' = \mathit{lset}\ \cup\ \{l\} \qquad \Sigma' = (\mathit{lset}', \mathit{st}, \phi)\end{array}}{\langle \Sigma | \mathbf{E}\,[\mathbf{readLIORef}\ x] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ e] \rangle}$$

$$\textsc{set}\quad \frac{\begin{array}{c}\Sigma = (\mathit{lset}, \mathit{st}, \phi)\\ \forall l \in \mathit{lset}\ .\ \neg\ \mathit{incUpperSet}\ (\mathit{st}, v, l) \qquad \Sigma' = (\mathit{lset}, v, \phi)\end{array}}{\langle \Sigma | \mathbf{E}\,[\mathbf{setState}\ v] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ ()] \rangle}$$

$$\textsc{get}\quad \frac{\Sigma = (\mathit{lset}, \mathit{st}, \phi)}{\langle \Sigma | \mathbf{E}\,[\mathbf{getState}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathbf{return}\ \mathit{st}] \rangle}$$

$$\textsc{toLabeled}\quad \frac{\Sigma = (\mathit{lset}, \mathit{st}, \phi)}{\begin{array}{c}\langle \Sigma | \mathbf{E}\,[\mathbf{toLabeled}\ l\ m] \rangle \longrightarrow\\ \langle \Sigma | \mathbf{E}\,[m \ggg\!= \mathbf{toLabeledRet}\ \mathit{lset}\ \mathit{st}\ l] \rangle\end{array}}$$

$$\textsc{toLabeledRet}\quad \frac{\Sigma = (\mathit{lset}, \mathit{st}, \phi) \qquad \mathit{lset}\ \sqsubseteq_{st}\ l}{\begin{array}{c}\langle \Sigma | \mathbf{E}\,[\mathbf{toLabeledRet}\ \mathit{ls}\ s\ l\ v] \rangle \longrightarrow\\ \langle (\mathit{ls}, s, \phi) | \mathbf{E}\,[\mathbf{return}\ (\mathtt{Lb}\ l\ v)] \rangle\end{array}}$$

Figure 6: SLIO semantics (standard $\lambda$-calculus rules elided).

ure 6 shows the relevant reduction rules for $\longrightarrow$. Intuitively, $\langle \Sigma | e \rangle \longrightarrow \langle \Sigma' | e' \rangle$ means that, starting from a configuration $\langle \Sigma | e \rangle$, it is possible to take a step to $\langle \Sigma' | e' \rangle$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The reduction rules for $\lambda_{\text{SLIO}}$ are specified using evaluation contexts in the style of Felleisen and Hieb [14]. Figure 7 defines the evaluation contexts for pure expressions ($E$) and monadic ($\mathbf{E}$) expressions for $\lambda_{\text{SLIO}}$. The definitions are mostly standard. Note that monadic expressions are evaluated only at the outermost use of bind ($\mathbf{E} \ggg\!= e$), as in Haskell.

$$\begin{array}{rl} E ::= & [\,]\ |\ E\ e\ |\ \mathbf{fix}\ E\ |\ \mathbf{if}\ E\ \mathbf{then}\ e\ \mathbf{else}\ e\\ & |\ \mathbf{label}\ E\ e\ |\ \mathbf{unlabel}\ E\ |\ \mathbf{labelOf}\ E\ |\ \mathbf{toLabeled}\ E\ e\\ & |\ \mathbf{newLIORef}\ E\ e\ |\ \mathbf{writeLIORef}\ E\ e\ |\ \mathbf{readLIORef}\ E\\ & |\ \mathbf{setState}\ E\\ \mathbf{E} ::= & E\ |\ \mathbf{E} \ggg\!= e \end{array}$$

Figure 7: Evaluation contexts for SLIO.

Rule (writeLIORef) is used to assign a value to a mutable reference. The rule looks up the reference in the memory store $\phi$, where it is represented as a labeled value $\mathit{Lb}\ l\ v$. Then, a security check is performed to ensure that the current label set flows to $l$ ($\mathit{lset}\ \sqsubseteq_{st}\ l$) (note that this is actually the conjunction of several checks, one per label in the label set). If the check passes, the memory store is updated with the new value.

Rule (readLIORef) reads a value from a mutable reference. The current label set is updated to include the label of the reference, to reflect the fact that the contents of the reference are now in scope and could potentially influence side-effects in the future. The rules for labeled values interact with the current label set in an analogous manner.

Rules (set) and (get) define the semantics for the new operations in $\lambda_{\text{SLIO}}$, namely **setState** and **getState**. As expected, they work by writing and reading the policy state in the $\lambda_{\text{SLIO}}$ state, except that **setState** additionally checks that the upper closure is not increased. This check is performed by a user-supplied function, as explained in Section 4.

Finally, the rule (toLabeled) binds the monadic computation $m$ to the internal-only expression **toLabeledRet**. The rule (toLabeledRet) resets the policy-relevant components to their value before $m$, returning the result of $m$ as a labeled value only if the current configuration allows information to flow to the label $l$ specified by the programmer.

## 6. Semantic Soundness

In this section we define a security condition for dynamic policies and show that it is guaranteed by $\lambda_{\text{SLIO}}$. We first present the attacker model, which is similar to the one used for static policies in LIO [33].

### 6.1 Attacker model

SLIO aims to provide security guarantees even in the presence of untrusted code. Following this assumption, we make configurations the observations of our model.

**Definition 1** (Trace). *A configuration produces a* trace *of configurations, written* $\langle \Sigma_0 | e_0 \rangle \Downarrow t$ *with* $t$ *a sequence of configurations* $\langle \Sigma_0 | e_0 \rangle \ldots \langle \Sigma_n | e_n \rangle$*, if there exists an evaluation* $\langle \Sigma_0 | e_0 \rangle \longrightarrow \ldots \longrightarrow \langle \Sigma_n | e_n \rangle$.

As in [33] we use a technique called *term erasure*. Attackers are represented by a security level $A$. The function $\varepsilon_A(t)$

$$\varepsilon_A(\langle\Sigma|e\rangle\cdot t) = \begin{cases} \langle\varepsilon^s_A(\Sigma)|\varepsilon^s_A(e)\rangle\cdot\varepsilon_A(t) & \text{if } obs_A(\langle\Sigma|e\rangle), \\ & \text{with } s = \Sigma.st \\ \varepsilon_A(t) & \text{otherwise} \end{cases}$$

$$\varepsilon^s_A(\Sigma) = \Sigma[\phi \mapsto \varepsilon^s_A(\Sigma.\phi)]$$

$$\varepsilon^s_A(\Sigma.\phi) = \{(x, \varepsilon^s_A(\Sigma.\phi\ (x))) \mid x \in \text{dom}(\Sigma.\phi)\}$$

$$\varepsilon^s_A(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \varepsilon^s_A(e) & \text{if } l \sqsubseteq_s A \\ \texttt{Lb}\ l\ \bullet & \text{otherwise} \end{cases}$$

Figure 8: Erasure function for non-trivial cases.

erases from the trace of configurations $t$ all the information which is not observable on level $A$.

Since the current label set protects all available information, $A$ can only observe configurations where the current label set can flow to $A$ (according to the current policy state).

**Definition 2** ($A$-observable configuration)**.** *A configuration $\langle\Sigma|e\rangle$ is* observable *to an attacker on level A, written $obs_A(\langle\Sigma|e\rangle)$, iff $\Sigma.lset \sqsubseteq_{\Sigma.st} A$.*

Configurations which are not observable to $A$ are removed from the trace entirely, as shown in Figure 8. From the configurations that are not removed, the erasure function erases only the information that cannot flow to $A$, so the erased configuration is $\langle\varepsilon^s_A(\Sigma)|\varepsilon^s_A(e)\rangle$. Here we fix $s$ as the current policy state in that configuration, i.e, $s = \Sigma.st$.

For most cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon^s_A(\ \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = \textbf{if}\ \varepsilon^s_A(e)\ \textbf{then}\ \varepsilon^s_A(e_1)\ \textbf{else}\ \varepsilon^s_A(e_2)$). The interesting cases for this function are displayed in Figure 8. The syntax node $\bullet$ represents an erased expression: information that is not observable to an attacker at level $A$. In particular, $\varepsilon^s_A(\texttt{Lb}\ l\ e)$ erases to $\texttt{Lb}\ l\ \bullet$ when $l \not\sqsubseteq_s A$.

### 6.2 Security Condition

The two-run noninterference condition associated with LIO does not translate well to a setting with dynamic policies. Instead, we find that epistemic properties [4, 5, 7] form a more natural basis for defining information flow conditions, in particular in the context of dynamic policies.

As a starting point we adapt the security condition from Askarov and Chong [1]. This condition extends from the notion of *gradual release* [4], which builds around the concept of a *knowledge set*: the set of initial inputs that could have resulted in the observations made by an attacker. Following Delft et al. [12] we instead talk about the *exclusion* knowledge set: the set of initial inputs that could *not* have resulted in these observations. This matches the intuition that a larger (exclusion) knowledge set implies more knowledge.

Since $\lambda_{\text{SLIO}}$ values can be SLIO computations, we let the initial expression take the role of initial (secret) input. Let $e$ be such a secret input which is evaluated in $\Sigma_0$, the initial state with $lset = \emptyset$ and $\phi = \emptyset$ – the initial value of $st$ varies between instantiations. Given $\langle\Sigma_0|e\rangle \Downarrow t$, i.e. this configuration produces a sequence of configurations $t$, let $o = \varepsilon_A(t)$ the observations made by attacker $A$. The exclusion knowledge of $A$ is then defined as the set of inputs that could not have produced the same observations:

$$ek_A(o) = \{e' \mid \neg\exists t'.\langle\Sigma_0|e'\rangle \Downarrow t' \text{ with } \varepsilon_A(t') = o\}$$

Now let $\langle\Sigma_0|e\rangle \Downarrow t\cdot\alpha$, with $obs_A(\alpha)$. What an attacker learns from this new configuration $\alpha$ can then be expressed as $ek_A(\varepsilon_A(t\cdot\alpha)) \setminus ek_A(\varepsilon_A(t))$: the set of inputs additionally excluded. To specify that the attacker does not learn anything new from this observation, we can simply require that $ek_A(\varepsilon_A(t\cdot\alpha)) \setminus ek_A(\varepsilon_A(t)) = \emptyset$.

This definition would also not allow the attacker to learn anything from the fact that the computation produced another output after producing trace $t$. SLIO, and LIO, however do not check for leaks via progress and allow computations to diverge based on sensitive information. This means that information might e.g. be leaked by the fact that a **toLabeled** computation terminated. Askarov and Chong present a termination-insensitive condition by introducing the attacker's *progress knowledge*. That is, we allow the attacker to exclude also those initial commands $e'$ that cannot produce another observation:

$$ek^+_A(o) = \{e' \mid \neg\exists t', \alpha'.\langle\Sigma_0|e'\rangle \Downarrow t'\cdot\alpha' \\ \text{with } \varepsilon_A(t') = o \text{ and } obs_A(\alpha')\}$$

Finally, we do allow the attacker to exclude *some* initial inputs using observation $\alpha$, as long as this is in accordance with the ordering determined by the state $s$ in which $\alpha$ was produced. Following Askarov and Chong, we allow the attacker to exclude those inputs that are not equal to $e$ when observed under state $s$.

**Definition 3** (Input release)**.** *Given input e, the state s allows an attacker A to exclude the set of inputs $I_A(e,s)$, where*

$$I_A(e,s) = \{e' \mid \varepsilon^s_A(e) \neq \varepsilon^s_A(e')\}$$

The security condition is then that for every $\langle\Sigma_0|e\rangle \Downarrow t\cdot\alpha$ with $obs_A(\alpha)$, $\alpha = \langle\Sigma_n|e_n\rangle$ and $\Sigma_n.st = s$, the attacker's increase in knowledge is bounded by $I_A(e,s)$:

$$ek_A(\varepsilon_A(t\cdot\alpha)) \setminus ek^+_A(\varepsilon_A(t)) \subseteq I_A(e,s)$$

***Relabeling support*** The examples from § 3.1 show that SLIO allows for persistent relabelings of data. By this we mean that we want to have the possibility to place a value of label $l_1$ in a container with label $l_2$, after which the value from this container is treated as if it has label $l_2$. We used this in a simple encoding of declassification, shown in Figure 3.

$$
\begin{array}{l}
\mathit{add}\ (\mathit{TopSecret}\ \sqsubseteq\ \mathit{Secret}) \\
\mathit{sec} \leftarrow \mathit{relabel}\ \mathit{Secret}\ \mathit{top} \\
\mathit{remove}\ (\mathit{TopSecret}\ \sqsubseteq\ \mathit{Secret}) \\
\mathit{add}\ (\mathit{Secret}\ \sqsubseteq\ \mathit{Public}) \\
\mathit{pub} \leftarrow \mathit{relabel}\ \mathit{Public}\ \mathit{sec}
\end{array}
$$

Figure 9: Relabeling example.

Such relabelings are a desirable feature of an IFC language. The fact that persistent relabelings are part of various policy languages that we would like SLIO to encode, notably including the DLM described in § 7, further motivates the need for a security condition that supports them.

It turns out that our direct adaptation of Askarov and Chong's security condition does *not* allow for persistent relabelings, as the example in Figure 9 shows. An attacker of level $\mathit{Public}$ observes the value of $\mathit{pub}$ (and therefore learns the value of $\mathit{top}$) when the current ordering does not allow flows from $\mathit{TopSecret}$ to $\mathit{Public}$. The security condition requires that a run started in a state with a different value for $\mathit{top}$ should yield the same value for $\mathit{pub}$ as in the observed run. Since this is not the case, the program violates the security condition. In the terminology of facets of dynamic policies [6], the condition does not allow for the *time-transitive* flows that we desire.

We conclude that we need to allow the attacker to exclude inputs based on relabeled information, *in addition to* the inputs described by $I_A(e, s)$.

Given the flow relation determined by policy state $s$, let $L$ be the set of levels from which $A$ is allowed to learn. That is, $L = \{l \mid l \sqsubseteq_s A\}$. To define the information that is collectively known by $L$, we introduce the erasure function on traces for multiple levels $\varepsilon_{\mathrm{L}}(t)$, shown in Figure 10. This function only erases labeled data or configurations if none of the levels in $L$ can observe it.

Given $\langle \Sigma_0 | e\rangle \Downarrow t \cdot \alpha$, with $\mathit{obs}_A(\alpha)$, and $s$ the policy state in $\alpha$. We can specify the information that is released to $A$ by relabelings in $t$ as follows.

**Definition 4** (Relabeling release). *Given a trace t, the policy state s allows an attacker A to exclude the set of inputs $R_A(t, s)$, where $L = \{l \mid l \sqsubseteq_s A\}$ in*

$$R_A(t, s) = \{e' \mid \neg\exists t', \alpha'. \langle \Sigma_0 | e'\rangle \Downarrow t' \cdot \alpha' \textit{ with } \mathit{obs}_A(\alpha') \textit{ and } \varepsilon_L(t) = \varepsilon_L(t')\}$$

We straightforwardly extend the security condition to additionally allow an attacker to learn information that has been released by relabelings.

**Definition 5** (Termination-insensitive security). *Command e is secure against an attacker A if for all traces t and configurations $\alpha$ such that $\langle \Sigma_0 | e\rangle \Downarrow t \cdot \alpha$ with $\mathit{obs}_A(\alpha)$, $\alpha = \langle \Sigma_n | e_n\rangle$ and $\Sigma_n.st = s$, the attacker's increase in knowledge is bounded by $I_A(e, s)$ and $R_A(t, s)$:*

$$\mathit{ek}_A(\varepsilon_A(t \cdot \alpha)) \setminus \mathit{ek}_A^{+}(\varepsilon_A(t)) \subseteq I_A(e, s) \cup R_A(t, s)$$

$$\varepsilon_L(\langle \Sigma | e\rangle \cdot t) = \begin{cases} \langle \varepsilon_{\mathrm{L}}^{s}(\Sigma) | \varepsilon_{\mathrm{L}}^{s}(e)\rangle \cdot \varepsilon_{\mathrm{L}}(t) \\ \quad \text{if } \exists l \in L \,.\, \mathit{obs}_l(\langle \Sigma | e\rangle), s = \Sigma.st \\ \varepsilon_{\mathrm{L}}(t) \quad \text{otherwise} \end{cases}$$

$$\varepsilon_{\mathrm{L}}^{s}(\mathtt{Lb}\ l\ e) = \begin{cases} \mathtt{Lb}\ l\ \varepsilon_{\mathrm{L}}^{s}(e) & \text{if } \exists l' \in L \,.\, l \sqsubseteq_s l' \\ \mathtt{Lb}\ l\ \bullet & \text{otherwise} \end{cases}$$

Figure 10: Multi-level erasure function for cases different from single-level erasure.

*Remark* 1. Our choice in defining the set $R_A(t, s)$ is not an arbitrary one. In Appendix B we list a collection of other possible definitions that also appear reasonable, but either do not support relabelings to the extent that we find natural, or allow for flows that we consider insecure, such as the release of information via conditional state change.

*Remark* 2. Askarov and Chong identify that a perfect recall attacker might learn less from an observation than an attacker who has forgotten part of the earlier knowledge (i.e. an attacker with some knowledge $\mathit{wk} \subset \mathit{ek}_A^{+}(\varepsilon_A(t))$. Although our definitions assume a perfect recall attacker, we observe that $\mathit{ek}_A^{+}(\varepsilon_A(t)) \subseteq R_A(t, s)$ since $L$ always includes $A$ itself. Therefore the security condition could be specified as $\mathit{ek}_A(\varepsilon_A(t \cdot \alpha)) \subseteq I_A(e, s) \cup R_A(t, s)$. Hence by allowing for relabeling release by Definition 4, a program that is secure by Definition 5 is also secure against even the most forgetful attacker with $\mathit{wk} = \emptyset$.

**Theorem 1.** *All $\lambda_{\scriptscriptstyle SLIO}$ computations are termination-insensitive secure.*

*Proof.* See Appendix A. □

## 7. Encodings

To demonstrate the genericity of SLIO we provide encodings for various policy specification frameworks. For each policy language, SLIO provides an enforcement mechanism in exchange for the relatively minor effort of encoding that language. This allows for easy exploration of policy languages, as well as the effects of modifying and extending them. We expect user applications to be typically written against such an encoding, rather than creating an *ad hoc* policy language using 'bare' SLIO (as we did in § 3.1). Using an existing policy language one can write natural policy labels with well-established semantics.

The following policy languages have been encoded in SLIO and are available from [9]: Two-Point Lattice, Flow-policies for non-disclosure [21], the Decentralized Label

Model (DLM) [27], Disjunction Category Labels [32] and Paralocks [7].

Rather than the dynamic policy-oriented Disjunction Category labels or Paralocks, we use this section to present the DLM encoding in more detail, for the following reasons.

- The DLM is well-known and widely used in research.
- All information relabelings need to pass a dedicated *declassify* function. We show how this common pattern can be enforced with dynamic policies using the right encoding in SLIO (following an encapsulation technique similar to [8]).
- Although typically not supported by implementations, the DLM does contain dynamic features. More specifically, the DLM includes a hierarchy among principals which is subject to change, but these changes are *'assumed to occur infrequently'* [25]. Jif, an extension to Java with support for the DLM, relies on this assumption when verifying that applications are information-flow secure. By encoding the DLM in SLIO we can guarantee security even in the presence of hierarchy change.

*Remark* 3. Since its introduction by Myers and Liskov [26], various information-flow concepts have been added to the DLM, such as robust declassification [35] and information erasure [10]. We consider the DLM as used in the first iteration of the Jif compiler [25], matching most closely the model described in [27].

***The DLM Language*** In DLM, the security label $l_1 = \{o_1 : r_2, r_3; o_2 : r_3, r_4\}$ specifies that data is *owned* by the *principals* $o_1$ and $o_2$. Each owner specifies a different set of principals they allow to *read* this data. Effectively, the only principal that they both allow to read the data is $r_3$. The DLM includes an ordering among principals, the *acts-for* hierarchy $\succeq$. In a setting where principal $r_2 \succeq r_4$, label $l_1$ is equivalent to the label $l_2 = \{o_1 : r_2, r_3;\ o_2 : r_2, r_3, r_4\}$. That is, since $o_2$ allows $r_4$ to read the data, $o_2$ implictly allows $r_2$ as well. Labels $l_1$ and $l_2$ are also equivalent in a setting where $o_1 \succeq o_2$. That is, each principal that is allowed to read data by $o_1$ is implicitly also allowed to read that data by $o_2$. This hierarchy may be modified at run time.

The DLM assumes the existence of a *declassify* statement which makes the label of the provided data more permissive, either by extending an owner's reader-set or by removing an owner's concern entirely. Declassification is only permitted if the owners for whom information is declassified allowed for this by giving the computation their *authority*.

***Representing the DLM in SLIO*** The state component of the SLIO encoding of the DLM contains *i*) a boolean indicating whether or not declassification is allowed; *ii*) a set of principals who have given authority to the current computation; and *iii*) the set of principal pairs indicating the current hierarchy $\succeq$, similar to the hierarchy among $User$s in Section 3.1. The DLM encoding does not expose the **setState** operation from SLIO directly to user code. Helper functions are provided to change the hierarchy, and information can be declassified using the exposed $declassify$ function. The $declassify$ function uses the boolean element of the state as in Figure 3 to relabel the information, but only if such declassification is allowed at that moment. By means of this encapsulation, we can provide the necessary guarantees on declassifications, even in the presence of a changing acts-for hierarchy. A more detailed discussion of the DLM encoding can be found in the technical report [9].

## 8. Related work

Supporting dynamic policies is the next step in the natural evolution of security conditions from noninterference and declassification [15, 22, 29]. Balliu [5], Broberg and Sands [7] and Askarov and Chong [1] construct conditions for dynamic policies on top of the epistemic gradual release property, originally created to support declassification [4]. Delft et al. [12] show that epistemic properties can be unfolded into two-run properties, a technique we also use in the proof the soundness of our enforcement system.

A different approach to defining dynamic security policies can be traced back to the early work of Goguen and Meseguer on conditional noninterference [16], where noninterference relations on machine models only need to hold provided that some condition on the execution history holds. Zhang [37] expands on this, presenting a set of unwinding relations that can be verified by existing proof assistants.

The dynamic policies considered by SLIO are of a synchronous nature. That is, the policy changes deterministically with program execution. Other work considers asynchronous policies, such as Hicks et al. [17] and Swamy et al. [34]. Both approaches do require some synchronisation mechanism between the policy and the program execution.

Concerning IFC libraries for Haskell, the seminal work by Li and Zdancewic [20] consists in a library for enforcing information-flow security using arrows [19], a generalisation of monads. Russo et al. [28] show a monadic IFC security library, which statically enforces noninterference by leveraging Haskell's type system. Stefan et al. [33] propose LIO, which uses monads to track information-flow dynamically. Morgenstern et al. [24] encode an IF-aware programming language in Agda, without considering computations with side-effects. Devriese and Piessens [13] use monad transformers and parametrised monads to enforce noninterference. Unlike SLIO, none of the approaches mentioned above support dynamic policies or declassification in their semantic conditions, although for practical reasons some of them provide special constructs for declassification in their implementation.

Breeze is a programming language with IFC proposed by Hritcu et al. [18] which, like LIO, is based on the floating-label approach. In this system, lowering labels on values or the program counter (c.f. current label in LIO) is a privileged operation that requires special authority. Given the design similarities with LIO [33], we believe that our results could be easily adapted to Breeze.

## 9. Conclusions and Future Work

We have explored dynamic policies in a dynamic IFC setting by presenting SLIO, a strict generalisation of LIO with support for generic enforcement of dynamic policies. We have shown SLIO sound with respect to an epistemic security condition for dynamic policies with relabelings. We also demonstrated its practical use by encoding multiple policy frameworks which are available on [9] together with the SLIO library and the technical report version of this paper.

As future work, we intend to generalise the singular labels on labeled values and references to become *sets* of labels, thereby making them more homogenous with the rest of the enforcement. That is, like the current label set, these labels become elements in the power set lattice of security labels.

We also propose to examine extensions of SLIO with more advanced language-level features, such as concurreny and exceptions. Supporting concurreny appears to be particularly challenging, since it is not clear whether the policy changes performed in one thread can be made available to other threads while preserving soundness.

Finally, we remark that the library presented here could serve as a convenient testbed for future encodings of policy frameworks and comparing their relative expressive power.

***Acknowledgments*** This paper benefited from the comments of David Sands, Niklas Broberg and the anonymous reviewers. This work is partly funded by the Swedish funding agencies SSF and VR.

## References

[1] A. Askarov and S. Chong. Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security-ESORICS 2008*, pages 333–348. Springer, 2008.

[3] A. Askarov and A. Sabelfeld. *Security-typed languages for implementation of cryptographic protocols: A case study*. Springer, 2005.

[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 207–221. IEEE, 2007.

[5] M. Balliu, M. Dam, and G. Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 6. ACM, 2011.

[6] N. Broberg, B. v. Delft, and D. Sands. The Anatomy and Facets of Dynamic Policies. In *Computer Security Foundations*, 2015. To appear.

[7] N. Broberg and D. Sands. Paralocks – Role-Based Information Flow Control and Beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[8] N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 217–232. Springer, 2013.

[9] P. Buiras and B. van Delft. Dynamic Enforcement of Dynamic Policies - TR. `http://slio.bitbucket.org`. Accessed: 2014-10-18.

[10] S. Chong and A. C. Myers. Language-based information erasure. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 241–254, 2005.

[11] E. Cohen. Information transmission in computational systems. In *ACM SIGOPS Operating Systems Review*, volume 11, pages 133–139. ACM, 1977.

[12] B. v. Delft, S. Hunt, and D. Sands. Very Static Enforcement of Dynamic Policies. In *Principles of Security and Trust*. Springer, 2015.

[13] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[14] 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.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *2012 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE Computer Society, 1982.

[16] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *2012 IEEE Symposium on Security and Privacy*, pages 75–75. IEEE Computer Society, 1984.

[17] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, volume 20, 2005.

[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*, 2013.

[19] J. Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129. Springer, 2005.

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

[21] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 226–240. IEEE, 2005.

[22] D. McCullough. Noninterference and the composability of security properties. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 177–186. IEEE, 1988.

[23] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

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

[25] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[26] A. C. Myers and B. Liskov. *A decentralized model for information flow control*, volume 31. ACM, 1997.

[27] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 186–197, 1998.

[28] A. Russo, K. Claessen, and J. Hughes. A library for lightweight information-flow security in Haskell. pages 13–24. ACM Press, Sept. 2008.

[29] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. In *In Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFW'05*, pages 255–269, 2005.

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

[31] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.

[32] D. Stefan, A. Russo, D. Mazieres, and J. C. Mitchell. Disjunction category labels. In *Information Security Technology for Applications*, pages 223–239. Springer, 2012.

[33] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

[34] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 13–pp. IEEE, 2006.

[35] S. Zdancewic and A. C. Myers. Robust declassification. In *Computer Security Foundations Workshop, IEEE*, pages 15–15, 2001.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006.

[37] C. Zhang. Conditional information flow policies and unwinding relations. In *Trustworthy Global Computing*, pages 227–241. Springer, 2012.

## A. Proof for Theorem 1

For brevity, we only present an outline of the proof. For more details we refer to the technical report [9].

***Notation*** As additional notation for this appendix, we introduce $\varepsilon_A(\langle\Sigma|e\rangle)$ as a shorthand for $\langle\varepsilon^s_A(\Sigma)|\varepsilon^s_A(e)\rangle$ where $s = \Sigma.st$.

The theorem states that for all commands $e$, for all traces $t$ and configurations $\alpha$ such that $\langle\Sigma_0|e\rangle \Downarrow t\cdot\alpha$ with $obs_A(\alpha)$, $\alpha = \langle\Sigma_n|e_n\rangle$ and $\Sigma_n.st = s$,

$$ek_A(\varepsilon_A(t\cdot\alpha)) \setminus ek^+_A(\varepsilon_A(t)) \subseteq I_A(e,s) \cup R_A(t,s)$$

Following [12], we rewrite the set containment using logical connectives. Which, after some simplifications, gives us:

$$\begin{array}{ll}
\forall e, e', t, t'. & \\
\left.\begin{array}{ll} & \langle\Sigma_0|e\rangle \Downarrow t\cdot\langle\Sigma_n|e_n\rangle \\ \wedge & obs_A(\langle\Sigma_n|e_n\rangle) \\ \wedge & \Sigma_n.st = s \end{array}\right\} & \text{Set up} \\
\left.\begin{array}{ll} \wedge & \langle\Sigma_0|e'\rangle \Downarrow t'\cdot\langle\Sigma_m|e_m\rangle \\ \wedge & obs_A(\langle\Sigma_m|e_m\rangle) \\ \wedge & \varepsilon_A(t) = \varepsilon_A(t') \end{array}\right\} & ek^+_A(\varepsilon_A(t)) \\
\left.\begin{array}{ll} \wedge & \varepsilon^s_A(e) = \varepsilon^s_A(e') \end{array}\right\} & I_A(e,s) \\
\left.\begin{array}{ll} \wedge & \varepsilon_{\mathrm{L}}(t) = \varepsilon_{\mathrm{L}}(t') \end{array}\right\} & R_A(t,s) \\
\left.\begin{array}{ll} \Rightarrow & \varepsilon_A(\langle\Sigma_n|e_n\rangle) = \varepsilon_A(\langle\Sigma_m|e_m\rangle) \end{array}\right\} & ek(\varepsilon_A(t\cdot\alpha))
\end{array}$$

We show this by induction on $n$ in

$$\langle\Sigma_0|e\rangle \longrightarrow^* \langle\Sigma_{n-1}|e_{n-1}\rangle \longrightarrow \langle\Sigma_n|e_n\rangle$$

- **Case** $n = 0$**:** Trivial by $I_A(e,s)$.
- **Case** $n > 0$**:** We have:

$$\begin{array}{c}
\langle\Sigma_0|e\rangle \longrightarrow^* \langle\Sigma_{n-1}|e_{n-1}\rangle \longrightarrow \langle\Sigma_n|e_n\rangle \\
\langle\Sigma_0|e'\rangle \longrightarrow^* \langle\Sigma_{m-1}|e_{m-1}\rangle \longrightarrow \langle\Sigma_m|e_m\rangle
\end{array}$$

  By cases on the reduced expression in $e_{n-1}$:

  - **Case** $e_{n-1} = \mathbf{E}\ [\textbf{setState}\ s]$**:** By $incUpperSet$ and $ek^+_A(\varepsilon_A(t))$ we can conclude that $obs_A(\langle\Sigma_{n-1}|e_{n-1}\rangle)$ and $obs_A(\langle\Sigma_{m-1}|e_{m-1}\rangle)$. By $R_A(t,s)$ we then have that $\varepsilon_{\mathrm{L}}(\langle\Sigma_{n-1}|e_{n-1}\rangle) = \varepsilon_{\mathrm{L}}(\langle\Sigma_{m-1}|e_{m-1}\rangle)$, and hence $\varepsilon_A(\langle\Sigma_n|e_n\rangle) = \varepsilon_A(\langle\Sigma_m|e_m\rangle)$.
  - **Case** $e_{n-1} = \mathbf{E}\ [\textbf{toLabeledRet}\ ls\ s\ l\ v]$**:** By $obs_A(\langle\Sigma_n|e_n\rangle)$ and $ek^+_A(\varepsilon_A(t))$ we can conclude that both runs entered the **toLabeled** computation in an $A$-equivalent configuration. Hence we only need to show that the changed references and returned value $v$ are also $A$-equal.

    For the cases where the label on the reference (or value) $l \not\sqsubseteq_s A$ both configuration erase the value to $\bullet$. When $l \sqsubseteq_s A$, we have $l \in L$ and equality follows by $R_A(t,s)$.

- **All other cases:** For the reductions in which the policy state is unchanged, we have by $obs_A(\langle\Sigma_n|e_n\rangle)$ and $obs_A(\langle\Sigma_m|e_m\rangle)$ that also $obs_A(\langle\Sigma_{n-1}|e_{n-1}\rangle)$ and $obs_A(\langle\Sigma_{m-1}|e_{m-1}\rangle)$. By $ek_A^+(\varepsilon_A(t))$ we have that $\varepsilon_A(\langle\Sigma_{n-1}|e_{n-1}\rangle) = \varepsilon_A(\langle\Sigma_{m-1}|e_{m-1}\rangle)$. By the Fixed-State Lemma 1 (below), this gives us $\varepsilon_A(\langle\Sigma_n|e_n\rangle) = \varepsilon_A(\langle\Sigma_m|e_m\rangle)$.

**Lemma 1** (Fixed-State Lemma)**.** *Given two single-step evaluations $\langle\Sigma_1|e_1\rangle \longrightarrow \langle\Sigma_2|e_2\rangle$ and $\langle\Sigma_1'|e_1'\rangle \longrightarrow \langle\Sigma_2'|e_2'\rangle$ with $\Sigma_1.st = \Sigma_2.st$. For all levels A, if $\varepsilon_A(\langle\Sigma_1|e_1\rangle) = \varepsilon_A(\langle\Sigma_1'|e_1'\rangle)$ then $\varepsilon_A(\langle\Sigma_2|e_2\rangle) = \varepsilon_A(\langle\Sigma_2'|e_2'\rangle)$.*

*Proof.* See technical report [9]. □

## B. Other Relabeling Release Definitions

Our definition of Relabeling Release (Definition 4) is not arbitrary. In this appendix we list a number of different definitions that, although sounding reasonable at first glance, do not match our intuition of what is released via relabelings.

### B.1 Release knowledge by $A$ and $s$

Consider an attacker $A_s$ who also makes observations on level $A$, but pretending that the policy state was $s$ for the duration of the whole execution. To allow the attacker $A$ to learn information resulting from relabelings to levels $l \sqsubseteq_s A$, we share the knowledge that $A_s$ has gained so far:

$$R_A(t,s) = \{e' \mid \neg\exists t', \alpha'.\langle\Sigma_0|e'\rangle \Downarrow t'\cdot\alpha' \text{ with } obs_A(\alpha') \text{ and } \varepsilon_A^s(t) = \varepsilon_A^s(t')\}$$

Here, $\varepsilon_A^s(t)$ fixes the policy state $s$ to consider already at the level of the trace, ignoring the actual state in each configuration:

$$\varepsilon_A^s(\langle\Sigma|e\rangle\cdot t) = \begin{cases} \langle\varepsilon_A^s(\Sigma)|\varepsilon_A^s(e)\rangle\cdot\varepsilon_A^s(t) & \text{if } \Sigma.lset \sqsubseteq_s A \\ \varepsilon_A^s(t) & \text{otherwise}\end{cases}$$

Although this does release the relabeling information from our example program in Figure 9, it does not allow all relabelings that we would intuitively mark secure. As an example, consider the following program (in the dynamic policy *User* setting from § 3):

```
setState $[(\text{Bob},\text{Carl})]$
$\_ \leftarrow$ toLabeled Bob (do
  setState $[(\text{Alice},\text{Bob})]$
  $a \leftarrow$ readLIORef $aliceRef$
  writeLIORef $bobRef\ a$)
```

When returning from this **toLabeled** computation, Carl learns the information that is in *aliceRef* since the current state allows him to see Bob's data, thus *bobRef*. We would argue that this is secure, since Bob learns Alice's data in a state where this was allowed, and Carl in turn learns Bob's (and thereby Alice's data via relabeling) in a state where this is allowed. However, the suggested set $R_A(t,s)$ does not release the value of *aliceRef* to Carl.

Consider the observer Carl$_s$ who observes as if the policy state is always $[(\text{Bob},\text{Carl})]$. After the instruction **readLIORef** *aliceRef* the current label becomes $\{\text{Alice}\}$, meaning that this and all configuration to the end of the **toLabeled** computation are not visible to Carl$_s$. Hence, Carl$_s$ does not learn the value of *aliceRef* and this is therefore not released to Carl.

### B.2 Release knowledge by $A$, $s$ and $lset$

As a possible correction to the $A_s$ attacker, we could consider the $A_s^{lset}$ attacker who also makes observations on level $A$, but pretending that the policy state was $s$ *and* the current label set was $lset$ for the duration of the whole execution. Here $lset$ is the current label set when the new observation $\alpha$ was produced – i.e. this attacker fixes all the policy-relevant components. To allow the attacker $A$ to learn information resulting from relabelings to levels $l \sqsubseteq_s A$, we share the knowledge that $A_s^{lset}$ has gained so far:

$$R_A(t,s) = \{e' \mid \neg\exists t', \alpha'.\langle\Sigma_0|e'\rangle \Downarrow t'\cdot\alpha' \text{ with } obs_A(\alpha') \text{ and } \varepsilon_A^{s,lset}(t) = \varepsilon_A^{s,lset}(t')\}$$

Here, $\varepsilon_A^{s,lset}(t)$ fixes the policy state $s$ amd the current label set, ignoring their actual values in each configuration. Hence since $obs_A(\alpha)$, all previous configurations are observable:

$$\varepsilon_A^{s,lset}(\langle\Sigma|e\rangle\cdot t) = \langle\varepsilon_A^s(\Sigma)|\varepsilon_A^s(e)\rangle\cdot\varepsilon_A^{s,lset}(t)$$

This indeed allows the secure program from § B.1, but also labels the following program secure, which we argued in § 4 to be clearly insecure due to conditionally allowing the flow from *High* to *Low*:

```
$leak\ highData =$ do
  $r \leftarrow$ newLIORef $Low$ 1
  $\_ \leftarrow$ toLabeled $High$ (do
    $h \leftarrow$ unlabel $highData$
    when $(h \equiv 0)$ (do
      setState $True$
      writeLIORef $r$ 0
    $v \leftarrow$ readLIORef $r$))
  return $v$
```

Although the $Low_s^{lset}$ attacker does not observe the value of $h$ when it is not 0, still this attacker learns that since the next expression to reduce after **unlabel** is **toLabeledRet**, that the value of $h$ was not 0. This is exactly the information leaked to *Low*, so this release policy allows for that leak.

### B.3 Release knowledge by all $l \sqsubseteq_s A$

Finally we consider one definition that is close to the one we selected. Rather than defining the multi-level erasure function $\varepsilon_{\mathrm{L}}(\cdot)$, we could say that we release the knowledge for each level $l \sqsubseteq_s A$ individually:

$$R_A(t, s) = \{e' \mid \neg\exists t', \alpha'.\langle\Sigma_0|e'\rangle \Downarrow t' \cdot \alpha' \text{ with } obs_A(\alpha') \\ \text{and } \varepsilon_l(t) = \varepsilon_l(t') \text{ for all } l \sqsubseteq_s A\}$$

This does disallow the leak via policy state change and it allows for the relabel examples shown in this paper so far. However, it does not consider the following program secure, which we would intuitively label as such:

```
setState [ ]
$one$ ← label Alice 1
$two$ ← label Alice 2
$bobData$ ← toLabeled Bob (do
   setState [(Carl, Bob)]
   $d$ ← unlabel $carlData$
   return (if $d$ then $one$ else $two$))
$daveData$ ← toLabeled Dave (do
   setState [(Bob, Dave)]
   $d$ ← unlabel $bobData$
   return $d$)
setState [(Alice, Dave)]
```

Returning from the second **toLabeled** command, information from Alice is not allowed to flow to Dave, so the value of *daveData* as observed by Dave is (`Lb` Dave (`Lb` Alice •)). After the last **setState** command Dave learns that the value at • was either 1 or 2, and from that gains knowledge about the value in *carlData*. The problem with the suggested definition is that it does allow Dave to learn this information, but not at the right point in the execution!

When unlabeling *bobData*, information may flow from Bob to Dave. Hence, at this point $R_A(t, s)$ allows Dave to learn what Bob has learned, which include the earlier observation of the value in *carlData*. However, Dave only sees (`Lb` Alice •) and does not learn this information yet. When Dave does learn the information, the state only allow information from Alice to flow to Dave. Alice has not been able to observe any configuration where *carlData* was unlabeled, so sharing Alice's knowledge with Dave does not allow Dave to learn anything about *carlData*.

The final definition for $R_A(t, s)$ given in Definition 4 resolves this by combining the observations from all levels $l \sqsubseteq_s A$ at each point. With $\mathrm{L} = \{\text{Alice}, \text{Dave}\}$ the projection $\varepsilon_{\mathrm{L}}(t)$ contains **unlabel** *bobData* which releases whether the value labeled with Alice is 1 or 2, as we desired.