



Liquid Information Flow Control

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

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.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → *Domain specific languages.*

Additional Key Words and Phrases: information flow control, liquid types, program synthesis

ACM Reference Format:
Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 105 (August 2020), 30 pages. https://doi.org/10.1145/3408987

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

Authors' addresses: Nadia Polikarpova, University of California, San Diego, USA, npolikarpova@eng.ucsd.edu; Deian Stefan, University of California, San Diego, USA, deian@cs.ucsd.edu; Jean Yang, Carnegie Mellon University, USA, jyang2@cs.cmu.edu; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Travis Hance, Carnegie Mellon University, USA, thance@cs.cmu.edu; Armando Solar-Lezama, Massachusetts Institute of Technology, USA, asolar@csail.mit.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/8-ART105
https://doi.org/10.1145/3408987

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 [Giffin et al. 2012] and Jacqueline [Yang et al. 2016] which separate the security policy from the application code and enforce the policy using dynamic information flow control (IFC). 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; alas, the IFC systems 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. 2008; 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 in 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 worlds: we present a static IFC system that automatically—without manual proofs or annotations—enforces Hails-like expressive, declarative 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. 2008; Vassena and Russo 2016] to enforce these policies. We do this in Lifty—short for Liquid Information Flow TYpes—a 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 *leak localization* mechanism that relies on the Lifty type-checker to infer an expected type 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 useful in practice.

Evaluation. We evaluated our prototype implementation of Lifty 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. For each of these programs, we write the code omitting all policy checks (i.e., as if all the data were publicly visible), and ask Lifty to repair the code—and make it secure. Our evaluation demonstrates that our solution supports expressive policies and

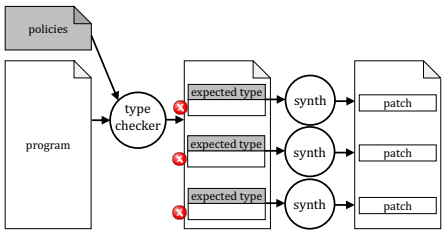


Fig. 1. Patching information leaks with Lifty.

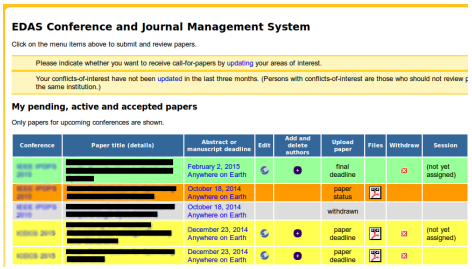


Fig. 2. Author's home screen in EDAS, shared with permission of Agrawal and Bonakdarpour [Agrawal and Bonakdarpour 2016].

is able to generate all necessary patches for our benchmarks within a reasonable time (within a minute for each of our case studies).

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 leak localization with type-driven synthesis to generate best-effort patches for the leaks.
(3) Prototype implementation: we implement a prototype of Lifty 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 is leaky: 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 applicatin. 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

```
1  showPaper ds client p = do
2    t   ⟵ getTitle ds p
3    dec ⟵ getDecision ds p
4    if dec = Accept
5      then do
6        ses ⟵ getSession ds p
7        print client (t + "␣" + ses)
8      else print client t
```

Fig. 3. Simplified version of the EDAS leak.

```
1  showPaper ds client p = do
2    t   ⟵ getTitle ds p
3    dec ⟵ do x1 ⟵ getPhase ds
4             if x1 = Done
5               then getDecision ds p
6               else return NoDecision
7    if dec = Accept
8      then do
9        ses ⟵ getSession ds p
10       print client (t + "␣" + ses)
11     else print client t
```

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

to `client` together with the title; otherwise, it only displays the title[1]. In deciding to display the session (or not), this code indirectly leaks the decision to `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 strew in such *policy-enforcing code* in the dozen of web request handlers that access papers, reviews, etc. Real web applications handle lots of sensitive data and have hundreads 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:

```
getDecision :: ds: Store ⟶ p: PaperId ⟶ TIO Decision <phase ds = Done, false>
```

The `TIO` type constructor is indexed by two security *labels*: the first label specifies when a user is 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 `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 this field with a default value—a constant `NoDecison`. 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

[1]The `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.

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. 2016]. 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 each sensitive value, which can have prohibitive runtime overhead. Moreover, Jeeves replaces sensitive values with default values implicitly, at run time—this makes it harder for the programmer to modify the tool's default mitigation strategy and find bugs due to default values.

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. 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 label* ℓ : `User → Bool` denotes information visible to users that satisfy ℓ[2]. For example, a key shared between `alice` and `bob` has a security label $\lambda u.u \in$ `[alice, bob]`. In the rest of the paper we will use a reserved variable v to range over users in security labels and omit the λ-binding; for example, the label introduced above is written as $v \in$ `[alice, bob]`. Note that the usual "can-flow-to" partial order in our security lattice corresponds to reverse implication: $l \sqsubseteq h$ iff $\forall v.h \Rightarrow l$, and hence the bottom (least secret) label is $\bot$ = true and the top (most secret) label is $\top$ = false. Our lattice can be seen as a generalization of DCLabels [Stefan et al. 2011a] and is thus at least as expressive [Montagu et al. 2013].

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. 2008; 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.

[2]For simplicity, we use a concrete type `User` to denote principals; our approach also supports leaving this type as a parameter.

```
getSharedKey :: TI String <v ∈ [alice,bob]>    print :: x:User → String → TO () <v = x>
getSSN :: x:User → TI String <v = x>
```

Fig. 5. Example input/output actions.

```
1  -- Primitives (trusted)
2  data TIO α <i,o> = -- hidden from clients
3  return :: ∀α . α → TIO α <true, false>
4  bind :: ∀α,β,i,j,o,p . TIO α <i, o>
5   → (α → TIO β <j, i ∧ p>)
6   → TIO β <i ∧ j, o ∨ (p ∧ i)>
7  downgrade :: ∀ c,i,o .
8    TIO {Bool | ν ⇒ c} <i ∧ c> <o>
9    → TIO {Bool | ν ⇒ c} <i> <o>
10 -- Auxiliary (typed-checked)
11 type TI α <i> = TIO α <i, False>
12 type TO α <o> = TIO α <True, o>
13 seq :: ∀α,β,i,j,o,p . TIO α <i,o>
14  → TIO β <j,p> → TIO β <j,o ∨ p>
15 mapM :: ∀α,β,i . (α → TIO β <i,i>)
16   → [α] → TIO [β] <i,i>
17 ... -- liftM, filterM, sortByM
```

Fig. 6. Excerpt from the TIO API.

More specifically, Lifty introduces the type constructor TIO ("tagged input-output"), indexed by a return type and two security labels, the *input label* i and the *output label* o. The type TIO T $\langle i, o\rangle$ 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 TIO String $\langle v \in$ [alice, bob], false$\rangle$ to indicate that its result should be viewed only by alice and bob, and that it does not have any (user-visible) output effects.

Subtyping. The TIO constructor is *covariant* in the input label and *contravariant* in the output label *wrt.* the lattice ordering, i.e., TIO T $\langle i_1, o_1\rangle$ <: TIO T $\langle i_2, o_2\rangle$ iff $i_1 \sqsubseteq i_2$ and $o_2 \sqsubseteq o_1$. For example, we can pass the getSharedKey computation from above to a function f with argument x: TIO String $\langle v =$ alice, $v =$ alice$\rangle$. Intuitively this is safe, because f requires the result of x to be visible at least to alice (while in fact the shared key is visible to both alice and bob), and counts on x to output at most to alice (while in fact getSharedKey does not perform any user-visible output). Because lattice ordering reduces to implication between refinement formulas, 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 alice and bob, reading a user's social-security number (which is visible only to the user themselves), and printing a string to a given user. TI and TO 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 the API shown in Fig. 6. The API consists of three core primitives; these form the trusted computing base of Lifty (together with the language runtime and type checker). The API also exposes several auxiliary functions that are built on top of the primitives and verified using the type checker.

```
1  ok1 =
2   do
3    k ← getSharedKey
4    print bob k
5    print alice k
```

```
1  bad1 =
2   do
3    b ← getSSN bob
4    print bob b
5    print alice b
```

```
1  bad2 =
2   do b ← getSSN  bob
3      print bob b
4      a ← getSSN alice
5      print alice a
```

```
1  ok2 =
2   do do b ← getSSN bob
3         print bob b
4      a ← getSSN alice
5      print alice a
```

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

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. To enforce this, we simply add a's input label `i` as a *conjunct* 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)[3], as well as `filterM` and `sortByM` (for filtering and sorting based on 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 (λ k . seq (print bob k) (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, v = `bob` $\vee$ v = `alice`, implies the input label of `getSharedKey`, $v \in$ [`alice`, `bob`]. More precisely, we can instantiate the type of **`bind`** with `i` $\mapsto v \in$ [`alice`, `bob`], `p`, `j` $\mapsto$ true, `o` $\mapsto$ 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.

[3]Why are the input and output labels of `mapM` the same? The only way to sequence two `TIO` computations and preserve both of their return values is to use **`bind`**. But **`bind`** requires that the input label of the first computation can flow to the output label of the second; hence `mapM` only verifies if its input label can flow to its output label. This is, of course, conservative: the later iterations of `mapM` do not actually use the results of the previous iterations! To express this fact, we could equip `TIO` with the *applicative* interface in addition to the *monadic* one, introducing a primitive operation for running two `TIO` computations independently and then combining their results. We decided not to pursue this path in the interest of simplicity.

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):

$$I \wedge \neg(v \in [\texttt{alice}, \texttt{bob}]) \Rightarrow \mathsf{false} \tag{1}$$

$$(O \vee P) \Rightarrow I \tag{2}$$

$$v = \texttt{alice} \Rightarrow P \tag{3}$$

$$v = \texttt{bob} \Rightarrow O \tag{4}$$

Here I, O, P are unknown predicates over the reserved variable v and any program variables in scope; these unknowns stand, respectively, 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-write unfolding of the rules: from (3) and (4) we infer the *strongest* assignment to O and P: $[O \mapsto v = \texttt{bob}, P \mapsto v = \texttt{alice}]$; substituting this into (2), we similarly infer $[I \mapsto (v = \texttt{alice} \vee v = \texttt{bob})]$; 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 `False`), because constants cannot leak information. Consider the following term:

```
downgrade (bind (getSSN x) (λ s . return valid(s) ∧ x = alice))
```

In Lifty, this term type-checks against the type `TI Bool` $\langle v$ `= alice`$\rangle$ even though it performs an input action of an incompatible type `TI Bool` $\langle v$ `= x`$\rangle$. Intuitively, this is safe because in all executions where the two input labels are indeed incompatible, it must be that `x` $\neq$ `alice`, and hence the *result* of the computation is always `False`. Lifty performs this reasoning automatically by instantiating the type of **`downgrade`** shown in Fig. 6 with $i \mapsto v$ `= alice`, $c \mapsto$ `x = alice`. Note that this type would no longer work if we removed the conjunct `x = 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

```
1  -- Datatypes and redaction functions
2  data Decision = Accept | Reject | NoDecision
3  redact NoDecision
4
5  -- Specification-only function for each field:
6  measure phase   :: Store ⟶ Phase
7  measure authors :: Store ⟶ PaperId ⟶ Set User
8
9  -- Field getters as TIO input actions:
10 getPhase    :: ds: Store              ⟶ TI {Phase | v = phase ds}          <True>
11 getTitle    :: ds: Store ⟶ p: PaperId ⟶ TI String                          <True>
12 getDecision :: ds: Store ⟶ p: PaperId ⟶ TI Decision                        <phase ds = Done>
13 getAuthors  :: ds: Store ⟶ p: PaperId ⟶ TI {[User] | elems v = authors ds p} <v ∈ authors ds p
14                                                                            ∨ phase ds = Done>
15 -- Field setters as TIO output actions:
16 setDecision :: ds: Store ⟶ dec: Decision ⟶ TO Store                        <phase ds = Done>
```

Fig. 8. Excerpt from the data store API for the conference manager.

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`:

```
seq a b = bind (downgrade (bind a (λ_ . return False))) (λ_ . 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 `phase ds = Done`, but this action is bound to

a computation with the output label v = `client`. For this occurrence of **`bind`** to be well-typed we need to prove that v = `client` $\Rightarrow$ `phase ds = Done` is valid, which does not hold.

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 `{Phase | v = phase ds}`. Hence the input action `getDecision ds p` is now being type-checked *under the assumption* `Done = 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 `showPaper`, 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. 2016, 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 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 the policies in Fig. 8 generates the following (simplified) system of CHCs:

$$I \wedge \texttt{phase ds = Done} \Rightarrow \text{false} \tag{5}$$

$$O \Rightarrow I \tag{6}$$

$$v = \texttt{client} \Rightarrow O \tag{7}$$

This system clearly has no solution; in particular, unfolding the rules (6)–(7) gives the strongest assignment $I \mapsto v$ = `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 generated which CHC. In this example, the query (5) is generated by `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 input 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 $I \mapsto v$ = `client`, hence replacing `getDecision ds p` with *any term* of type `TI Decision` $\langle v$ `= client`$\rangle$ 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 `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 `TI Decision` $\langle v$ `= client`$\rangle$, 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 space of possible redacted values by generating

```
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).

the branches in a restricted environment, which only contains the original term and explicitly designated *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 `Decision`). As a result, our running example generates only two branches:

```
getDecision ds p :: TI Decision ⟨phase ds = Done⟩
return NoDecision :: TI Decision ⟨true⟩
```

Next, for every branch, Lifty attempts to abduce a condition that would make the branch typecheck against the expected type. In our example, the second branch is correct unconditionally, while the first branch generates the following *logical abduction* problem: $C \wedge v = \texttt{client} \Longrightarrow \texttt{phase ds} = \texttt{Done}$, where C is an unknown formula over only the program variables, i.e., it cannot mention the user variable v. Lifty uses existing techniques [Polikarpova et al. 2016] to find the following solution to the abduction problem: $C \mapsto \texttt{phase ds} = \texttt{Done}$. 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 `bind getPhase (λx1 . x1 = 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 measures from 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 Lifty's `filterM` function, giving it the following, very strong type:

$$\texttt{filterM} :: \forall \alpha, i, f.(x : \alpha \to \texttt{TI}\ \{\texttt{Bool} \mid v \Rightarrow f\ x\}\ \langle f\ x \wedge i\rangle) \to [\alpha] \to \texttt{TI}\ [\{\alpha \mid f\ v\}]\ \langle i\rangle$$

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 in 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.

Attacker model. Lifty'a attacker model is standard, and similar to that of sequential LIO [Stefan et al. 2017]. In particular, we assume a language-level attacker who supplies `TIO` actions (which we ensure to be well-typed as `TIO ()` $\langle \top, \bot \rangle$). We assume that policies are correct (i.e., the type annotations on input-output actions) and that the underlying Lifty infrastructure (in particular the type checker, SMT solver, runtime, operating system, etc.) are secure. Like most previous work on language-level IFC, we consider side channels and transient execution attacks out of scope.

4 THE CORE CALCULUS

We now formalize a core language λ^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 noninterference guarantee, which we accomplish by reduction to LIO [Stefan et al. 2017].

4.1 Syntax of λ^L

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

Refinements

$$r, l ::= \mathsf{true} \mid \mathsf{false} \mid x \mid \neg r \mid r \wedge r \mid a \quad \textit{Ref. terms}$$

Types

$$B ::= \mathtt{()} \mid \mathtt{Bool} \mid \mathtt{User} \quad \textit{Base types}$$

$$T ::= \{B \mid r\} \mid T_1 \to T_2 \mid \mathtt{TIO}\ T\ \langle l_1, l_2 \rangle \mid \mathtt{Field}\ T\ \langle l \rangle \quad \textit{Types}$$

Programs

$$b ::= () \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{U}_1 \mid \mathsf{U}_2 \mid \ldots \quad \textit{Base Values}$$

$$f ::= \mathsf{F}_1 \mid \mathsf{F}_2 \mid \ldots \quad \textit{Fields}$$

$$v ::= b \mid f \mid \lambda x.\ t \mid \mathsf{TIO}\ t \quad \textit{Values}$$

$$t ::= v \mid x \mid t_1\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \textbf{return}\ t \mid \textbf{bind}\ t_1\ t_2 \mid \textbf{downgrade}\ t_1\ t_2 \mid \mathtt{get}\ t \mid \mathtt{set}\ t_1\ t_2 \quad \textit{Terms}$$

Fig. 11. Syntax of the core language λ^L.

Refinements. As is common in refinement types literature [Polikarpova et al. 2016; Rondon et al. 2008; Vazou et al. 2014b], λ^L distinguishes between program terms t and refinement terms r, used inside types; for readability, we use the meta-variable l instead of r for those refinement terms that represent labels. 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 functions, a consists of equalities such as $x = \mathtt{phase}\ ds$. 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[4].

Types. The base types B in λ^L include the unit type, the booleans, and the type of users. As is standard in refinement types systems, types include refined base types and function types. In a refined base types $\{B \mid r\}$, 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 $\{B \mid \mathsf{true}\}$. Although the Lifty implementation supports dependent function types of the form $x\colon T_1 \to T_2$ (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 λ^L for simplicity.

One non-standard feature of λ^L is the type of sensitive computations $\mathtt{TIO}\ T\ \langle l_1, l_2 \rangle$, indexed by the return type T, and input and output *labels* l_1 and l_2. Both labels are refinement predicates over the program variables and a special *user variable* v. Since it is convenient to think of labels as elements of a lattice, we define the following lattice-theoretic syntax for logical operations on labels:

$$l_1 \sqcup l_2 \triangleq l_1 \wedge l_2 \qquad l_1 \sqcap l_2 \triangleq l_1 \vee l_2 \qquad \top \triangleq \mathsf{false} \qquad \bot \triangleq \mathsf{true}$$

Another non-standard feature is a dedicated type of *fields* $\mathtt{Field}\ T\ \langle l \rangle$, which is used to model Lifty's atomic input/output actions and intuitively represents a resource at security level l that stores values of type T (where T is restricted to refined base types).

Program terms. Base values b include unit, booleans, and a finite set of user literals U_i. We also assume a finite set of field literals f, each of which has a fixed type and label, denoted by $\mathsf{ty}(f)$ and $\mathsf{lab}(f)$, respectively. We denote the set of all base values as $\mathcal{B}$ and the set of all fields as $\mathcal{F}$. In addition to base values and fields, values v include lambda abstractions and monadic actions $\mathsf{TIO}\ t$. Like in prior work [Buiras et al. 2015; Stefan et al. 2017; Vassena and Russo 2016], the TIO data constructor is not part of the surface syntax.

Terms t include values, variables, function application, conditionals, as well as monadic primitives **`return`**, **`bind`**, and **`downgrade`** introduced in Sec. 3. Atomic input/output actions are represented in λ^L as two universal actions `get` and `set` parameterized by the field to read from or write to. The **`downgrade`** construct in λ^L also takes an additional field argument, whose role is simply to specify

[4]Our implementation uses the theory of arrays, uninterpreted functions, and linear integer arithmetic.

Evaluation $\boxed{\langle \Sigma \mid t \rangle \downarrow \langle \Sigma' \mid v \rangle}$

$$\text{RET}\ \frac{}{\langle \Sigma \mid \texttt{return}\ t \rangle \downarrow \langle \Sigma \mid \text{TIO}\ t \rangle} \qquad \text{DOWN}\ \frac{\langle \Sigma \mid t_2 \rangle \downarrow \langle \Sigma' \mid \text{TIO}\ b \rangle}{\langle \Sigma \mid \texttt{downgrade}\ t_1\ t_2 \rangle \downarrow \langle \Sigma' \mid \text{TIO}\ b \rangle}$$

$$\text{BIND}\ \frac{\langle \Sigma \mid t_1 \rangle \downarrow \langle \Sigma' \mid \text{TIO}\ t_1' \rangle \qquad \langle \Sigma' \mid t_2\ t_1' \rangle \downarrow \langle \Sigma'' \mid v \rangle}{\langle \Sigma \mid \texttt{bind}\ t_1\ t_2 \rangle \downarrow \langle \Sigma'' \mid v \rangle}$$

$$\text{GET}\ \frac{\langle \Sigma \mid t \rangle \downarrow \langle \Sigma \mid f \rangle}{\langle \Sigma \mid \texttt{get}\ t \rangle \downarrow \langle \Sigma \mid \text{TIO}\ \Sigma[f] \rangle} \qquad \text{SET}\ \frac{\langle \Sigma \mid t_1 \rangle \downarrow \langle \Sigma \mid f \rangle \qquad \langle \Sigma \mid t_2 \rangle \downarrow \langle \Sigma \mid b \rangle}{\langle \Sigma \mid \texttt{set}\ t_1\ t_2 \rangle \downarrow \langle \Sigma[f := b] \mid \text{TIO}\ () \rangle}$$

Fig. 12. Big-step operational semantics of IFC constructs in λ^L.

the label to downgrade to: intuitively, **`downgrade`** $f\ t$ downgrades the result of t to $\text{lab}(f)$. In the Lifty implementation this label is implicit and inferred by the type checker; the reason λ^L needs to reify this label in the term language will become clear in Sec. 4.4. We omit recursion from λ^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.

4.2 Dynamic Semantics of λ^L

To model input and output actions, we define the run-time behavior of λ^L in terms of a *store* $\Sigma : \mathcal{F} \to \mathcal{B}$ that maps fields to base values. A *program configuration* $\langle \Sigma \mid t \rangle$ consists of a store and a term. Fig. 12 defines a big-step evaluation relation $\downarrow$ on configurations. The behavior of monadic primitives is mostly straightforward; in particular, since λ^L only tracks labels statically, there is no label propagation or access checks at run time. The evaluation order of pure and monadic terms is mostly standard for a call-by-name calculus [Peyton Jones 2001], in particular: **`bind`** is strict in its first argument, as expected; `set` is strict to ensure that only values are written to the store. **`downgrade`** fully evaluates its second argument—including under the TIO constructor—but ignores its field argument; the purpose of the field argument will be clear from the noninterference proof in Sec. 4.4. The rules for pure terms are standard and therefore omitted, and the rule for **`downgrade`** is slightly simplified for exposition (see extended version [Polikarpova et al. 2020] for the full set of rules). Note that evaluating a pure term—i.e., a term of a non-`TIO` type—does not change the store.

4.3 Static Semantics of λ^L

Fig. 13 shows a subset of typing rules for λ^L that are relevant to IFC. Other rules are standard for languages with decidable refinement types and deferred to the extended version. 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.

Well-formedness. We show well-formedness rules for base and `TIO` types. They rely on the auxiliary *sorting* judgment $\Gamma \vdash r : B$, which depends on the underlying refinement logic, but necessarily checks that r only mentions variables in Γ. The two rules formalize the distinction between logical refinements and labels: the former can mention the value variable ν and the latter can mention the user variable v. Well-formedness premises are implicit in all typing rules described below.

Subtyping. The rule <:-TIO specifies that tagged types are covariant in their input label and contra-variant in the output label *wrt.* the *can-flow-to* order on labels. This order, $\Gamma \vdash l \sqsubseteq l'$ is defined as the logical validity of reverse implication between label predicates, under the assumptions stored in the environment (which include path conditions and refinements on program variables).

Well-formedness $\boxed{\Gamma \vdash T}$

$$\textsc{wf-base}\ \frac{\Gamma, v : B \vdash r : \texttt{Bool}}{\Gamma \vdash \{B \mid r\}} \qquad \textsc{wf-tio}\ \frac{\Gamma \vdash T \qquad \Gamma, v : \texttt{User} \vdash l_i \wedge l_o : \texttt{Bool}}{\Gamma \vdash \texttt{TIO}\ T\ \langle l_i, l_o \rangle}$$

Subtyping and Flow $\boxed{\Gamma \vdash T <: T'}$ $\boxed{\Gamma \vdash l \sqsubseteq l'}$

$$<:\text{-TIO}\ \frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash l_1 \sqsubseteq l_2 \qquad \Gamma \vdash l_2' \sqsubseteq l_1'}{\Gamma \vdash \texttt{TIO}\ T_1\ \langle l_1, l_1' \rangle <: \texttt{TIO}\ T_2\ \langle l_2, l_2' \rangle} \qquad \textsc{flow}\ \frac{\Gamma, v : \texttt{User} \vDash l' \Rightarrow l}{\Gamma \vdash l \sqsubseteq l'}$$

Typing $\boxed{\Gamma \vdash t :: T}$

$$\textsc{t-ret}\ \frac{\Gamma \vdash t :: T}{\Gamma \vdash \textbf{return}\ t :: \texttt{TIO}\ T\ \langle \bot, \top \rangle}$$

$$\textsc{t-bind}\ \frac{\Gamma \vdash t_1 :: \texttt{TIO}\ T_1\ \langle l_1, l_1' \rangle \qquad \Gamma \vdash t_2 :: T_1 \to \texttt{TIO}\ T_2\ \langle l_2, l_2' \rangle \qquad \Gamma \vdash l_1 \sqsubseteq l_2'}{\Gamma \vdash \textbf{bind}\ t_1\ t_2 :: \texttt{TIO}\ T_2\ \langle l_1 \sqcup l_2, l_1' \sqcap l_2' \rangle}$$

$$\textsc{t-down}\ \frac{\Gamma \vdash t_1 :: \texttt{Field}\ _\ \langle l \rangle \qquad \Gamma \vdash t_2 :: \texttt{TIO}\ \{\texttt{Bool} \mid v \Rightarrow r\}\ \langle l \sqcup r, l' \rangle}{\Gamma \vdash \textbf{downgrade}\ t_1\ t_2 :: \texttt{TIO Bool}\ \langle l, l' \rangle}$$

$$\textsc{t-get}\ \frac{\Gamma \vdash t :: \texttt{Field}\ T\ \langle l \rangle}{\Gamma \vdash \texttt{get}\ t :: \texttt{TIO}\ T\ \langle l, \top \rangle} \qquad \textsc{t-set}\ \frac{\Gamma \vdash t_1 :: \texttt{Field}\ T\ \langle l \rangle \qquad \Gamma \vdash t_2 :: T}{\Gamma \vdash \texttt{set}\ t_1\ t_2 :: \texttt{TIO ()}\ \langle \bot, l \rangle}$$

Fig. 13. Static semantics of IFC constructs in λ^L.

For example, the judgment a: User, b: {User $\mid v = a$} $\vdash v = a \sqsubseteq v = b$ reduces to a formula $\forall a, b, v.b = a \wedge v = b \Rightarrow v = a$, and hence is valid. Recall that under our assumptions on the refinement logic, the validity of such formulas is decidable. Note that for a given Γ, the set of labels with $\sqsubseteq, \sqcup, \sqcap, \top, \bot$ forms a lattice.

Subtyping for base types and function types is standard. Importantly, field types are *invariant* in both the value type and the label (since fields are used for reading and writing).

Term typing. The rest of Fig. 13 defines the typing judgment for program terms $\Gamma \vdash t :: T$. The first three rules encode the same label propagation logic we have introduced in Fig. 6; the difference is that in λ^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 **bind** slightly: here we add an explicit premise $\Gamma \vdash l_1 \sqsubseteq l_2'$, whereas in Fig. 6 we encode this implicitly by representing l_2' as $l_1 \sqcup p$. The last two rules type the universal actions `get` and `set` using the type of the corresponding field.

4.4 Noninterference in λ^L

In this section we show that Lifty programs cannot leak sensitive data by proving *noninterference* for the core calculus. Instead of proving noninterference from first principles, we accomplish this by reducing λ^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 *instrumented* operational semantics that adds dynamic IFC to λ^L; next, we argue that the instrumented λ^L is a subset of core LIO and hence exhibits noninterference; finally, we show that well-typed terms behave equivalently under the original and instrumented semantics.

Instrumented Evaluation $\boxed{\langle \Sigma, \ell_c \mid t\rangle \Downarrow \langle \Sigma', \ell_c' \mid v\rangle}$

$$\textsc{i-ret}\ \frac{}{\langle \Sigma, \ell_c \mid \texttt{return}\ t\rangle \Downarrow \langle \Sigma, \ell_c \mid \mathsf{TIO}\ t\rangle}$$

$$\textsc{i-bind}\ \frac{\langle \Sigma, \ell_c \mid t_1\rangle \Downarrow \langle \Sigma', \ell_c' \mid \mathsf{TIO}\ t_1'\rangle \qquad \langle \Sigma', \ell_c' \mid t_2\ t_1'\rangle \Downarrow \langle \Sigma'', \ell_c'' \mid v\rangle}{\langle \Sigma, \ell_c \mid \texttt{bind}\ t_1\ t_2\rangle \Downarrow \langle \Sigma'', \ell_c'' \mid v\rangle}$$

$$\textsc{i-down-1}\ \frac{\langle \Sigma, \ell_c \mid t_1\rangle \Downarrow \langle \Sigma, \ell_c \mid f\rangle \qquad \langle \Sigma, \ell_c \mid t_2\rangle \Downarrow \langle \Sigma', \ell_c' \mid \mathsf{TIO}\ b\rangle \qquad \mathsf{lab}(f) = \ell \qquad \ell_c' \sqsubseteq \ell \sqcup \ell_c}{\langle \Sigma, \ell_c \mid \texttt{downgrade}\ t_1\ t_2\rangle \Downarrow \langle \Sigma', \ell \sqcup \ell_c \mid \mathsf{TIO}\ b\rangle}$$

$$\textsc{i-down-2}\ \frac{\langle \Sigma, \ell_c \mid t_1\rangle \Downarrow \langle \Sigma, \ell_c \mid f\rangle \qquad \langle \Sigma, \ell_c \mid t_2\rangle \Downarrow \langle \Sigma', \ell_c' \mid \mathsf{TIO}\ b\rangle \qquad \mathsf{lab}(f) = \ell \qquad \ell_c' \not\sqsubseteq \ell \sqcup \ell_c}{\langle \Sigma, \ell_c \mid \texttt{downgrade}\ t_1\ t_2\rangle \Downarrow \langle \Sigma', \ell \sqcup \ell_c \mid \mathsf{TIO}\ \mathsf{False}\rangle}$$

$$\textsc{i-get}\ \frac{\langle \Sigma, \ell_c \mid t\rangle \downarrow \langle \Sigma, \ell_c \mid f\rangle \qquad \mathsf{lab}(f) = \ell}{\langle \Sigma, \ell_c \mid \mathtt{get}\ t\rangle \Downarrow \langle \Sigma, \ell \sqcup \ell_c \mid \mathsf{TIO}\ \Sigma[f]\rangle}$$

$$\textsc{i-set}\ \frac{\langle \Sigma, \ell_c \mid t_1\rangle \downarrow \langle \Sigma, \ell_c \mid f\rangle \qquad \langle \Sigma, \ell_c \mid t_2\rangle \Downarrow \langle \Sigma, \ell_c \mid b\rangle \qquad \ell_c \sqsubseteq \mathsf{lab}(f)}{\langle \Sigma, \ell_c \mid \mathtt{set}\ t_1\ t_2\rangle \Downarrow \langle \Sigma[f := b], \ell_c \mid \mathsf{TIO}\ ()\rangle}$$

Fig. 14. Instrumented operational semantics of IFC constructs in λ^L.

4.4.1 Instrumented Semantics. An *instrumented configuration* k is a triple $\langle \Sigma, \ell_c \mid t\rangle$, where ℓ_c is the *current (program counter) label*. Intuitively, ℓ_c starts out at $\bot$ and then gradually rises as the evaluation progresses, keeping track of the most sensitive field that the computation has read so far and blocking output to any field not above ℓ_c. In the interest of clarity, we introduce a distinct syntactic category ℓ of *run-time labels*, which are labels that do not mention any variables except v; the *can-flow-to* judgment between run-time labels does not require an environment, so we write it simply as $\ell \sqsubseteq \ell'$. Note that all labels of field literals $\mathsf{lab}(f)$ are naturally run-time labels.

Fig. 14 defines a big-step evaluation relation $\Downarrow$ on instrumented configurations. The rules for pure terms keep the current label intact and are omitted (the full set of rules can be found in the extended version). The core mechanism for propagating and checking run-time labels is captured in the rules I-GET, I-SET, and I-BIND. The rule I-GET raises the current label by ℓ—the label of the field being read. The premise of I-SET checks that the current label can flow to the label of the field being written. Finally the rule 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 **`downgrade`** $t_1\ t_2$, expressed in I-DOWN-1 and I-DOWN-2. Both of these rules start by fully evaluating t_1 to f (which changes neither the store nor the current label, since t_1 is a pure term) and then t_2 to TIO b (i.e., either TIO True or TIO False). The latter evaluation raises the current label to ℓ_c'. Instead of adopting ℓ_c' as the new current label, however, both rules *downgrade* it to $\ell \sqcup \ell_c$, effectively only raising the current label by the manifest label $\mathsf{lab}(f)$ of the **`downgrade`** operation. But won't such downgrading leak information at level ℓ_c' through b? This is where the I-DOWN-2 comes in: if in fact ℓ_c' can 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 **`downgrade`** can be encoded in LIO as follows:

```
get f       ≡ readLIORef f
set f t     ≡ writeLIORef f t
```

```
downgrade f t ≡ do lc ⟵ getLabel
                   lb ⟵ toLabeled ((labelOf f) ⊔ lc) t
                   catchLIO (unlabel lb) (λ_ ⟶ return False)
```

Both `get` and `set` simply read and write a reference `f`, labeled with `l` = lab(`f`) (created at the start of the program with `newLIORef l`). The more interesting case is **`downgrade`**. The `toLabeled` primitive returns a labeled value—whose label is `l` ⊔ `lc`—that either contains the result of `t` or a "delayed" exception if `t` reads data more sensitive than `l` ⊔ `lc`. `unlabel` raises the current label to `l` ⊔ `lc` and either returns the value computed within the `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 λ^L inherits its guarantee. Noninterference is formalized in terms of ℓ*-equivalence* on instrumented configurations, which is formally defined in the extended version. Intuitively, two stores are considered ℓ-equivalent if they only differ in fields whose label is not below ℓ.

Lemma 1 (Noninterference of instrumented semantics)**.** Instrumented evaluation from ℓ-equivalent configurations leads to ℓ-equivalent configurations: if $k_1 \approx_\ell k_2$, $k_1 \Downarrow k_1'$, and $k_2 \Downarrow k_2'$, then $k_1' \approx_\ell k_2'$

4.4.3 Simulation. The core of our noninterference argument is a proof that instrumented execution simulates original execution for well-typed terms. The full proofs can be found in the extended version; here we only state the key lemma and give the intuition for the proof.

Lemma 2 (Simulation)**.** If $\epsilon \vdash t :: \texttt{TIO}\ T\ \langle \ell_i, \ell_o \rangle$ and $\langle \Sigma \mid t \rangle \downarrow \langle \Sigma' \mid v \rangle$, then for any $\ell_c \sqsubseteq \ell_o$, there exists a new current label ℓ_c' such that (1) $\langle \Sigma, \ell_c \mid t \rangle \Downarrow \langle \Sigma', \ell_c' \mid v \rangle$, (2) $\ell_c' \sqsubseteq \ell_c \sqcup \ell_i$, (3) $\epsilon \vdash v :: \texttt{TIO}\ T\ \langle \ell_i, \ell_o \rangle$.

Intuitively, this lemma says that executing a well-typed monadic term t from a configuration where the current label ℓ_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 instrumented execution would only raise the current label by the static input label of the computation (2). The proof is by induction on the derivation of big-step evaluation. Interesting cases include `set`, where we show that the runtime check never fails (this follows from $\ell_c \sqsubseteq \ell_o$); **`bind`**, where we show that after executing the first action, the current label remains below the output label of the second action (this follows from property (2) and the last premise of T-BIND); and **`downgrade`**, where we show that whenever I-DOWN-2 applies, b is False anyway.

Finally we can combine Lemma 1, Lemma 2, and LIO's noninterference proof to show noninterference of λ^L programs:

Theorem 1 (Noninterference for λ^L)**.** Evaluating a computation t statically visible to ℓ from ℓ-equivalent stores leads to ℓ-equivalent stores: If $\Sigma_1 \approx_\ell \Sigma_2$, $\epsilon \vdash t :: \texttt{TIO}\ T\ \langle \ell, _ \rangle$, $\langle \Sigma_1 \mid t \rangle \downarrow \langle \Sigma_1' \mid v_1 \rangle$, and $\langle \Sigma_2 \mid t \rangle \downarrow \langle \Sigma_2' \mid v_2 \rangle$, we have $v_1 = v_2$ and $\Sigma_1' \approx_\ell \Sigma_2'$.

Technically, our noninterference guarantee is *termination-insensitive*: we state that final configurations are equivalent only as long as both initial configurations evaluate to a value. However in λ^L this is not an issue, since *all* well-typed programs evaluate to a value: progress and termination can be shown by a straightforward extension of proofs of these properties for simply-typed lambda calculus. More interestingly, the full LIFTY language also enjoys this property: although it supports recursion, it uses refinement types to prove that all recursive calls terminate.

5 LEAK REPAIR IN λ^L

We now formalize LIFTY's leak repair mechanism for the core calculus λ^L. Fig. 15 shows the pseudocode of the algorithm ENFORCE, which performs type-checking and repair of an individual

```
1:  ENFORCE(Γ, t)
2:      t̂ ← LOCALIZE(Γ, t)
3:      for ⟨T_e ◃ T_a⟩t_a ∈ t̂ do
4:          t_p ← GENERATE(x : T_a, Γ, T_e)[x ↦ t_a]
5:          t̂ ← t̂[⟨T_e ◃ T_a⟩t_a ↦ t_p]
6:  LOCALIZE(Γ, t)
7:      t̂ ← t
8:      (rules, queries) ← CHC(Γ ⊢ t :: TIO () ⟨⊤, ⊥⟩)
9:      A ← HORNSOLVER(rules)
10:     for Q ← queries | A ⊭ Q do
11:         (t_a, sub) ← source(Q)
12:         if sub = TI B ⟨i⟩ <: TI B ⟨l⟩ then
13:             t̂ ← [t_a ↦ ⟨TI B ⟨A[l]⟩ ◃ TI B ⟨i⟩⟩t_a]t̂
14:         else fail
15: GENERATE(Γ_R, Γ_G, TI B ⟨l⟩)
16:     Γ_R ← Γ_R ∪ R
17:     branches ← SYNTHALL(Γ_R ⊢ ?? :: TI B ⟨⊤⟩)
18:     conds ← ABDUCE(Γ_G, ?? ⊢ t_b :: TI B ⟨l⟩)
19:                 for t_b ← branches
20:     ((t_d, r_g) : guarded) ← SORT(branches, conds)
21:     if r_g ⇔ true then
22:         t ← t_d
23:     else fail
24:     for (r_g, t_b) ← guarded do
25:         T_g ← TI {Bool | ν ⇔ r_g} ⟨l⟩
26:         t_g ← SYNTH(Γ_G ⊢ ?? :: T_g)
27:         t ← bind t_g (λy.if y then t_b else t)
28:     return t
```

Fig. 15. Leak repair algorithm.

$$\textsc{t-cast}\ \frac{}{\Gamma \vdash \langle T' \triangleleft T\rangle :: T \to T'} \qquad \textsc{l-get}\ \frac{\Gamma \vdash \texttt{get}\ t :: T \quad T = \texttt{TIO}\ B\ \langle l, \top\rangle \quad T' = \texttt{TIO}\ B\ \langle l', \top\rangle}{\Gamma \vdash \texttt{get}\ t \hookrightarrow \langle T' \triangleleft T\rangle\ (\texttt{get}\ t) :: T'}$$

Fig. 16. Cast typing and cast insertion for λ^L.

controller function. More precisely, the algorithm takes as input a typing environment Γ and a program t, and determines whether t can be patched to produce a well-typed TIO computation, i.e., a term t' such that $\Gamma \vdash t' :: \texttt{TIO}\ ()\ \langle\top, \bot\rangle$. 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). Then, the algorithm replaces each type cast in $\hat{t}$ with an appropriate patch, generated by the procedure GENERATE (Sec. 5.2).

5.1 Leak Localization

Type casts. For the purpose of leak localization, we extend the values of λ^L with type casts:

$$v ::= \cdots \mid \langle T' \triangleleft T\rangle$$

Statically, our casts are similar to those in prior work [Knowles and Flanagan 2010]; in particular, the cast $\langle T' \triangleleft T\rangle$ has type $T \to T'$, as indicated in Fig. 16. However, the dynamic semantics of casts in λ^L is undefined: casts are inserted solely for the purpose of leak localization, and, if repair succeeds, are completely eliminated. We restrict the notion of *type-safe* λ^L programs to those that are well-typed are *free of type casts*.

Cast insertion. Declaratively, leak localization can be formalized using a cast insertion judgment $\Gamma \vdash t \hookrightarrow \hat{t} :: T$, which informally means that inserting type casts into term t can yield a term $\hat{t}$ of type T. Unlike prior work, our cast insertion is specific to IFC and our intended repair strategy—guarding and redacting unsafe input actions. As a result, we only allow inserting casts around `get` expressions, as shown in the L-GET rule in Fig. 16; for all other terms the judgment is defined homomorphically. Furthermore, the rule L-GET imposes two important restrictions: (1) the cast can only change the input label of the action (intuitively, it downgrades l into l'), and (2) the cast must be *functionally oblivious*: the result type $\{B \mid r\}$ of TIO must have a trivial refinement, i.e., $r = \text{true}$. As we explain below, these restrictions are crucial for the efficiency of repair, and although they introduce incompleteness, we found them to work well in practice. In particular, functionally

oblivious casts make sense in our use case because patch generation will redact the input action anyway, so we are unlikely to be able to satisfy any functional property r of the original action.

Minimal sound localizations. Using the typing and cast insertion rules, we can show that if $\Gamma \vdash t \hookrightarrow \hat{t} :: T$ then $\Gamma \vdash \hat{t} :: T$. We refer to $\hat{t}$ as a *sound localization* of t at type T in Γ. A sound localization $\hat{t}$ of t is also *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. The following lemma follows directly from well-typing of $\hat{t}$:

Lemma 3 (Localization)**.** If $\Gamma \vdash t \hookrightarrow \hat{t} :: T$, replacing each subterm of the form $\langle T_i' \triangleleft T_i \rangle\ t_i$ in $\hat{t}$ with a type-safe term of type T_i' yields a type-safe program.

Once a sound localization is found, this lemma enables patch generation to proceed *independently* for each type cast (taking the type T_i' as the expected type). If the localization is also minimal, patch generation has the highest chance to succeed. Hence the goal of the leak localization algorithm is to find a minimal sound localization of t at type $\texttt{TIO}\ ()\ \langle \top, \bot \rangle$ in Γ.

In a general-purpose type-driven repair setting, a term might have many minimal sound localizations, and the repair engine would have to explore all of them, until it finds one where all type casts can be patched, leading to inefficiency. For our domain-specific repair strategy, however, there is no need to search through localizations: in fact, given the restrictions we introduced on cast insertion, any λ^L term has *at most one* minimal sound localization (up to equivalence of refinement terms).

To see why, recall that sound localizations can only differ in expected labels of atomic input actions. From the typing rules we know that in the typing derivation of $\hat{t}$, the expected label l' can only appear on the right-hand side of an implication $\Gamma \vDash l \Rightarrow l'$; hence the weakest expected type for each cast can be chosen independently: it is the type with the highest label l' that satisfies the above constraint. If we omitted the requirement that casts be functionally oblivious and allowed the expected type to be any $\texttt{TIO}\ \{B \mid r\}\ \langle l', \bot \rangle$, the uniqueness property would be violated. This is because, unlike labels, refinements r can appear in a typing derivation as environment assumptions. Hence cast insertion would have a trade-off: picking a stronger type for one action (with a stronger r) might validate a choice of a weaker type for another action.

Inferring the localization. Given the restrictions outlined above, finding the minimal sound localization for a λ^L term, amounts to inferring the highest expected label for each unsafe access. Procedure Localize formalizes our new algorithm that infers expected labels efficiently during type checking. It first uses the λ^L 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 t_a 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 t_a to some (possibly unknown) label l, then we insert a type-cast around t_a, taking the expected label to be $\mathcal{A}[l]$, i.e., the valuation of l in $\mathcal{A}$. If, on the other hand, Q is not derived from a can-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 $\langle T_e \triangleleft T_a \rangle\ t_a$ with a patch term t_p 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. Given $\texttt{TI}\ B\ \langle l \rangle$ as the goal type, GENERATE first uses SYNQUID [Polikarpova et al. 2016] to synthesize the set of all terms up to certain size of type $\texttt{TI}\ B\ \langle \top \rangle$, i.e., with the right content type, but with no restriction on the label (line 17). Branches are generated in a restricted environment Γ_R, 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 e.g., functions 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. Consider a branch t_b synthesized by SYNTHALL: its actual, strongest type is some $\texttt{TI}\ B\ \langle l' \rangle$, while the expected type of the patch is $\texttt{TI}\ B\ \langle l \rangle$. GENERATE now attempts to synthesize the optimal guard that would make t_b respect the expected type. At a high level, this guard must be logically equivalent to the weakest refinement formula r_g, such that (1) $\Gamma_G, r_g \vdash l' \sqsubseteq l$ and (2) $\Gamma_G \vdash r_g : \texttt{Bool}$ (i.e., r_g does not mention the user variable v). This formula can be inferred using existing techniques, such as logical abduction [Dillig and Dillig 2013]. In particular, GENERATE relies on SYNQUID's *liquid abduction* mechanism to infer r_g for each branch in line 19.

In line 20 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 Γ_R contains a pure value b of type B, in which case **`return`** b 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 r_g, and (2) *no leaky enforcement*: the input label of the guard itself may flow to the expected label l of the patch. To ensure both conditions, we again use SYNQUID, this time with the goal type $\texttt{TI}\ \{\texttt{Bool} \mid \nu \Leftrightarrow r_g\}\ \langle l \rangle$ to synthesize the guard.

Lemma 4 (Safe patch generation)**.** If GENERATE succeeds, it produces a type-safe term of the expected type $\texttt{TI}\ B\ \langle l \rangle$.

Assuming correctness of SYNTH and ABDUCE, we can use the typing rules of Sec. 4 to show that the invariant $\Gamma_R \cup \Gamma_G \vdash \mathit{patch} :: \texttt{TI}\ B\ \langle l \rangle$ is established in line 22 and maintained in line 27. In particular, the type of the bound variable y in line 27 is $\{\nu : \texttt{Bool} \mid \nu \Leftrightarrow r_g\}$, hence, **`then`** branch is checked under the path condition $r_g \Leftrightarrow \text{true}$. Since r_g is the result of abduction, we know that $\Gamma_G, r_g \vdash \texttt{TI}\ B\ \langle l' \rangle <: \texttt{TI}\ B\ \langle l \rangle$, and hence $\Gamma_G, r_g \vdash b :: \texttt{TI}\ B\ \langle l \rangle$.

5.3 Guarantees and Limitations

In this section we summarize the soundness guarantee of leak repair in λ^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 3 and 4 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

Table 1. Microbenchmarks.

	Benchmark	Code size (AST nodes)		Time		
		Original	LIFTY	Localize	Generate	Total
1	EDAS	66	25	0.1s	0.1s	0.3s
2	EDAS-Multiple	87	50	0.3s	0.3s	0.6s
3	EDAS-Self-Ref	87	76	0.3s	0.9s	1.2s
4	Search	76	25	0.8s	0.2s	1.0s
5	Sort	64	58	0.5s	0.6s	1.1s
6	Broadcast	27	25	0.1s	0.2s	0.2s
7	HotCRP	68	22	0.5s	0.0s	0.5s
8	AirBnB	52	33	0.2s	0.1s	0.4s
9	Instagram	73	42	0.5s	0.9s	1.4s

domain-specific repair algorithm. GENERATE can fail in lines 23 and 26. The first failure indicates that Γ_R 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 "decision" 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 "decision" at least as restrictive as the one on "phase").

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 label of output actions due to imprecisions of refinement type inference; and (2) the guard condition r_g 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.

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

(a) Conference Management System — Policy size (AST nodes): 247

Benchmark	Code size (AST nodes)			Time			
	Original	Manual	Lifty	Verify	Localize	Generate	Total
Register user	22	0	0	0.1s	0.1s	0.0s	0.1s
View users	26	16	25	0.5s	0.2s	0.2s	0.4s
Paper submission	65	0	0	2.0s	2.2s	0.0s	2.2s
Search papers	123	77	96	29.0s	6.5s	5.4s	11.9s
Show paper record	63	61	85	8.0s	0.8s	2.0s	2.8s
Show reviews for paper	96	61	70	14.6s	5.0s	0.8s	5.8s
User profile: GET	46	0	0	0.2s	0.2s	0.0s	0.2s
User profile: POST	20	0	0	0.1s	0.1s	0.0s	0.1s
Submit review	103	0	0	9.0s	7.8s	0.0s	7.8s
Assign reviewers	63	0	0	0.6s	0.6s	0.0s	0.6s
Total	627	215	276	64.1s	23.5s	8.4s	31.9s

(b) Gradr—Course Management System — Policy size (AST nodes): 75

Benchmark	Code size (AST nodes)		Time		
	Original	Lifty	Localize	Generate	Total
Display the home page (static content)	17	0	0.0s	0.0s	0.0s
View a user's profile (owner)	119	59	3.0s	0.8s	3.8s
View a user's profile (any user)	120	59	3.1s	1.5s	4.6s
Instructor: view scores for an assignment	73	103	0.9s	1.4s	2.3s
Instructor: view top scores for an assignment	99	147	2.4s	2.4s	4.8s
Student: view all scores for user	100	112	3.7s	9.5s	13.2s
Total	528	480	13.2s	15.6s	28.8s

(c) HealthWeb—Health Information Portal — Policy size (AST nodes): 95

Benchmark	Code size (AST nodes)		Time		
	Original	Lifty	Localize	Generate	Total
Search a record by id	27	161	0.1s	13.0s	13.1s
Search a record by patient	74	339	1.2s	41.6s	42.8s
Show authored records	76	0	1.2s	0.0s	1.2s
Update record	25	0	0.0s	0.0s	0.0s
List patients for a doctor	80	87	1.3s	3.5s	4.8s
Total	282	587	3.8s	58.1s	61.9s

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. 2016].
- **Performance.** We show that the Lifty compiler is reasonably efficient at leak localization and patch synthesis: Lifty is able to generate all necessary patches for each of our case studies in 30–60 seconds.
- **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 the extended version [Polikarpova et al. 2020].

Benchmark 1 is our running example from Sec. 2; benchmarks 2–3 are its variations with multiple unsafe accesses in the same controller 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 `filterM`'s predicate.

`Sort` displays the list of all conference submissions sorted by their score, using a higher-order `sortM` 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 decision, 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, whose policy requires that both accounts be visible to the viewer:

```
measure following :: Store → User → Set User
getIsFollowing :: ds: Store → who: User → whom: User →
  TI {Bool | ν = (whom in following ds who)} <canSee ds ν who ∧ canSee ds ν whom>
inline canSee ds x y  =  x = y ∨ isPublic ds y ∨ y ∈ following ds x
```

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 examples from Jacqueline [Yang et al. 2016], and a health portal based on the HealthWeb example from Fine [Swamy et al. 2010]. Tab. 2 lists the controllers we implemented for each case study, together with their sizes in AST nodes. More precisely, the column "Original" refers to the size of manually-written code without policy checks, and "Lifty" refers to the size of auto-generated policy-enforcing code; for an example, see grayed-out vs highlighted code in Fig. 4. Conference manager additionally reports the size of manually written policy-enforcing code in the column "Manual". For each case study, we also report the size of the policy, which is the cumulative size of all refinements of input and output actions, plus the size of **`inline`** macro definitions; we do not include the size of the Lifty standard library into the policy size.

Conference manager. We implemented two versions of a basic academic conference manager: one where the programmer enforces the policies by hand (and Lifty only verifiers 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".

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:

```
inline recordPolicy w v rid = v = recordAuthor w rid ∨ v = recordPatient w rid ∨
 (¬(shouldWithhold w rid) ∧ isDoctor w v ∧
 (isTreating w v (recordPatient w rid) ∨ ¬(recordIsPsychiatric w rid)))
getRecordPatient :: ds: Store → rid: RecordId
  → TI {User | v = recordPatient ds rid} <recordPolicy ds v 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 leak 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 a side-effect of the restriction imposed by our repair algorithm (Sec. 5.1) that expected types for patches be functionally oblivious. Thanks to this restriction, automatically-generated leak patches can be verified independently from the rest of the controller (and from each other); on the other hand, with manual policy enforcement Lifty verifies the controller function as a whole, which is less efficient but more precise.

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 different leaks inside one function, which allows Lifty to scale to functions that require multiple patches. For a stress test, we created a benchmark that sequences together N reads of a sensitive field, and then a `print` to an arbitrary user. Lifty's job is to patch each of the N leaks. Both leak localization and patch generation scale non-linearly but relatively well: as N grows from 1 to 16, the total compilation time increases from less than a second to just over 20 seconds. The repair time is non-linear because with the sequential structure of our benchmark each read introduces a new variable, which is visible in all the following patches, and hence increases the search space for policy checks. You can find the detailed results of this experiment is the extended version.

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. 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.

6.5 Discussion and Limitations

We conclude this section with a discussion of Lifty's limitations.

Policy language. The expressiveness of Lifty policies is limited by the underlying SMT theory of quantifier-free linear arithmetic, uninterpreted functions, and arrays. For example, policies cannot involve non-linear arithmetic or arbitrary recursive functions over data (e.g., refer to the maximum of all paper scores). Since all functions are uninterpreted, Lifty does not automatically know, for example, that paper reviewers cannot be in conflict with a paper. This could result in generating redundant checks but is also easy to avoid by adding postconditions to the input action `getReviewers`, relating it not only to the `reviewers` measure but to the `conflicts` measure. Perhaps most importantly, complex policies are most naturally expressed using existential quantification; for example, to state that a review r is visible only to reviewers that have submitted a reviews for the same paper, we would like to write: $\exists r'.\texttt{reviewPaper}\ r = \texttt{reviewPaper}\ r' \wedge v = \texttt{reviewer}\ r'$. This policy is currently not supported by Lifty: instead, a programmer would need to introduce "inverse measures" for `reviewPaper` and `reviewer` and write $\texttt{reviewsBy}\ v \cap \texttt{reviewsFor}\ (\texttt{reviewPaper}\ r) \neq \emptyset$ (and add postconditions to getters to connect direct and inverse measure).

Programming model. While the present work lays a foundation for static IFC with liquid types, our ultimate goal is to turn Lifty into a realistic web framework for Haskell, building upon LiquidHaskell [Vazou et al. 2014a]. Apart from a significant engineering effort, there are several research challenges involved in achieving this goal: most importantly, encoding a realistic database model that supports provably secure interaction with the data store through SQL-like *queries* rather than retrieving fields "one at a time". This is challenging because the type checker has to infer an aggregate label for all the data returned by the query, which has to be precise enough to verify common data retrieval patterns. In addition, we will need a more convenient way to specify existential policies mentioned above, a way to generate input-output actions automatically from a declarative description of the application's *model* (i.e., database schema with policies), as well as a way to tie the `TIO` monad into a server framework. We leave all these improvements to future work.

7 RELATED WORK

Lifty builds upon several lines of prior work, most notably in static information flow control, program synthesis and repair, and type error localization. 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. 2017; Stefan et al. 2017, 2011b] and Jeeves [Austin et al. 2013; Yang et al. 2016, 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. 2008] 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. 2008; 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 leak localization, crucial for repair. 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 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. 2017] and Jacqueline [Yang et al. 2016]) 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 techniques for synthesizing provably correct programs from formal specifications [Alur et al. 2017; Kneuss et al. 2013; Kuncak et al. 2010; Manna and Waldinger 1980; Polikarpova et al. 2016]. 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 deductive program repair [Kneuss et al. 2015], but 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. Finally, our repair technique is based on CHCs and uses a Horn solver to infer the optimal expected type; [Hojjat et al. 2016] also propose a Horn-based repair technique but for a different domain (software-defined networks).

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 $\mathcal{R}$ of redaction functions. The Lifty type system, however, is far more expressive 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 leak 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.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and our shepherd, Nikhil Swamy, for their valuable feedback on earlier drafts of this paper. We are also grateful to Marco Vassena for suggesting how to simplify and strengthen the noninterference theorem. This work was supported by a gift from Cisco and by the National Science Foundation under Grants No. 1911149 and 1943623.

REFERENCES

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. 1999. A Core Calculus of Dependency. In *POPL*. ACM.
Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *CSF*.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*. 319–336.

O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. 2012. Sharing Mobile Code Securely with Information Flow Control. In *Oakland*.

Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted execution of policy-agnostic programs. In *PLAS*.

Niklas Broberg, Bart van Delft, and David Sands. 2017. Paragon–Practical programming with information flow control. *Journal of Computer Security* 25, 4-5 (2017), 323–365.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ICFP*. 289–301.

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*.

Adam Chlipala. 2010. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *OSDI*.

Catalin Cimpanu. 2020. Walgreens says mobile app leaked users' personal data. https://www.zdnet.com/article/walgreens-says-mobile-app-leaked-users-personal-data/.

Benjamin Cosman and Ranjit Jhala. 2017. Local refinement typing. *PACMPL* 1, ICFP (2017), 26:1–26:27.

K. Crary, A. Kliger, and F. Pfenning. 2005. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming* 15, 2 (March 2005).

Dominique Devriese and Frank Piessens. 2011. Information flow enforcement in monadic libraries. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM.

Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *CAV*. 684–689.

Cory Doctorow. 2015. United website breach let fliers see each others' private data. https://boingboing.net/2015/01/28/united-website-breach-let-flie.html.

Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas W. Reps, Phillip A. Porras, Hassen Saïdi, and Vinod Yegneswaran. 2012. Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In *CAV*.

Vinod Ganapathy, Trent Jaeger, and Somesh Jha. 2006. Retrofitting Legacy Code for Authorization Policy Enforcement. In *SP*.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. 2017. Hails: Protecting Data Privacy in Untrusted Web Applications. *Journal of Computer Security* 25 (2017).

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *OSDI*. 47–60.

William R. Harris, Somesh Jha, and Thomas Reps. 2010. DIFC Programs by Automatic Instrumentation. In *CCS*.

Kashmir Hill. 2017. How Facebook Outs Sex Workers. https://gizmodo.com/how-facebook-outs-sex-workers-1818861596

Hossein Hojjat, Philipp Rümmer, Jedidiah McClurg, Pavol Cerný, and Nate Foster. 2016. Optimizing horn solvers for network repair. In *FMCAD*. 73–80.

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

Troy Hunt. 2020. Have I Been Pwned: Check if your email has been compromised in a data breach. https://haveibeenpwned.com/.

Limin Jia and Steve Zdancewic. 2009. Encoding information flow in Aura. In *PLAS*.

Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *CAV*.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *OOPSLA*. 407–426.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete functional synthesis. In *PLDI*.

Peng Li and Steve Zdancewic. 2005. Downgrading Policies and Relaxed Noninterference. (2005).

Peng Li and Steve Zdancewic. 2006. Encoding Information Flow in Haskell. In *CSFW*.

J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. 2009. Fabric: a platform for secure distributed computation and storage. In *SOSP*. ACM.

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *OOPSLA*. ACM.

Luísa Lourenço and Luís Caires. 2014. Information flow analysis for valued-indexed data security compartments. In *Trustworthy Global Computing*. Springer, 180–198.

Luísa Lourenço and Luís Caires. 2015. Dependent information flow types. In *POPL*. ACM, 317–328.

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980).

Simon Marlow. 2010. Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/

Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. 2013. A Theory of Information-Flow Labels. In *CSF*.

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*.

Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

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

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *PLDI*.

Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *CoRR* abs/1607.03445 (2020). arXiv:1607.03445 http://arxiv.org/abs/1607.03445

François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *POPL*. 319–330.

Privacy Rights Clearinghouse. 2020. Data Breaches. https://www.privacyrights.org/data-breach/.

Vineet Rajani and Deepak Garg. 2020. On the expressiveness and semantics of information flow types. *Journal of Computer Security* 28 (2020).

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*.

Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell. In *ICFP*.

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-weight Information-flow Security in Haskell. In *Haskell Symposium*.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003).

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to blame: localizing novice type errors with data-driven diagnosis. *PACMPL* 1, OOPSLA (2017), 60:1–60:27.

E.G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F.B. Schneider. 2011. Logical attestation: an authorization architecture for trustworthy computing. In *SOSP*. 249–264.

Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *NDSS*. The Internet Society.

Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *J. Funct. Program.* 27 (2017).

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2011a. Disjunction Category Labels. In *Nordic Conference on Security IT Systems (NordSec)*. Springer.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011b. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*. ACM SIGPLAN.

Nikhil Swamy, Juan Chen, and Ravi Chugh. 2010. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *ESOP*.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *ICFP*.

Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. 2009. A Theory of Typed Coercions and Its Applications. In *ICFP*. ACM.

Marco Vassena and Alejandro Russo. 2016. On Formalizing Information-Flow Control Libraries. In *PLAS*, Toby C. Murray and Deian Stefan (Eds.). ACM, 15–28.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC: a verified static information-flow control library. *Journal of logical and algebraic methods in programming* 95 (2018), 148–180.

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *ESOP*.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: experience with refinement types in the real world. In *Haskell Symposium*.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014b. Refinement types for Haskell. In *ICFP*.

Chelsea Voss. 2016. private email communication.

Jean Yang. 2017. James Comey's Twitter Security Problem Is Your Problem, Too. https://www.technologyreview.com/s/604286/james-comeys-twitter-security-problem-is-your-problem-too

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *PLDI*.

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies.

Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving application security with data flow assertions. *SOSP* (2009).

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing type errors with class. In *PLDI*.

Lantian Zheng and Andrew C. Myers. 2007. Dynamic security labels and static information flow control. *International Journal of Information Security* 6, 2 (2007), 67–84.