Enforcing Declarative Policies with Targeted Program Synthesis

Nadia Polikarpova
University of California, San Diego
npolikarpova@eng.ucsd.edu

Jean Yang
Carnegie Mellon University
jyang2@cs.cmu.edu

Shachar Itzhaky
Technion
shachari@cs.technion.ac.il

Travis Hance
Carnegie Mellon University
thance@cs.cmu.edu

Armando Solar-Lezama
Massachusetts Institute of Technology
asolar@csail.mit.edu

Abstract

We present a technique for static enforcement of declarative information flow policies. Given a program that manipulates sensitive data and a set of declarative policies on the data, our technique automatically inserts policy-enforcing code throughout the program to make it provably secure with respect to the policies. We achieve this through a new approach we call *targeted program synthesis*, which enables the application of traditional synthesis techniques in the context of global policy enforcement. The key insight is that, given an appropriate encoding of policy compliance in a type system, we can use type inference to decompose a global policy enforcement problem into a series of small, local program synthesis problems that can be solved independently.

We implement this approach in Lifty, a core DSL for data-centric applications. Our experience using the DSL to implement three case studies shows that (1) Lifty's centralized, declarative policy definitions make it easier to write secure data-centric applications, and (2) the Lifty compiler is able to efficiently synthesize all necessary policy-enforcing code, including the code required to prevent several reported real-world information leaks.

1 Introduction

From social networks to health record systems, today's software manipulates sensitive data in increasingly complex ways. To prevent this data from leaking to unauthorized users, programmers sprinkle *policy-enforcing code* throughout the system, whose purpose is to hide, mask, or scramble sensitive data depending on the identity of the user or the state of the system. Writing this code is notoriously tedious and error-prone. Static information flow control techniques [10, 23, 28, 32, 41, 51] mitigate this problem by allowing the programmer to state a high-level *declarative policy*, and statically verify the code against this policy. These techniques, however, only address part of the problem: they can check whether the code as written leaks information, but they do not help programmers write leak-free programs in the first place. In this work, we are interested in alleviating the programmer burden associated with writing policy-enforcing code.

In recent years, *program synthesis* has emerged as a powerful technology for automating tedious programming tasks [7, 14, 20, 39, 47]. In this paper we explore the possibility of using this technology to enforce information flow security by construction: using a declarative policy as a specification, our goal is to automatically inject provably sufficient policy-enforcing code throughout the system. This approach seems especially promising, since each individual policy-enforcing snippet is usually short, side-stepping the scalability issues of program synthesizers.

The *challenge*, however, is that our setting is significantly different from that of traditional program synthesis. Existing synthesis techniques [3, 4, 15, 25, 33, 34, 39] target the generation of self-contained functions from end-to-end specifications of their input-output behavior. In contrast, we are given a global specification of *one aspect* of the program behavior: it must not leak information. This specification says nothing about where to place the policy-enforcing snippets, let alone what each snippet is supposed to do.

Targeted program synthesis. In this paper we demonstrate how to bridge the gap between global policies and local enforcement via a new approach that we call *targeted program synthesis*. Our main insight is that a carefully designed type system lets us leverage error information from type-checking the original unsafe program to infer local, end-to-end specifications for sufficient policy-enforcing snippets (or *leak patches*). More specifically, (1) the location of a type error indicates *where* to insert a leak patch and (2) the expected type corresponds to the *local specification* for the patch. Moreover, it is possible to guarantee that any combination of patches that satisfy their respective local specifications yields a provably secure program. In other words, we show how to decompose the problem of policy enforcement into several independent program synthesis problems, which can then be tackled by state-of-the-art synthesis techniques.

Type system. The main technical challenge in making targeted synthesis work is the design of a type system that, on

Draft, 2017, USA
2017 ACM ISBN ...$15.00
https://doi.org/

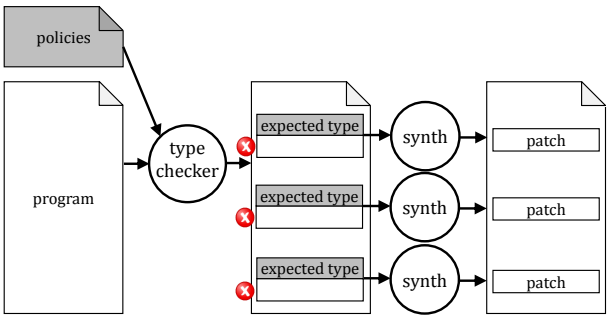

Figure 1. Policy enforcement in Lifty.

the one hand, is expressive enough to reason about the policies of interest, and on the other hand, produces appropriate type errors for the kinds of patches we want to synthesize. For our policy language, we draw inspiration from the Jeeves language [6, 49], which supports rich, *context-dependent* policies, where the visibility of data might depend both on the identity of the viewer and the state of the system. For example, in a social network application, a user's location can be designated as visible only to the user's friends who are within a certain distance of that location. In Jeeves, these policies are expressed directly as predicates over users and states. Our technical insight is that static reasoning about Jeeves-style policies can be encoded in a decidable refinement type system by indexing types with policy predicates. Moreover, we show how to instantiate the Liquid Types framework [36] to infer appropriate expected types at the error locations.

The Lifty language. Based on this insight, we developed Lifty[1], a core DSL for writing secure data-centric applications. In Lifty, the programmer implements the core functionality of an application without having to worry about information leaks. Separately, they provide a *policy module*, which associates declarative policies with some of the fields (columns) in the data store, by annotating their types with policy predicates. Given the source program and the declarative policies, Lifty automatically inserts leak patches across the program, so that the resulting code provably adheres to the policies (Fig. 1). To that end, Lifty's type inference engine checks the source program against the annotated types from the policy module, flagging every *unsafe access* to sensitive data as a type error. Moreover, for every unsafe access the engine infers the most restrictive policy that would make this access safe. Based on this policy, Lifty creates a local specification for the leak patch, and then uses a variant of type-driven synthesis [34] to generate the patch.

Evaluation. To demonstrate the practical promise of our approach, we implemented a prototype Lifty-to-Haskell compiler. We evaluated our implementation on a series of small but challenging micro-benchmarks, as well as three case studies: a conference manager, a health record system, and a student grade record system. The evaluation demonstrates that our solution supports expressive policies, reduces the burden placed on the programmer, and is able to generate all necessary patches for our benchmarks within a reasonable time (26 seconds for our largest case study). Importantly, the evaluation confirms that the patch synthesis time scales linearly with the size of the source code (more precisely, with the number of required leak patches), suggesting the feasibility of applying this technique to real-world code bases.

[1] Lifty stands for Liquid Information Flow TYpes.

2 Lifty by Example

To introduce Lifty's programming model, type system, and the targeted synthesis mechanism, we use an example based on a leak from the EDAS conference manager [1]. We have distilled our running example to a bare minimum to simplify the exposition of how Lifty works under the hood; at the end of the section, we demonstrate the flexibility of our language through more advanced examples.

2.1 The EDAS Leak

Figure 2 shows a screenshot from the EDAS conference manager. On this screen, a user can see an overview of all their papers submitted to upcoming conferences. Color coding indicates paper status: green papers have been accepted, orange have been rejected, and yellow is used before author notifications are out, indicating that the decision is still pending. An author is not supposed to learn about the decision before the notifications are out, yet from this screen, the user can infer that the first one of the pending papers has been tentatively accepted, while the second one has been tentatively rejected. They can make this conclusion because the two rows differ in the value of the "Session" column (which displays the conference session where the paper is to be presented), and the user knows that sessions are only displayed for accepted papers.

The EDAS leak is particularly insidious because it provides an example of an *implicit flow*: the "accepted" status does not appear anywhere on the screen, but rather influences the output via a conditional. To prevent such leaks, it is insufficient to simply examine output values; rather, sensitive values must be tracked through control flow.

Fig. 3 shows a simplified version of the code that has caused this leak. This code retrieves the `title` and `status` for a paper `p`, then retrieves `session` only if the paper has been accepted, and finally displays the title and the session to the currently logged-in `client`. The leak happened because the programmer forgot to insert *policy-enforcing code* that would prevent the true value of `status` from influencing the output, unless the conference is in the appropriate phase (notifications are out). It is easy to imagine, how in an application that manipulates a lot of sensitive data, such policy-enforcing

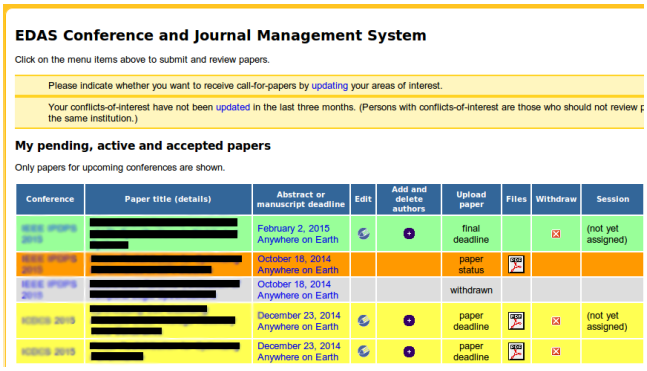

Figure 2. Author's home screen in EDAS, shared with permission of Agrawal and Bonakdarpour [1].

```
1  showPaper client p =
2    let row = do
3      t ← get (title p)
4      st ← get (status p)
5      ses ← if st = Accepted
6              then get (session p) else ""
7      t + "␣" + ses in
8    print client row
```

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

```
module ConfPolicy where

title :: PaperId → Ref ⟨String⟩^any

status :: PaperId → Ref ⟨Status⟩^λ(s,u).s[phase] = Done

session :: PaperId → Ref ⟨String⟩^any
```

Figure 4. Snippet from a policy module for a conference manager.

code become ubiquitous, imposing a significant burden on the programmer and obscuring the application logic.

2.2 Programming with Lifty

Lifty liberates the programmer from having to worry about policy-enforcing code. Instead, they provide a separate *policy module* that describes the data layout and associates sensitive data with declarative policies. For example, Fig. 4 shows a policy module for our running example.

The Lifty type system is equipped with a special type constructor $\langle T \rangle^{\pi}$ ("T tagged with policy π"), where $\pi : (\Sigma, \texttt{User}) \to \texttt{Bool}$ is a predicate on contexts, *i.e.* pairs of states and users. The type $\langle T \rangle^{\pi}$ denotes values of type T that are only visible to a user u in a state s such that $\pi\ (s, u)$ holds. For example, to express that a paper's status is only

```
1  showPaper client p =
2    let row = do
3      t ← get (title p)
4      st ← let x0 = get (status p) in do
5        x1 ← get phase
6        if x1 = Done then x0 else NoDecision
7      ses ← if st = Accepted
8        then get (session p) else ""
9      t + "␣" + ses in
10   print client row
```

Figure 5. With a patch inserted by Lifty to protect against the EDAS leak.

visible when the conference phase is `Done`, the programmer defines its type as a reference to `Status` tagged with policy $\lambda(s, u).s[\texttt{phase}] = \texttt{Done}$. Hereafter, we elide the binders (s, u) from policy predicates for brevity, and simply write $\lambda.p$. The predicate `any` $= \lambda.\texttt{True}$ annotates fields as public (*i.e.* visible in any context).

Given the code in Fig. 3 and the policy module, Lifty injects policy-enforcing code required to patch the EDAS leak; the result is shown in Fig. 5 with the new code highlighted. This code guards the access to the sensitive field `status` with a *policy check*, and if the check fails, it substitutes the true value of `status` with a *redacted value* (a constant `NoDecison`). Lifty guarantees that this code is provably correct with respect to the policies in the policy module.

2.3 Targeted Program Synthesis

Can the code in Fig. 5—and its correctness proof—be synthesized using existing techniques? Several synthesis systems [25, 34] can generate provably correct programs, but require a full functional specification (which is not available for `showPaper`) and might fail to scale to larger functions. Prior approaches to sound program repair [24] use *fault localization* to focus synthesis on small portions of the program, responsible for the erroneous behavior. These existing focusing techniques, however, are not applicable in our setting, because (1) they rely on testing, which is challenging for information flow security, and (2) they are not *precise* enough, *i.e.* they would not be able to pinpoint `get (status p)` as the unsafe term.

In this section we show how a careful encoding of information flow security into a type system (Sec. 2.3.1) allows us to instead use type inference for *precise* fault localization (Sec. 2.3.2). Concretely, type-checking the code in Fig. 3 against the policy module, leads to a type error in line 5, which flags the term `get (status p)` as unsafe, and moreover, specifies the expected type, which can be used as the local specification for patch synthesis (Sec. 2.3.3).

2.3.1 Type System

The Lifty type system builds upon existing work on *security monads* [37, 41], where sensitive data lives inside a monadic type constructor (in our case, $\langle\cdot\rangle$), parameterized by a security level; proper propagation of levels through the program is ensured by the type of the monadic bind. In contrast with prior work, our security levels correspond directly to policy predicates, which allows Lifty programs to express complex context-dependent policies directly as types, instead of encoding them into an artificial security lattice.

Subtyping. Moreover, unlike prior work, Lifty features subtyping between tagged types, which is *contravariant* in the policy predicate, *i.e.* $\langle T\rangle^{\lambda.p} <: \langle T\rangle^{\lambda.q}$ iff $q \Rightarrow p$. This allows a "low" value (with a less restrictive policy) to appear where a "high" value (with more restrictive policy) is expected, but not the other way around. Lifty restricts the language of policy predicates to decidable logics; hence, the subtyping between tagged types can be automatically decided by an SMT solver.

Tagged primitives. The type error for the EDAS leak is generated due to the typing rules for primitive operations on tagged values, **`print`** and **`bind`**. The latter is present in Fig. 3 *implicitly*: our Haskell-like **`do`**-notation desugars into invocations of **`bind`** in a standard way [31] (see appendix for the desugared version). The typing rule for **`bind`** can be informally stated as follows: if we want a sequence of two computations to produce a result visible in a given set of contexts, then both computations better produce results visible at least in those contexts. The rule for **`print`** allows displaying messages tagged with any π that holds of the current state and the viewer. We formalize these rules in Sec. 3.

Type inference. The Lifty type inference engine is based on the Liquid Types framework [12, 36, 44]. As such, it uses the typing rules to generate a system of subtyping constraints over tagged types, and then uses the definition of contravariant subtyping to reduce them to the following system of implications or *Horn constraints* over policy predicates:

$$B \Rightarrow s[\texttt{phase}] = \texttt{Done} \quad (1)$$

$$P \Rightarrow B \quad (2)$$

$$u = \texttt{client} \wedge s = \sigma \Rightarrow P \quad (3)$$

where P, B are unknown predicates that correspond to the policies of **`print`** and **`bind`**[2]. Horn constraints are solved using a combination of unfolding and predicate abstraction.

In this case, however, the system clearly has no solution, since the consequent of (1), which represents the policy on `status`, is not implies by the antecedent of (3), which is derived from the invocation of **`print`** and reflects what we know about the output context (*i.e.* that the viewer is `client` and the output state is the same as the current state, σ). Intuitively, it means that the code is trying to display a sensitive value in a context where its policy doesn't hold.

[2]There's a separate unknown for each invocation of **`bind`**, but in this example they are equivalent, and we simplify for readability.

2.3.2 Fault Localization

Unlike existing refinement type checkers [12, 36, 44], Lifty is not content with finding that a type error is present: it needs to identify the term to blame and infer its expected type. Intuitively, declaring constraint (3) as the cause of the error corresponds to blaming **`print`** for displaying its sensitive message in too many contexts, while picking constraint 1, corresponds to blaming the access **`get`** `(status p)` for returning a value that is too sensitive. For reasons explained shortly, Lifty decides to blame the access. To infer its expected type, it has to find an assignment to B, which works for the rest of the program (*i.e.* is a solution to constraints (2)–(3)). This new system has multiple solutions, including a trivial one $[P, B \mapsto \top]$. The optimal solution corresponds to the *least restrictive* expected type, in other words—due to contravariance—the *strongest* solution for policy predicates: $[P, B \mapsto u = \texttt{client} \wedge s = \sigma]$. Substituting this solution into the subtyping constraint that produced (1), results in the desired type error:

`get` `(status p)` :
expected type: $\langle\texttt{Status}\rangle^{\lambda.u = \texttt{client} \wedge s = \sigma}$
and got: $\langle\texttt{Status}\rangle^{\lambda.s[\texttt{phase}] = \texttt{Done}}$

Note that picking constraint (3) as the cause instead, and inferring the weakest solution to constraints (1)–(2) ($[P, B \mapsto s[\texttt{phase}] = \texttt{Done}]$) would give rise to a different patch: guarding the whole message `row` with a policy check. This would fix the leak but have an undesired side effect of hiding the paper title along with the session. Data-centric applications routinely combine multiple pieces of data with different policies in a single output; therefore, in this domain it makes more sense to guard the access, which results in "redacting" as little data as possible (and also mirrors the Jeeves semantics). Hence, Lifty always chooses to blame negative constraints (such as (1)), even though its inference engine could support the alternative behavior as well.

2.3.3 Patch Synthesis

From the expected type, Lifty obtains a type-driven synthesis problem [34]:

$$\Gamma \vdash \texttt{??} :: \langle\texttt{Status}\rangle^{\lambda.u = \texttt{client} \wedge s = \sigma}$$

Here Γ is a *typing environment*, which contains a set of components—variables and functions that can appear in the patch—together with their refinement types. A solution to this problem is any program term t that type-checks against the expected type in the environment Γ. As we show in Sec. 4, any such t, when substituted for **`get`** `(status p)` in Fig. 3, would produce a provably secure program; hence the synthesis problem is *local* (can be solved in isolation).

Even though *any* solution is secure, not all solutions are equally desirable: for example, returning `NoDecision` unconditionally is a solution (and so is returning `Accepted`). Intuitively, a desirable solution returns the original value whenever allowed, and otherwise either redacts some information from that value or returns a reasonable default. To synthesize this solution, Lifty enumerates all terms of type $\langle \texttt{Status} \rangle^{\pi}$ up to a fixed size and arranges them into a list of branches according to the strength of their policies. To pick reasonable default values, we require user annotations in the policy file that essentially pick a single `Status` constructor to be added to Γ. As a result, our running example generates only two branches:

$$\texttt{get (status s)} :: \langle \texttt{Status} \rangle^{\lambda s.s[\texttt{phase}] = \texttt{Done}}$$
$$\texttt{NoDecision} :: \langle \texttt{Status} \rangle^{\texttt{any}}$$

Next, for every branch, Lifty abduces a condition that would make the branch type-check against the expected type. For example, our first branch generates the following abduction problem:

$$C \wedge u = \texttt{client} \wedge s = \sigma \Longrightarrow s[\texttt{phase}] = \texttt{Done}$$

where C is an unknown formula that cannot mention the policy parameters s and u. Lifty uses existing techniques [34] to find the following solution $C \mapsto \sigma[\texttt{phase}] = \texttt{Done}$. It then uses the abduced condition to synthesize a *guard, i.e.* a program that computes the monadic version of the condition. In our case, the guard is `bind (get phase) (x1 . x1 = Done)`. Finally, Lifty combines the synthesized guards and branches into a single conditional, which becomes the patch that replace the original unsafe access.

2.4 Scaling Up to Real-World Policies

In the rest of the section, we demonstrate more challenging scenarios, where (1) a function contains several unsafe accesses, (2) the policy check itself uses sensitive data, and hence proper care must be taken to ensure that policy-enforcing code does not introduce new leaks, (3) the redacted value is not just a constant, or (4) the policy check depends on the eventual viewer and the state at the time of output (which need not equal the state at the time of data retrieval).

2.4.1 Multiple Leaks

Consider a variant of our running example, where in addition to the paper's title and session, we display its authors. Also assume our conference is double-blind, so `authors` is a a sensitive field with a policy similar to that of `status`. When checking this extended version of `showPaper`, Lifty generates two type errors, one for `get (status p)` and one for `get (authors p)`, each with the same expected type (since they flow into the same `print` statement). This gives rise to two patch synthesis problems, which can be solved independently, because their expected types only depend on the output context, and are not affected by the rewriting. More generally, local synthesis is possible in this example because the correctness property of interest does not depend on the content of the unsafe terms but only on their policy, and hence the content can be freely replaced without affecting the correctness of the rest of the program. As we detail in Sec. 4, this does not hold in general, but it holds for our intended use case.

2.4.2 Complex Policies

Continuing with our extended example, assume that we want to allow a paper's author to see the author list even before the notifications are out. This is an example of a policy that depends on a sensitive value; moreover, in this case the policy is *self-referential* because it guards access to the field `authors` in a way that depends on the *value* of `authors`. Enforcing such complex policies manually is particularly challenging, because the policy-enforcing code itself retrieves and computes over sensitive values, and hence, while trying to patch one leak, it might inadvertently introduce another.

In Lifty, the programmer expresses this complex policy in a straightforward way:

$$\texttt{authors} :: p : \texttt{PaperId} \longrightarrow \texttt{Ref}\ \langle [\texttt{User}] \rangle^{\lambda s.s[\texttt{phase}] = \texttt{Done} \vee u \in s[\texttt{authors}\ p]}$$

Note that the policy predicate can talk about the true value of the author list using the refinement term $s[\texttt{authors}\ p]$, which is only available in specifications. Given this policy, Lifty generates a provably correct patch:

```
auts ← let x0 = get (authors p) in do
    c1 ← do x1 ← get phase; x2 ← x0
            x1 = Done ∨ elem client x2
    if c1 then x0 else []
```

Intuitively, this code is secure *despite* the fact that the policy check `c1` depends on the value of `authors p`, because for any paper whose authors `client` is not allowed to see, `c1` is always false—independently on the actual author list—so it does not reveal any secrets. In Sec. 3 we show how a novel downgrading construct enables Lifty to perform this nontrivial reasoning automatically.

2.4.3 Nontrivial Patches

When sensitive data has more interesting structure, the optimal redacted value can be more complex than just a constant. Consider the example of an auction where bids are only revealed once all participants have bid [37]. Now consider a more interesting policy: once a participant has bid and before all bids are fully revealed, they can see who else has bid, but not how much. One way to encode this in Lifty is to store the bid in an option type, `Maybe Int`, and associate different policies with the option and its content:

$$\texttt{bid} :: \texttt{User} \longrightarrow \texttt{Ref}\ \langle \texttt{Maybe}\ \langle \texttt{Int} \rangle^{\lambda s.s[\texttt{phase}] = \texttt{Done}} \rangle^{\lambda s.s[\texttt{bid}\ u] \neq \emptyset}$$

With this definition, Lifty generates the following patch inside a function `showBid client p`, which displays participant `p`'s bid to `client`:

```
1 b ⟵ let x0 = get (bid p) in do
2   x1 ⟵ get phase; x2 ⟵ get (bid client); x3 ⟵ x0
3   if x1 = Done
4     then x0
5     else if isJust x2
6       then mapJust (λ_ . 0) x3
7       else Nothing
```

This patch has three branches, of which the *second* one (line 6) is the most interesting: whenever `client` has bid but the bidding is not yet `Done`, Lifty only redacts the value that might potentially be stored inside `x3`, but not whether `x3` is `Nothing` or `Just`. Note that Lifty reasons about this patch based solely on the generic type of `mapJust`:

$$\texttt{mapJust} :: (\alpha \to \beta) \to \texttt{Maybe}\ \alpha \to \texttt{Maybe}\ \beta$$

2.4.4 State Updates

Continuing with the auction example, consider the implementation of the function `placeBid client b`, which first retrieves everyone's current bids, then calls **`set`** `(bid client) b`, and finally displays all the bids to `client`. In this case, reusing the patch from above would be wrong and would result in hiding too much, since `x3` would reflect `client`'s (missing) bid at the time of retrieval; by the time of output, however, `client` has already bid and has the right to see who else did. Lifty would insert a correct repair, since it can reason about how the call to **`set`** affects the state, and in this case can statically determine that s[`bid` u] holds of the output context.

3 The λ^L Type System

We now formalize the type system of a core security-typed language λ^L, which underlies the design of Lifty. The main novelty of this type system is representing security labels as policy predicates. This brings two important benefits: on the one hand, our type system directly supports context-dependent policies; on the other hand, we show how to reduce type checking of λ^L problems to *liquid type inference* [36]. As a result, our type system design enables automatic verification of information flow security against complex, context-dependent policies, and requires no auxiliary user annotations. Moreover, Sec. 4 also demonstrates how this design enables precise fault localization required for targeted synthesis of policy-enforcing code.

Another novelty of the λ^L type system is its support for policies that depend on sensitive values, including self-referential policies (Sec. 2.4.2). Until now, this kind of policies were only handled by run-time techniques such as Jeeves [6, 49]. To support safe enforcement of these policies, λ^L includes a novel *safe downgrading* construct (Sec. 3.2), and features a custom security guarantee, which we call *contextual noninterference* (Sec. 3.3).

Program Terms

$v ::= c \mid \lambda x.\ e$	*Values*
$e ::= v \mid x \mid e\ e \mid$ **`if`** e **`then`** e **`else`** e	*Expressions*
$\mid$ **`get`** $x \mid$ **`bind`** $e\ e \mid \lfloor e \rfloor$	
$s ::=$ **`skip`** $\mid$ **`let`** $x = e$ **`in`** s	*Statements*
$\mid$ **`set`** $x\ x\ ;\ s \mid$ **`print`** $x\ x\ ;\ s$	

Types

$B ::= \texttt{Bool} \mid \texttt{User} \mid \cdots \mid \langle T \rangle^{\pi} \mid \texttt{Ref}\ T$	*Base types*
$T ::= \{B \mid r\} \mid T \to T$	*Types*

Refinements

$r ::= \top \mid \bot \mid \neg r \mid r \oplus r$	
$\mid r[r] \mid r[r := r] \mid x \mid r\ r \mid \cdots$	
where $\oplus \in \{= \mid \wedge \mid \vee \mid \Longrightarrow\}$	
$\pi ::= \lambda(s, u).\ r$	*Policy predicates*

Figure 6. Syntax of the core language λ^L.

This section introduces the syntax of λ^L (Sec. 3.1) and its typing rules (Sec. 3.2), and shows that well-typed λ^L programs satisfy contextual non-interference (Theorem 3.2). The runtime behavior of λ^L programs is straightforward; we provide an operational semantics in Appendix A.2.

3.1 Syntax of λ^L

λ^L is a simple core language with references, extended with several information-flow specific contructs. We summarize the syntax of λ^L in Fig. 6.

Program terms. λ^L differentiates between expressions and statements. Expressions include store read (**`get`**), monadic bind (**`bind`**), and *downgrading* ($\lfloor \cdot \rfloor$), which we describe in detail below. A statement can modify the store (**`set`**) or output a value to a user (**`print`**). Keeping expressions pure avoids the usual complications associated with implicit flows, which in λ^L can be encoded by passing conditional expressions as arguments to **`print`**.

Types. λ^L supports static information flow tracking via *tagged types*. The type $\langle T \rangle^{\pi}$ ("T tagged with π") attaches a policy predicate $\pi : (\Sigma, \texttt{User}) \to \texttt{Bool}$ to a type T (here Σ is the type of stores, which map locations to values). A tagged type is similar to a labeled type in existing security-typed languages [35, 38, 41], except the domain of labels is not an abstract lattice, but rather the lattice of predicates over stores and users. Intuitively, a value of type $\langle T \rangle^{\lambda(s,u).p}$ can be revealed in any store s to any user u, such that p holds. Here p is a *refinement predicate* over the program variables in scope and the policy parameters s and u. The exact set of refinement predicates depends on the chosen refinement logic; the only requirement is that the logic be decidable to enable automatic type checking. We assume that the logic at least includes the theories of uninterpreted functions (x and

$r\ r$) and arrays ($r[r]$ and $r[r := r]$), which λ^L uses to encode policy predicates and store reads/writes, respectively.

Other types of λ^L include primitive types, references, function types, and *refinement types*, which are standard [26, 36]. 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.

Constants. To formalize the Lifty's notion of policy module, we assume that the syntactic category of constants, c, includes a predefined set of store *locations* and *fields* (functions that return locations). The type of each constant c is determined by an auxiliary function $\mathrm{ty}(c)$. For example, in a conference manager we define $\mathrm{ty}(\texttt{title}) = \texttt{PaperId} \to \langle \texttt{String} \rangle^{\lambda(s,u).\top}$. Since λ^L programs do not allocate new references at run time, the type of any location l is known a-priori and can be obtained through $\mathrm{ty}(l)$, which is why our typing rules do not keep track of a "store environment". Besides locations and fields, constants include values of primitive types and built-in functions on them.

3.2 Typing rules for λ^L

Fig. 7 shows a subset of subtyping and type checking rules for λ^L that are relevant to information flow tracking. Other rules are standard for languages with decidable refinement types [36, 44–46] and deferred to Appendix 11. In Fig. 7, a *typing environment* $\Gamma ::= \bullet \mid \Gamma, x : T \mid \Gamma, r$ maps variables to types and records *path conditions* r.

Subtyping. We only show subtyping rules for tagged types. The rule <:-Tag1 allows to tag a pure type with any well-formed policy. The rule <:-Tag2 specifies that tagged types are *contravariant* in their policy parameter; this relation allows "upgrading" a term with a less restrictive policy (more public) into one with a more restrictive policy (more secret) and not the other way around. The premise $\Gamma \vDash r' \Rightarrow r$ checks implication between the policies under the assumptions stored in the environment (which include path conditions and refinements on program variables). By restricting refinement predicates to decidable logic, we make sure that this premise can be validated by an SMT solver. To our knowledge, λ^L is the first security-typed language that supports both expressive policies and automatic upgrading.

Term typing. The rest of Fig. 7 defines the typing judgments for expressions ($\Gamma; \sigma \vdash e :: T$) and statements ($\Gamma; \sigma \vdash e :: T$). Since λ^L is stateful, both judgments keep keeps track of σ, the variable that stands for the current store. This variable is used in the rule T-Get and T-Set to, respectively, relate the retrieved value to the current store and record the effect on the store. The rule for conditionals (P-If) is standard, but we include it because verification of programs with policy checks relies crucially on its path-sensitivity: note how the branches are type-checked in an environment extended with a path condition, derived from the refinement of the guard.

Subtyping $\boxed{\Gamma \vdash T <: T'}$

$$\text{<:-Tag1}\ \frac{\Gamma \vdash \langle T \rangle^{\pi}}{\Gamma \vdash T\ <:\ \langle T \rangle^{\pi}}$$

$$\text{<:-Tag2}\ \frac{\Gamma \vdash T\ <:\ T' \qquad \Gamma \vDash r' \Rightarrow r}{\Gamma \vdash \langle T \rangle^{\lambda(s,u).r}\ <:\ \langle T' \rangle^{\lambda(s,u).r'}}$$

Expression Typing $\boxed{\Gamma; \sigma \vdash e :: T}$

$$\text{T-Get}\ \frac{\Gamma; \sigma \vdash x :: \texttt{Ref}\ \{B \mid r\}}{\Gamma; \sigma \vdash \textbf{get}\ x :: \{B \mid r \wedge v = \sigma[x]\}}$$

$$\text{T-If}\ \frac{\Gamma; \sigma \vdash e :: \{\texttt{Bool} \mid r\} \qquad \Gamma, [v \mapsto \top]r \vdash e_1 :: T \qquad \Gamma, [v \mapsto \bot]r \vdash e_2 :: T}{\Gamma; \sigma \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 :: T}$$

$$\text{T-Bind}\ \frac{\Gamma; \sigma \vdash e_1 :: \langle T_1 \rangle^{\pi} \qquad \Gamma; \sigma \vdash e_2 :: T_1 \to \langle T_2 \rangle^{\pi}}{\Gamma; \sigma \vdash \textbf{bind}\ e_1\ e_2 :: \langle T_2 \rangle^{\pi}}$$

$$\text{T-}\lfloor\cdot\rfloor\ \frac{\Gamma; \sigma \vdash e :: \langle \{\texttt{Bool} \mid v \Rightarrow r\} \rangle^{\lambda(s,u).\pi[(s,u)] \wedge r}}{\Gamma; \sigma \vdash \lfloor e \rfloor :: \langle \{\texttt{Bool} \mid v \Rightarrow r\} \rangle^{\pi}}$$

Statement Typing $\boxed{\Gamma; \sigma \vdash s}$

$$\text{T-Print}\ \frac{\Gamma; \sigma \vdash x_1 :: \langle \{\texttt{User} \mid \pi[(\sigma, v)]\} \rangle^{\pi} \qquad \Gamma; \sigma \vdash x_2 :: \langle \texttt{Str} \rangle^{\pi} \qquad \Gamma; \sigma \vdash s}{\Gamma; \sigma \vdash \textbf{print}\ x_1\ x_2\ ;\ s}$$

$$\text{T-Set}\ \frac{\Gamma; \sigma \vdash x_1 :: \texttt{Ref}\ T \qquad \Gamma; \sigma \vdash x_2 :: T \qquad \Gamma, \sigma' : \{\Sigma \mid v = \sigma[x_1 := x_2]\}\ ;\ \sigma' \vdash s \qquad \sigma' \text{ is fresh}}{\Gamma; \sigma \vdash \textbf{set}\ x_1\ x_2\ ;\ s}$$

Figure 7. Typing rules of λ^L.

The core of information flow checking in λ^L are the rules T-Bind and T-Print, which, in combination with contravariant subtyping, guarantee that tagged values only flow into allowed contexts. To this end, T-Bind postulates that applying a sensitive function to a sensitive value, yields a result that is at least as secret as either of them. According to T-Print, a **print** statement takes as input a tagged user (which may be computed from sensitive data) and a tagged result. The rule requires both arguments to be tagged with the same policy π, and crucially, π must hold of the viewer in the current store (*i.e.* both the viewer identity and the message must be visible to the viewer). Here $\pi[(\sigma, v)]$ stands for "applying" the policy predicate; formally $(\lambda(s,u).p)[(\sigma, v)] \doteq p[s \mapsto \sigma, u \mapsto v]$.

Downgrading. The *safe downgrading* construct, $\lfloor e \rfloor$, is a novel feature of λ^L, which we introduced specifically to support static verification of programs with self-referential policies (Sec. 2.4.2). Informally, the idea is that we can we can safely downgrade a tagged term (i.e. weaken its policy), whenever we can prove that the term is constant, since constants cannot leak information. Whereas this property

is hard to check automatically in the general case, a special case where e is a tagged *boolean* turns out to be both amenable to automatic verification and particularly useful for self-referential policies. The rule T-$\lfloor\cdot\rfloor$ allows tagging $\lfloor e \rfloor$ with $\lambda(s,u).p$ as long as there exists a refinement predicate r over program variables, such that e can be tagged with $\lambda(s,u).p \wedge r$ and the value of e implies r. This operation is safe because in any execution where r holds, the two policies are the same; while any execution where r does not hold, the value of e is guaranteed to be `false` (a constant).

To illustrate the application of this rule, consider a simplified version of the patch form Sec. 2.4.2 where `authors` has a self-referential policy $\pi \doteq \lambda.u \in s$[`authors` p]. In this case, to decide whether to show the author list to `client`, the patch has to check whether `client` is on the list, *i.e.* compute `bind (get (authors p)) (x2 . elem client x2)`. Since this term retrieves the author list, it has to be itself tagged with π, preventing the patch form type-checking. Wrapping the policy check in $\lfloor\cdot\rfloor$ breaks this circularity and allows tagging it with $\lambda.u =$ `client` $\wedge\, s = \sigma$, (since its value implies `client` $\in \sigma$[`authors` p]), causing the patch to type-check.

Algorithmic type checking. As is customary for expressive type systems, the rules in Fig. 7 are not algorithmic: they require "guessing" appropriate policy predicates for intermediate terms (when applying rules T-Print and T-Bind), as well as the predicate r in T-$\lfloor\cdot\rfloor$. Our insight is that we can re-purpose liquid type inference [12, 36, 44], which has been previously used to automatically discover unknown refinements, to also discover these unknown predicates. To this end, our typing rules are carefully designed to respect the restrictions imposed by Liquid Types, such as that all unknown predicates occur positively in specifications. As a result, we obtain fully automatic verification for programs with (decidable) context-dependent policies.

3.3 Contextual Noninterference in λ^L

We want to show that well-typed λ^L programs indeed do not leak information. In the presence of context-dependent policies, defining what exactly constitutes a leak is non-trivial: we cannot directly apply the traditional notion of noninterference because our policies can depend on the sensitive values they protect. Instead, we enforce *contextual non-interference*, a guarantee similar to that of the Jeeves language. In the interest of space, this section formalizes contextual non-interference in the absence of store updates and omits proofs; the full version of our formalization can be found in Appendix A.4.

Intuitively, we require that an observer o : `User` cannot observe the difference between two stores that only differ in locations secret from o. However, which locations are "secret" depends on the store. Following Jeeves, we only require that o cannot observe a difference in location l if l is secret in *both stores* (*e.g.* it's fine if I notice the difference between a real paper status I can see and a default status `NoDecision`).

1: Enforce(Γ, e, T)
2: $\quad\hat{e} \leftarrow$ Localize($\Gamma \vdash e :: T$)
3: $\quad$**return** Patch($\hat{e}$)

4: Patch(**let** $x = \langle T_e \triangleleft T_a \rangle d$ **in** e)
5: $\quad d' \leftarrow$ Generate($[x_0 : T_a], \Gamma, T_e$)
6: $\quad$**return let** $x =$ (**let** $x_0 = d$ **in** d') **in** Patch(e)
7: Patch(e)
8: $\quad$recursively call Patch on subterms of e

9: Generate($\Gamma_B, \Gamma_G, \langle T \rangle^\pi$)
10: $\quad\Gamma_B \leftarrow \Gamma_B \cup$ redaction functions for T
11: $\quad$*branches* $\leftarrow$ Synthesize($\Gamma_B \vdash$ `??` $:: \langle T \rangle^{\mathsf{none}}$)
12: $\quad$(*dflt* : *guarded*) $\leftarrow$ sort *branches* by policy
13: $\quad$**if** Check($\Gamma \vdash$ *dflt* $:: \langle T \rangle^\pi$) **then**
14: $\quad\quad$*patch* $\leftarrow$ *dflt*
15: $\quad$**else fail**
16: $\quad$**for** $b \leftarrow$ *guarded* **do**
17: $\quad\quad\psi \leftarrow$ Abduce(Γ_G, `??` $\vdash b :: \langle T \rangle^\pi$)
18: $\quad\quad T_g \leftarrow \langle \{\nu :$ `Bool` $\mid \nu \Leftrightarrow \psi\} \rangle^\pi$
19: $\quad\quad$*guard* $\leftarrow$ Synthesize($\Gamma \vdash$ `??` $:: T_g$)
20: $\quad\quad$*patch* $\leftarrow$ **bind**(*guard*)($\lambda g.$**if** g **then** b **else** *patch*)
21: $\quad$**return** *patch*

Figure 8. Policy enforcement algorithm

Definition 3.1 (observational equivalence)**.** For some *observer* o: `User`, two stores σ_1, σ_2 are *o-equivalent*—written $\sigma_1 \sim_o \sigma_2$—if they hold the same value at every location l visible to o in either store:

$$\forall l.\ \mathrm{ty}(l) = \mathtt{Ref}\ \langle T \rangle^\pi \wedge \big(\pi[(\sigma_1, o)] \vee \pi[(\sigma_2, o)]\big) \Rightarrow \sigma_1[l] = \sigma_2[l]$$

Theorem 3.2 (contextual noninterference)**.** *Let s be a λ^L program and let σ_1, σ_2 be two stores. Assume that running s on $\sigma_{1,2}$ produces outputs $\langle o_{i,j}, v_{i,j} \rangle$, for $i \in 1..2, j \in 1..k$, where $o_{i,j}$ is the viewer of output $v_{i,j}$.*

For any observer o, if $\sigma_1 \sim_o \sigma_2$, then for all $j \in 1..k$, $o_{1,j} = o \Leftrightarrow o_{2,j} = o$ and $o_{1,j} = o \Rightarrow v_{1,j} = v_{2,j}$.

4 Targeted Synthesis for λ^L

We now turn to the heart of our system: the algorithm Enforce (shown in Fig. 8), which takes as input a type environment Γ, a program e (in A-normal form), and a top-level type annotation T, and determines whether policy-enforcing code can be injected into e to produce e', such that $\Gamma \vdash e' :: T$. 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" $\hat{e}$ (Sec. 4.1). Second, procedure Patch traverses $\hat{e}$ (line 3) replacing each type cast with an appropriate patch, generated by the procedure Generate (Sec. 4.2).

4.1 Fault Localization

Type casts. For the purpose of fault 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 [26]; in particular, the cast $\langle T \triangleleft T' \rangle$ has type $T' \to T$. However, the dynamic semantics of casts in λ^L is undefined. The idea is that casts are inserted solely for the purpose of targeting synthesis, and, if synthesis 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*.

Sound localizations. Algorithm Localize first uses liquid type inference [12, 36, 44] to reduce the problem of checking the source program e against type T to a system of Horn constraints. If the constraints have a solution, it returns e unmodified; otherwise, instead of simply signaling an error like existing liquid type checkers, it attempts to construct a *sound localization* of e, which is a program $\hat{e}$ that satisfies the following properties: (1) $\hat{e}$ is obtained from e by inserting type casts, *i.e.* replacing one or more subterms e_i in e by $\langle T_i \triangleleft T_i' \rangle e_i$; (2) it is type correct, *i.e.* $\Gamma \vdash \hat{e} :: T$ In particular, note that (2) implies that each e_i has type T_i'.

Lemma 4.1 (Localization). *Replacing each subterm of the form $\langle T_i \triangleleft T_i' \rangle\ e_i$ in a sound localization of e with a type-safe term of type T_i, yields a type-safe program.*

This lemma follows directly form (2) and a standard substitution lemma for refinement types [26]. Crucially, it shows that once a sound localization has been found, patch generation can proceed independently for each type cast.

Minimal localizations. Among sound localizations, not all are equally desirable. Intuitively, we would like to make minimal changes to the behavior of the original program. Formalizing and checking this directly is hard, so we approximate it with the following two properties. A sound localization is *syntactically minimal* if no type cast can be removed or moved to a subterm[3]without breaking soundness. Picking syntactically minimal localizations leads to patching smaller terms, rather than trying to rewrite the whole program.

Once the unsafe terms are fixed, we can still pick different expected types T_i. Intuitively, the more restrictive the T_i, the less likely are we to find the patch to replace the cast. A *minimal localization* is syntactically minimal, and all its expected types cannot be made any less restrictive without breaking soundness. In general, there can be multiple minimal localizations, and a general program repair engine would have to explore them all, leading to inefficiency. For the specific problem of policy enforcement, however, there is a reasonable default, which we infer as shown below.

[3]In this context, the definition of a let-bound variable is considered a subterm of the let body

Inferring the localization. Given an unsatisfiable system of Horn constraints, Localize first makes sure that all conflicting clauses have been generated by implication checks on policy predicate, and removes those clauses that were generated by the smallest term. It then re-runs the fixpoint solver [12, 36] on the remaining system, inferring strongest solutions for policy predicates, after which it reset the non-policy refinements of the removed terms to $\top$ and re-check the validity of the constraints. If the constraints are satisfied, we have obtained a sound and minimal localization (the expected types are the least restrictive because policies are strongest, and other refinements are $\top$). If the constraints are violated, it indicates that the program depends on some functional property of the unsafe term we want to replace. We consider such programs out of scope: if a programmer wants to benefit form automatic policy enforcement, they have to give up the ability to reason about functional properties of sensitive values, since our language reserves the right to substitute them with other values.

4.2 Patch Generation

Next, we describe how our algorithm replaces a type-cast $\langle T_e \triangleleft T_a \rangle\ d$ with a type-safe term d' of the expected type T_e, using the patch generation procedure Generate (line 5). At a high level, the goal of this step is to generate a term from a given refinement type T_e, which is the problem tackled by type-driven synthesis as implemented in Synquid [34]. Unfortunately, Generate cannot use Synquid out of the box, because the expected type T_e is not a full functional specification: this type only contains policies but no type refinements, allowing trivial solutions to the synthesis problem, such as unconditionally returning an arbitrary constant with the right type shape.

To avoid such undesired patches, procedure Generate implements a specialized synthesis strategy: first, it generates a list of *branches*, which return the original term redacted to a different extent; then, for each branch, it infers an optimal *guard* (a policy check), that makes the branch satisfy the expected type; finally, it constructs the patch by arranging the properly guarded branches into a (monadic) conditional.

Synthesis of branches. In line 11, Generate uses Synquid to synthesize the set of all terms up to certain size with the right content type, but with no restriction on the policy (here, `none` = λ.`False`). Note that branches are generated in a restricted environment Γ_B, which contains only the original faulty term, the "default" value of type T, and a small set of *redaction functions* for this type (such as `mapJust` for `Maybe` in Sec. 2.4.3). This restriction makes the branch synthesis both more predictable and more efficient.

Default branch. Once the branches have been generated, we sort them according to their actual policy predicate, form weakest to strongest (*i.e.* in the reverse order of how they are going to appear in the program). In line 13, we check

that the first branch can be used as the *default branch*, *i.e.* it satisfies the expected type unconditionally. This property is always satisfied as long as Γ_B contains a value v of type T, since in our type system $T <: \langle T \rangle^\pi$ for any π. For this check, we use the original liquid type checking unmodified.

Synthesis of guards. For each of the other branches b (which include at least the original term), GENERATE attempts to synthesize the optimal guard that would make b respect the expected type. At a high level, this guard must be logically equivalent to a formula ψ, such that (1) $\psi \wedge p \Leftrightarrow q$, where $\lambda(s,u).p \doteq \pi$ is the expected policy of the patch, and $\lambda(s,u).q$ is the actual policy of branch b; (2) ψ does not mention the policy parameters s and u. This predicate can be inferred using existing techniques, such as logical abduction [13]. In particular, GENERATE relies on SYNQUID's *liquid abduction* mechanism [34] to infer ψ in line 17.

The main challenge of guard synthesis, however, 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 ψ, and (2) *no leaky enforcement*: the guard itself respects the expected policy π of the patch. To ensure both conditions, we obtain the guard via type-driven synthesis, providing $\langle \{v : \texttt{Bool} \mid v \Leftrightarrow \psi\} \rangle^\pi$ as the target type.

Lemma 4.2 (Safe patch generation). *If GENERATE succeeds, it produces a type-safe term of the expected type $\langle T \rangle^\pi$.*

Assuming correctness of SYNTHESIZE and ABDUCE, we can use the typing rules of Sec. 3 to show that the invariant $\Gamma \vdash \mathit{patch} :: \langle T \rangle^\pi$ is established in line 14 and maintained in line 20. In particular, the type of bound variable g in line 20 is $\{v : \texttt{Bool} \mid v \Leftrightarrow \psi\}$, hence, **then** branch is checked under the path condition $\psi \Leftrightarrow \top$, which implies $\Gamma_G \vdash b :: \langle T \rangle^\pi$.

We would also like to provide a guarantee that a patch produced by GENERATE is *minimal*, *i.e.*, in each concrete execution, its return value retains the maximum information allowed by π from the original term. We can show that, for a fixed set of generated branches, the patch will always pick the most sensitive one that is allowed by π, since the guards characterize precisely when each branch is safe. Of course, the set of generated branches is restricted to terms of certain size constructed from components in Γ_B. The original term, however, is always in Γ_B, hence we are guaranteed to retain the original value whenever allowed by π.

4.3 Guarantees and Limitations

In this section we summarize the soundness guarantee of targeted synthesis in λ^L and then discuss the limitations on its completeness and minimality.

Theorem 4.3 (Soundness of targeted synthesis). *If procedure ENFORCE succeeds, it produces a program that satisfies contextual noninterference.*

Benchmark	Compilation time		
	Localize	Generate	Total
Basic policy	0.04s	0.09s	0.14s
Self-referencing policy	0.00s	0.17s	0.19s
Implicit flow	0.01s	0.29s	0.30s
Filter by author	0.06s	0.61s	0.68s
Sort by score	0.03s	0.86s	0.90s
Send to multiple users	0.00s	0.34s	0.35s
Interleaved reads/writes	0.01s	0.72s	0.74s
Copy a private field	0.00s	0.19s	0.20s

Table 1. Micro-benchmarks, with compile-time statistics.

This is straightforward by combining Lemmas 4.1 and 4.2 with Theorem 3.2.

Completeness. When does procedure ENFORCE fail? Sec. 4.1 explains how LOCALIZE can fail when the inferred localization is not safe. GENERATE can fail in lines 13, 17, and 19. The first failure indicates that Γ_B does not contain any sufficiently public terms (in particular, there is no default value). The second failure can happen if the abduction engine is not powerful enough (this does not happen in our case studies). The third failure is the most interesting one: it happens when no guard satisfies both functional and security requirements, indicating that the policy is *not enforceable* without leaking some other sensitive information.

Minimality. We would like to show that the changes made by ENFORCE are minimal: in any execution where e did not cause a leak, e' would output the same values as e. Unfortunately, this is not true, even though we have shown that LOCALIZE produces least restrictive expected types and GENERATE synthesizes minimal patches. The reason is that even the least restrictive expected type might over-approximate the set of output contexts, because of the imprecisions of refinement type inference. In these cases, ENFORCE is conservative: *i.e.* it hides more information than is strictly necessary. One example is if the state is updated in between the **get** and the **print** by calling a function for which no precise refinement type can be inferred. Another example is when the same sensitive value with a viewer-dependent policy is displayed to multiple users. In our case studies, we found that in the restricted class of data-centric applications that LIFTY is intended for, these patterns occur rarely. One approach to overcoming this limitation would be to combine targeted synthesis with runtime techniques similar to Jeeves.

5 Evaluation

We implemented a set of microbenchmarks and larger case studies and demonstrate the following.

Expressiveness of policy language. We use LIFTY to implement a conference manager, a grading application, and a health portal[4]. The two authors who developed the case

[4]The first two applications are based on case studies for the policy-agnostic Jacqueline system [48]. The health portal is based on the HealthWeb example from the Fine paper [41].

(a) Conference Management System — Policy size (tokens): 345

	Program size (tokens)			Time			
		Policy enforcing		Manual	Auto		
Benchmark	Original	Manual	Auto	Verify	Localize	Generate	Total
Register user	10	0	0	0.00s	0.00s	0.00s	0.00s
View users	20	9	24	0.01s	0.01s	0.58s	0.59s
Paper submission	45	0	0	0.00s	0.00s	0.00s	0.00s
Search papers	77	96	82	0.90s	0.08s	7.03s	7.11s
Show paper record	53	46	82	0.34s	0.04s	7.89s	7.94s
Show reviews for paper	57	54	45	0.39s	0.07s	0.58s	0.66s
User profile: GET	66	0	0	0.03s	0.03s	0.00s	0.03s
User profile: POST	17	0	0	0.00s	0.00s	0.00s	0.00s
Submit review	40	0	0	0.00s	0.00s	0.00s	0.00s
Assign reviewers	47	0	0	0.01s	0.01s	2.25s	2.26s
Totals	432	205	233	1.71s	0.27s	18.36s	18.64s

(b) Gradr—Course Management and Interactive Grading System — Policy size (tokens): 141

	Program size (tokens)		Time		
		Policy enforcing			
Benchmark	Original	(Lifty)	Localize	Generate	Total
Display the home page	12	0	0.00s	0.00s	0.00s
Student: get classes	13	0	0.04s	0.00s	0.04s
Instructor: get classes	13	0	0.00s	0.00s	0.00s
Get class information for a user	29	0	0.01s	0.00s	0.01s
View a user's profile (owner)	23	0	0.00s	0.00s	0.00s
View a user's profile (any user)	25	19	0.01s	0.04s	0.06s
Instructor: view scores for an assignment	36	17	0.00s	0.03s	0.05s
Student: view all scores for user	36	17	0.00s	0.03s	0.05s
Instructor: view top scores for an assignment	62	17	0.02s	0.03s	0.06s
Totals	253	70	0.13s	0.17s	0.31s

(c) HealthWeb—Health Information Portal — Policy size (tokens): 194

	Program size (tokens)		Time		
		Policy enforcing			
Benchmark	Original	(Lifty)	Localize	Generate	Total
Search a record by id	23	293	0.00s	12.10s	12.11s
Search a record by patient	56	293	0.03s	13.10s	13.14s
Show authored records	45	0	0.02s	0.00s	0.02s
Update record	34	0	0.00s	0.15s	0.15s
List patients for a doctor	52	44	0.03s	0.09s	0.13s
Totals	216	630	0.10s	25.46s	25.57s

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

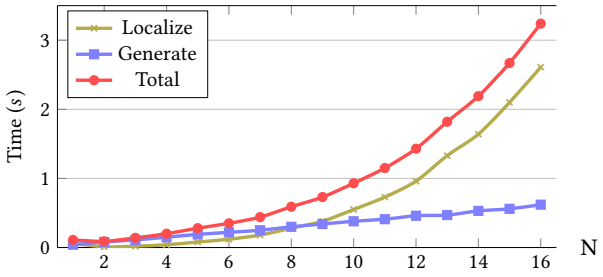

Figure 9. Scalability—N accesses in a single function.

studies were not involved in the development of Lifty. We demonstrate that Lifty's policy language supports the desired policies for these systems.

Scalability. We demonstrate that the Lifty compiler is sufficiently efficient at error localization and synthesis to use for systems of reasonable size: Lifty is able to generate all necessary checks for our conference manager (424 lines of Lifty) in about 20 seconds. Furthermore, we show that synthesis times are linear in the number of required patches.

Quality of patches. We compare the code generated by Lifty to a version with manually written policy checks and show that not only does Lifty allow for policy descriptions to be centralized and concise, but also the compiler is able

to recover *all* necessary checks, without reducing the functionality.

5.1 Microbenchmarks and Case Studies

We implemented the following code using Lifty.

Microbenchmarks. To exercise the flexibility of our language, we implemented a series of small but challenging microbenchmarks, described in Tab. 1.

Conference manager. We implemented two versions of a basic academic conference manager: one where the programmer enforces the policies by hand and one where Lifty is responsible for injecting the policy checks. The manager handles confidentiality policies for papers in different phases of the conference and different paper statuses, based on the role of the viewer. Policies depend on this state, as well as additional properties such as conflicts with a particular paper. The system provides features for displaying the paper title and authors, status, list of conflicts, and conference information conditional on acceptance. Information may be displayed to the user currently logged in or sent via various means to different users. The system contains 888 lines of code in total (524 Lifty + 364 Haskell).

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 example for the Fine language [41], we implemented a system that supports the enforcement of information flow policies in the context of viewing and searching over health records. The complexity of the policies and functions of this case study make it interesting. We describe it in more detail in Appendix A.5.

5.2 Performance Statistics

We show running times for the microbenchmarks in Tab. 1, and for the case studies in Tab. 2. We break them down into fault localization (including type checking) and patch synthesis. For the conference manager, we show a comparison between the version with manual policy checks and a version with automatically generated checks. For the version that contains manual checks, we show only verification time, as Lifty skips the other phases. Notice that Lifty is able to determine that six of our benchmarks required no patches at all: in particular, all store updates are safe.

Scaling. Because of the way targeted synthesis works, synthesizing patches for each function is independent. Cross effects arise only from (1) interactions between policies and (2) having more generic components in scope, as the synthesizer needs to search over this space. For this reason, Lifty scales *linearly* with respect to the number of functions in the program. For a stress test, we created a benchmark test that performs N reads (of the same field, for convenience) and then a `print` to an arbitrary user. Lifty's job is to patch all of the `get` locations with a conditional. We show in Fig. 9 that patch generation time is linear in N. Verification (including error localization) is still quadratic in N. This currently dominates the compilation time.

5.3 Measuring the Quality of Patches

We compared the two versions of our conference manager (Tab. 2). The size of the checks confirms our hypothesis that for data-centric applications, much of the programming burden is in policy-enforcing code. Our results reveal that while manual checks are more concise than Lifty-generated checks, the bloat in the generated code comes from unnecessary verbosity, and affects neither its functionality nor performance. The manual and automatic checks were semantically equivalent across our benchmarks.

6 Related Work

Program synthesis and repair. Our approach differs from existing program synthesis techniques [2–4, 15, 16, 22, 27, 30, 33, 34, 39], which synthesize programs from scratch, from end-to-end functional specifications, while Lifty performs synthesis for the cross-cutting program concern of information flow. Our goal is similar to that of sound program repair [24], but in the specific setting of policy enforcement, Lifty is able to perform a much more precise fault localization, and synthesize all patches locally, which makes it more scalable. Prior work on rewriting programs based on security concerns [17, 18, 21, 40] does not involve reasoning about expressive information-flow policies.

Information flow control. Lifty provides a high-level programming interface to support security guarantees based on a long history of work in language-based information flow [38]. Both static and dynamic label-based approaches [5, 8, 11, 19, 29, 32, 35, 50, 51] allow programmers to label data with security levels and check programs for unsafe flows. Labels are, however, low level: they trust the programmer to correctly express high-level policies in terms of label-manipulating code. More importantly, none of these approaches address the issue of or programmer burden: static approaches simply prevent unsafe programs from compiling; the dynamic approaches raise exceptions or silently fail.

Lifty takes a policy-agnostic approach [6, 48, 49] and factors information flow out from core program functionality, allowing programmers to implement policies as high-level predicates over the program state. Prior work uses runtime enforcement, which yields nontrivial runtime overheads and makes it difficult to reason about program behavior. Lifty addresses these issues by providing a synthesis-driven approach to support the Jeeves semantics statically, supporting an analogous security property.

Type systems. Lifty's type system, as well as the monadic encoding for information flow, are inspired by other value-dependent type systems [9, 10, 23, 41, 42]. The key difference is the decidability of both type inference, which yields automated verification and fault localization, crucial for targeted synthesis. Lifty's type inference engine is built on top of the Liquid Types framework [12, 36, 44–46], and extends it with a fault localization mechanism tailored towards security types. Our technique for using types for program rewriting resembles both hybrid type checking [26] and type-directed coercion insertion [43]. Both our types and rewriting capabilities are more advanced and thus can handle global, cross-cutting concerns such as information flow.

References

[1] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *CSF*.

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*.

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV*.

[4] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *CAV*.

[5] 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*.

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

[7] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 218–228.

[8] N. Broberg and David Sands. 2006. Flow Locks: Towards a core calculus for Dynamic Flow Policies. In *ESOP (LNCS)*, Vol. 3924. Springer Verlag.

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

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

[11] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged Information Flow for Javascript. In *PLDI*.

[12] Benjamin Cosman and Ranjit Jhala. 2017. Local refinement typing. *PACMPL* 1, ICFP (2017), 26:1–26:27. https://doi.org/10.1145/3110270

[13] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 684–689.

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*.

[16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *POPL*.

[17] 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*.

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

[19] 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 *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 47–60.

[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330.

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

[22] Jeevana Priya Inala, Xiaokang Qiu, Ben Lerner, and Armando Solar-Lezama. 2015. Type Assisted Synthesis of Recursive Transformers on Algebraic Data Types. *CoRR* abs/1507.05527 (2015).

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

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

[25] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 407–426.

[26] Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages. https://doi.org/10.1145/1667048.1667051

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

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

[29] 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.

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

[31] Simon Marlow. 2010. Haskell 2010 language report. (2010). https://www.haskell.org/onlinereport/haskell2010/

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

[33] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *PLDI*.

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

[35] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems* 25, 1 (Jan. 2003).

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

[37] Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-weight Information-flow Security in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

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

[39] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*.

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

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

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

[43] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. 2009. A Theory of Typed Coercions and Its Applications. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA.

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

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

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

[47] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26. https://doi.org/10.1145/3133887

[48] 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*.

[49] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. (2012).

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

[51] 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.

A Appendix

A.1 Desugaring do-notation

This is code from Fig. 3 with the **do**-notation desugared into invocations of **bind**.

```
showPaper client p =
  let row =
    bind (get (title p)) (λ t .
    bind (get (status p)) (λ st .
    bind (if st = Accepted
      then get (session p) else return "") (λ ses .
    return (t + "␣" + ses))))
  print client row
```

A.2 Operational Semantics of λ^L

The runtime behavior of λ^L programs is straightforward and is summarized in Fig. 10. Expression evaluation happens in the context of a store σ : (Loc ⟶ Value) ∪ (User ⟶ Value), which has two components, mapping location to values and users to their corresponding output. The statements **set** and **print** modify the two components of the store respectively.

The dynamic semantics of tagged primitives is not very interesting, which is not surprising, since λ^L only tracks policies statically. $\lfloor\cdot\rfloor$ simply returns its argument, while **bind** calls its second argument on the first. At runtime a tagged computation is indistinguishable from a computation on untagged values (in fact, you might have noticed that **bind** corresponds to the bind of the identity monad).

A.3 The λ^L Type System

We show the full typing rules in Fig. 11.

A.4 Contextual Noninterference with Store Updates

In the presence of store updates, there is an additional subtlety in the definition of contextual noninterference. As the program executes and writes to the store, some previously secret locations can become visible, hence we only require that o cannot observe a difference in location l if l is secret *throughout* both program executions (*e.g.* it's fine if I notice the difference in paper status if the phase advanced halfway through the program execution and it became visible).

Definition A.1 (observational equivalence)**.** For some *observer* o : `User` and a set of stores $\Delta = \{\Delta_1, \cdots, \Delta_n\}$, two stores σ_1, σ_2 are $\langle o, \Delta\rangle$-*equivalent* — written $\sigma_1 \sim_{o,\Delta} \sigma_2$ — if

$$\forall l, p.\ \mathrm{ty}(l) = \texttt{Ref}\ \langle T\rangle^p \wedge \left(\bigvee_{\Delta_i \in \Delta} p(\Delta_i, o)\right) \Rightarrow \sigma_1[l] = \sigma_2[l]$$

That is: at every location l visible to o in any Δ_i, the stores hold the same value.

When Δ is omitted, it means $\Delta = \{\sigma_1, \sigma_2\}$.

In order to discuss the privacy properties of the language, we need to add some annotation to program terms.

Definition A.2 (semantic annotations)**.** $\lambda^{\langle L\rangle}$ is the language obtained from λ^L by adding one more case to v:

$$v ::= \cdots \mid \langle v\rangle^p$$

The operational semantics rules remain the same, ignoring and bypassing any $\langle\cdot\rangle^p$ annotations. The rule for **let** is slightly changed so that the substituted value is annotated with p whenever the type of the bound variable is $\langle T\rangle^p$.

Definition A.3 (observational equivalence for program terms)**.** For two terms t_1, t_2 (either expressions or statements), and for an observer o and stores Δ as before, the terms are $\langle o, \Delta\rangle$-*equivalent* — $t_1 \sim_{o,\Delta} t_2$ — when:

- They are syntactically identical, except at annotated values;
- For corresponding annotated values $\widehat{v_1} = \langle v_1\rangle^p$, $\widehat{v_2} = \langle v_2\rangle^{p'}$ they agree on the tag $p = p'$, and

$$\left(\bigvee_{\Delta_i \in \Delta} p(\Delta_i, o)\right) \Rightarrow v_1 = v_2$$

We formalize our contextual noninterference theorem as follows.

First we state the full theorem, which includes writes and reasons about λ^L programs containing **set** statements.

Theorem A.4 (contextual noninterference)**.** *Let s be a λ^L program and let σ_1, σ_2 be two stores. Observe the two $\lambda^{\langle L\rangle}$-traces of s on these stores:*

$$\sigma_j, s \longrightarrow \sigma_j^{(1)}, s^{(1)} \longrightarrow \sigma_j^{(2)}, s^{(2)} \longrightarrow \cdots \longrightarrow \sigma_j^{(k)}, s^{(k)}$$

(notice that the traces must be of equal length) and define $\Delta = \bigcup_{j\in\{1,2\}, i\in 1..k}\{\sigma_j^{(i)}\}$.

If $\sigma_1 \sim_{o,\Delta} \sigma_2$, then $\sigma_1^{(i)} \sim_{o,\Delta} \sigma_2^{(i)}$ for all $i \in 1..k$.

Notice that this generalizes our previous handling of **print** statements, since the output displayed to each observer can be modeled by an array of `Ref`s, one per user, with the policy that only that user can see them, such that **print** appends to these stores.

We now prove four lemmas, one for expressions, and one for each type of statements. In all of them, we implicitly assume terms are well-typed. The proofs are rather boring so only a brief sketch is given.

Lemma A.5 (contextual noninterference for expressions)**.** *Let*

- *$\sigma_1, \sigma_2, \Delta$ stores such that $\sigma_1 \sim_{o,\Delta} \sigma_2$;*
- *e_1, e_2 expression such that $e_1 \sim_{o,\Delta} e_2$;*
- *Γ, y, T, p such that $\Gamma; y \vdash e_j :: \langle T\rangle^p$ for $j \in \{1,2\}$ and i such that $p(\Delta_i, o)$;*
- *$\sigma_j, e_j \longrightarrow^* c_j$ for $j \in \{1,2\}$;*

then $c_1 = c_2$.

Proof. This lemma constitutes the core of the non-interference property. We the standard technique of Pottier and Simonet [35]: define an auxiliary language λ_2^L where every tagged value has two components, one for each of σ_j, and show that a term

Expression Evaluation $\boxed{\sigma, e \longrightarrow e}$

$$\text{e-}\beta\ \frac{}{\sigma, (\lambda x : T.\ e_1)\ e_2 \longrightarrow [x \mapsto e_2]e_1} \qquad \text{e-get}\ \frac{}{\sigma, \mathbf{get}\ l \longrightarrow \sigma[l]}$$

$$\text{e-true}\ \frac{}{\sigma, \mathbf{if}\ \text{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow e_1} \qquad \text{e-false}\ \frac{}{\sigma, \mathbf{if}\ \text{false}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow e_2}$$

$$\text{e-bind}\ \frac{}{\sigma, \mathbf{bind}\ e_x\ (\lambda x : T.e) \longrightarrow [x \mapsto e_x]e} \qquad \text{e-downgrade}\ \frac{}{\sigma, \lfloor e \rfloor \longrightarrow e}$$

$$\text{e-ctx}\ \frac{e \longrightarrow e'}{C[e_1] \longrightarrow C[e_2]}$$

where $C ::= \bullet \mid C\ e \mid e\ C \mid \lambda x : T.\ C \mid \mathbf{get}\ C \mid \mathbf{if}\ C\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{bind}\ e\ C$

Statement Execution $\boxed{\sigma, s \longrightarrow \sigma, s}$

$$\text{let}\ \frac{\sigma, e \longrightarrow^* v}{\sigma, \mathbf{let}\ x = e\ \mathbf{in}\ S \longrightarrow \sigma, [x \mapsto v]S}$$

$$\text{set}\ \frac{}{\sigma, \mathbf{set}\ l\ v\ ;\ S \longrightarrow \sigma[l := v], S} \qquad \text{print}\ \frac{}{\sigma, \mathbf{print}\ u\ v\ ;\ S \longrightarrow \sigma[u \mathrel{+}= v], S}$$

Figure 10. λ^L operational semantics.

visible to the observer o will evaluate to a value with equal components, assuming that the stores hold equal values at references visible to o.

We omit the full formalization because it is mostly tedious. We just note that it relies crucially on the λ^L_2 definition of the tagged primitives **bind** and $\lfloor\cdot\rfloor$; in particular, **bind** v f executes f on both components of v, and $\lfloor\cdot\rfloor$ increases visibility exactly for those cases where both components are known to be equal. The subtyping rules make sure that "upcasts" can only strengthen the policy tag p, so tagged values with non-equal components can never become visible again.

Lemma A.6 (contextual noninterference for "print" statements). *Let* $\sigma_1 \sim_{o,\Delta} \sigma_2$,
$s_j = \mathbf{print}\ \langle u_j \rangle^p\ \langle v_j \rangle^p\ ;\ t_j$ *for* $j \in \{1, 2\}$,
such that $s_1 \sim_{o,\Delta} s_2$ *and* $\sigma_j, s_j \longrightarrow \sigma'_j\ ;\ s'_j$.
Assume $\sigma_{1,2}, \sigma'_{1,2} \in \Delta$.
Then $\sigma'_1 \sim_{o,\Delta} \sigma'_2$ *and* $s'_1 \sim_{o,\Delta} s'_2$.

Proof. The semantics of **print** is that it modifies the location u_j in the store. From the typing rules for **print** we know that $p(\sigma_j, u_j)$. So if either $u_1 = o$ or $u_2 = o$, we get $p(\sigma_j, o)$, therefore from $\langle o, \Delta \rangle$ equivalence $u_1 = u_2 = o$ and $v_1 = v_2$.

By statement execution rules, $s'_j = t_j$ which are substatements of s_j and equivalence follows from the definition.

Lemma A.7 (contextual noninterference for "set" statements). *Let* $\sigma_1 \sim_{o,\Delta} \sigma_2$,
$s_j = \mathbf{set}\ l\ \langle v_j \rangle^p\ ;\ t_j$ *such that* $\text{ty}(l) = \mathit{Ref}\ \langle T \rangle^p$ *for* $j \in \{1, 2\}$,
such that $s_1 \sim_{o,\Delta} s_2$ *and* $\sigma_j, s_j \longrightarrow \sigma'_j\ ;\ s'_j$.
Assume $\sigma_{1,2}, \sigma'_{1,2} \in \Delta$.
Then $\sigma'_1 \sim_{o,\Delta} \sigma'_2$ *and* $s'_1 \sim_{o,\Delta} s'_2$.

Proof. Notice that in this case the location itself is not tagged so both executions alter the same key in the store. If $p(\Delta_i, o)$ for some i, then we know that $v_1 = v_2$; otherwise the mutated location is not observed hence the values are insignificant.

As in the **print** case, $s'_j = t_j$ and the rest is the same.

Lemma A.8 (contextual noninterference for "let" statements). *Let* $\sigma_1 \sim_{o,\Delta} \sigma_2$,
$s_j = \mathbf{let}\ x = e_j\ \mathbf{in}\ t_j$ *for* $j \in \{1, 2\}$,
such that $s_1 \sim_{o,\Delta} s_2$ *and* $\sigma_j, s_j \longrightarrow \sigma'_j\ ;\ s'_j$.
Assume $\sigma_{1,2}, \sigma'_{1,2} \in \Delta$.
Then $\sigma'_1 \sim_{o,\Delta} \sigma'_2$ *and* $s'_1 \sim_{o,\Delta} s'_2$.

Proof. If x does not have a tagged type, the theorem is trivial. Otherwise, let $x :: \langle T \rangle^p$. From Lemma A.5, if $p(\sigma_j, o)$ holds (for either $j \in \{1, 2\}$) then e evaluates to the same value on both stores; otherwise two tagged values $\langle c_j \rangle^p$ are created and substituted into s_j, and since $p(\sigma_j, o)$ this does not violate $\langle o, \Delta \rangle$ equivalence.

In both cases, **let** does not mutate the store, so $\sigma'_j = \sigma_j$, and obviously $\sigma'_1 \sim_{o,\Delta} \sigma'_2$.

Proof by induction on i, starting at $i = 0$ denoting the initial state. For each derivation step, either Lemma A.6, A.7, or A.8 applies.

With Theorem 3.2 we can be certain that if the permissions are set correctly, then no information flow can violate the policy throughout the execution of the program. The requirement is that any value that becomes public at any point, should be equal on the two *initial* stores. This is important because policies depend on the state of the store; so if a program grants permission to view a field that previously was

Well-formedness $\boxed{\Gamma \vdash r}$ $\boxed{\Gamma \vdash B}$ $\boxed{\Gamma \vdash S}$

$$\text{WF-}r\ \frac{\Gamma \vdash r : \texttt{Bool}}{\Gamma \vdash r} \qquad \text{WF-Tag}\ \frac{\Gamma \vdash T \qquad \Gamma, y : \texttt{Store}, u : \texttt{User} \vdash r}{\Gamma \vdash \langle T \rangle^{\lambda y.\lambda u.r}}$$

Subtyping $\boxed{\Gamma \vdash T <: T'}$ $\boxed{\Gamma \vdash B <: B'}$

$$<:\text{-Sc}\ \frac{\Gamma \vdash B\ <:\ B' \qquad \Gamma \vDash r \Rightarrow r'}{\Gamma \vdash \{B \mid r\}\ <:\ \{B' \mid r'\}} \qquad <:\text{-Fun}\ \frac{\Gamma \vdash T'_x\ <:\ T_x \qquad \Gamma \vdash T\ <:\ T'}{\Gamma \vdash T_x \to T\ <:\ T'_x \to T'}$$

$$<:\text{-Tag1}\ \frac{\Gamma \vdash \langle T \rangle^p}{\Gamma \vdash T\ <:\ \langle T \rangle^p} \qquad <:\text{-Tag2}\ \frac{\Gamma \vdash T\ <:\ T' \qquad \Gamma \vDash r' \Rightarrow r}{\Gamma \vdash \langle T \rangle^{\lambda y.\lambda u.r}\ <:\ \langle T' \rangle^{\lambda y.\lambda u.r'}}$$

$$<:\text{-Refl}\ \frac{}{\Gamma \vdash B\ <:\ B}$$

Expression Typing $\boxed{\Gamma; y \vdash e :: T}$

$$\text{T-C}\ \frac{}{\Gamma; y \vdash c :: \text{ty}(c)} \qquad \text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma; y \vdash x :: T} \qquad \text{T-}\lambda\ \frac{\Gamma \vdash T_x \qquad \Gamma, x : T_x; y \vdash e :: T}{\Gamma; y \vdash \lambda x.\ e :: T_x \to T}$$

$$\text{T-App}\ \frac{\Gamma; y \vdash e_1 :: T_x \to T \qquad \Gamma; y \vdash e_2 :: T_x}{\Gamma; y \vdash e_1\ e_2 :: T} \qquad \text{T-Get}\ \frac{\Gamma; y \vdash x :: \texttt{Ref}\ \{B \mid r\}}{\Gamma; y \vdash \textbf{get}\ x :: \{B \mid r \wedge v = y[x]\}}$$

$$\text{T-If}\ \frac{\Gamma; y \vdash e :: \{\texttt{Bool} \mid r\} \qquad \Gamma, [v \mapsto \top]r \vdash e_1 :: T \qquad \Gamma, [v \mapsto \bot]r \vdash e_2 :: T}{\Gamma; y \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 :: T} \qquad \text{T-Bind}\ \frac{\Gamma; \sigma \vdash e_1 :: \langle T_1 \rangle^\pi \qquad \Gamma; \sigma \vdash e_2 :: T_1 \to \langle T_2 \rangle^\pi}{\Gamma; \sigma \vdash \textbf{bind}\ e_1\ e_2 :: \langle T_2 \rangle^\pi}$$

$$\text{T-}\lfloor\cdot\rfloor\ \frac{\Gamma; \sigma \vdash e :: \langle \{\texttt{Bool} \mid v \Rightarrow r\} \rangle^{\lambda(s,u).\pi[(s,u)] \wedge r}}{\Gamma; \sigma \vdash \lfloor e \rfloor :: \langle \{\texttt{Bool} \mid v \Rightarrow r\} \rangle^\pi} \qquad \text{T-}<:\ \frac{\Gamma; y \vdash e :: T' \qquad \Gamma \vdash T' <: T}{\Gamma; y \vdash e :: T}$$

$$T\forall\ \frac{\Gamma, \alpha; y \vdash e :: S}{\Gamma; y \vdash e :: \forall \alpha.\ S} \qquad T\text{-Inst}\ \frac{\Gamma; y \vdash e :: \forall \alpha.\ S \qquad \Gamma \vdash T}{\Gamma; y \vdash e :: [\alpha \mapsto T]S}$$

Statement Typing $\boxed{\Gamma; y \vdash s}$

$$\text{T-Let}\ \frac{\Gamma; y \vdash e :: T \qquad \Gamma, x : T; y \vdash s}{\Gamma; y \vdash \textbf{let}\ x = e\ \textbf{in}\ s} \qquad \text{T-Print}\ \frac{\Gamma; y \vdash x_1 :: \langle \{\texttt{User} \mid p(y, v)\} \rangle^p \qquad \Gamma; y \vdash x_2 :: \langle \texttt{Str} \rangle^p \qquad \Gamma; y \vdash s}{\Gamma; y \vdash \textbf{print}\ x_1\ x_2\ ;\ s}$$

$$\text{T-Set}\ \frac{\Gamma; y \vdash x_1 :: \texttt{Ref}\ T \qquad \Gamma; y \vdash x_2 :: T \qquad \Gamma, y' : \{\texttt{Store} \mid v = y[x_1 := x_2]\}\ ;\ y' \vdash s \qquad y' \text{ is fresh}}{\Gamma; y \vdash \textbf{set}\ x_1\ x_2\ ;\ s} \qquad \text{T-Skip}\ \frac{}{\Gamma; y \vdash \textbf{skip}}$$

Figure 11. λ^L static semantics.

secret, and this field had two different values, then clearly the result of the program would differ.

A.5 Health Portal Case Study

Our health portal case study, based on the HealthWeb case study in the Fine [41] paper, is particularly interesting because it showcases many Lifty capabilities and because of its complex policies guarding health records. We show the type signatures for some of the functions in Figure 12. As you can see, the policy on a health record is quite complex, depending on both the identity of the viewer, whether they are a patient, whether there is a withholding relationship on the record, and whether there is a psychologist and treatment relationship between the viewer and the patient whose record it is. For this example, the complexity of the policy makes the generated policy check significantly larger than the size of the original code.

```
getRecord :: rid : RecordId ⟶ Ref ⟨RecordId⟩^λ(s,u).u=author (s[rid])∨
                (isPatient s u∧u=patient s[rid]∧¬(shouldWithhold u s[rid]))∨
                (isDoctor s u∧¬(shouldWithhold u s[rid]))∨
                (isPsychiatrist s u∧isTreating u s[rid]∧isPsychiatristRecord s[rid]∧¬(shouldWithhold u s[rid]))
getIsTreating :: u : User ⟶ w : User ⟶ Ref ⟨Bool⟩^λ(s,v).isPsychiatrist s v ∧ v=u
getAuthoredRecordIds :: User ⟶ Ref ⟨[RecordId]⟩^any
```

Figure 12. Function signatures from the HealthWeb case study.

The health portal code takes advantage of Lifty's ability to generate checks as close to the data source as possible. One view function, `showRecordsForPatientView`, uses a filter over the list of all records to find the records that have a specified patient, and then outputs the result. The repair works as expected: the repaired version of the function generates a complex check (corresponding to the above) and runs it on each element of the list, so that only those records that pass the check will be shown.

We also found Lifty to handles sensitive values in policies appropriately. The policy for `getIsTreating` depends on the result of `isTreating`, but our `getIsTreating` function has a policy of its own that says that the patients of a psychiatrist can only be seen by that psychiatrist. However, the generated policy check still works fine, because in the `getRecord`'s policy, the `isTreating` predicate is checked only after `isPsychiatrist` is checked.

The `showAuthoredRecordsView` function is also interesting because it demonstrates how relying on automatic patch generation can potentially reduce the number of checks necessary in the code. In our code, the `showAuthoredRecordsView` function first gets all the IDs of records authored by the session user. The `getRecord` policy says that a record may always be seen by its author. Because Lifty can verify this policy against the code, it is able to determine that the `showAuthoredRecordsView` function satisfies policies without even needing to add a check.