Liquid Information Flow Control

NADIA POLIKARPOVA, University of California, San Diego
DEIAN STEFAN, University of California, San Diego
JEAN YANG, Carnegie Mellon University
SHACHAR ITZHAKY, Technion
TRAVIS HANCE, Carnegie Mellon University
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology

We present Lifty, a domain-specific language for data-centric applications that manipulate sensitive data. A Lifty programmer annotates the sources of sensitive data with declarative security policies, and the language statically and automatically verifies that the application handles the data according to the policies. Moreover, if verification fails, Lifty suggests a provably correct repair, thereby easing the programmer burden of implementing policy enforcing code throughout the application.

The main insight behind Lifty is to encode information flow control using liquid types, an expressive yet decidable type system. Liquid types enable fully automatic checking of complex, data dependent policies, and power our repair mechanism via type-driven error localization and patch synthesis. Our experience using Lifty to implement three case studies from the literature shows that (1) the Lifty policy language is sufficiently expressive to specify many real-world policies, (2) the Lifty type checker is able to verify secure programs and find leaks in insecure programs quickly, and (3) even if the programmer leaves out all policy enforcing code, the Lifty repair engine is able to patch all leaks automatically within a reasonable time.

ACM Reference Format:

1 INTRODUCTION

Modern applications handle sensitive user data in complex ways, subject to increasingly complex security policies. For example, social networks like Twitter and Facebook must ensure that they handle user data according to GDPR, health record systems (e.g., the Dexcom diabetes management system) must abide by HIPAA, and financial applications like Stripe and Mint must ensure they are PCI compliant. In most cases, these applications even allow users to restrict who can access their data—e.g., on Facebook a user can restrict access to (part of) their profile to their friends. Unfortunately, many applications specify and enforce these policies by strewing checks throughout application code—an error prone process that has lead to many inadvertent data leaks [Cimpanu 2020; Doctorow 2015; Hunt 2020; Privacy Rights Clearinghouse 2020].

A promising approach to tackling this challenge is to use web frameworks like Hails and Jacqueline which separate the security policy from the application code and enforce the policy using dynamic information flow control (IFC) [Giffin et al. 2012; Yang et al. 2016a]. In such IFC frameworks, the programmer declaratively specifies expressive data-dependent policies; the language runtime—or in the case of Hails, the LIO monad [Stefan et al. 2017, 2011b]—then automatically enforces these policies to prevent leaks (e.g., by throwing an exception or replacing sensitive values with defaults).
Unfortunately, enforcing policies dynamically as in Hails and Jacqueline has inherent limitations. First, the IFC systems often perform redundant checks. In many applications, developers already insert checks in the application code to, for example, implement the user interface; the IFC systems, unfortunately, do not know about these checks and will perform similar checks when enforcing the policy. These checks impose unnecessary performance overheads, i.e., they tax the application latency a second time. Second, errors due to policy violations only manifest at runtime: the programmer doesn’t know if their policy is too strict until their application crashes at runtime.

Static IFC systems (e.g., [Buiras et al. 2015; Chlipala 2010; Jia and Zdancewic 2009; Li and Zdancewic 2005; Myers 1999; Russo et al. 2008a; Swamy et al. 2010; Zheng and Myers 2007]) precisely address these limitations: they do not impose unnecessary runtime checks and catch errors early—at compile time. Unfortunately existing static IFC systems either lack support for expressive data-dependent policies (necessary to modern applications), or they require manual proofs or annotations to be strewed thought the application code.

In this paper, we take the best from both: we present a static IFC system that automatically—without manual proofs or annotations—enforces Hails-like expressive, declarative high-level policies.

The Lifty Language. Our first insight is that we can encode static IFC into the framework of liquid types [Rondon et al. 2008; Vazou et al. 2013, 2014b], an expressive yet decidable type system. To this end, we use predicates from decidable logics to directly specify expressive policies and adopt security monads [Li and Zdancewic 2006; Russo et al. 2008a; Vassena and Russo 2016] to enforce these policies. We do this in Lifty—short for Liquid Information Flow TYpes—a core domain-specific language (DSL) for writing secure data-centric applications. With Lifty, programmers (1) write code in our custom IO monad called TIO and (2) specify policies in a decidable logic when declaring sources of sensitive data. The Lifty type-checker uses liquid types to verify the program against the policies and flags any unsafe access to sensitive data as a type error.

Leak Repair. By taking a static approach to enforcing information flow, Lifty can also help programmers repair their unsafe code. Our insight here is to use type-errors to localize the source of each leak and suggest a best-effort leak patch (Fig. 1). The suggested patches guard the unsafe access with a policy check and, for the failure case, implement a safe escape—e.g., they return a default value. The key to the efficiency of our repair technique is a new, domain-specific error localization mechanism that relies on the Lifty type-checker to infer an expected policy for each unsafe access. While efficient, this approach is necessarily limited: although the patch is guaranteed to fix the leak, the generated policy check might be conservative, or the repair attempt might fail (e.g. when it cannot find a safe escape). Nevertheless, we find this best-effort still useful in practice.
Evaluation. To demonstrate the practical promise of our approach, we implemented a prototype LIFTY-to-Haskell compiler. We evaluated our implementation on a series of small, but representative micro-benchmarks, as well as three case studies: a conference manager, a health portal, and a student grade record system. The evaluation demonstrates that our solution supports expressive policies, reduces the annotation burden placed on the programmer, and is able to generate all necessary patches for our benchmarks within a reasonable time (within a minute for each of our case studies). Importantly, the evaluation confirms that the patch synthesis time scales linearly with the size of the source code (more precisely, with the number of required leak patches), suggesting the feasibility of applying this technique to real-world code bases.

Contributions. In summary, we make the following contributions:

1. The TIO Monad: an encoding of static IFC into liquid types, that supports fully automatic verification of expressive policies.
2. Leak repair: we combine novel policy abduction with type-driven synthesis to generate best-effort patches for the leaks.
3. Prototype implementation: we implement a prototype of LIFTY with a type checker and a compiler to Haskell and evaluate our system on several micro-benchmarks and case studies.

2 MOTIVATING EXAMPLE

We motivate LIFTY using an example based on a leak from the EDAS conference manager [Agrawal and Bonakdarpour 2016]. In this section, we describe the anatomy of this leak, show how LIFTY can detect it at compile-time given a declarative security policy, and demonstrate how our tool automatically synthesizes a patch for this leak. The core technical innovation that enables automatic verification and synthesis—the LIFTY type system—is introduced in Sec. 3, together with more advanced examples that demonstrate the flexibility of our language.

2.1 The EDAS Leak

Fig. 2 shows a screenshot of the EDAS conference manager home screen. On the home screen, users are presented with an overview of all their submitted papers (both old and new). Color coding indicates PC decisions: green papers have been accepted, orange have been rejected, and yellow papers are awaiting notification. As usual, users are not supposed to learn the acceptance decision of their papers before the notifications are out. But, the site leaks: in the figure, we can infer that the first one of the yellow papers has been tentatively accepted, while the second one has been tentatively rejected. We can make this conclusion because the two rows differ in the value of the "Session" column—and sessions are only displayed for accepted papers.

This leak is particularly insidious—indeed, it’s an example of an implicit flow: the “accepted” decision does not appear anywhere on the screen, but it does conditionally influence the output of the web application. To prevent such leaks, it is insufficient to simply examine output values; we must track the flow of sensitive information throughout the system.

Fig. 3 shows a simplified version of the leaky code (in LIFTY syntax) that is used to display the description of individual papers. This code retrieves the title and decision for paper \( p \) and, if the paper has been accepted, it retrieves the session where the paper will be presented and displays it to \( \text{client} \) together with the title; otherwise, it only displays the title. In deciding to display the session (or not), this code indirectly leaks the decision to \( \text{client} \).

The easiest way to fix this leak is to check if the conference is in an appropriate phase (reviewing is done), and only then display the session. But, even for a simple example like EDAS, we must

---

1The \( ds \) parameter models the state of the data store; it is only used for specification purposes and is threaded through LIFTY programs explicitly for simplicity.
showPaper ds client p = do
  t ← getTitle ds p
  dec ← getDecision ds p
  if dec = Accept
    then ses ← getSession ds p
    print client (t + " + ses)
  else print client t

Fig. 3. Snippet of the core functionality of a conference manager (in Lifty syntax).

Fig. 4. With a leak patch inserted by Lifty.

strew in such poly-enforcing code in the dozen of web request handler that access papers, reviews, etc. Real web applications handle lots of sensitive data and have hundreds to thousands of such handlers; getting all the right checks in all the right places is notoriously hard.

2.2 Programming with Lifty

In Lifty, the programmer explicitly associates sensitive data with declarative policies by annotating input actions that retrieve data from the store with appropriate types. For example, to specify that PC decisions are visible to everyone once the conference phase is Done but not otherwise, the action getDecision in the EDAS example can be annotated with the type:

\[
\text{getDecision} :: \text{ds: Store} \to \text{p: PaperId} \to \text{TIO Decision} < \lambda u \cdot \text{phase ds = Done}, \lambda u \cdot \text{false}>
\]

The TIO type constructor is indexed by two security labels: the first label specifies the set of users who are allowed to see the result of this action; we postpone the explanation of the second label to Sec. 3. Labels can be expressive, value-dependent predicates as in this example—here, the label depends on the state of the data store ds—or simple predicates (e.g., input actions for public fields “title” and “session” are labeled \(\lambda u \cdot \text{true}\)).

Given such policy-annotated actions, Lifty would statically reject the code in Fig. 3 with a type error: the value of dec obtained on line 3 is flowing to client, but is not visible to client in the current state ds. Moreover, Lifty would suggest a patch for this leak, as shown in Fig. 4. The fixed code guards the access to the sensitive field “decision” with a policy check, and if the check fails, it substitutes the true value of decision with a default value—a constant NoDecision. Lifty, in turn, guarantees that the patched code respects the declared policies.

To our knowledge, only two other information-flow control tools support similarly expressive declarative policies. The first is LIO, as used in the Hails web framework [Giffin et al. 2017, 2012], or the LWeb framework [Parker et al. 2019]. LIO does not check policies statically. Hence, the equivalent code of Fig. 3 would compile without errors. Instead, at run time, LIO would throw an exception on either line 7 or 8, when trying to execute the print action (after inspecting the decision). Though this successfully prevents the leak, it also means the programmer won’t find out that their application is broken until run time.

The second tool is Jeeves as used in Jaqueline framework [Yang et al. 2016a]. As with LIO, the equivalent Jeeves code of Fig. 3 would compile without errors. The runtime behavior of the Jeeves code, however, is identical to the version patched by Lifty, i.e. the value of dec will be replaced by a default whenever the conference is not Done. Unlike Lifty though, Jeeves achieves this using faceted execution [Yang et al. 2012], i.e., by performing the computation on multiple versions of
3 OVERVIEW

In this section we first describe how LIFTY uses liquid types to enforce static information flow control. We then explain how to use this IFC mechanism to implement secure data-centric applications with complex, data-dependent policies. Next, we give the intuition for how LIFTY’s type-driven repair engine generates leak patches for these application (the details of the repair algorithm can be found in Sec. 5). We wrap up the section with more advanced examples from the conference manager.

3.1 Static IFC with TIO

Liquid types. LIFTY builds upon a pure functional language with liquid types [Rondon et al. 2008; Vazou et al. 2013]—types decorated with predicates from SMT-decidable logics. For example, we can define the type of natural numbers as type Nat = {Int | 0 ≤ ν}, where ν is a reserved value variable, which ranges over the values of the refined type. State-of-the-art liquid type systems feature subtyping (e.g. Nat <: Int), as well as type constructors that can be indexed by both types and refinement predicates. For example, we can define a type constructor List α <p: α → α → Bool> for lists with elements of type α, where each in-order pair of elements satisfies a binary predicate p; then the type List Nat <λ x y . x ≤ y> denotes a sorted list of natural numbers. Concretely our implementation builds upon the SYNQUID language [Polikarpova et al. 2016], but a similar technique could be used to add static IFC to LIQUIDHASKELL [Vazou et al. 2014a].

Security lattice. Like almost all IFC systems, LIFTY uses a security lattice to distinguish between data with different levels of confidentiality. We, however, fix the security lattice to be the lattice of refinement-logic predicates over principals. More precisely, a security level ℓ : User → Bool denotes information visible to users that satisfy ℓ. For example, a key shared between Alice and Bob has a security level λu. u ∈ [Alice, Bob]. In the rest of the paper we will use Scala-like syntactic sugar for lambda terms and write _ ∈ [Alice, Bob]. Note that the usual “can-flow-to” partial order in our security lattice corresponds to reverse implication: l ⊑ h iff ∀u.h u ⇒ l u, and hence the bottom (least secret) level is ⊥ = true and the top (most secret) level is ⊤ = false.

Our label model can be seen as a generalization of DCLabels [Stefan et al. 2011a] and is thus at least as expressive [Montagu et al. 2013]. Without loss of generality, we however focus only on secrecy.

The TIO monad. To extend the base language with static IFC, we follow a long line of work on security monads [Li and Zdancewic 2006; Russo 2015; Russo et al. 2008a; Vassena and Russo 2016]—in particular SLIO [Buiras et al. 2015]—where sensitive computations are wrapped in a special datatype indexed by a security label. Proper assignment of labels and their propagation through the program are ensured by the types of primitives in the API of the security monad.

More specifically, LIFTY introduces the type constructor TIO ("tagged input-output"), indexed by a return type and two refinement predicates, i, o : User → Bool, which we refer to as the input and the output label. The type TIO T (i, o) denotes a secure computation that may read from resources at security level i (and below) and may write to resources at security level o (and above). For example, a computation getSharedKey that reads a key shared between Alice and Bob can have the type

For simplicity, we use a concrete type User to denote principals; our approach also supports leaving this type as a parameter.
The TIO constructor is **covariant** in the input label and **contravariant** in the output label wrt. the label ordering, i.e. $\text{TIO} T \langle i_1, o_1 \rangle <: \text{TIO} T \langle i_2, o_2 \rangle$ iff $i_1 \subseteq i_2$ and $o_2 \subseteq o_1$. For example, we can pass the `getSharedKey` computation from above to a function $f$ with argument $x$: $\text{TIO} \text{String} \langle \_ = \text{alice}, \_ = \text{false} \rangle$. Intuitively this is safe, because $f$ promises to reveal the result of $x$ only to $\text{alice}$ (while in fact the shared key can be safely revealed to both $\text{alice}$ and $\text{bob}$), and counts on $x$ to output only to $\text{alice}$ (while in fact `getSharedKey` does not perform any user-visible output). Because label ordering reduces to implication between refinement predicates, the subtyping between TIO types can be automatically decided by the base language type checker (with the help of an SMT solver).

### Input/output actions
LIFTY programmers identify sources and sinks of sensitive information by providing a set of domain-specific atomic input and output actions, annotated with labels that are assumed to be correct. Fig. 5 gives an example set of atomic actions that includes reading the shared key of users $\text{alice}$ and $\text{bob}$, reading a user’s social-security number (which is visible only to the user themselves), and printing a string to a given user. T1 and T0 are type synonyms for TIO computations that perform only input or only output, respectively (see Fig. 6).

Similar to other IFC systems (e.g., LIO and SLIO), we leave LIFTY agnostic to the particular choice of input/output actions; depending on the domain, actions can be used to model reading and writing to mutable memory, file system, or database, as well as HTTP responses, sending emails, etc.

### TIO Primitives
Client code manipulates TIO computations through an API shown in Fig. 6. The API consists of three core primitives; together with the language runtime and type checker, these form the trusted computing base of LIFTY. We also expose several auxiliary functions that are built on top of the primitives and verified using the type checker.

The first core primitive, `return`, simply embeds a pure value into a sensitive computation, and hence has the strongest possible labels: input label $\bot$ and output label $\top$. The `bind` primitive (the analogue of >>*= in Haskell) sequences two sensitive computations, $a$ and $b$, such that $a$’s result flows into $b$. To prevent leaks, we need to guarantee that $a$’s input label can flow to $b$’s output label.
1 ok1 = 1 1 bad2 = 1 ok2 =
2 do 2 do 2 do b ← getSSN bob 2 do do b ← getSSN bob
3 k ← getSharedKey 3 b ← getSSN bob 3 print bob b 3 print bob b
4 print bob k 4 print bob b 4 a ← getSSN alice 4 a ← getSSN alice
5 print alice k 5 print alice b 5 print alice a 5 print alice a

Fig. 7. Examples of well-typed (ok1, ok2) and ill-typed (bad1, bad2) TIO computations.

To enforce this, we simply add $a$’s input label $i$ as a conjunction to $b$’s output label. Moreover, we set the input label of bind $a b$ to the join (conjunction) of the input labels of $a$ and $b$, and its output label to the meet (disjunction) of the output labels of the two. Finally, the downgrade primitive supports safe declassification of boolean terms; we postpone its presentation to the end of this section.

**Auxiliary TIO API functions.** Often a Lifty program sequences two computations $a$ and $b$, but no information is actually flowing from $a$ to $b$. To relax the restrictions imposed by bind in this case, we provide an API function seq (the analogue of >> in Haskell). The type of seq does not enforce any relationship between the labels of $a$ and $b$. Note that seq is not a primitive, and can be implemented using bind and downgrade.

The rest of the TIO API contains monadic combinators such as liftM (for lifting a pure function into TIO), mapM (for mapping a sensitive computation over a list), as well as filterM and sortByM (for filtering and sorting based on a sensitive criteria). All auxiliary functions are implemented in terms of the three primitives and type-checked automatically by Lifty in less than a second.

**Examples.** To gain some intuition about our IFC encoding, consider simple examples of TIO computations in Fig. 7. Lifty’s surface syntax supports Haskell-like do-notation [Marlow 2010], which desugars into invocations of bind and seq in a standard way. For example, ok1 is desugared into bind getSharedKey $(\lambda k . \text{seq} \ (\text{print} \ bob \ k) \ (\text{print} \ alice \ k))$. Note that the desugaring uses seq (with its more permissive type) instead of bind whenever the return value of a line is not bound.

These snippets use atomic actions defined in Fig. 5. The snippet ok1 is well-typed in Lifty because the output label of the sequence of two print actions, $\bot = bob \lor \bot = alice$, implies the input label of getSharedKey, $\bot \in [alice, bob]$. More precisely, we can instantiate the type of bind with $i \mapsto \bot \in [alice, bob]$, $j \mapsto \text{true}$, and $o \mapsto \text{false}$. On the other hand, the snippet bad1 is ill-typed because the implication no longer holds—indeed, this would be leaking bob’s SSN to alice.

Perhaps surprisingly, the snippet bad2 is also ill-typed: we cannot bind getSSN bob to the rest of the computation, whose output label permits output to alice. The code does not actually leak b to alice. Instead, Lifty flags this snippet because our information flow tracking is coarse-grained [Buiras et al. 2015; Stefan et al. 2017; Vassena and Russo 2016], and restricts outputs based on all the data in scope; in particular, the output to alice on line 5 should be rejected because b, which is not visible to alice, is still in scope on line 5.

This coarse-grained approach to IFC is simpler, but as expressive as fine-grained IFC [Rajani and Garg 2020]. We simply need to tweak the code to express the desired program: ok2 performs the same actions in the same order, but with a different binding structure. Here lines 2 and 3 handling bob’s data are grouped into one TIO computation, while lines 4 and 5 handling alice’s data are grouped into another TIO computation; the two actions are then sequenced with seq, which lets the type checker know that no information flows between the two, leading to a well-typed program.

**Type checking** The Lifty type checker is based on the Liquid Types inference framework [Cosman and Jhala 2017; Rondon et al. 2008]. To type-check a TIO computation, it uses the types of the API functions to generate a system of subtyping constraints over TIO types. It then relies on the definition...
of co- and contravariant subtyping to reduce subtyping constraints to a system of constrained Horn clauses (CHCs), i.e. implications between (possibly unknown) refinement predicates. For example, the snippet ok1 from Fig. 7 generates the following CHCs (trivial constraints are omitted):

\[
\begin{align*}
I \land \neg (u \in \{ \text{alice}, \text{bob} \}) & \Rightarrow \text{false} \\
(O \lor P) & \Rightarrow I \\
u = \text{alice} & \Rightarrow P \\
u = \text{bob} & \Rightarrow O
\end{align*}
\]

Here \(I, O, P\) are unknown predicates over a variable \(u: \text{User}\), and stand for the instantiations of indexes \(i\) in bind on line 3 and \(o\) and \(p\) in seq on line 4. Specifically, constraint (1) relates bind to getSharedKey, constraint (2) relates the output label of the seq computation to the input label of the left-hand-side of bind, while the last two constraints relate the output labels of print actions to the output label of seq. The left- and right-hand sides of the implication are called the body and the head of a CHC, respectively. CHCs can be divided into rules (head is an unknown, like 2–4) and queries (head is false, like 1). In this case, the CHCs are non-recursive, and hence can be solved by simple left-to-right unfolding of the rules: from (3) and (4) we infer the strongest assignment to \(O\) and \(P\):

\[
[O \leftarrow \text{bob}, P \leftarrow \text{alice}]
\]

In this case, the CHCs are non-recursive, and hence can be solved by simple left-to-right unfolding of the rules: from (3) and (4) we infer the strongest assignment to \(O\) and \(P\): \([O \leftarrow \text{bob}, P \leftarrow \text{alice}];\) substituting this into (2), we similarly infer \([I \leftarrow (u = \text{alice} \lor u = \text{bob})]\); finally, with this assignment the query (1) is valid, hence this assignment is a (strongest) solution. Recursive CHCs are a bit more involved, but we can still find the strongest solution using a combination of unfolding and predicate abstraction, as shown in [Cosman and Jhala 2017].

**Safe downgrading.** We leverage the power of refinement types to support safe, i.e. non-leaky, declassification of boolean terms. We do this with the `downgrade` primitive. The intuition behind `downgrade` \(t\) is that the input label of \(t\) can be safely lowered as long as we can prove that in all relevant executions \(t\) always returns the same value (in particular, the value \(\text{false}\)), because constants cannot leak information. Consider the following term:

\[
downgrade (\text{bind} (\text{getSSN} \ u) (\lambda s. \text{return valid}(s) \land u = \text{alice}))
\]

In LIFTY, this term type-checks against the type \(\text{Tl} \text{Bool} \langle \_ = \text{alice} \rangle\) even though it performs an input action of an incompatible type \(\text{Tl} \text{Bool} \langle \_ = u \rangle\). Intuitively, this is safe because in all executions where the two input labels are indeed incompatible, it must be that \(u \neq \text{alice}\), and hence the result of the computation is always \(\text{false}\). LIFTY performs this reasoning automatically by instantiating the type of `downgrade` shown in Fig. 6 with \(i \leftarrow \_ = \text{alice}, c \leftarrow u = \text{alice}\). Note that this type would no longer work if we remove the conjunct \(u = \text{alice}\) from the term.

Safe downgrading in LIFTY is restricted to boolean terms, which lets us rely on existing machinery of liquid type inference to discover all intermediate labels completely automatically. Although such a restrictive mechanism might not appear very useful at first, it turns out to be indispensable for supporting applications with complex data-dependent policies, as we demonstrate in Sec. 3.4. Finally, as we mentioned above, `downgrade` can be used to implement `seq`:

\[
\text{seq} \ a \ b = \text{bind} \ (\text{downgrade} \ (\text{bind} \ a (\lambda _. \text{return} \text{false}))) (\lambda _. \ b)
\]

### 3.2 Encoding Policies in Data-Centric Applications

The TIO monad is particularly suitable for enforcing secure information flow in data-centric web applications (such as a conference manager or a social network). Such applications are built around a data store, where different fields have different visibility policies, which might depend on the data itself. Web application are typically structured as a set of controllers or request handlers, i.e.
functions called by a user request that read data from the store, process it and then respond to the user. A LIFTY programmer can encode store reads and writes as atomic input and output actions, and responses as output actions. They can directly express data-dependent policies as types, instead of translating them into an intermediate security lattice.

**Conference manager.** Let us revisit the conference manager example from Sec. 2 and demonstrate how a LIFTY programmer would specify its data-dependent policies. Fig. 8 shows an excerpt from the data store API for this system. To encode the policies, we introduce an uninterpreted type Store, which models the state of the data store. Next, for each field of the store we introduce a measure, i.e., an uninterpreted function that models the value of the field. Note that both of these are specification-only constructs, introduced solely for the purpose of expressing policies.

The actual program-level API of the data store contains a getter and a setter for each field of the store, encoded as atomic TIO actions. The programmer, however, can use the uninterpreted measures to relate the return values of the getters to the labels of the actions. For example, in Fig. 8, we use the measure phase both in the return type of getPhase and in the input label of getDecision (resp. output label of setDecision), thereby relating the two actions.

The reason LIFTY considers the EDAS leak example from Fig. 3 ill-typed is now clear: the input action getDecision ds p on line 3 has the input label \( \lambda u.\text{phase} \) \( ds = \text{Done} \), but this action is bound to a computation with the output label \( \lambda u.u = \text{client} \). For this occurrence of bind to be well-typed we need to prove that \( \forall u.u = \text{client} \Rightarrow \text{phase} \) \( ds = \text{Done} \), which is not valid.

On the other hand, the patched code in Fig. 4 is well-typed. To understand why, note that due to the polymorphic type of bind, the type of the binder x1 on line 3 is \( \{\text{Phase} \mid \nu = \text{phase} \) \( ds \} \). Hence the input action getDecision ds p is now being type-checked under the assumption \( \text{Done} = \text{phase} \) \( ds \), which makes the above implication valid.

### 3.3 Patching the Leaks

How does LIFTY patch the leak in Fig. 3? Intuitively, our goal is to eliminate the type error in this program by breaking the insecure flow from the input action getDecision ds p on line 3 into the output actions print client on lines 7 and 8. There are, of course, many ways to break this flow, and
in the absence of a complete functional specification for showSession, LIFTY cannot be sure what the programmer indented. However, in the domain of data-centric applications there is a reasonable default: LIFTY can guard the offending input action with a policy check, and redact the sensitive value whenever the policy check fails. LIFTY borrows this repair strategy from Jeeves [Yang et al. 2016a, 2012], which enforces policies by replacing secret values with public defaults. Note, however, that a LIFTY programmer does not have to use this strategy: because repair happens statically, they can inspect the suggested patch and if necessary replace it with a manual fix.

LIFTY implements this general leak repair strategy in three steps: (1) localize leaky input actions, (2) for each such input action, infer the expected type of the patch (i.e. the weakest type that would eliminate the type error), and (3) generate a term of the expected type by filling a domain-specific template. We describe these steps in the rest of this section.

**Localizing leaks.** Type-checking the code in Fig. 3 against with the policies in Fig. 8 generates the following (simplified) system of CHCs:

\[
\begin{align*}
I \land \text{phase ds} = \text{Done} & \Rightarrow \text{false} \\
O & \Rightarrow I \\
\text{u} = \text{client} & \Rightarrow O
\end{align*}
\]

This system clearly has no solution; in particular, unfolding the rules (6)–(7) gives the strongest assignment \( I \mapsto \text{u} = \text{client} \), but this assignment does not validate the query (5). Our insight is that each such invalid query corresponds to an atomic input action whose input label is too high for the output action it is flowing to. Using this insight, the LIFTY type checker can identify all leaky input actions at the same time, with just a little extra bookkeeping—namely, tracking which term generate which CHC. In this example, the query (5) is generated by \text{getDecision ds p} , so this is the action we need to guard.

**Inferring the expected type.** From the same CHCs we can infer not only the offending term, but also the highest label a replacement term can have for the program to type-check. We obtain this label from the strongest assignment computed from the rule clauses. In our example, the strongest assignment has \( I \mapsto \text{u} = \text{client} \), hence replacing \text{getDecision ds p} with any term of type \text{TI Decision} \( \_ = \text{client} \) would fix the leak. We refer to this type as the expected type of the patch.

**Synthesizing the patch.** Even though any term of the expected type is secure, not all solutions are equally desirable: for example, we wouldn’t want the patch to return \text{Accept} unconditionally. Intuitively, a desirable solution returns the original value whenever it is safe, and otherwise replaces it with a reasonable redacted value. LIFTY achieves this through a combination of two measures. First, instead of synthesizing a single term of type \text{TI Decision} \( \_ = \text{client} \), it generates a set of candidate branches (by enumerating all branch-free terms of this type up to a fixed size). Second, LIFTY gives the programmer control over the set of possible redacted values by generating the branches in a restricted environment, which only contains the original term and the components explicitly designated by the programmer as redaction functions. When defining a new datatype, the programmer is expected to designate one or more constructors (or functions) of this type as redactions (Fig. 8 shows an example for type \text{Decision}). As a result, our running example generates only two branches:

\[
\begin{align*}
\text{getDecision ds p} & :: \text{TI Decision} \langle \text{phase ds} = \text{Done} \rangle \\
\text{return NoDecision} & :: \text{TI Decision} \langle \text{true} \rangle
\end{align*}
\]

Next, for every branch, LIFTY attempts to abduce a condition that would make the branch type-check against the expected type. In our example, the second branch is correct unconditionally, while the first branch generates the following logical abduction problem: \( C \land \text{u} = \text{client} \Rightarrow \text{phase ds} = \)
showMyPapers ds client = let include p = do auts ← getAuthors ds p return (elem client auts) in do papers ← filterM include allPapers titles ← mapM (getTitle ds) papers print client (unlines titles)

Fig. 9. A controller that displays all client’s papers (well-typed).

showMyAccepts ds client = let include p = do auts ← getAuthors ds p dec ← getDecision ds p return (elem client auts \& dec = Accepted) in do papers ← filterM include allPapers titles ← mapM (getTitle ds) papers print client (unlines titles)

Fig. 10. A controller that displays all client’s accepted papers (ill-typed).

\( \text{Done} \), where \( C \) is an unknown formula over only the program variables, \( i.e. \) it cannot mention the user variable \( u \). LIFTY uses existing techniques [Polikarpova et al. 2016] to find the following solution to the abduction problem: \( C \mapsto \text{phase \, ds} = \text{Done} \). It then sorts all successfully abduced branch conditions from strongest to weakest, and uses each condition to synthesize a guard, \( i.e. \) a program that computes the monadic version of the condition. In our case, the guard for the first branch is \( \text{bind \, getPhase \, \lambda x1 \cdot x1 = \text{Done}} \). Finally, LIFTY combines the synthesized guards and branches into a single conditional, which becomes the patch and replaces the original offending input action.

3.4 Advanced Policies

We conclude the overview of LIFTY with another example from the conference manager, which illustrates the kinds of policies we need to realistically express.

**Self-referential policies.** Consider the action getAuthors in Fig. 8 that retrieves the author list of a given submission. Assuming that our conference is double-blind, we would like to enforce a policy that the author list is only visible to the authors themselves until the reviewing is done (and afterwards visible to everyone). Unlike the policy on field “decision”, which depends on a public field “phase”, this is an example of a policy that depends on a sensitive field; moreover, in this case the policy is self-referential: it guards access to “authors” in a way that depends on the value of “authors”. By separating between measures and input/output actions, LIFTY makes it easy to express such self-referential policies: the programmer simply uses the authors measure in both the return type and the label of getAuthors.

**Checking self-referential policies via downgrading.** Consider the controller showMyPapers in Fig. 9, which takes as argument a user client and displays to them the titles of all the papers they authored. To that end, showMyPapers filters the list of all paper identifiers allPapers with a monadic predicate include \( p \), which returns True iff client is an author of \( p \).

At a first glance, this program should be rejected: showMyPapers reads the author lists of every paper, even those that client is not allowed to see. Moreover, because of the self-referential policy, the programmer finds themselves in a catch-22 situation: in order to check whether the policy holds for a paper \( p \), they must retrieve the author list of \( p \), but that retrieval itself violates the policy! Observe, however, that showMyPapers does not in fact leak anything to the user. In particular, it does not allow client to distinguish between two data stores that only differ in author lists that
client is not allowed to see. So, we would like showMyPapers to be well-typed, but that requires the type checker to perform nontrivial reasoning about the values returned by include p.

This is exactly where the LIFTY’s downgrade primitive comes in: it allows the type checker to downgrade the input label on include p, because it always returns False for those papers that client is not allowed to see. This is not a coincidence: in fact, any (correctly implemented) runtime check of a self-referential policy is well-typed (only) if it is wrapped in a downgrade. In our example, the programmer does not use downgrading explicitly because it is built into our filterM function:

$$\text{filterM} :: \forall \alpha, i, f. \ (x :: \alpha \rightarrow \text{TI0} \ (\text{Bool} \ | \ v \Rightarrow f \ x) < f \ x \land i _=, \ false>)$$

$$\rightarrow \ [\alpha] \rightarrow \text{TI0} \ [(\alpha \ | \ f \ v)] < i, \ false>$$

This type permits the filter predicate to have a higher label than the overall computation, as long as for each list element x, the difference in labels f x is implied by the return value of the predicate. This, in turns, allows the code in Fig. 9 to type-check in LIFTY completely automatically.

**Information leaks through search.** Now consider the controller showMyAccepts in Fig. 10, which is similar but only shows client their accepted papers. This code has an information leak of the similar nature as our original example in Fig. 3: the decision leaks to client through the list of paper titles before the reviewing is done. This example is inspired by real-world leaks through search and recommendation functionality of data-centric applications (see Sec. 6 for concrete examples).

As expected, the LIFTY type-checker identifies the input action getDecision ds p as the cause of the leak, and replaces it with the same leak patch it generated for the EDAS leak. With this patch, showMyAccepts always returns an empty list if called before reviewing is done; the programmer deems this behavior acceptable and therefore accepts the patch.

## 4 THE CORE CALCULUS

We now formalize a core language $\lambda^L$, which captures the essence of LIFTY’s IFC mechanism. We first present its syntax, as well as dynamic and static semantics. The main goal of this section is to prove a non-interference guarantee, which we accomplish by reduction to LIO [Stefan et al. 2017].

### 4.1 Syntax of $\lambda^L$

$\lambda^L$ is a pure call-by-name $\lambda$-calculus, equipped with refinement types and IFC constructs. We summarize its syntax in Fig. 11.

**Refinements.** As is common in refinement types literature [Polikarpova et al. 2016; Rondon et al. 2008; Vazou et al. 2014b], $\lambda^L$ distinguishes between refinement terms $r$ used inside types and program terms $t$. We assume a syntactic category of atomic refinement terms $a$ drawn from a first-order theory. For example, for the theory of equality and uninterpreted function, $a$ consists of
equalities such as \( x = \text{phase} \, \ell \). Our formalization is agnostic to the choice of theory, as long as validity of universal formulas of the form \( \forall x_1, \ldots, x_n. r \) is decidable\(^3\).

**Labels.** We introduce a syntactic category \( \ell \in \mathcal{L} \) for *labels*, which are lambda abstractions over refinements. In a label \( \lambda u. r \), the refinement \( r \) can mention the program variables and the bound variable \( u \). Note that \( \langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle \) forms a lattice with

\[
\ell_1 \sqsubseteq \ell_2 \triangleq \forall u. \ell_1 u \\Rightarrow \ell_2 u \quad \ell_1 \sqcup \ell_2 \triangleq \lambda u. \ell_1 u \land \ell_2 u \quad \ell_1 \sqcap \ell_2 \triangleq \lambda u. \ell_1 u \lor \ell_2 u
\]

\( T \triangleq \lambda u. \text{false} \quad \bot \triangleq \lambda u. \text{true} \)

**Types.** The base types \( B \) in \( \lambda^L \) include the unit type and the booleans; we omit other base types (and values) since they are irrelevant to IFC. Types \( T \) include the type of sensitive computations \( \text{TIO} \, T \langle \ell_1, \ell_2 \rangle \), indexed by return type \( T \), and input and output labels \( \ell_1 \) and \( \ell_2 \), respectively. The rest of the types are either refined base types or function types. In a refined base types \( \{ B | r \} \), \( r \) is a refinement predicate over the program variables and a special value variable \( v \), which denotes the bound variable of the type. We sometimes write \( B \) as a shortcut for \( \{ B | \text{true} \} \). Although the LIftY implementation supports dependent function types of the form \( x : T_1 \rightarrow T_2 \) (where the refinement of \( T_2 \) can mention the argument \( x \)), and indeed we use them to specify policies in data-centric applications, they are not central to our formalization, and hence the dependency is omitted from \( \lambda^L \) for simplicity.

**Program terms.** Base values \( b \in B \) include unit and booleans. Values \( v \) additionally include lambda abstractions and monadic actions \( \text{TIO} \, t \); like in prior work [Buiras et al. 2015; Stefan et al. 2017; Vassena and Russo 2016], the \( \text{TIO} \) constructor is not part of the surface syntax. Terms \( t \) include values, variables, function application, conditionals, as well as monadic primitives \( \text{return} \), \( \text{bind} \), and \( \text{downgrade} \) introduced in Sec. 3. The \( \text{downgrade} \) in \( \lambda^L \) has an explicit label argument—the label to downgrade to—for simplicity; in LIftY this label is inferred by the type checker. Atomic input/output actions are represented in \( \lambda^L \) as two universal actions \( \text{get} \) and \( \text{set} \) that also have explicit labels as first arguments; intuitively, each label represents the security level of the resource the action is reading from or writing to. We omit recursion from \( \lambda^L \), since it does not present distinct challenges for static IFC. The LIftY implementation supports recursion, and uses refinement types to prove that all recursive functions terminate.
To model input and output actions, we define the run-time behavior of $\lambda$ values are written to the store,

\[ \Gamma \vdash T_i <: T_2 \]

Subtyping

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

Typing

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash \text{return } t ::= \text{TIO } T \langle \bot, T \rangle \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-RET} \quad \Gamma \vdash \text{return } t ::= \text{TIO } T \langle \bot, T \rangle \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

Fig. 13. Static semantics of IFC constructs in $\lambda^L$ (see Appendix A for the full semantics).

4.2 Dynamic Semantics of $\lambda^L$

To model input and output actions, we define the run-time behavior of $\lambda^L$ in terms of a store $\Sigma: L \rightarrow B$ that maps labels to base values. This is similar to split memory used in MAC [Vassena and Russo 2016], which is also partitioned into security levels. Our store is much simpler, however: at each level $\ell$, it only stores a single value of a fixed base type $B_\ell$.

A program configuration $\langle \Sigma \mid t \rangle$ consists of a store and a term. Fig. 12 defines a small-step evaluation relation $\rightarrow$ on configurations. We will write $\rightarrow^*$ for a recursive transitive closure of this relation. The rules for pure terms are standard and therefore omitted (the full set of rules can be found in Appendix A). The behavior of monadic primitives is mostly straightforward; in particular, since $\lambda^L$ only tracks labels statically, there is no label propagation or access checks at run time. Of note are the rules for bind and downgrade, which transitively evaluate (one of) their subterms to a value; this follows prior work [Buiras et al. 2015; Stefan et al. 2017], and the reason for this choice will be clear from the non-interference argument in Sec. 4.4. To ensure that only values are written to the store, set is strict. Other than that, evaluation order of pure and monadic terms is standard [Peyton Jones 2001].

4.3 Static Semantics of $\lambda^L$

Fig. 13 shows a subset of typing rules for $\lambda^L$ that are relevant to IFC. Other rules are standard for languages with decidable refinement types [Rondon et al. 2008] and deferred to Appendix A. In the figure, a typing environment $\Gamma ::= \epsilon \mid \Gamma, x : T \mid \Gamma, r$ maps variables to types and records path conditions $r$, which arise when checking conditional terms.

Subtyping. We only show subtyping rules for TIO types. The rule $\ll T \rightarrow$ TIO specifies that tagged types are covariant in their input label and contra-variant in the output label wrt. the lattice ordering. The premise $\Gamma \vdash \ell' \subseteq \ell$ denotes the validity of label ordering under the assumptions stored in the environment (which include path conditions and refinements on program variables). For example, the judgment $a : \text{User}, b : \{\text{User} \mid v = a\} \vdash (\lambda u.u = a) \subseteq (\lambda u.u = b)$ reduces to a

\[ \text{Subtyping} \]

\[ \Gamma \vdash T_i <: T_2 \]

\[ \Gamma \vdash \ell_1 \subseteq \ell_2 \quad \Gamma \vdash \ell_2' \subseteq \ell_1' \]

\[ \Gamma \vdash \text{TIO } T_1 \langle \ell_1, \ell_1' \rangle <:\text{TIO } T_2 \langle \ell_2, \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash \text{return } t ::= \text{TIO } T \langle \bot, T \rangle \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]

\[ \text{T-DOWM} \quad \Gamma \vdash \text{downgrade } \ell t ::= \text{TIO } \langle \ell, \ell' \rangle \]

\[ \text{T-BIND} \quad \Gamma \vdash \text{bind } t_1 t_2 ::= \text{TIO } T_2 \langle \ell_1 \sqcup \ell_2, \ell_1' \cap \ell_2' \rangle \]

\[ \text{T-RET} \quad \Gamma \vdash t ::= T \]

\[ \text{T-GET} \quad \Gamma \vdash \text{get } \ell ::= \text{TIO } B_\ell \langle \ell, T \rangle \]

\[ \text{T-SET} \quad \Gamma \vdash \ell ::= T \]
formula $\forall a, b, u, v = a \land u = b \Rightarrow u = a$ (which is valid). Recall that under our assumptions on the refinement logic, the validity of such formulas is decidable.

**Term typing.** The rest of Fig. 13 defines the typing judgments for program terms $\Gamma :: t :: T$. The first three rules encode the same label propagation logic we have introduced in Fig. 6; the difference is that in $\lambda_L$ the monadic primitives are built-in and have custom typing rules, whereas in the actual LIFTY implementation they are represented as library functions with polymorphic types. Note that we leverage the flexibility of custom typing rules to simplify the static semantic of $\text{bind}$ slightly: here we add an explicit premise $\Gamma :: t_1 :\leq t_2$, whereas in Fig. 6 we encode this implicitly by representing $\ell'_2$ as $\ell_1 :\leq \ell_2$. The last two rules type the universal actions $\text{get}$ and $\text{set}$ as expected: reading/writing from the store at security level $\ell$ yields a computation with input/output label $\ell$, respectively. Recall that $B_\ell$ is the type of values stored in $\Sigma$ at level $\ell$.

### 4.4 Noninterference in $\lambda_L$

In this section we show that LIFTY programs cannot leak sensitive data by proving noninterference for the core calculus, i.e. we show that evaluating an appropriately typed $\lambda_L$ term $t$ starting in two $\ell$-equivalent stores yields $\ell$-equivalent stores\footnote{Two stores are considered $\ell$-equivalent if they only differ in keys above $\ell$.}. Instead of proving noninterference from first principles, we accomplish this by reducing $\lambda_L$ to core LIO, a language with dynamic IFC, whose noninterference proof has been mechanized in Coq [Stefan et al. 2017]. Our proof strategy is a as follows: first, we present instrumen
ted operational semantics that adds dynamic IFC to $\lambda_L$; next, we argue that the instrumented $\lambda_L$ is a subset of core LIO and hence exhibits non-interference; finally, we show that well-typed terms behave equivalently under the original and instrumented semantics.

#### 4.4.1 Instrumented Semantics.

To add dynamic IFC to $\lambda_L$, we define an instrumented configuration $k$ as a triple $(\Sigma, \ell_c :: t)$, where $\ell_c$ is the current (program counter) label. Intuitively, $\ell_c$ starts out at $\bot$ and then gradually rises as the evaluation progresses, keeping track of the most sensitive resource that the computation has read so far and blocking output to any resource below $\ell_c$. Fig. 14 defines a small-step evaluation relation $\rightsquigarrow$ on instrumented configurations; as usual, we write $\rightsquigarrow^*$ for its recursive transitive closure. The rules for pure terms keep the current label intact and are therefore omitted (the full set of rules can be found in Appendix A).

The core mechanism for propagating and checking run-time labels is captured in the rules $\text{I-}\text{get}$, $\text{I-}\text{set}$, and $\text{I-}\text{bind}$. The rule $\text{I-}\text{get}$ raises the current label by $\ell$—the label of the resource being read. The premise of $\text{I-}\text{set}$ checks that the current label can flow to the label of the resource. Note that

\[
\begin{align*}
\text{Instrumented Evaluation} & \quad \frac{}{\langle \Sigma, \ell_c :: t \rangle \rightsquigarrow \langle \Sigma', \ell'_c :: t' \rangle} \\
\text{1-RET} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{return } t \rangle \rightsquigarrow \langle \Sigma, \ell_c :: \text{TIO } t \rangle} \\
\text{1-BIND} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{bind } t_1 t_2 \rangle \rightsquigarrow^* \langle \Sigma', \ell'_c :: \text{TIO } t'_1 \rangle} \\
\text{1-DOWN-1} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{downgrade } \ell \ t \rangle \rightsquigarrow \langle \Sigma', \ell \cup \ell_c :: \text{TIO } b \rangle} \\
\text{1-DOWN-2} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{downgrade } \ell \ t \rangle \rightsquigarrow \langle \Sigma', \ell \cup \ell_c :: \text{TIO } \text{False} \rangle} \\
\text{1-GET} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{get } \ell \rangle \rightsquigarrow \langle \Sigma, \ell \cup \ell_c :: \text{TIO } \Sigma[\ell] \rangle} \\
\text{1-SET} & \quad \frac{}{\langle \Sigma, \ell_c :: \text{set } \ell \ b \rangle \rightsquigarrow \langle \Sigma[\ell := b], \ell_c :: \text{TIO } () \rangle}
\end{align*}
\]
before performing such a runtime check the labels are evaluated to closed terms; hence, unlike a static check \( \Gamma \vdash \ell_1 \subseteq \ell_2 \), a dynamic check \( \ell_1 \subseteq \ell_2 \) does not require an environment. Finally the rule \( \text{I-BIND} \) uses the final label of the first action as the starting label of the second action.

The most interesting part of the instrumented semantics is the behavior of \( \text{downgrade} \ \ell \ t \), expressed in \( \text{I-DOWN-1} \) and \( \text{I-DOWN-2} \). As in the uninstrumented case, both of these rules start by fully evaluating \( t \) to \( \text{TIO} \ b \) (i.e., either \( \text{TIO} \ True \) or \( \text{TIO} \ False \)); this evaluation raises the current label to \( \ell' \). Instead of adopting \( \ell' \) as the new current label, however, both rules \( \text{downgrade} \) it to \( \ell \cup \ell_c \), effectively only raising the current label by the manifest label \( \ell \) of the \( \text{downgrade} \) operation. But won’t such downgrading leak information at level \( \ell \)? Set \( \ell \) will remain below the new output label (this follows from property (2) of Lemma 3); and \( \ell \) won’t such downgrading leak information at level \( \ell' \)? This is where the \( \text{I-DOWN-2} \) comes in: if it turns out that in fact \( \ell' \) does not flow to the downgraded label, then the true value of \( b \) is discarded and False is returned instead.

### 4.4.2 From Instrumented Semantics to LIO

We argue that the semantics in Fig. 14 is equivalent to a subset of sequential LIO with references, as defined in [Stefan et al. 2017]. For pure terms, as well as return and bind the equivalence is straightforward by comparing the evaluation rules. The remaining primitives get, set, and \( \text{downgrade} \) can be encoded in LIO as follows:

\[
\begin{align*}
\text{get} \ l & \equiv \text{readLIORef} \ ref \\
\text{set} \ l \ t & \equiv \text{writeLIORef} \ ref \ t \\
\text{downgrade} \ l \ t & \equiv \begin{cases} 
\text{do} \ lc \leftarrow \text{getLabel} \\
\text{catchLIO} \ (\text{unlabel} \ lb) \ (\lambda \_ \rightarrow \text{return} \ False)
\end{cases}
\end{align*}
\]

Both get and set simply read and write a singleton reference \( ref \) labeled \( l \) (and created with \( \text{newLIORef} \)). The more interesting case is \( \text{downgrade} \). The \( \text{toLabeled} \) primitive returns a labeled value—whose label is \( l \cup lc \)—that either contains the result of \( t \) or a "delayed" exception if \( t \) reads data more sensitive than \( l \cup lc \). \( \text{unlabel} \) raises the current label to \( l \cup lc \) and either returns the value computed within the \( \text{toLabeled} \) block or throws the delayed exception—hence the need to catch the exception and return False.

Since noninterference in LIO has been proven wrt. an arbitrary security lattice, the instrumented \( \lambda^L \) inherits its guarantee (\( \ell \)-equivalence on instrumented configurations is defined in Appendix A):

**Lemma 1** (Noninterference of instrumented semantics). Evaluation from \( \ell \)-equivalent configurations leads to \( \ell \)-equivalent configurations: if \( k_1 \approx_{\ell} k_2, k_1 \rightarrow^n k'_1 \), and \( k_2 \rightarrow^n k'_2 \), then \( k'_1 \approx_{\ell} k'_2 \).

### 4.4.3 Simulation

The core of our non-interference argument is a proof that instrumented execution simulates original execution for well-typed terms. The full proofs can be found in Appendix A; here we only state the key lemmas and give the intuition for the proofs.

**Lemma 2** (Single-step simulation). If \( \Gamma \vdash t :: \text{TIO} \ T \langle \ell_i, \ell_o \rangle \) and \( \langle \Sigma \mid t \rangle \rightarrow \langle \Sigma' \mid t' \rangle \), then for any \( \ell_c \subseteq \ell_o \), there exists a new current label \( \ell'_c \), and a new type \( \text{TIO} \ T' \langle \ell'_i, \ell'_o \rangle \) such that

1. \( \langle \Sigma, \ell_c \mid t \rangle \rightarrow \langle \Sigma' \mid \ell'_i \cup \ell_i \rangle \),
2. \( \ell'_c \subseteq \ell_c \cup \ell_l \),
3. \( \Gamma \vdash t' :: \text{TIO} \ T' \langle \ell'_i, \ell'_o \rangle \)
4. \( \text{TIO} \ T \langle \ell_i, \ell_o \rangle \)

Intuitively, this lemma says that executing a well-typed monadic term \( t \) from a configuration where \( \ell_c \) is not too-high (with respect to the static output label) leads to the same result under the instrumented semantics (1). In addition, we also show that the new current label remains below the new static output label (5); this allows us to chain multiple evaluation steps together in Lemma 3. The proof is by induction on the typing derivation. Interesting cases include set, where we show that the runtime check never fails (this follows from \( \ell_c \subseteq \ell_o \); \( \text{bind} \)), where we show that the new current label remains below the new output label (this follows from property (2) of Lemma 3); and \( \text{downgrade} \), where we show that whenever \( \text{I-DOWN-2} \) applies, \( b \) is False anyway.
We now formalize Lifty with holes (i.e., noninterference for Lemma 3 and Lemma 3 (Multi-step simulation). If \( \Gamma \vdash t :: \text{TI0} \left< \ell_1, \ell_o \right> \) and \( \langle \Sigma | t \rangle \rightarrow^a \langle \Sigma' | t' \rangle \), then for any \( \ell_c \subseteq \ell_o \), there exists a new current label \( \ell'_c \) and a new type \( \text{TI0} T' \left< \ell'_1, \ell'_o \right> \) such that (1) \( \langle \Sigma, \ell_c | t \rangle \rightarrow^a \langle \Sigma', \ell'_c | t' \rangle \), (2) \( \ell'_c \subseteq \ell_c \cup \ell_l \), (3) \( \Gamma \vdash t' :: \text{TI0} T' \left< \ell'_1, \ell'_o \right> \) (4) \( T \text{I0} T' \left< \ell'_1, \ell'_o \right> :: \text{TI0} \left< \ell_1, \ell_o \right> \), and (5) \( \ell'_c \subseteq \ell'_o \).

The proof is straightforward, by induction on the number of steps. Finally we can combine Lemma 1, Lemma 3, and LIO’s noninterference proof to prove noninterference of \( \lambda^L \) programs:

**Theorem 1** (Noninterference for \( \lambda^L \)). Evaluating a computation \( t \) statically visible to \( \ell \) from \( \ell \)-equivalent stores leads to \( \ell \)-equivalent stores: if \( \Sigma_1 \approx_{\ell} \Sigma_2, \Gamma \vdash t :: \text{TI0} \left< \ell_1, \ell_o \right> \), and \( \langle \Sigma_1 | t \rangle \rightarrow^a \langle \Sigma_2 | t' \rangle \), we have \( t'_1 = t'_2 \) and \( \Sigma'_1 \approx_{\ell} \Sigma'_2 \).

5 LEAK REPAIR IN \( \lambda^L \)

We now formalize Lifty’s leak repair mechanism for the core calculus \( \lambda^L \). Fig. 15 shows the pseudocode of the algorithm ENFORCE, which performs type-checking and repair of an individual controller function. More precisely, the algorithm takes as input a typing environment \( \Gamma \) and a program \( t \), and determines whether \( t \) can be patched to produce a well-typed TI0 computation, i.e. a term \( t' \) such that \( \Gamma \vdash t' :: \text{TI0} \left< \ell, \ell_\bot \right> \). The algorithm proceeds in two steps. First, procedure LOCALIZE identifies unsafe terms (line 2), replacing them with type casts to produce a “program with holes” \( \hat{t} \) (Sec. 5.1). Second, the algorithm replaces each type cast in \( \hat{t} \) with an appropriate patch, generated by the procedure GENERATE (Sec. 5.2).

5.1 Fault Localization

Type casts. For the purpose of fault localization, we extend the values of \( \lambda^L \) with type casts:

\[
\nu ::= \cdots | \left< T \ast T' \right>
\]

Statically, our casts are similar to those in prior work [Knowles and Flanagan 2010]; in particular, the cast \( \left< T \ast T' \right> \) has type \( T' \rightarrow T \). However, the dynamic semantics of casts in \( \lambda^L \) is undefined. The idea is that casts are inserted solely for the purpose of fault localization, and, if repair succeeds, are completely eliminated. We restrict the notion of type-safe \( \lambda^L \) programs to those that are well-typed are free of type casts.

Minimal sound localizations. The goal of algorithm LOCALIZE is to infer a minimal sound localization of the term \( t \). We say that a term \( \hat{t} \) is a sound localization of \( t \) at type \( T \) in \( \Gamma \), if (1) \( \hat{t} \) is obtained...
from $t$ by inserting type casts (i.e., replacing one or more subterms $t_i$ in $t$ by $\langle T_i \triangleleft T'_i \rangle t_i$), and (2) it is type correct, i.e., $\Gamma \vdash \hat{t} :: T$. In particular, note that (2) implies that each $t_i$ has type $T'_i$. Hence, the following lemma follows directly from the substitution lemma for refinement types [Knowles and Flanagan 2010]:

**Lemma 4** (Localization). Replacing each subterm of the form $\langle T_i \triangleleft T'_i \rangle t_i$ in a sound localization of $t$ with a type-safe term of type $T_i$, yields a type-safe program.

A sound localization $\hat{t}$ of $t$ is minimal, if replacing any $T_i$ in $\langle T_i \triangleleft T'_i \rangle$ in $\hat{t}$ with its supertype prevents $\hat{t}$ from type-checking against $T$. Finding a sound localization is important because patch generation can then proceed independently for each type cast (taking the type $T_i$ as the expected type); minimality is important because it gives patch generation the highest chance to succeed.

In general, a term can have multiple minimal sound localizations, and a general program repair engine would have to explore them all, leading to inefficiency. Our goal, however, is to implement a very specific repair strategy—guarding and redacting unsafe input actions. Hence, we can restrict the set of candidate localizations as follows: first, for terms $t_i$ we only consider input actions (i.e., terms of the form $\text{get } \ell'$); second, for the expected types $T_i$ we only consider types of the form $\text{T10 } B \langle \ell, T \rangle$, where $\ell$ is the expected label (the highest label that makes the localization sound).

Importantly, we do not consider types where the result of the $\text{T10}$ monad is refined: because we are going to redact the input action, we are unlikely to be able to satisfy any of the functional properties of the original action. Although these restrictions might lead to missing valid repairs in rare cases, they are crucial for efficiency: in fact, given these restrictions there exist at most one minimal sounds localization. To infer this localization, we simply have to find the expected label for each unsafe access, which can be done efficiently during type checking.

**Inferring the localization.** Algorithm LOCALIZE first uses the $\lambda^T$ typing and subtyping rules to reduce the problem of checking the source program $t$ to a system of constrained Horn clauses (CHCs) over unknown refinements and labels. As we explained in Sec. 3, CHCs can be divided into rules and queries. In line 9, we use an existing CHC solver [Cosman and Jhala 2017] to obtain the strongest assignment $\mathcal{A}$ of refinement terms to unknowns. If this assignment satisfies all the queries, then $t$ is well-typed, and LOCALIZE terminates without modifying it. Otherwise, for each query $Q$ that does not hold under $\mathcal{A}$, we obtain its source, i.e., the term $d$ and the subtyping constraint that generated the query. Now if the query was generating by a can-flow check from the input label $i$ of term $d$ to some label $P$, then we insert a type-cast around $d$, taking the expected label to be $\mathcal{A}[P]$, i.e., the valuation of $P$ in $\mathcal{A}$. If, on the other hand, $Q$ is not caused by a may-flow check, then the type error is not caused by an information leak, hence LOCALIZE fails.

### 5.2 Patch Generation

Next, we describe how our algorithm replaces a type-cast $\langle T_e \triangleleft T_o \rangle d$ with a type-safe term $d'$ of the expected type $T_e$. using the patch generation procedure GENERATE (line 4). GENERATE implements a domain-specific synthesis strategy: first, it generates a list of branches, which return the original term redacted to a different extent; then, for each branch, it infers an optimal guard (a policy check) that makes the branch satisfy the expected type; finally, it constructs the patch by arranging the properly guarded branches into a (monadic) conditional.

**Synthesis of branches.** In line 17, GENERATE uses SYNQUID [Polikarpova et al. 2016] to synthesize the set of all terms up to certain size with the right content type, but with no restriction on the policy. Branches are generated in a restricted environment $\Gamma_B$, which contains only the original faulty term and a small set of redaction functions $\mathcal{R}$. This set is specified by the programmer, and typically includes a “default value” of each type, but may also include custom redaction functions,
e.g. that sanitize strings (we show examples in Sec. 6). This restriction gives the user control over the space of patches and also makes the synthesis more efficient.

**Synthesis of guards.** For each of the branches \( b \) (which include at least the original term), `Generate` attempts to synthesize the optimal guard that would make \( b \) respect the expected type. At a high level, this guard must be logically equivalent to the weakest formula \( c \), such that (1) \( c \land p \Rightarrow q \), where \( \lambda u.p \) is \( \ell \), the expected input label of the patch, and \( \lambda u.q \) is the actual input label of branch \( b \); (2) \( c \) does not mention the user variable \( u \). This predicate can be inferred using existing techniques, such as logical abduction [Dillig and Dillig 2013]. In particular, `Generate` relies on `Synqid`'s *liquid abduction* mechanism to infer \( c \) for each branch in line 19.

Next, we topologically sort the branches according to their abduced conditions, from weakest to strongest (i.e. in the reverse order of how they are going to appear in the program). In line 21, we check that the first branch can be used as the *default branch*, i.e. it is correct unconditionally. This property is always satisfied as long as \( \Gamma_B \) contains a pure value \( v \) of type \( T \), in which case `return v` is a valid default branch.

The main challenge of guard synthesis, is that the guard itself must be monadic, since it might need to retrieve and compute over some data from the store. Since the data it retrieves might itself be sensitive, we need to ensure that two conditions are satisfied (1) *functional correctness*: the guard returns a value equivalent to \( c \), and (2) *no leaky enforcement*: the input label of the guard itself may flow to the expected label \( \ell \) of the patch. To ensure both conditions, we again use `Synqid`, this time with the goal type \( \text{TII} \{ \text{Bool} \mid v \Leftrightarrow c \} \langle \ell \rangle \).

**Lemma 5 (Safe patch generation).** If `Generate` succeeds, it produces a type-safe term of the expected type \( \text{TII} T \langle \ell \rangle \).

Assuming correctness of `Synthesize` and `Abduce`, we can use the typing rules of Sec. 4 to show that the invariant \( \Gamma \vdash \text{patch} :: \text{TII} T \langle \ell \rangle \) is established in line 22 and maintained in line 27. In particular, the type of bound variable \( g \) in line 27 is \( \{ v : \text{Bool} \mid v \Leftrightarrow c \} \), hence, then branch is checked under the path condition \( c \Leftrightarrow \text{true} \), which implies \( \Gamma_G \vdash b :: \text{TII} T \langle \ell \rangle \).

### 5.3 Guarantees and Limitations

In this section we summarize the soundness guarantee of leak repair in \( \lambda L \) and then discuss the limitations on its completeness and minimality.

**Theorem 2 (Soundness of leak repair).** If procedure `Enforce` succeeds, it produces a program that satisfies noninterference.

This is straightforward by combining Lemmas 4 and 5 with Theorem 1.

**Completeness.** When does procedure `Enforce` fail? `Localize` fails when it cannot find a safe localization satisfying our domain-specific restrictions, which happens if (1) the program contains an error unrelated to information flow, or (2) the program depends on a functional property of an unsafe input action we want to redact. We consider both of these cases out of scope of our domain-specific repair algorithm. `Generate` can fail in lines 23 and 26. The first failure indicates that \( \Gamma_B \) does not contain any sufficiently public terms; in this case, `Lifty` prompts the programmer to add a default value of an appropriate type. The second failure happens when no guard satisfies both functional and security requirements; this commonly indicates that the policy is *not enforceable* without leaking some other sensitive information. For instance, in the EDAS leak example, if the programmer declared "phase" to be only visible to the program chair, no program could precisely check the policy on "status" without leaking the information about "phase". In this case, `Lifty` prompts the programmer to change the policies in a way that respects dependencies between sensitive fields (i.e. to make the policy on "status" at least as restrictive as the one on "phase").
Table 1. Microbenchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code size (AST nodes)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Localize</td>
</tr>
<tr>
<td></td>
<td>Original</td>
<td>Policy enforcing</td>
</tr>
<tr>
<td>1 EDAS</td>
<td>66</td>
<td>25</td>
</tr>
<tr>
<td>2 EDAS-Multiple</td>
<td>87</td>
<td>50</td>
</tr>
<tr>
<td>3 EDAS-Self-Ref</td>
<td>87</td>
<td>76</td>
</tr>
<tr>
<td>4 Search</td>
<td>76</td>
<td>25</td>
</tr>
<tr>
<td>5 Sort</td>
<td>64</td>
<td>58</td>
</tr>
<tr>
<td>6 Broadcast</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>7 HotCRP</td>
<td>68</td>
<td>22</td>
</tr>
<tr>
<td>8 AirBnB</td>
<td>52</td>
<td>33</td>
</tr>
<tr>
<td>9 Instagram</td>
<td>73</td>
<td>42</td>
</tr>
</tbody>
</table>

Minimality. Ideally, we would like to show that the changes made by Enforce are minimal: in any execution where t did not cause a leak, t' would output the same values as t. Unfortunately, this is not true: Enforce is conservative and might hide more information than is strictly necessary. The reason for the imprecision is two-fold: (1) the minimal localization inferred by Localize might over-approximate the actual runtime level of output actions due to imprecisions of refinement type inference; and (2) the guard condition c abduced by Generate might be overly strong due to the limitations of the abduction engine. In the latter case, the programmer can provide a more precise guard manually; the former is a fundamental limitation of all static IFC systems.

6 EVALUATION

Implementation. We have implemented a prototype LIFTY compiler by extending the SYNQUID program synthesizer [Polikarpova et al. 2016]. From SYNQUID, LIFTY inherits a liquid type checker and a type-driven synthesis mechanism. On top of this, our implementation adds (1) the TIO library, which implements the API shown in Fig. 6 plus some standard output actions and redaction functions (90 lines of LIFTY code); (2) the implementation of the Enforce algorithm from Sec. 5 that calls out to the type-checker and the synthesizer; (3) a SYNQUID-to-Haskell translator, which can link LIFTY code with other Haskell modules. Thanks to the translator, a possible usage scenario for LIFTY is to serve as a language for the data-centric application core, while low-level libraries can be implemented directly in Haskell.

Programs. To evaluate the LIFTY compiler, we implemented (1) a set of microbenchmarks that highlight challenging scenarios and model reported real-world leaks; and (2) three larger case studies based on existing applications from the literature. For each of these programs, we specified the security policies in the style of Fig. 8 and implemented the basic logic of the controllers omitting all policy checks. Hence, for each controller, LIFTY must localize unsafe data accesses and generate leak patches. For one of our case studies, we additionally implemented a non-leaky version with manually written policy checks.

Evaluation criteria. Our goal is to evaluate the following parameters:

- Expressiveness of policy language. We demonstrate that LIFTY is expressive enough to support interesting policies from in a range of problem domains, including conference management, course management, health records, and social networks. In particular, we were able to replicate all the desired policies in three case studies from prior work [Swamy et al. 2010; Yang et al. 2016a].
- Scalability. We show that the LIFTY compiler is reasonably efficient at error localization and patch synthesis: LIFTY is able to generate all necessary patches for each of our case studies.
### (a) Conference Management System

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Policy enforcing</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original Manual Auto</td>
<td>Manual Localize Generate Total</td>
</tr>
<tr>
<td>Register user</td>
<td>22 0 0</td>
<td>0.1s 0.1s 0.0s 0.1s</td>
</tr>
<tr>
<td>View users</td>
<td>26 16 25</td>
<td>0.5s 0.2s 0.2s 0.4s</td>
</tr>
<tr>
<td>Paper submission</td>
<td>65 0 0</td>
<td>2.0s 2.2s 0.0s 2.2s</td>
</tr>
<tr>
<td>Search papers</td>
<td>123 77 96</td>
<td>29.0s 6.5s 5.4s 11.9s</td>
</tr>
<tr>
<td>Show paper record</td>
<td>63 61 85</td>
<td>8.0s 0.8s 2.0s 2.8s</td>
</tr>
<tr>
<td>Show reviews for paper</td>
<td>96 61 70</td>
<td>14.6s 5.0s 0.8s 5.8s</td>
</tr>
<tr>
<td>User profile: GET</td>
<td>46 0 0</td>
<td>0.2s 0.2s 0.0s 0.2s</td>
</tr>
<tr>
<td>User profile: POST</td>
<td>20 0 0</td>
<td>0.1s 0.1s 0.0s 0.1s</td>
</tr>
<tr>
<td>Submit review</td>
<td>103 0 0</td>
<td>9.0s 7.8s 0.0s 7.8s</td>
</tr>
<tr>
<td>Assign reviewers</td>
<td>63 0 0</td>
<td>0.6s 0.6s 0.0s 0.6s</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td>627 215 276</td>
<td>64.1s 23.5s 8.4s 31.9s</td>
</tr>
</tbody>
</table>

### (b) Gradr—Course Management System

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code size (AST nodes)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the home page (static content)</td>
<td>17 0</td>
<td>0.0s 0.0s 0.0s 0.0s</td>
</tr>
<tr>
<td>View a user’s profile (owner)</td>
<td>119 59</td>
<td>3.0s 0.8s 0.8s 3.8s</td>
</tr>
<tr>
<td>View a user’s profile (any user)</td>
<td>120 59</td>
<td>3.1s 1.5s 1.5s 4.6s</td>
</tr>
<tr>
<td>Instructor: view scores for an assignment</td>
<td>73 103</td>
<td>0.9s 1.4s 1.4s 2.3s</td>
</tr>
<tr>
<td>Instructor: view top scores for an assignment</td>
<td>99 147</td>
<td>2.4s 2.4s 2.4s 4.8s</td>
</tr>
<tr>
<td>Student: view all scores for user</td>
<td>100 112</td>
<td>3.7s 9.5s 9.5s 13.2s</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td>528 480</td>
<td>13.2s 15.6s 15.6s 28.8s</td>
</tr>
</tbody>
</table>

### (c) HealthWeb—Health Information Portal

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code size (AST nodes)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search a record by id</td>
<td>27 161</td>
<td>0.1s 13.0s 13.0s 13.0s</td>
</tr>
<tr>
<td>Search a record by patient</td>
<td>74 339</td>
<td>1.2s 41.6s 41.6s 42.8s</td>
</tr>
<tr>
<td>Show authored records</td>
<td>76 0</td>
<td>1.2s 0.0s 0.0s 1.2s</td>
</tr>
<tr>
<td>Update record</td>
<td>25 0</td>
<td>0.0s 0.0s 0.0s 0.0s</td>
</tr>
<tr>
<td>List patients for a doctor</td>
<td>80 87</td>
<td>1.3s 3.5s 3.5s 4.8s</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td>282 587</td>
<td>3.8s 58.1s 58.1s 61.9s</td>
</tr>
</tbody>
</table>

Table 2. Case studies: conference management, course manager, health portal.

in 30–60 seconds. Furthermore, we show that synthesis times are linear in the number of required patches.

- **Quality of patches.** We compare the code generated by Lifty to a version with manual policy enforcement and show that it is able to recover all necessary policy-enforcing code, without reducing functionality.
6.1 Microbenchmarks

To exercise the flexibility of our language, we implemented a series of small but challenging microbenchmarks, summarized in Tab. 1. The code of each benchmark, with leak patches inserted by LIFTY, is available in Appendix C.

Benchmark EDAS is our running example from Sec. 2; benchmarks 2–3 are its variations with multiple unsafe accesses and with a self-referential policy on the “authors” field, respectively. Benchmarks 4 and 5 exercise tricky cases of implicit flow through higher-order functions. Search is the controller from Fig. 10, which displays the titles of all client’s accepted papers; here LIFTY inserts a patch inside the filter’s predicate.

Sort displays the list of all conference submissions sorted by their score, using a higher-order sort function with a custom comparator. The order of submissions might leak paper scores to a conflicted reviewer. To prevent this leak, LIFTY rewrites the comparator to return a default score if a paper is conflicted with the viewer. Interestingly, this benchmark features a negative self-referential policy for the list of conflicted reviewers: this list is visible only to users who are not on the list. Such policies are not supported by Jeeves, since they are incompatible with its fixpoint interpretation of self-referential policies (Sec. 7); in LIFTY, the semantics of policies is decoupled from their evaluation, hence this example presents no difficulty.

Broadcast sends a decision notification to all authors of a given paper. This benchmark tests LIFTY’s ability to handle messages sent to multiple users; LIFTY infers that all those users are authors of the paper, and hence are allowed to see its status, as long as the phase is done. An additional challenge is that the list of recipients is itself sensitive, since the conference is double-blind; LIFTY infers that no additional check is needed, since authors are always allowed to see themselves.

The last three benchmarks model reported real-world leaks. HotCRP models a leak in the HotCRP conference manager, first reported in [Yip et al. 2009], where the conference chair could send password reminder emails to PC members, and then glean their passwords from the email preview. LIFTY repairs this leak by masking the password in the preview (but not in the actual email), since the preview is flowing to the chair, while the email is flowing to the owner of the password.

AirBnB models a leak in the AirBnB website [Voss 2016]. The website redacts phone numbers from user messages (presumably to keep people from going around the site), but phone numbers appear unredacted in message previews. In LIFTY we model the AirBnB messaging system by designating the message text visible only to its sender and the site administrator, and introducing a special redaction function scrubPhoneNumbers, whose result is additionally visible to the message recipient. With these policies, whenever a message is displayed to the recipient, LIFTY inserts a check whether they are the administrator, and otherwise redacts the text with scrubPhoneNumbers.

Instagram is inspired by several reported cases, where sensitive social network data was revealed through recommendation algorithms [Hill 2017; Yang 2017]. In particular, if an Instagram account is private, their photos and “following” relations are supposedly only visible to their followers (which have to be approved by the user). Yet, journalist Ashley Feinberg was able to identify the private Instagram account of the former FBI director James Comey, because Instagram mistakenly revealed that James was followed by his son Brien (whose account is public). In LIFTY, we model the Instagram “following” relation using a getter

```plaintext
inline canSee ds x y = x = y ∨ isPublic ds y ∨ y in following ds x
getIsFollowing :: ds: Store → who: User → whom: User →
  TI {Bool | v = (whom in following ds who)} <->canSee ds _ who ∧ canSee ds _ whom
```

whose policy requires that both accounts be visible to the viewer. When the recommendation system attempts to retrieve all accounts followed by Brien Comey, LIFTY injects a check that those accounts be visible to the viewer, and otherwise replaces the true value of “is following” with false.

6.2 Case Studies

We use LIFTY to implement three larger case studies: a conference manager and a course manager, both based on the case studies for the policy-agnostic Jacqueline system [Yang et al. 2016a], and a health portal based on the HealthWeb example from Fine [Swamy et al. 2010].

Conference manager. We implemented two versions of a basic academic conference manager: one where the programmer enforces the policies by hand (and LIFTY only verifies correctness) and one where the programmer omits all policy-enforcing code (and LIFTY is responsible for injecting leak patches). The former version contains 247 lines of LIFTY code while the latter contains 216; both systems share 364 lines of Haskell code that implement non-security-critical functionality. The manager handles confidentiality policies for user profiles, submissions, and reviews, and enforces policies such as: “a user profile is only visible to that user and the conference chair” or “the list of PC members conflicted with a submission is only visible to PC members who are not conflicted”. The list of controllers for this and the other two case studies is provided in Tab. 2 (together with their sizes in AST nodes).

While Jacqueline only supports constant default values, we decided to deviate from the original system to experiment with nontrivial redaction functions. In our version, reviewer names that are hidden for any reason are displayed as “Reviewer A”, “Reviewer B”, etc., following common convention. This is implemented by representing each reviewer entry as a pair of (index, name), where the redaction replaces name with “Reviewer x” according to index.

Course manager. We implemented a system for sending grades to students based on their course enrollment and assignment status. An example policy is that a student can see their own scores, whereas instructors can see scores for all of their students.

Health portal. Based on the HealthWeb case study from [Swamy et al. 2010], we implemented a system that supports viewing and searching over health records. This case study is interesting because of the complexity of the associated policies. For example, the policy that guards patients associated with health records states that the viewer must be the author of the record or the patient; otherwise non-withheld records can be viewed by a doctor, but psychiatric records can only be viewed by the doctor actually treating the patient:

\[
\text{inline recordPolicy w v rid = v = recordAuthor w rid} \lor \ v = \text{recordPatient w rid} \lor \\
\neg(\text{shouldWithhold w rid}) \land \text{isDoctor w v} \land \\
(\text{isTreating w v (recordPatient w rid)} \lor \neg(\text{recordIsPsychiatric w rid})))
\]

getRecordPatient :: ds: Store → rid: RecordId
→ TI {User | v = recordPatient ds rid} <recordPolicy ds 0 rid>

As a result, the generated policy enforcement code for this study is significantly larger than the original program, and takes twice as long to generate as in the conference manager.

6.3 Performance Statistics

We show compilation times for the microbenchmarks in Tab. 1, and for the case studies in Tab. 2. We break them down into fault localization (including type checking) and patch synthesis. LIFTY was able to patch each of the microbenchmarks in under two second. For each of the three case studies, LIFTY takes 30–60 seconds. Interestingly, the version of the conference manager with manual policy enforcement takes longer to verify than the leaky version takes to repair (64 vs 32
seconds). This is because verification times scale non-linearly (roughly quadratically) with the size of a controller, hence extending the controller with several policy checks may increase its verification time significantly. On the other hand, when generating leak patches automatically, Lifty verifies them independently from the rest of the controller; this is possible because, as we explained in Sec. 5, Lifty makes a simplifying assumption that the rest of the code cannot rely on the functional properties of the patch.

**Scalability.** Note that Lifty verifies and patches each top-level function in a program completely independently. Moreover, unlike prior work on program repair, patch synthesis proceeds independently for multiple leaks inside one function. For a stress test, we created a benchmark that performs $N$ reads (of the same field, for convenience) and then a `print` to an arbitrary user. Lifty’s job is to patch each of the $N$ leaks. We have confirmed that patch generation time is linear in $N$ (verification is still quadratic in the size of the function, and dominates the compilation time in this case); you can find the results of this experiment in Appendix B.

### 6.4 Quality of Patches

We compared the two versions of our conference manager (Tab. 2). The column “Original” shows the size of the code, in AST nodes, without any policy enforcement. We also show the cumulative size of policy-enforcing code, both hand-written and generated by Lifty. The size of policy predicates is given as “Policy size”. Note that the size of policy-enforcing code often approaches or exceeds the size of the core functionality, which motivates the Lifty repair engine as an approach to reducing the programmer burden. Manual inspection reveals that while the two versions of policy-enforcing code are syntactically different, they differ in neither functionality nor performance.

### 7 RELATED WORK

Lifty builds upon several lines of prior work, most notably in static information flow control, sound program synthesis and repair, and type-directed coercion insertion. Each of these areas has a rich history, but until now they have developed relatively independently.

#### 7.1 Information Flow Control

The Lifty type system builds upon a long history of work in language-based information flow control [Sabelfeld and Myers 2003]. Though we (indirectly) borrow some ideas from dynamic IFC systems—in particular Hails/LIO [Giffin et al. 2012; Stefan et al. 2017, 2011b] and Jeeves [Austin et al. 2013; Yang et al. 2016a, 2012]—Lifty enforces security policies using an information-flow type system. We see our work as complimentary to previous efforts on static information flow type systems. For example, Jif [Myers 1999], Fabric [Arden et al. 2012; Liu et al. 2009] and Paragon [Broberg et al. 2017] have been used to enforce IFC for Java programs, FlowCaml [Pottier and Simonet 2002] for OCaml, and SLIO [Buiras et al. 2015], MAC [Vassena et al. 2018], and others [Devriese and Piessens 2011; Hughes 2000; Li and Zdancewic 2006; Russo 2015; Russo et al. 2008b] for Haskell. To our knowledge, Lifty is the first system to encode IFC into the framework of liquid types—and while our implementation is for Synquid, we think Lifty can be similarly be implemented in other languages with liquid types (e.g. LiquidHaskell [Vazou et al. 2014a]).

Our IFC encoding shares some similarities to SLIO [Buiras et al. 2015], Fine [Chen et al. 2010; Swamy et al. 2010] and F* [Swamy et al. 2011]. Like Lifty, all three use a monadic encoding of information flow; many others share a similar encoding [Crary et al. 2005; Li and Zdancewic 2006; Russo 2015; Russo et al. 2008a; Vassena and Russo 2016], going back as far as the Dependency Core Calculus [Abadi et al. 1999]. Fine [Chen et al. 2010; Swamy et al. 2010], F* [Swamy et al. 2011], and others [Lourenço and Caires 2014, 2015] additionally support value-dependent security
types. The key difference is that our system uses (SMT-decidable) predicates as security labels, which supports (1) a direct encoding of Hails- and Jeeves-like policies, and (2) fully automatic verification and fault localization, crucial for type-targeted synthesis. UrFlow [Chlipala 2010] is the only automated verification system that supports a similar flavor of policies, but it does not provide a sound treatment of self-referential policies. More importantly, none of these approaches address the issue of or programmer burden: they simply prevent unsafe programs from compiling, but do not help programmers write policy-enforcing code.

Our policies are closer to policies used in IFC web frameworks (e.g., Hails [Giffin et al. 2012] and Jacqueline [Yang et al. 2016b]) than most IFC systems. Indeed, most IFC systems track the flow of information by associating labels with data and thus need to keep labels simple to be efficient. While existing label models can be used to encode web application policies [Montagu et al. 2013; Myers and Liskov 2000; Stefan et al. 2011a], high-level declarative policies like Lifty’s are usually instantiated to these labels. Jeeves [Yang et al. 2012] and Nexus [Sirer et al. 2011] are exceptions to these—they respectively encode policies in SMT-decidable and first-order logics—but enforce these policies at run time. Beyond runtime overhead, such rich policies are also harder to debug at runtime—e.g. in Jeeves this is the case because the runtime replaces sensitive values with default values when the policy is not satisfied.

7.2 Program Synthesis and Repair
Lifty is related to sound program synthesis techniques [Alur et al. 2017; Kneuss et al. 2013; Kuncak et al. 2010; Manna and Waldinger 1980; Polikarpova et al. 2016], which take a formal specification as input and synthesize a provably correct program. These techniques, however, generate programs from scratch, from end-to-end functional specifications, while Lifty injects code into an existing program based on the cross-cutting concern of information flow.

Our problem statement is similar to that of sound program repair [Kneuss et al. 2015], but as we explain in Sec. 2, in the specific setting of policy enforcement, Lifty is able to infer a local specification for each patch, and synthesize all patches independently, which makes it more scalable. There has been prior work on program repair for security concerns [Fredrikson et al. 2012; Ganapathy et al. 2006; Harris et al. 2010; Son et al. 2013], but it does not involve reasoning about expressive information-flow policies, and hence, both the search space for patches and their verification is much less complex.

7.3 Type Coercions and Type Error Localization
Our use of type errors to target program rewriting resembles type-directed coercion insertion [Swamy et al. 2009]; in particular, their coercion insertion and coercion generation mechanisms are similar to our fault localization and patch synthesis, respectively, and their coercion set is similar to our set $R\!$ of redaction functions. The Lifty type system, however, is far more complex than the type systems explored in that work. In particular, the combination of polymorphism and subtyping complicates type error localization (since there are many valid type derivations), while refinements complicate coercion generation (which becomes a refinement type inhabitation problem).

Hybrid type checking [Knowles and Flanagan 2010] can be viewed as coercion insertion for refinement types. In fact, their coercions also amount to wrapping the original value in a conditional, however, in their case both the guard and the alternative branch are straightforward.

Existing work on type error localization for expressive types systems [Loncaric et al. 2016; Seidel et al. 2017; Zhang et al. 2015] is in a more general—yet more forgiving—context of giving feedback to programmers. Our localization technique (removing constraints that make the system unsatisfiable) is similar to [Loncaric et al. 2016], but for our specific purpose we have more information to decide between possible error locations.


Chelsea Voss. 2016. private email communication.


Liquid Information Flow Control

Evaluation

\[
\begin{align*}
\text{RET} & : \langle \Sigma \mid t \rangle \rightarrow \langle \Sigma' \mid t' \rangle \\
\text{BIND} & : \langle \Sigma \mid \text{bind} \ t_1 \ t_2 \rangle \rightarrow \langle \Sigma' \mid t_1 \ t_2' \rangle \\
\text{DOWN} & : \langle \Sigma \mid \text{down} \ t \rangle \rightarrow \langle \Sigma' \mid \text{TIO} \ b \rangle \\
\text{GET} & : \langle \Sigma \mid \text{get} \ \ell \rangle \rightarrow \langle \Sigma \mid \text{TIO} \ \Sigma[\ell] \rangle \\
\text{SET} & : \langle \Sigma \mid \text{set} \ b \rangle \rightarrow \langle \Sigma \mid \ell := b \mid \text{TIO} \ () \rangle \\
\text{IF-TRUE} & : \langle \Sigma \mid \text{if} \ \text{True} \ \text{then} \ t_1 \ \text{else} \ t_2 \rangle \rightarrow \langle \Sigma \mid t_1 \rangle \\
\text{IF-FALSE} & : \langle \Sigma \mid \text{if} \ \text{False} \ \text{then} \ t_1 \ \text{else} \ t_2 \rangle \rightarrow \langle \Sigma \mid t_2 \rangle \\
\text{APP} & : \langle \Sigma \mid (\lambda x. \ t_1) \ t_2 \rangle \rightarrow \langle \Sigma \mid [x := t_2] \ t_1 \rangle \\
\text{CTX} & : \langle \Sigma \mid \text{C}[t] \rangle \rightarrow \langle \Sigma \mid \text{C}[t'] \rangle \\
\end{align*}
\]

where \( C ::= [] \mid C \cdot \mid \text{if} \ C \ \text{then} \ t_1 \ \text{else} \ t_2 \mid \text{set} \ C \)

Fig. 16. \( \lambda^L \) operational semantics.

A. THE LANGUAGE \( \lambda^L \)

A.1 Operational Semantics of \( \lambda^L \)

The full operational semantics of \( \lambda^L \) is given in Fig. 16.

A.2 The \( \lambda^L \) Type System

The full static semantics of \( \lambda^L \) is given in Fig. 17.

A.3 Non-interference

Instrumented Semantics. The full instrumented operational semantics is given in Fig. 18.

From Instrumented Semantics to LIO. To formalize the non-interference guarantee, we extend our calculus with erased terms, denoted with \( \bullet \). We define the erasure function on stores and instrumented configurations as follows:

\[
\varepsilon_\ell(\Sigma)[\ell'] = \begin{cases} 
\Sigma[\ell'] & \ell' \subseteq \ell \\
\bullet & \text{otherwise}
\end{cases}
\]

\[
\varepsilon_\ell(\langle \Sigma, \ell_c \mid t \rangle) = \begin{cases} 
\langle \varepsilon_\ell(\Sigma), \ell_c \mid t \rangle & \ell_c \subseteq \ell \\
\langle \bullet, \bullet \mid \bullet \rangle & \text{otherwise}
\end{cases}
\]

We say that two stores are \( \ell \)-equivalent (\( \Sigma_1 \approx_\ell \Sigma_2 \)) iff \( \varepsilon_\ell(\Sigma_1) = \varepsilon_\ell(\Sigma_2) \). Similarly, we say that two instrumented configurations are \( \ell \)-equivalent (\( k_1 \approx_\ell k_2 \)) iff \( \varepsilon_\ell(k_1) = \varepsilon_\ell(k_2) \)

Lemma 6 (Noninterference of instrumented semantics). If \( k_1 \approx_\ell k_2 \) and \( k_1 \rightarrow^* k'_1 \) and \( k_2 \rightarrow^* k'_2 \), then \( k'_1 \approx_\ell k'_2 \)

This lemma follows directly from Theorem 1 in [Stefan et al. 2017], if we implement \text{get}, \text{set}, and \text{downgrade} as specified in Sec. 4.
Subtyping \( \Gamma \vdash T <: T' \) \( \Gamma \vdash B <: B' \)

\[\vdash B <: B' \quad \Gamma \vdash r \Rightarrow r' \]

\[\Gamma \vdash \{B \mid r\} <: \{B' \mid r'\} \]

\[\vdash T_1 <: T_2 \quad \Gamma \vdash \ell_1 \subseteq \ell_2 \quad \Gamma \vdash \ell_2' \subseteq \ell_1' \]

\[\vdash \tau_1 T_1 (\ell_1, \ell_1') <: \tau_1 T_2 (\ell_2, \ell_2') \]

\[\vdash B <: \]

Typing \( \Gamma \vdash t :: T \)

\[\vdash \text{true} :: \{\text{Bool} \mid \nu\} \]

\[\vdash \text{false} :: \{\text{Bool} \mid \neg \nu\} \]

\[\vdash () :: () \mid \text{true} \]

\[\vdash \lambda x : T \cdot t :: T_1 \rightarrow T_2 \quad \Gamma \vdash \lambda x : T \cdot t :: T_1 \rightarrow T_2 \]

\[\vdash \text{if-true} \quad \Gamma \vdash \text{if-false} \quad \Gamma \vdash \text{if-false} \]

\[\vdash \text{if-true} \quad \Gamma \vdash \text{if-false} \quad \Gamma \vdash \text{if-true} \]

\[\vdash \text{bind} \quad \Gamma \vdash \text{get} \quad \Gamma \vdash \text{set} \]

\[\vdash \text{downgrade} \]

\[\vdash \text{return} \quad \Gamma \vdash \text{return} \]

Simulation. We say that \( t \) is a pure term if its top-level constructor is none of \( \tau \), \( \text{get} \), \( \text{set} \), \( \text{return} \), \( \text{bind} \), or \( \text{downgrade} \).

Lemma 7 (Pure simulation). If \( \Gamma \vdash t :: T \), \( t \) is a pure term, and \( \langle \Sigma \mid t \rangle \rightarrow \langle \Sigma' \mid t' \rangle \), then \( \Sigma' = \Sigma \) and for any \( \ell_c \):

\( 1 \) \( \langle \Sigma, \ell_c \mid t \rangle \rightarrow \langle \Sigma, \ell_c \mid t' \rangle \),

\( 2 \) and \( \Gamma \vdash t' :: T \).

Proof. Since \( t \) steps, it must be either a conditional or an application. \( 1 \) follows trivially by comparing rules \( \text{IF-TRUE}, \text{IF-FALSE}, \text{APP}, \text{CTX} \) with \( \text{I-IF-TRUE}, \text{I-IF-FALSE}, \text{I-APP}, \text{I-CTX} \) respectively. \( 2 \) follows from preservation for the base language with refinement types [Vazou et al. 2014b].
The only possible evaluation step is \( \langle \ell \text{-get} \rangle \) and the new current label \( \ell \text{-get} \).

\[ \langle \Sigma, \ell_c \mid t \rangle \rightarrow \langle \Sigma', \ell'_c \mid t' \rangle \]

**Lemma 8** (Single-step simulation). If \( \Gamma \vdash t :: \text{TIO} \ T \langle \ell_1, \ell_o \rangle \) and \( \langle \Sigma \mid t \rangle \rightarrow \langle \Sigma' \mid t' \rangle \), then for any \( \ell_c \subseteq \ell_o \), there exists a new current label \( \ell'_c \) and a new type \( \text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle \) such that

1. \( \langle \Sigma, \ell_c \mid t \rangle \rightarrow \langle \Sigma', \ell'_c \mid t' \rangle \),
2. \( \ell'_c \subseteq \ell_c \cup \ell_i \),
3. \( \Gamma \vdash t' :: \text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle \)
4. \( \text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle < :: \text{TIO} \ T \langle \ell_1, \ell_o \rangle \),
5. and \( \ell'_c \subseteq \ell'_o \).

**Proof.** By induction on the derivation of \( \Gamma \vdash t :: \text{TIO} \ T \langle \ell_1, \ell_o \rangle \).

**Case T-GET:** The root of the derivation is:

\[ \text{T-GET} \quad \Gamma \vdash \text{get} \mathrel{:} \text{TIO} \text{Bool} \langle \ell, \top \rangle \]

The only possible evaluation step is \( \langle \Sigma \mid \text{get} \ell \rangle \rightarrow \langle \Sigma \mid \text{TIO} \Sigma[\ell] \rangle \) by rule GET. Let \( \ell_c \) be arbitrary, and the new current label \( \ell'_c = \ell \cup \ell_c \) and the new type be \( \text{TIO} \text{Bool} \langle \bot, \top \rangle \).

- By rule 1-GET: \( \langle \Sigma, \ell_c \mid \text{get} \ell \rangle \rightarrow \langle \Sigma, \ell_c \cup \ell_c \mid \Sigma[\ell] \rangle \), hence (1) holds.
- \( \ell_c \subseteq \ell \cup \ell_c \), hence (2) holds.
- By rule T-TIO: \( \Gamma \vdash \Sigma \ell \mid \text{TIO} \text{Bool} \langle \bot, \top \rangle \), hence (3) holds.
- By rule <:TIO: \( \text{TIO} \text{Bool} \langle \bot, \top \rangle < :: \text{TIO} \text{Bool} \langle \ell, \top \rangle \), hence (4) holds.
- \( \ell_c \subseteq \ell \cup \ell_c \subseteq \top \), hence (5) holds.

**Case T-SET:** The root of the derivation is:

\[ \text{T-SET} \quad \Gamma \vdash t :: B \]

\[ \Gamma \vdash \text{set} \mathrel{:} \text{TIO} \text{Bool} \langle \bot, \top \rangle \]
Pick some \( \ell_c \subseteq \ell \).

Let us first consider the case when \( t \) is a value \( b \). The only possible evaluation step is \( \langle \Sigma \mid \text{set } \ell \ b \rangle \rightarrow \langle \Sigma[\ell := b] \mid \text{TIO }() \rangle \) by rule set. Let the new current label \( \ell' = \ell_c \) and the new type be \( \text{TIO }() \langle \bot, \top \rangle \).

- Since \( \ell_c \subseteq \ell \), then by rule \text{r-set}: \( \langle \Sigma, \ell_c \mid \text{set } \ell \ b \rangle \rightarrow \langle \Sigma[\ell := b] \mid \text{TIO }() \rangle \), hence (1) holds.
- \( \ell_c \subseteq \ell_c \subseteq \bot \), hence (2) holds.
- By rule \text{T-TIO}: \( \Gamma \vdash \text{TIO }() \langle \bot, \top \rangle \), hence (3) holds.
- By rule \text{<-TIO}: \( \text{TIO }() \langle \bot, \top \rangle \emptyset \vdash \text{TIO }() \langle \bot, \top \rangle \), hence (4) holds.
- \( \ell_c \subseteq \top \), hence (5) holds.

Now consider the case when \( t \) is not a value. Then the only possible evaluation step is \( \langle \Sigma \mid \text{set } \ell \ t \rangle \rightarrow \langle \Sigma \mid \text{set } \ell \ t' \rangle \) as long as \( \langle \Sigma \mid \text{set } \ell \ t \rangle \rightarrow \langle \Sigma \mid \text{set } \ell \ t' \rangle \) by rule \text{ctx}. Since \( \Gamma \vdash t :: B \), then \( t \) must be a pure term by inspection of the typing rules. Hence by Lemma 7, we get \( \langle \Sigma, \ell_c \mid t \rangle \rightarrow \langle \Sigma, \ell_c \mid t' \rangle \) (a) and \( \Gamma \vdash t' :: B \) (b). Let the new current label and the new type stay the same, i.e. \( \ell_c \) and \( \text{TIO }() \langle \bot, \top \rangle \).

- By (a) and rule \text{r-ctx}: \( \langle \Sigma, \ell_c \mid \text{set } \ell \ t \rangle \rightarrow \langle \Sigma, \ell_c \mid \text{set } \ell \ t' \rangle \), hence (1) holds.
- \( \ell_c \subseteq \ell_c \subseteq \bot \), hence (2) holds.
- By (b) and rule \text{r-set}: \( \Gamma \vdash \text{set } \ell \ t' :: \text{TIO }() \langle \bot, \top \rangle \), hence (3) holds.
- By rule \text{<-TIO}: \( \text{TIO }() \langle \bot, \top \rangle \emptyset \vdash \text{TIO }() \langle \bot, \top \rangle \), hence (4) holds.
- \( \ell_c \subseteq \top \), hence (5) holds.

**Case \text{t-ret}:** The root of the derivation is:

\[
\begin{array}{c}
\text{t-ret} \\
\Gamma \vdash t :: T \\
\Gamma \vdash \text{return } t :: \text{TIO } T \langle \bot, \top \rangle
\end{array}
\]

The only possible evaluation step is \( \langle \Sigma \mid \text{return } t \rangle \rightarrow \langle \Sigma \mid \text{TIO } t \rangle \). Pick any \( \ell_c \), and let the new current label and the new type be unchanged, i.e. \( \ell_c' = \ell_c \) and \( \text{TIO } T \langle \bot, \top \rangle \).

- By rule \text{r-ret}: \( \langle \Sigma, \ell_c \mid \text{return } t \rangle \rightarrow \langle \Sigma, \ell_c \mid \text{TIO } t \rangle \), hence (1) holds.
- \( \ell_c \subseteq \ell_c \subseteq \bot \), hence (2) holds.
- By (a) and rule \text{T-TIO}: \( \Gamma \vdash \text{TIO } T :: \text{TIO } T \langle \bot, \top \rangle \), hence (3) holds.
- By rule \text{<-TIO}: \( \text{TIO } B \langle \bot, \top \rangle \emptyset \vdash \text{TIO } B \langle \bot, \top \rangle \), hence (4) holds.
- \( \ell_c \subseteq \top \), hence (5) holds.

**Case \text{t-bind}:** The root of the derivation is:

\[
\begin{array}{c}
\text{t-bind} \\
\Gamma \vdash t_1 :: \text{TIO } T_1 \langle \ell_1, \ell_1' \rangle \\
\Gamma \vdash t_2 :: \text{TIO } T_2 \langle \ell_2, \ell_2' \rangle \\
\Gamma \vdash \ell_1 \subseteq \ell_2 \\
\Gamma \vdash \text{bind } t_1 t_2 :: \text{TIO } T_2 \langle \ell_1 \cup \ell_2, \ell_1' \cap \ell_2' \rangle
\end{array}
\]

The only possible evaluation step is \( \langle \Sigma \mid \text{bind } t_1 t_2 \rangle \rightarrow \langle \Sigma' \mid t_2 t_1' \rangle \), so it must be that \( \langle \Sigma \mid t_1 \rangle \rightarrow^* \langle \Sigma' \mid \text{TIO } t_1' \rangle \) (d). Pick some \( \Gamma \vdash \ell_c \subseteq \ell_1' \cap \ell_2 \). Now let the new type be \( \text{TIO } T_2 \langle \ell_2, \ell_2' \rangle \).

From (a) and (d) by Lemma 9, there exists \( \ell_c' \) such that \( \langle \Sigma, \ell_c \mid t_1 \rangle \rightarrow^* \langle \Sigma', \ell_c' \mid \text{TIO } t_1' \rangle \) (e), \( \Gamma \vdash \ell_c' \subseteq \ell_c \cup \ell_1 \) (f), and \( \Gamma \vdash \text{TIO } T_1 \langle \bot, \top \rangle \) (g) (using rule \text{t-<}). We will pick \( \ell_c' \) as the new current label.

- By rule \text{r-bind} and (e): \( \langle \Sigma, \ell_c \mid \text{bind } t_1 t_2 \rangle \rightarrow \langle \Sigma', \ell_c' \mid t_2 t_1' \rangle \), hence (1) holds.
- From (f) we conclude that \( \ell_c' \subseteq \ell_c \cup \ell_1 \cup \ell_2 \), hence (2) holds.
- From (g) by rule \text{T-TIO} we have \( \Gamma \vdash t_1' :: T_1 \). From that and (b) by rule \text{T-App} we have:
  \( \Gamma \vdash t_2 t_1' :: \text{TIO } T_2 \langle \ell_2, \ell_2' \rangle \), hence (3) holds.
- By rule \text{<-TIO}: \( \text{TIO } T_2 \langle \ell_2, \ell_2' \rangle \emptyset \vdash \text{TIO } T_2 \langle \ell_1 \cup \ell_2, \ell_1' \cap \ell_2' \rangle \), hence (4) holds.
The only possible uninstrumented evaluation step is \( \ell TIO Case \langle \ell r \rangle \).

\[ \begin{aligned}
\ell TIO Case 0: & \text{ Trivial with Lemma 9 and as the new type we pick } \sqsubseteq \ell'(0) \text{ and } \Gamma (3) \langle (1) \ell (2) \rangle.
\end{aligned} \]

Proof. By induction on the number of steps.

Case T-DOWN: The root of the derivation is:

\[ \begin{aligned}
\Gamma t TIO & \{0bool | v \Rightarrow r\} \langle \ell \sqcup \lambda x.r, \ell' \rangle \quad (a)
\end{aligned} \]

The only possible uninstrumented evaluation step is \( \langle \Sigma | downgrade \ell t \rangle \rightarrow \langle \Sigma' | TIO b \rangle \), so it must be that \( \langle \Sigma | t \rangle \rightarrow^* \langle \Sigma' | TIO b \rangle \) (b).

Pick some \( \ell_c \subseteq \ell' \). As the new label we pick \( \ell_c' = \ell \sqcup \ell_c \) and as the new type we pick TIO Bool \( \langle \bot, \ell' \rangle \).

To prove that the downgrade steps to the same term under the instrumented semantics, we consider two cases. First, if \( \ell_c'' \subseteq \ell \sqcup \ell_c \), then by rule i-DOWN-1, we have \( \langle \Sigma, \ell_c | downgrade \ell t \rangle \rightarrow \langle \Sigma', \ell_c \sqcup \ell | TIO b \rangle \), hence (1) holds in this case.

Otherwise, by rule i-DOWN-2, we have \( \langle \Sigma, \ell_c | downgrade \ell t \rangle \rightarrow \langle \Sigma', \ell_c \sqcup \ell | TIO b \rangle \), hence (3) holds.

From (e) and (b) by Lemma 9, there exists \( \ell_c'' \) such that \( \langle \Sigma, \ell_c | t \rangle \rightarrow^* \langle \Sigma', \ell_c'' | TIO b \rangle \) (d), \( \ell_c'' \subseteq \ell_c \sqcup \ell \sqcup \lambda x.r \) (d), and \( \Gamma \vdash b : \{0bool | v \Rightarrow r\} \) (e) (using subtyping and t-TIO).

From (c) and typing rules we know that \( b \) is necessarily either True or False; so let us assume \( b = True \) and arrive at a contradiction.

If \( b = True \), then by rule t-TRUE, \( \Gamma \vdash b : \{0bool | v\} \); then from (c) and subtyping we get \( \Gamma \vdash \{0bool | v\} : \{0bool | v \Rightarrow r\} \), and hence \( \Gamma \vdash v \Rightarrow r \). Now from (d) and \( \ell_c'' \not\subseteq \ell \sqcup \ell_c \) we know that necessarily \( \Gamma \vdash \ell_c'' \subseteq \lambda x.r \), in other words \( \Gamma, x : r \Rightarrow \ell_c'' \). Hence we have \( \Gamma, x : r \Rightarrow \ell_c'' \). However, \( \ell_c'' \) cannot mention \( v \), hence the only way this could hold is if \( \ell_c'' = \lambda x.true = \bot \), which contradicts the assumption that \( \ell_c'' \not\subseteq \ell \sqcup \ell_c \).

\[ \begin{aligned}
\cdot \ell \sqcup \ell_c \subseteq \ell_c, \text{ hence (2) holds.}
\cdot \text{From (c) by rule t-TIO and subtyping: } \Gamma \vdash TIO b : \{0bool | \bot, \ell, \ell'\}, \text{ hence (3) holds.}
\cdot \text{By rule } \leq,TIO: TIO Bool \langle \bot, \ell, \ell' \rangle \leq \langle \bot, \bot, T \rangle, \text{ hence (4) holds.}
\cdot \ell \sqcup \ell_c \subseteq T, \text{ hence (5) holds.}
\end{aligned} \]

\[ \square \]

Lemma 9 (Multi-step simulation). If \( \Gamma \vdash t : TIO T \langle \ell_1, \ell_0 \rangle \) and \( \langle \Sigma | t \rangle \rightarrow^* \langle \Sigma' | t' \rangle \), then for any \( \ell_c \subseteq \ell_o \), there exists a new current label \( \ell_c' \) and a new type \( TIO T' \langle \ell_1', \ell_0' \rangle \) such that

\[ \begin{aligned}
(1) & \langle \Sigma, \ell_c | t \rangle \rightarrow^* \langle \Sigma', \ell_c' | t' \rangle, \\
(2) & \ell_c' \subseteq \ell_c \sqcup \ell_1, \\
(3) & \Gamma \vdash t' \in TIO T' \langle \ell_1', \ell_0' \rangle \\
(4) & TIO T' \langle \ell_1', \ell_0' \rangle \leq TIO T \langle \ell_1, \ell_o \rangle, \\
(5) & \text{and } \ell_c' \subseteq \ell_o'.
\end{aligned} \]

Proof. By induction on the number of steps.

Case 0: Trivial with \( \ell_c' = \ell_c \) and \( TIO T' \langle \ell_1', \ell_0' \rangle = TIO T \langle \ell_1, \ell_0 \rangle \).

Case \( n + 1 \). Let \( \langle \Sigma | t \rangle \rightarrow \langle \Sigma'' | t'' \rangle \rightarrow^* \langle \Sigma' | t' \rangle \). Then by Lemma 8, there exist \( \ell_c'' \) and TIO \( T'' \langle \ell_1'', \ell_0'' \rangle \), such that

\[ \begin{aligned}
\cdot \langle \Sigma, \ell_c | t \rangle \rightarrow \langle \Sigma'', \ell_c'' | t'' \rangle \quad (a), \\
\ell_c'' \subseteq \ell_c \sqcup \ell_1 \quad (b), \\
\Gamma \vdash t'' \in TIO T'' \langle \ell_1'', \ell_0'' \rangle \quad (c), \\
TIO T'' \langle \ell_1'', \ell_0'' \rangle \leq TIO T \langle \ell_1, \ell_0 \rangle \quad (d), \\
\text{and } \ell_c'' \subseteq \ell_o'. \quad (e).
\end{aligned} \]
Now thanks to (c) and (e), we can apply the IH to $\langle \Sigma'' | t'' \rangle \longrightarrow^n \langle \Sigma' | t' \rangle$ and get a new label $\ell'_c$ and type $\text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle$ such that:

- $\langle \Sigma'', \ell''_c | t'' \rangle \longrightarrow^n \langle \Sigma', \ell'_c | t' \rangle$ (i),
- $\ell'_c \subseteq \ell''_c \cup \ell''_o$ (g),
- $\Gamma \vdash t' :: \text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle$ (h),
- $\text{TIO} \ T' \langle \ell'_1, \ell'_o \rangle \prec \text{TIO} \ T' \langle \ell'_1, \ell''_o \rangle$ (i),
- and $\ell'_c \subseteq \ell'_o$ (j).

Now (1) follows by definition of $\longrightarrow^n$; (3) follows directly from (h); (4) follows from (d) and (i) by transitivity of subtyping; (5) follows directly from (j). What remains to show is (2).

We have (b) $\ell''_c \subseteq \ell_c \cup \ell_i$ and (g) $\ell'_c \subseteq \ell''_c \cup \ell''_o$, so we can prove (2) by calculating:

$\ell'_c \subseteq \ell''_c \cup \ell_i \subseteq \ell''_c \subseteq \ell_c \cup \ell_i$

Now (2) follows by definition of $\longrightarrow^n$.

**Noninterference.** Putting it all together, we obtain the following non-interference theorem:

**Theorem 3** (Noninterference for $\lambda^t$). Take two $\ell$-equivalent stores $\Sigma_1 \approx^t \Sigma_2$. Then for any terms $t$ such that $\Gamma \vdash t :: \text{TIO} \ T \langle \ell, _\_ \rangle$, $\langle \Sigma_1 | t \rangle \longrightarrow^n \langle \Sigma'_1 | t'_1 \rangle$, and $\langle \Sigma_2 | t \rangle \longrightarrow^n \langle \Sigma'_2 | t'_2 \rangle$, we have $t'_1 = t'_2$ and $\Sigma'_1 \approx^t \Sigma'_2$.

**Proof.** Since $\langle \Sigma_1 | t \rangle \longrightarrow^n \langle \Sigma'_1 | t'_1 \rangle$ and $\Gamma \vdash t :: \text{TIO} \ T \langle \ell, _\_ \rangle$, then by Lemma 9 we have $\langle \Sigma_1, \perp | t \rangle \longrightarrow^n \langle \Sigma'_1, \ell'_c | t'_1 \rangle$ and $\ell'_c \subseteq \ell$. Similarly for the other store we get: $\langle \Sigma_2, \perp | t \rangle \longrightarrow^n \langle \Sigma'_2, \ell'_c | t'_2 \rangle$ and $\ell'_c \subseteq \ell$. Because $\langle \Sigma_1, \perp | t \rangle \approx^t \langle \Sigma_2, \perp | t \rangle$, then by Lemma 6 we get: $\langle \Sigma'_1, \ell'_c | t'_1 \rangle \approx^t \langle \Sigma'_2, \ell'_c | t'_2 \rangle$. Now since $\ell'_c \subseteq \ell$ and $\ell'_c \subseteq \ell$, then by definition of configuration erasure we get that $\Sigma'_1 \approx^t \Sigma'_2$ and $t'_1 = t'_2$.

**B SCALABILITY**

Fig. 19 shows the dependency of verification and patch generation times on the number of leaky input actions to patch.

**C MICROBENCHMARKS**

Below you can find the code for all our microbenchmarks with LIFTY-generated patches in gray.
**Benchmark 1 (EDAS):** Show data for paper p to client.

1. -- | Conference phase (public)
2. predicate phase :: Store → Phase
3. getPhase :: ds: Store → TIO {Phase | \( \nu = \) phase ds} \(<\text{True}>\) \(<\text{False}>\)
4. -- | Paper title (public)
5. getPaperTitle :: ds: Store → p: PaperId → TIO String \(<\text{True}>\) \(<\text{False}>\)
6. -- | Paper status (only visible when phase is done)
7. getPaperDecision :: ds: Store → p: PaperId → TIO Decision \(<\text{phase ds = Done}>\) \(<\text{False}>\)
8. -- | Paper session (public)
9. getPaperSession :: ds: Store → p: PaperId → TIO String \(<\text{True}>\) \(<\text{False}>\)
10. redact {NoDecision}

11. showSession :: Store → User → PaperId → TIO Unit \(<\text{False}>\) \(<\text{True}>\)
12. showSession = \(\lambda\) ds . \(\lambda\) client . \(\lambda\) p .
13. do
14. t ← getPaperTitle ds p
15. dec ← do
16. g0 ← downgrade (do
17. x5 ← getPhase ds
18. return (eq Done x5))
19. if g0
20. then getPaperDecision ds p
21. else return NoDecision
22. if dec = Accepted
23. then do
24. ses ← getPaperSession ds p
25. print client (unwords [t, ses])
26. else print client (unwords [t, emptyString])

Benchmark 2 (EDAS-Multiple): Same as (EDAS), but multiple terms need to be patched.

```haskell
... -- as in EDAS
getPaperAuthors :: ds: Store → p: PaperId → TIO [User] {phase ds = Done} {False}

showSession :: Store → User → PaperId → TIO Unit {False} {True}
showSession = \ds . \client . \p .
do
t ← getPaperTitle ds p
auts ← do
g0 ← downgrade (do
   x5 ← getPhase ds
   return (eq Done x5))
if g0
   then getPaperAuthors ds p
   else return Nil
dec ← do
g1 ← downgrade (do
   x11 ← getPhase ds
   return (eq Done x11))
if g1
   then getPaperDecision ds p
   else return NoDecision
if dec = Accepted
   then do
      ses ← getPaperSession ds p
      print client (unwords [t, show auts, ses])
   else print client (unwords [t, show auts, emptyString])
```
**Benchmark 3 (EDAS-Self-Ref)**: Same as (EDAS-Multiple), but with a self-referential policy on authors.

... -- as in EDAS

-- | Paper authors (only visible to themselves or when phase is done)

**predicate paperAuthors :: Store → Map PaperId (Set User)**

**getPaperAuthors :: ds: Store → p: PaperId →**

TIO (List User | ν in (paperAuthors ds)[p]) | elems ν = (paperAuthors ds)[p]

<{} in (paperAuthors ds)[p] | phasε ds = Done>> <False>)

**showSession :: Store → User → PaperId → TIO Unit <False> <True>**

**showSession = λ ds . λ client . λ p .**

**do**

t ← getPaperTitle ds p

**auts ← do**

g0 ← downgrade (do

   x5 ← getPhase ds

   return (eq Done x5))

**g1 ← downgrade (do**

   x10 ← getPaperAuthors ds p

   return (elem client x10))

if g0 ∨ g1

then getPaperAuthors ds p

else return Nil

dec ← do

g2 ← downgrade (do

   x16 ← getPhase ds

   return (eq Done x16))

if g2

then getPaperDecision ds p

else return NoDecision

if dec = Accepted

then do

   ses ← getPaperSession ds p

   print client (unwords [t, show auts, ses])

else print client (unwords [t, show auts, emptyString])
**Benchmark 4 (Search):** Show client all their accepted papers. Repairs a leak through a filter.

```plaintext
... -- as in EDAS-Self-Ref

showMyAcceptedPapers :: Store → User → TIO Unit <False> <True>
showMyAcceptedPapers = λ ds . λ client .
  let isMyAccepted = λ p .
    downgrade (do
      auts ← getPaperAuthors ds p
      dec ← do
        g0 ← downgrade (do
          x5 ← getPhase ds
          return (eq Done x5))
        if g0 then getPaperDecision ds p
        else return NoDecision
      return ((elem client auts) ∧
        (dec = Accepted))) in
    do
      allPaperIDs ← getAllPaperIds ds
      paperIDs ← filterM isMyAccepted
        allPaperIDs
      titles ← mapM (λ p . getPaperTitle ds p) paperIDs
      print client (unlines titles)
```
**Benchmark 5** (Sort) Sort papers by their score, which is hidden from conflicted reviewers. Repairs a leak through the order of sorted submission. Contains a negative self-referential policy.

```
-- | Paper title (public)
getPaperTitle :: ds: Store → p: PaperId → TIO String <(True)> <(False)>

dPredicate paperConflicts :: Store → pid: PaperId
   → TIO (List User | elems ν = (paperConflicts ds)[pid])
<(<(λ.0 ∈ (paperConflicts ds)[pid]))> <(False)>

-- | Paper conflicts (public)
getPaperConflicts :: ds: Store → pid: PaperId
    -> TIO (List User | elems ν = (paperConflicts ds)[pid])
<(<(λ.0 ∈ (paperConflicts ds)[pid]))> <(False)>

-- | Paper score (only visible if not in conflict)
getPaperScore :: ds: Store → pid: PaperId → TIO Int
    <(<(λ.0 ∈ (paperConflicts ds)[pid]))> <(False)>

-- | All papers in the conference
getAllPaperIds :: ds: Store → TIO [PaperId] <(True)> <(False)>

sortPapersByScore :: Store → User → TIO Unit <(False)> <(True)>
sortPapersByScore = λ ds . λ client .
   do
      x1 ← do
         g0 ← downgrade (do
            x7 ← getPaperConflicts ds pid1
            return (not (elem client x7)))
         if g0
            then getPaperScore ds pid1
            else return zero
         x2 ← do
            g1 ← downgrade (do
            x15 ← getPaperConflicts ds pid2
            return (not (elem client x15)))
            if g1
               then getPaperScore ds pid2
               else return zero
            return (x1 ≤ x2) in
         do
            pids ← getAllPaperIds ds
            sortedPids ← sortByM cmpScore pids
            titles ← mapM (getPaperTitle ds) sortedPids
         print client (unlines titles)
```
Benchmark 6 (Broadcast): Send status notification to authors. Sending message to multiple viewers, viewers are sensitive.

1: Conference phase (public)
2  predicate phase :: Store → Phase
3  getPhase :: ds: Store → TIO {Phase | \( v = \text{phase } ds \)} <|True|> <|False|>
4  -- | Paper title (public)
5  getPaperTitle :: ds: Store → p: PaperId → TIO String <|True|> <|False|>
6  -- | Paper authors (only visible to themselves or when phase is done)
7  predicate paperAuthors :: Store → Map PaperId (Set User)
8  getPaperAuthors :: ds: Store → p: PaperId →
9    TIO {List {User | \( v \in (\text{paperAuthors } ds)[[p]] \) | \( \text{elems } v = (\text{paperAuthors } ds)[[p]] \)}
10   <|0 \in (\text{paperAuthors } ds)[[p]] \lor \text{phase } ds = \text{Done}|> <|False|>
11  -- | Paper status (only visible when phase is done)
12  getPaperDecision :: ds: Store → p: PaperId → TIO Decision <|\text{phase } ds = \text{Done}|> <|False|>
13  -- | Paper session (public)
14  getPaperSession :: ds: Store → p: PaperId → TIO String <|True|> <|False|>
15  redact {NoDecision}
16
17  notifyAuthors :: ds:Store → p:PaperId → TIO Unit <|False|> <|True>
18  notifyAuthors = \lambda ds . \lambda p .
19        do
20          status ← do
21              g0 ← downgrade (do
22                  x5 ← getPhase ds
23                  return (eq Done x5))
24
25              if g0
26                then getPaperDecision ds p
27                else return NoDecision
28              authors ← getPaperAuthors ds p
29              printAll authors (show status)
Liquid Information Flow Control

**Benchmark 7 (HotCRP):** HotCRP password leak: chair could see other people’s passwords in message preview.

```plaintext
-- | Mask a password
mask :: TIO Password <(False)> <(False)> → TIO Password <(True)> <(False)>
-- | User name (public)
getUserName :: ds: Store → u: User → TIO String <(True)> <(False)>
-- | User password (only visible to the user)
getUserPassword :: ds: Store → u: User → TIO Password <(_0 = u)> <(False)>
-- | PC chair (public)
predicate chair :: Store → User
getchair :: ds: Store → TIO {User | _v = chair ds} <(True)> <(False)>
redact {mask}

sendPasswordReminder :: Store → User → TIO Unit <False> <True>
sendPasswordReminder = λ ds . λ u .
do
    ch ← getChair ds
    preview ← liftM2 strcat
        (getUserName ds u) (liftM show
            (do
                g0 ← downgrade (return (eq ch
                                  u))
                if g0
                    then getUserPassword ds u
                    else mask (getUserPassword ds u)))
    print ch preview
    message ← liftM2 strcat
        (getUserName ds u) (liftM show (getUserPassword ds u))
    print u message
```
Benchmark 8 (AirBnB) AirBnB bug: they scrape phone numbers from user messages, but forgot to do so in the preview. This example features a custom redaction function that makes a message text visible to the recipient, but not completely public. It also showcases expressive functional reasoning with higher-order functions, since correctness depends on the argument of filterM.

```haskell
getAllMessageIDs :: ds :: Store → [MessageId]
-- | AirBnB admin
getAdmin :: ds :: Store → TIO {User | v = admin ds} <-{True}> <-{False}>
-- | Message sender
getSender :: ds :: Store → m :: MessageId
→ TIO {User | v = (sender ds)[m] ∧ v ≠ (recipient ds)[m]} <-{True}> <-{False}>
-- | Message recipient
getRecipient :: ds :: Store → m :: MessageId
→ TIO {User | v = (recipient ds)[m] ∧ v ≠ (sender ds)[m]} <-{True}> <-{False}>
-- | Message text (only visible to the admin and the sender)
getText :: ds :: Store → m :: MessageId
→ TIO String <-{_0 = admin ds ∨ _0 = (sender ds)[m]}<-{False}>
-- | Scrape phone numbers from the message, making it visible to the recipient
scrapePhoneNumbers :: ds :: Store → m :: MessageId
→ TIO String <-{_0 = admin ds ∨ _0 = (sender ds)[m]}<-{True}>
redact {scrapePhoneNumbers}

viewInbox :: Store → User → TIO Unit <-{False}> <-{True}>
viewInbox = λ ds . λclient .
  let isMyMessage = λ m . do
    to ← getRecipient ds m
    return (to = client) in
  do
    myMIDs ← filterM isMyMessage
    (getAllMessageIDs ds)
    messages ← mapM (λ m . do
      g0 ← downgrade (do
        x37 ← getAdmin ds
        return (eq client x37))
      if g0 then getText ds m
      else scrapePhoneNumbers ds m (getText ds m)) myMIDs
    print client (unlines messages)
```

Benchmark 9 (Instagram): The James Comey Instagram leak: the follow-relationships of private accounts leak through recommendation algorithms.

```liquid
getAllUsers :: ds: Store → [User]
-- | User name (visible to all)

predicate name :: Store → Map User String
getName :: ds: Store → u: User → TIO (String | ν = (name ds)[u]) <True> <False>
-- | Is user's account private? (visible to all)

predicate isPrivate :: Store → Map User Bool
getIsPrivate :: ds: Store → u: User → TIO (Bool | ν = (isPrivate ds)[u]) <True> <False>
-- | Who this user follows (for private accounts: only visible to followers)

predicate following :: Store → Map User (Set User)
getIsFollowing :: ds: Store → who: User → whom: User → TIO (Bool | ν = (whom in (following ds)[who]) <True> <False>

setIsFollowing :: ds: Store → who: User → whom: User → val: Bool → TIO (Bool | ν = (whom in (following ν)[who]) = val ∧ name ν = name ds ∧ isPrivate ν = isPrivate ds) <True> <False>

-- | Is account u public? Yes if it's not private
inline isPublic ds u = ¬(isPrivate ds)[u]
-- | Can x see y? Yes if they are the same, y is public, or x is following y
inline canSee ds x y = x = y ∨ isPublic ds y ∨ y in (following ds)[x]
```

do

ds' ← setIsFollowing ds client newFriend true

showRecommendations ds' client newFriend