Type-Driven Repair for Information Flow Security

Nadia Polikarpova
Massachusetts Institute of Technology
polikarn@csail.mit.edu

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

Shachar Itzhaky
Armando Solar-Lezama
Massachusetts Institute of Technology
shachari,asolar@csail.mit.edu

Abstract

We present LIFTY, a language that uses type-driven program repair to enforce information flow policies. In LIFTY, the programmer specifies a policy by annotating the source of sensitive data with a refinement type, and the system automatically inserts access checks necessary to enforce this policy across the code. This is a significant improvement over current practice, where programmers manually implement access checks, and any missing check can cause an information leak.

To support this programming model, we have developed (1) an encoding of information flow security in terms of decidable refinement types that enables fully automatic verification and (2) a program repair algorithm that localizes unsafe accesses to sensitive data and replaces them with provably secure alternatives. We formalize the encoding and prove its noninterference guarantee. Our experience using LIFTY to implement a conference management system shows that it decreases policy burden and is able to efficiently synthesize all necessary access checks, including those required to prevent a set of reported real-world information leaks.

1. Introduction

From conference management systems to health record management systems, today's software manipulates sensitive data in increasingly complex ways. To protect sensitive data, programmers must enforce fine-grained, evolving confidentiality policies. For example, policies for the very system used to submit this paper include "a reviewer may see the score for the submission if there is no conflict of interest" and "authors may see reviews for their own papers during or after the rebuttal phase."

To some, it may seem surprising that conference management systems still leak information. If we consider the code, however, security bugs are not so shocking. Traditionally programmers enforce confidentiality by inserting policy checks across the program. For example, inside a function that displays the status of all submissions to a reviewer, the programmer must check, for *each submission*, if there is a conflict. Then there is the issue of the viewer: there is a documented bug in the HotCRP conference management system [52], for instance, where users were allowed to send themselves password reminder emails—from *any user in the system*. As such, both the placement and content of the checks are nontrivial to determine. The many opportunities for accidental information leaks supports the case for automatic information flow control.

We divide existing approaches to information flow control into four categories. The first two include program analysis techniques that check a given program against a high-level, declarative description of a policy. Dynamic techniques [8, 25, 52] insert checking code into otherwise unencumbered programs, with the downside of unpredictable runtime behavior, as failed policy checks cause exceptions or silent failures. Runtime behavior is predictable using static techniques [10, 22, 28, 33, 45, 54], but these either lack support for complex policies present in realistic applications, or require programmers to provide auxiliary annotations. Moreover, none of the analysis approaches frees the programmer from both writing policy checks across the program and writing code to handle cases when a viewer does not have access.

Addressing the problem of programmer burden, the other two categories of techniques aim to go beyond checking a given program, and instead modify the program—either at runtime or at compile time—in order to avoid leaks. With the *policy-agnostic paradigm* [6, 50, 51] the programmer is free to omit policy checks from the code and instead associates policy functions with data definitions. The language runtime is responsible for steering dynamic behavior to adhere to policies, which are expressive and may depend on sensitive values. Unfortunately, this solution incurs potentially prohibitive performance overheads and may result in unexpected dynamic behavior. Static *program repair* techniques avoid these pitfalls, but none of the existing repair techniques for security [18, 20, 43] target policies as expressive as those that policy-agnostic programming supports.

In this paper we present LIFTY, a new security-typed programming language that combines advantages of all four categories of existing approaches. LIFTY provides a static solution to policy-agnostic programming: the programmer provides high-

[Copyright notice will appear here once 'preprint' option is removed.]

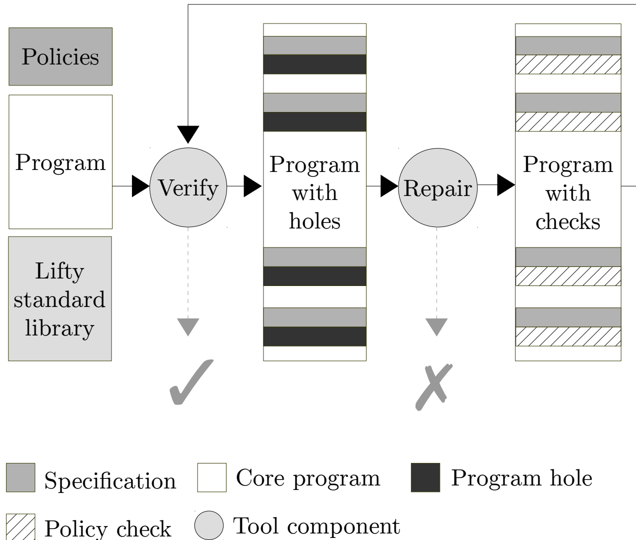

Figure 1. Type-driven repair for LIFTY programs.

level, policy specifications, expressed directly in the form of logical predicates over viewers and states; the compiler automatically inserts policy checks across the code in such as way that the resulting program provably satisfies the policies. Like other static approaches, LIFTY has no run-time overhead relative to the necessary policy checks. At the same time, it supports complex, representative policies, allowing both policies and policy evaluation context (*e.g.* the viewer) to depend on sensitive values.

Providing a static, repair-based solution to policy-agnostic programming poses two major challenges: program verification and policy synthesis. Because we want a verification algorithm for the purpose of repair, we need a fully automatic static analysis technique that supports expressive policies that may depend on sensitive values. In addition, it is nontrivial to determine optimal policy checks statically, as they depend both on the source of sensitive data and on the viewer to whom the data is flowing.

Our core insight is that we can encode information flow security using *liquid types* [38, 47], an expressive yet decidable refinement type system. We take advantage of two key aspects of liquid types: (1) the support for decidable type-checking and error localization and (2) the support for sound and automatic program synthesis [36]. Our solution has two main aspects. First, we present an encoding of information flow using liquid types that is both flexible enough to verify programs with complex policies. Second, we present a repair algorithm that decomposes the global problem of policy enforcement into independent, local program synthesis problems, hereby overcoming the scalability limitations of program synthesis.

In this paper, we describe the input language, information flow encoding, and compiler implementation for LIFTY (which stands for Liquid Information Flow TYpes). In Fig. 1 we show an overview of LIFTY's work flow. The programmer specifies information flow policies as refinement types associated with sources of sensitive data. LIFTY's verifier uses a variant of liquid type inference [38] to produce a program with holes, where each hole corresponds to an unsafe data access paired with a local policy specification. Our repair algorithms extends on an existing type-based program synthesis technique [36] to produce a policy check for each hole.

This paper makes the following contributions:

- **Repair-based approach to information flow control.** We present the first technique that statically repairs programs to ensure information flow security with respect to fine-grained, state-dependent policies.
- **Verification of expressive policies.** We developed an encoding of information flow security in the formalism of liquid types, which is sound, requires no programmer annotations besides policy specifications, and supports fine-grained, state-dependent policies, which may depend on sensitive values. Verification against such policies is out of scope for existing static analysis techniques.
- **Formalization and proof of security guarantee.** We formalize *contextual noninterference*: a generalization of noninterference for viewer- and state-dependent policies, and prove that LIFTY guarantees this property.
- **Demonstration of practical promise.** We implement a LIFTY-to-Haskell compiler and demonstrate through the implementation of a conference management system that our solution supports expressive policies, reduces the burden placed on the programmer, is able to generate all necessary checks for our benchmarks (about seven minutes for the entire conference management system), and can repair programs to prevent reported real-world leaks.

2. Overview

LIFTY is a pure, call-by-value functional language with Haskell-like syntax. The implementation of a typical LIFTY application consists of a data module defining data schemas and information flow policies and other modules implementation application functionality in a policy-agnostic manner. Only the data module is trusted: the LIFTY compiler verifies the remainder of the program, inserting missing policy checks where necessary.

2.1 Programming in LIFTY

We use simplified snippets from our conference management system case study to introduce the LIFTY language, our encoding of information flow control, and the repair mechanism. Recall the password-reminder-of-any-user bug [52] from our Introduction. Here we implement similar functionality to send to send an email to a given member of program committee with the list of all conference submissions and their current scores. In Fig. 2 we show excerpts form the data module (`DM`) and application code (`Server`) for implementing this functionality.

Policy-Agnostic Application Code. In the `Server` module (line 12), we implement the function `sendReminder` to send an email with the list of all conference submissions and their

```
module DM where -- Mediates data access
import Tagged     -- uses info flow library

getTitle :: World → PaperId →
  Tagged String <True>
getAuthors :: World → pid: PaperId →
  Tagged [User] <λw _ . phase w ≥ rebuttal>
getScore :: World → pid: PaperId →
  Tagged Int <λw u . u in pc w ∧
                   u ∉ conflicts w pid>

module Server where -- Application logic
import Tagged, DB

sendReminder :: World → UserId → World
sendReminder w i =
  let paperRow pid =
        do t ← getTitle w pid
           as ← getAuthors w pid
           s ← getScore w pid
           return (unwords [t, show as, show s])
      out =
        do allPids ← getAllPaperIds w
           rows ← mapM paperRow allPids
           return (unlines rows)
      mem = getPCMember w i
  in print w mem out -- Send message out to mem
```

Figure 2. Snippets from two modules of the conference management system.

current scores. The function first defines a helper function `paperRow pid` that retrieves the title, authors, and score of the submission `pid` from the data store and concatenates them into a line of text. Next, the function computes the entire output by mapping `paperRow` onto the list of all papers and concatenating the resulting list of lines. Finally, it retrieves the identity of the PC member and outputs the text (using **`print`** from LIFTY's standard library). Note that the function has both read effects (retrieval from the data store) and write effects (output to user). In our pure language we capture these effects by propagating a single additional argument `w` (of type `World`) through the code. We assume that a `World` value encapsulates both the state of the data store and the observations made by the users. As we explain below, we encode information flow control using a monad; hence the code uses monadic construct such as **`do`** and `mapM`, which are however standard and familiar to Haskell programmers.

What is remarkable about the body of `sendReminder` is that it accesses sensitive data, using accessor functions such as `getTitle`, `getAuthors`, and `getScore`, *without implementing policy checks*! In a conference management system, implementing double-blind review requires hiding the list of paper authors from reviewers until the rebuttal phase, as well as hiding the score of a paper from those reviewers who have a conflict of interest. And because we are implementing a function that sends email to an arbitrary user, implementing checks for such policies involves additionally keeping track of the viewer. But `sendReminder` as it is written does not seem to take this policy into account at all. This is because, as we will show, the LIFTY compiler is responsible for inserting these checks automatically, based on programmer specifications.

Policy Specifications. The LIFTY programmer provides specifications of information flow policies by associating *types* with *accessor functions* in the data module. The programmer designates a data value as sensitive by *tagging it*: wrapping the return type of the corresponding accessor function in a `Tagged` type constructor that is parameterized by a predicate corresponding to the confidentiality policy.

In Fig. 2 we show how to specify the policies mentioned above. The type `Tagged` α `<`P`>`, where P is a binary predicate, stores a value (of type α) that can only be seen by a user u in a world w provided that $P\ w\ u$ holds. For example, we gave the result of `getScore` the type

```
Tagged Int <λw u . u ∈ pc w ∧ u ∉ conflicts w pid>
```

This policy says that a viewer `u` may see the return value of the function as long as at the time of output, `u` is in the PC and not in conflict with the paper. Policy predicates can directly refer to the fields of the data store (such as `conflicts` on line 10); all accesses to sensitive values in executable parts the program obtain `Tagged` versions through the accessor functions. The LIFTY compiler uses the associated types to insert necessary policy checks into the program.

For implementing policy specifications LIFTY supports *abstract refinement types* in the style of Vazou *et al.* [47]. Our framework is agnostic to the exact logic of refinement predicates, as long as it is decidable. Our implementation uses the quantifier-free logic of arrays (used to model sets and maps), uninterpreted functions, and linear integer arithmetic. In our experience this was sufficient to express all policy predicates for the conference management system are a direct translation of the intuitive statement of the policy.

2.2 Inserting Policy Checks

We now describe how the LIFTY compiler inserts missing policy checks. Our key insight in developing LIFTY is that we can use *the results of a failed verification attempt* to both (1) identify where in the code we need to guard sensitive data and (2) determine the appropriate check.

Step 1: Verification. To compile `sendReminder`, the LIFTY compiler first attempts to verify the body with respect to the specified policies using a variant of liquid type inference [38]. It detects, for instance, that the authors list obtained from calling `getAuthors` (line 6) with policy `λw _ . phase w ≥ rebuttal` flows into the argument `out` of **`print`**, which is required to have the policy

```
λw u . u = pcMember w i ∧ u ∈ pc w
```

(or weaker) resulting from type information propagated back from evaluating `mem = getPCMember w i` (line 26). Lifty uses

```
sendReminder w i =
  let paperRow pid =
        do t ← getTitle w pid
           as ← ifM (do t1 ← getPhase w
5                       return (rebuttal ≤ t1))
                    (getAuthors w pid)
                    (return []))
           s ← ifM (do t2 ← getPCMember w i
                       t3 ← getConflicts w pid
10                     return (not (elem t2 t3)))
                    (getScore w pid)
                    (return -100)
           return (unwords [t, show as, show s])
      out = ... -- as before
15    mem = getPCMember w i
  in print w mem out
```

Figure 3. Example implementation after repair (injected policy checks are highlighted).

the SMT solver to check whether the latter policy implies the former. (Note that this is always decidable thanks to the restrictions on the logic of policy predicates.) Since this is not the case, LIFTY deems this flow unsafe.

Step 2: Error Localization. What is the best way to prevent this leak? One option is to wrap the **print** invocation itself in a conditional. This would prevent the leak, but will have an undesired side effect of hiding the titles and the scores along with the authors. Even worse, in this example there is actually another sensitive value—the score—with a different policy flowing into the same **print** operations, making such a solution even more clumsy. Our goal for the LIFTY compiler is to preserve as much of the original program behavior as possible, and so LIFTY always chooses to guard the smallest possible subterm, *i.e.* the invocation of the accessor method. In this case, even though several data elements flow into the same **print** operation, LIFTY is able to determine only two of them are sensitive and require policy checks, and associates the checks with `getAuthors w pid` and `getScore w pid`. Such precision becomes increasingly important with larger programs that access many pieces of data.

Key to our precise error localization is our encoding of information flow in terms of liquid types. Our encoding enables LIFTY to leverage the *type checker* to perform two important functions. First, we can use type error localization to identify each offending term. Second, we can use type inference to propagate the information about the eventual viewer from the invocation of **print** back to the source in order to infer a *local policy specification* for each offending term (*i.e.* what is the most restrictive policy the repaired term must satisfy in order to avoid the leak). In this example, the local policy specification for both offending terms is λ`w u . u` $=$ `pcMember w i` $\wedge$ `u` $\in$ `pc w`, since they are flowing into the same **print** operation.

Step 3: Repair. The next step is guard each offending access with a policy check so that the guarded access adheres to the local policy specification derived through type inference. LIFTY uses a modification of the SYNQUID program synthesizer [36] to generate the weakest guard that is sufficient to make the access safe. Using a program synthesizer as opposed to syntactically deriving checks from policy predicates, makes LIFTY more robust and allows it to avoid redundant checks. For example, the policy for `getScore` contains a conjunct `u` $\in$ `pc w`, but LIFTY determines from the local policy specification that in this context the viewer is always a PC member and omits this redundant check. Similarly, if the programmer has already implemented the checks in the program, LIFTY will leave the program as is (it is also able to enhance existing partial checks by adding conditionals inside of the existing ones).

In Fig. 3 we show the repaired version of the code, where the invocations of `getAuthors` and `getScore` are guarded with an appropriate checks. The monadic operation `ifM` branches on a sensitive Boolean expression. In case the policy is violated, the guarded access returns a *default value* (in this example, `-100` for the score in line 8). The programmer designates a default for every sensitive accessor method.

Step 4: Verification to Prevent Leaky Enforcement. As a final step, LIFTY verifies the resulting program to ensure that the insertion of policy checks did not introduce *new* leaks. It does this to address the problem of *leaky enforcement*, which occurs when policy checks leak information about the sensitive values they depend on. Leak enforcement may occur in our system because LIFTY allows policies to depend on sensitive values. Note that policies that depend on sensitive values are becoming common in realistic applications, but have been largely ignored in previous work on static verification of information flow security. (For more details, see Sec. 6.)

To illustrate leaky enforcement, consider how the policy associated with the `getScore` function depends on the `conflicts` field. Now suppose a programmer specified the following policy for `getConflicts`, which allows only the PC chair to view it:

```
getConflicts :: World → pid: PaperId →
  Tagged Int <λw u . u = chair w>
```

If the programmer really intended such a policy, then there is a leak of information, since an ordinary, non-chair PC member now sees scores differently depending on whether they have a conflict, thus leaking information about the conflict field. In this program there is no way to precisely check the policy as stated without leaking information, and so LIFTY will detect a leak in the repaired code. In this case LIFTY and alerts the programmer that the policy for `score` might be ill-formed.

Reasoning About Self-Referential Policies Supporting policies that depend on sensitive values involves handing *self-referential* policies. As an example, consider the list of conflicts for a submission that the injected code in Fig. 3 accesses. It is natural to demand, in a double-blind conference, that a reviewer cannot access the conflicts list of a submission when they are

themselves in conflict. We can express this with the following policy specification on `getConflicts`:

```
getConflicts :: World → pid: PaperId →
  Tagged Int <λw u . u ∉ conflicts w pid>
```

This policy is self-referential because it guards access to the field `conflicts` in a way that depends on the value of `conflicts`.

Self-referential policies pose a challenge in defining a security guarantee. The traditional notion of noninterference cannot apply: it would be too restrictive to demand that a reviewer cannot tell whether they are in conflict with a submission or not. Instead all we care about is that a conflicted reviewer cannot tell *who else is in conflict*; in other words, a principal must not be able to distinguish between two values of a sensitive field that they are not allowed to see. This relaxed notion of noninterference was first introduced in dynamic policy-agnostic programming [50]. In the context of LIFTY, we refer to it as *contextual noninterference* and formalize it in Sec. 3. LIFTY is able to reason about these self-referential policies statically and automatically in a way that is sound with respect to contextual noninterference.

2.3 Encoding Information Flow with Liquid Types

Key to our solution is our encoding of information flow using liquid types. Goals for our solution included (1) strong formal guarantees, (2) support for expressive high-level policies, (3) automated verification, and (4) precise error localization.

For strong formal guarantees, we build on a line of work [22, 39, 44, 45] that encodes information flow policies using value-dependent data types. To ensure soundness, the module system disallows client code to explicitly construct or destruct values of this type. Instead clients manipulate labeled values using a predefined *monadic library*, which only allows secure manipulations. In LIFTY, the module `Tagged`, imported by `DM` and `Server` in Fig. 2, is such a monadic library defining the `Tagged` datatype, along with primitive monadic operations (**`return`** and **`bind`**), derived operations (such as `liftM`, `ifM`, and `mapM`), and well as the output function **`print`** for extracting values from `Tagged`. The type signatures of these operations propagate labels through the code in a sound way. For example, the signature of **`bind`** prescribes that applying a sensitive function to a sensitive value yields a result that is at least as secret as either of them; the signature of **`print`** imposes the requirement that the sensitive value it consumes is visible to the target of the output. In our meta-theory we prove that these signatures guarantee contextual noninterference.

We fulfilled goals (2) through (4) using a careful encoding of information flow using liquid types. In LIFTY, the programmer encodes policies as predicates over the program state, as opposed to less directly in terms of labels and axioms, as in Fable [44] and Fine [45]. Our encoding allows us to take advantage of a combination of subtyping and predicate abstraction available in liquid types to infer all auxiliary policy annotations completely automatically. The encoding also facilitates precise error localization, relying on type inference to identify the data sources.

Modifying Data Store A LIFTY application can perform both reads and writes. When the code writes a value, LIFTY makes sure that it does not introduce an information leak by reading a private field and then writing it into a more public one. Note that this is separate from access control (*i.e.* which users are allowed write access), which can be implemented by normal preconditions, and is not the focus of this paper. To establish these checks, the data module also declares *setters*, for example

```
setScore :: World → PaperId → Tagged Int
   <λw u . u ∈ pc w ∧ u ∉ conflicts w pid> → World
```

which complements `getScore` and has the same policy. If `setScore` is called, *e.g.*, with a value visible only to the chair, it would be detected as a type error. Such type annotations preserve the global invariant that only values with sufficiently weak policies are stored.

Note that the presence of mutable stores necessitates an interpretation of every policy with respect to the world at the time of output, which is why our policies accept `w: World` as an argument. A case where a value is public when it is read, but private when it is printed, will be detected and handled correctly by LIFTY.

3. Formal Semantics and Guarantees

We present the semantics and guarantees of LIFTY in two steps. First, we present the static semantics of $\mathcal{BL}$, a simple pure functional language that extends λ_P, the core language of *abstract refinement types* [47], with type constructors (polymorphic data types) that are parameterized by types and predicates and obey nominal subtyping rules. Our extension is sufficiently minimal that we can take advantage of λ_P's decidable type-checking and automatic type inference.

Polymorphic data types allow us to encode tagging values with information flow policies directly in $\mathcal{BL}$, rather than extending the language. We first show how to implement tagging in $\mathcal{BL}$ as the information flow monad `Tagged`. We then use a new proof technique we have developed to prove non-interference, introducing the $\texttt{Tagged}^2$ monad that relates pairs of executions and showing that type-checking with $\texttt{Tagged}^2$ implies contextual non-interference with `Tagged`. Since the repair phase always generates type-correct programs, this is sufficient for verifying the correctness of LIFTY's repair.

3.1 Syntax and Types of $\mathcal{BL}$

We now present $\mathcal{BL}$. Like λ_P, $\mathcal{BL}$'s type system features decidable refinement types, as well as type- and predicate-polymorphism. Our presentation of the syntax, types, and semantics closely follows Vazou *et al.*'s presentation of λ_P [47]. $\mathcal{BL}$ additionally includes a formalization of type constructors parameterized both by types and by predicates. These type constructors, combined with the subtyping rules we define for them, are crucial for supporting the phantom predicates necessary for our solution.

We show the $\mathcal{BL}$ syntax in Fig. 4.

$$
\begin{array}{lll}
v & ::= x \mid \lambda x{:}T.e & \textit{Values} \\
e & ::= v \mid \textbf{let}\ x = v\ v\ \textbf{in}\ e & \textit{Expressions} \\
 & \quad \mid \textbf{if}\ x\ \textbf{then}\ e\ \textbf{else}\ e & \\
 & \quad \mid \textbf{match}\ x\ \textbf{with}\ D\ \bar{x} \rightarrow e & \\
\psi & ::= & \textit{Formulas:} \\
 & \quad \mid \top \mid \bot \mid 0 \mid + \mid \ldots\ \textit{(varies)} & \text{interpreted symbol} \\
 & \quad \mid f & \text{uninterpreted symbol} \\
 & \quad \mid \psi\ \psi & \text{application} \\
a & ::= \psi \mid \pi\ \bar{x} \mid \psi \Rightarrow a & \textit{Atomic refinement} \\
r & ::= a \mid a \wedge r & \textit{Refinement} \\
p & ::= r \mid \lambda x{:}T.p & \textit{Parametric refinement} \\
 & & \\
B & ::= & \textit{Base types:} \\
 & \quad \mid () \mid \texttt{Bool} \mid \texttt{Int} & \text{primitive} \\
 & \quad \mid \alpha & \text{type variable} \\
 & \quad \mid D\ \bar{T}\ \langle \bar{p} \rangle & \text{data type} \\
T & ::= \{B \mid r\} \mid x{:}\ T \rightarrow T & \textit{Types} \\
\circ & ::= \oplus \mid \ominus \mid \odot & \textit{Variance} \\
S & ::= T \mid \forall_\circ \alpha.S \mid \forall_\circ \langle \pi{:}T \rangle.S & \textit{Type schemas}
\end{array}
$$

Figure 4. Terms and types.

Expressions. We differentiate between program terms and refinement terms. The former include values (variables and abstractions) as well as let-bindings, conditionals, and pattern-matching. All $\mathcal{BL}$ programs are in A-normal form [16]: application only appears in let-bindings and are built out of values, not arbitrary expressions (this is important for refinement type checking).

For simplicity of presentation we omit recursion and assume our data types are record types (*i.e.* have a single constructor); hence the **`match`** expression, which binds the fields of the record to variables, only has one case. Our implementation supports both recursion and proper algebraic data types (tagged unions); extending the formalism to include these features would be straightforward.

Refinements. Refinements are built up from formulas ψ of the refinement logic and applications of predicate variables π. Inside formulas, the exact set of interpreted symbols depends on the chosen refinement logic; the only requirement is that the logic be decidable to enable automatic type checking. Predicate variables always appear positively inside refinements to enable type inference.

Types and Schemas. $\mathcal{BL}$ types include refined base types $\{B \mid r\}$ and dependent function types $x{:}\ T_x \rightarrow T$. Here r is a *refinement predicate* over the program variables and a special *value variable* ν, which denotes the bound variable of the type, and x may appear in the refinement predicates of T. Base types include primitives, type variables, and data types. A data type is an application of a type constructor D to zero or more types and zero or more parametric refinements. Schemas are obtained by universally quantifying types over type and predicate variables. We explicitly label each quantification with its variance: covariant ($\oplus$), contravariant ($\ominus$), or invariant ($\odot$). $\oplus$ is the default variance and may be omitted.

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

$$
\text{WF-}\pi\ \frac{\Gamma \vdash \pi\ \bar{x}{:}\texttt{Bool} \qquad \Gamma(\pi) \neq T[\ominus]}{\Gamma \vdash \pi\ \bar{x}} \qquad \text{WF-}\alpha\ \frac{\Gamma(\alpha) \neq \ominus}{\Gamma \vdash \alpha}
$$

$$
\text{WF-FUN}\ \frac{\Gamma^- \vdash T_x \qquad \Gamma; x{:}T_x \vdash T}{\Gamma \vdash T_x \rightarrow T}
$$

$$
\text{WF-D}\ \frac{\Gamma(D) = \overline{\forall_\circ \alpha_i}.\overline{\forall_\circ \langle \pi_j{:}U_j \rangle}.T \qquad |T_i| = |\alpha_i| \qquad \Gamma \vdash p_j : U_j}{\Gamma \vdash D\ \overline{T_i}\ \langle \overline{p_j} \rangle}
$$

$$
\text{WF-}\forall\alpha\ \frac{\Gamma;\alpha{:}\circ \vdash S}{\Gamma \vdash \forall_\circ \alpha.S} \qquad \text{WF-}\forall\pi\ \frac{\Gamma;\pi{:}T[\circ] \vdash S}{\Gamma \vdash \forall_\circ \langle \pi{:}T \rangle.S}
$$

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

$$
<:\text{-SC}\ \frac{\Gamma \vdash B <: B' \qquad \mathsf{Valid}([\![\Gamma]\!] \wedge r \Rightarrow r')}{\Gamma \vdash \{B \mid r\} <: \{B' \mid r'\}}
$$

$$
<:\text{-D}\ \frac{\Gamma(D) = \overline{\forall_{\circ_i} \alpha_i}.\overline{\forall_{\circ_j} \langle \pi_j \rangle}.T \qquad \Gamma \vdash T_i \sim_{\circ_i} T_i' \qquad \Gamma \vdash p_j \sim_{\circ_j} p_j'}{\Gamma \vdash D\ \overline{T_i}\ \langle \overline{p_j} \rangle <: D\ \overline{T_i'}\ \langle \overline{p_j'} \rangle}
$$

$$
\frac{\Gamma \vdash T <: T'}{\Gamma \vdash T \sim_\oplus T'} \quad \frac{\Gamma \vdash T' <: T}{\Gamma \vdash T \sim_\ominus T'} \quad \frac{\Gamma \vdash T <: T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash T \sim_\odot T'}
$$

$$
\frac{\Gamma; x{:}T \vdash p \sim_\circ p'}{\Gamma \vdash \lambda x{:}T.p \sim_\circ \lambda x{:}T.p'} \qquad \frac{\Gamma \vdash \{() \mid r\} \sim_\circ \{() \mid r'\}}{\Gamma \vdash r \sim_\circ r'}
$$

Type Checking $\boxed{\Gamma \vdash e :: S}$

$$
\text{IF}\ \frac{\Gamma \vdash x :: \{\texttt{Bool} \mid r\} \qquad \Gamma;[\top/\nu]r \vdash e_1 :: T \qquad \Gamma;[\bot/\nu]r \vdash e_2 :: T}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 :: T}
$$

$$
\text{T-GEN}\ \frac{\Gamma;\alpha{:}\circ \vdash e :: S}{\Gamma \vdash e :: \forall_\circ \alpha.S} \qquad \text{T-INST}\ \frac{\Gamma \vdash e :: \forall_\circ \alpha.S \qquad \Gamma \vdash \{B \mid r\}}{\Gamma \vdash e :: [\{B \mid r\}/\alpha]S}
$$

$$
\text{P-GEN}\ \frac{\Gamma;\pi{:}T[\circ] \vdash e :: S \qquad \Gamma \vdash T}{\Gamma \vdash e :: \forall_\circ \langle \pi{:}T \rangle.S}
$$

$$
\text{P-INST}\ \frac{\Gamma \vdash e :: \forall_\circ \langle \pi{:}T \rangle.S \qquad \Gamma \vdash p : T}{\Gamma \vdash e :: [p \triangleright \pi]S}
$$

Figure 5. Relevant rules from the static semantics of $\mathcal{BL}$: well-formedness, subtyping, and type-checking.

3.2 $\mathcal{BL}$ Static Semantics

In Fig. 5 we show the relevant subset of well-formedness, subtyping, and type checking rules for $\mathcal{BL}$ (the omitted rules can be found in supplementary material). These rules deviate from the standard semantics is in the way we track variances of type and predicate parameters of polymorphic schemas; explicit variance annotations are required to control the subtyping relation for data types with phantom predicate parameters, which we use to encode policies. Note that while our extensions to λ_P are standard, they are important for deriving our safety property.

In our semantics, a *typing environment* Γ maps variables to type schemas $(x : S)$, bound type variables to their variances $(\alpha : \circ)$, and bound predicate variables to their types and variances $(\pi : T[\circ])$. We assume that for each type constructor D the environment contains a data constructor with the same name; the type schema of the constructor has the form $\forall_\circ \bar{\alpha}.\forall_\circ \overline{\pi : T}.T_1 \to \ldots \to T_n \to \{D\ \bar{\alpha}\ \overline{\pi}\ \overline{x} \mid r\}$and determines the type and predicate parameters of the type constructor D (here and below, $\overline{a}$ for any syntactic element a denotes a sequence $a_1,...,a_n$).

Well-Formedness. A refinement r is *well-formed* in the environment Γ, written $\Gamma \vdash r$, if it sort-checks to Boolean and none of its predicate variables are bound in a contravariant manner in Γ (rule WF-π). We use a judgment $\Gamma \vdash r : T$ in the premise of WF-π to denote simple sort checking of refinement terms, as opposed to $\Gamma \vdash e :: T$, which denotes refinement type checking of program terms. Well-formedness extends to base types, types, and type schemas. The well-formedness rules ensure that variance annotations on type and predicate parameters are consistent with how those parameters are used inside the type (*i.e.* whether they appear positively, negatively, or in both positions); to this end, Γ^- in the premises of rules for function types inverts variance annotations for all type and predicate variables in the environment.

Subtyping. The *subtyping* relation $\Gamma \vdash T <: T'$ is standard (Fig. 5) except for data types. Rule $<:$-SC reduces subtyping between scalar types to implication between their refinements, under the assumption extracted from the environment. Since the refinements are drawn from a decidable logic, this implication is decidable. Refinement assumption is simply a conjunction of all refinements of scalar variables:

$$[\![\Gamma]\!] = \bigwedge_{x:\{B|r\}\in\Gamma} [x/\nu]r$$

Rule $<:$-D reduces subtyping between two instantiations of the same type constructor to a relation between their type and predicate arguments. Each argument is compared according to its variance annotation in the corresponding data constructor.

Type Checking and Inference. Type checking rules are standard; rules for variables, abstractions, application/let-bindings, and matches are omitted. We show the rule for conditionals (P-IF) to demonstrate that $\mathcal{BL}$ is path-sensitive, which is important for verifying policy checks (in this rule, we use a shortcut $\Gamma;r$ for $\Gamma;x : \{\texttt{()} \mid r\}$, where x is a fresh variable name). The most interesting rule is P-INST, which instantiates a term of predicate-polymorphic type with a parametric refinement p of an appropriate type. The operation $[\pi \triangleright p]S$ can be understood as substituting the lambda-term p for every occurrence of π in S and then "beta-reducing" the result using the actual arguments of π (see [47] for details).

Note that rules T-INST and P-INST are non-deterministic: they guess appropriate instantiations for type and predicate variables. In practice these instantiations are inferred automatically by liquid type inference (see Sec. 4).

```
module Tagged where

-- | Tagged data constructor
private Tagged: ∀α . ∀⊖<p: W → U → Bool> .
5   val:α → Tagged α <p>

return: ∀α . ∀⊖<p: W → U → Bool> .
  α → Tagged α <p>

10 bind: ∀α β . ∀⊖<p: W → U → Bool> .
       ∀<f:α → β → Bool> .
  x: Tagged α <p>
  → (y: α → Tagged {β | f y ν} <p>)
  → Tagged {β | f (val x) ν} <p>
15
print: ∀α . ∀⊖<p: W → U → Bool> .
  w: W → viewer: Tagged {U | p w ν}<p>
  → msg: Tagged α <p> → W

20 downgrade: ∀⊖<p: W → U → Bool> . ∀<c: Bool> .
  x: Tagged {Bool | ν ⇒ c} <λw u . p w u ∧ c>
  → Tagged {Bool | ν ⇒ c} <p>
```

Figure 6. The `Tagged` monad. U denotes the type of principals; W encapsulates state and observations by the viewers.

3.3 Encoding Information Flow in $\mathcal{BL}$

We track information flow by wrapping sensitive values inside a data type `Tagged` $\alpha\ \langle \lambda w.\lambda u.r \rangle$. The predicate parameter $\langle \lambda w.\lambda u.r \rangle$, which we refer to as *policy*, encodes which whether a principal u is allowed to see the wrapped value in a world w. We show the type of the corresponding data constructor together with the *primitive monadic operations* in Fig. 6. To prevent user code from matching on a tagged value and freely extracting the protected sensitive value, we place a restriction that the `Tagged` constructor is not accessed from other modules; this is similar to prior work [22, 39, 44, 45]. The four primitive operations in Fig. 6 are the only functions that use the `Tagged` constructor directly, and thus have to be proven secure in the meta theory.

An important feature of our encoding is that the policy parameter of the `Tagged` constructor is *contravariant*. As a result the subtyping relation `Tagged` $\alpha\ \langle \pi \rangle <:$ `Tagged` $\alpha\ \langle \rho \rangle$ holds when $\rho \Rightarrow \pi$, which is resolved automatically by the SMT solver. In other words, we use the subtyping relation of $\mathcal{BL}$ to enforce that a value with a less restrictive tag (*i.e.* visible to more users, more public) is allowed to flow into a variable with a more restrictive tag (more secret) and not the other way around. This is different from prior work that uses dependent type for information flow [22, 44, 45], which lack subtyping between types with different labels; this complicates the encoding of operations on labeled types and conversion between types with different labels.

Manipulating tagged values. Policy-agnostic code manipulates tagged values using the operations **`return`**, **`bind`**, and **`print`**

shown in Fig. 6. Their type signatures (in combination with contravariant subtyping) ensure proper propagation of tags. The signature of **return** allows tagging a non-sensitive value with an arbitrary policy. The signature of **bind** means that applying a sensitive function to a sensitive value yields a result that is at least as secret as either of them; the additional predicate parameter f of **bind** allows the type checker to reason about the functional properties of a Tagged computation, alongside its policies.

Output. The signature of **print** is responsible for imposing the requirement that the sensitive value it outputs is visible to the target of the output. In addition to the sensitive message msg, the function takes as argument the viewer who is going to observe the output. The type of **print** is parameterized by a policy p, which labels both the viewer and the message. The rationale is that the identity of the viewer may itself depend on sensitive information. When checking an application **print** w u x, the type checker must infer a concrete instantiation of p that is at least as restrictive as the policies guarding both u and x, but at the same time p w u must hold (as expressed by the refinement on the type of viewer).

Downgrading policy checks. The signature of **bind** requires that all steps in a computation over sensitive values carry the same policies as the result. This can be overly restrictive when we want to type-check policy checks that depend on sensitive values. Consider the check for getScore from Fig. 3, desugared to use only bind **bind**:

```
  let
    mem = getPCMember w i
    check = bind mem (λt2 . bind (getConflicts w pid)
                     (λt3 . return (not (elem t2 t3))))
5   s = bind check (λc . if c
                          then getScore w pid
                          else return -100)
    ...
  in print w mem out
```

Here, the value getConflicts w pid flows, via check, s, and out, to the PC member mem, and thus the type checker will require that it be visible to mem. But getConflicts has a self-referential policy, as is only visible to mem when mem is not one of the conflicts, which is not known statically: in fact, that's precisely what we are trying to check! To type-check this code we need to wrap check in an invocation of **downgrade**. The type of **downgrade** is parametrized by an additional nullary predicate c, and has the effect of removing a conjunct c from a policy of a sensitive Boolean x. This type is sound and does not allow leaking any information: in a world where c holds, x is visible to the same viewers as **downgrade** x, while in a world where c does not hold, the value inside x is always false, and thus x it cannot leak information. The **downgrade** operation is not meant to be used directly by LIFTY programmers, but instead is used to implement ifM, which LIFTY uses in the generated policy checks.

```
   module Tagged² where

   private cw: W, cu: U -- Current context

5  -- | Tagged data constructor
   private Tagged²: ∀α . ∀⊖<p:  W → U → Bool> .
     l:α → r:α → prop: ({() | p cw cu} → {() | l = r})
     → Tagged α <p>

10 print²: ∀α . ∀⊖<p:  W → U → Bool> .
     w: W → u: Tagged² {U | p w ν}<p> → x: Tagged² α <p> →
   W
   print² = λw . λu . λx .
     match u with Tagged² ul ur up →
       if w ≠ cw ∨ (ul ≠ cu ∧ ur ≠ cu) then w
15       else if ul ≠ ur then fail (up ())
           else match x with Tagged² xl xr xp →
             if xl ≠ xr then fail (xp ()) else doPrint w xl
```

Figure 7. The Tagged^2 monad, which keeps track of two projections, used in the proof of noninterference.

3.4 Proving Non-Interference Using Tagged^2

We now prove that executions involving the Tagged monad preserve *contextual noninterference*: if a sensitive value v may not flow to a given viewer, then any pair of executions involving different assignments to v should yield equivalent outputs.

Reasoning directly about noninterference is inconvenient because it requires talking about two executions. We simplify our noninterference proof using a technique similar to that of Pottier and Simonet [37]: we introduce auxiliary constructs that allow us to reason about two executions in one. Being able to encode security labels as a library makes the formalization particularly nice: the only auxiliary construct we need for the proof is an alternative definition of the Tagged monad. We introduce the Tagged^2 monad with new implementations of the four primitive operations, yielding the property that if a program type-checks with Tagged^2, then it preserves contextual noninterference with Tagged.

The Tagged^2 monad. To simplify formalization of noninterference, we parameterize the semantics of $\mathcal{BL}$ by the context, *i.e.* the principal who is observing the execution and the world at the time of output. More concretely, we assume that the environment always contains two variables cw$:W$ and cu$:U$; when a program executes, it executes with all possible values of cw and cu "in parallel", but in each of these parallel threads, **print** only performs the output when its arguments match cw and cu, so this parametric semantics has no effect on the output.

We first construct a *phantom encoding*: a new information flow monad, Tagged^2, that explicitly relates pairs of program executions. The intuition behind Tagged^2 is as follows: it represents two versions of a sensitive value from two different executions of the program as seen by the current context. Mirroring what we want for our noninterference property, the two versions are only allowed to differ for those sensitive values that are *not visible*

in the context. The `Tagged`^{2} constructor accepts two α values, `l` and `r`, which we call *projections*. Its third argument `prop` serves as a proof of the property p `cwcu` $\Rightarrow l = r$, that is, if the policy holds of the current context, the two projections must be equal.

A `Tagged`^{2} value with different projections corresponds to Pottier and Simonet's "bracket value" in [37], and the `prop` requirement corresponds to their rule that all bracket values are assigned high security labels. The main conceptual difference of our treatment is that the division between high and low security, as well as the notion of a leak, is context-specific.

We show an excerpt from the implementation of the `Tagged`^{2} in Fig. 7. The phantom encoding provides alternative implementations for the four primitive operations. The function **`return`**^{2} gives the same value for both projections, while **`bind`**^{2} applies the function projection-wise (these two operations are omitted from Fig. 7 in the interest of space). There is also a definition of **`downgrade`**^{2} (also omitted), that allows the checker to prove that downgrading policies is safe.

The function **`print`**^{2} is designed to *fail* when it detects interference: namely, whenever the target of the output is different in the two executions (`ul` $\neq$ `ur`) or because it outputs two different values (`xl` $\neq$ `xr`). We assume that **`fail`** has the type `{() | False}`$\rightarrow$`a`, so the only way to type-check **`print`**^{2} is to prove that both failing branches are unreachable, which the $\mathcal{BL}$ type checker successfully accomplishes. To understand why the first failing branch is unreachable, recall that from the type of `u` we know that `p w ul` $\wedge$ `p w ur`; we also know that `w` $=$ `cw` and `ul` $=$ `cu` $\vee$ `ur` $=$ `cu` from the path condition, thus `p cw cu` holds, which gives `ul` $=$ `ur` guaranteed by the `Tagged`^{2} constructor.

Contextual noninterference. It remains to prove that type-checking with `Tagged`^{2} implies contextual noninterference with `Tagged`. Because the `Tagged`^{2} functions type-check and because the type system of $\mathcal{BL}$ is sound [47], we know that no type-correct program that manipulates `Tagged`^{2} values can go wrong, *i.e.* attempt to print the results of two executions that are different. Now we only have to formally connect computations with `Tagged` values and those with `Tagged`^{2} values, and show how type safety of the latter implies noninterference for the former.

We first show that replacing a `Tagged`^{2} value with its projection in `Tagged` at the beginning of an execution yields the same result as projecting at the end of an execution. A *projection* of an expression e (written $\lfloor e \rfloor_j$, for $j = \{l,r\}$) is an expression where every occurrence of `Tagged`^{2} $x_l\ x_r$ _ in e is replaced by `Tagged` x_j.

Lemma 1 (Projection)**.** If $e \rightarrow^* e'$ then $\lfloor e \rfloor_j \rightarrow^* \lfloor e' \rfloor_j$, for $j = \{l,r\}$.

Theorem (Contextual Noninterference)**.** Let $\Gamma; x:$ `Tagged` $\alpha\ \langle p \rangle \vdash e :: W$, and $\neg(p$ `cw cu`$)$. Let for $j \in \{l,r\}$, $\Gamma \vdash v_j :: \alpha$ and $[($`Tagged` $v_j)/x]e \rightarrow^* w_j$. Then $w_l = w_r$.

We omit the proofs in the interest of space; more details can be found in the supplementary material.

Algorithm 1 Repair

```
 1: REPAIR(Γ,e,T)
 2:     leaks ← VERIFY(Γ,e,T)
 3:     for (x,T′) ← leaks do
 4:         e ← FIX(Γ,x,T′,e)
 5:     leaks′ ← VERIFY(Γ,e,T)
 6:     if leaks′ = [] then return e
 7:     else fail

 8: FIX(Γ,x,T,let x = f v̄ in e)
 9:     ψ ← ABDUCE(Γ;ψ ⊢ f v̄ :: T)
10:     c ← SYNTHESIZE(Γ′ ⊢ c :: {Bool | ν ⇔ ψ})
11:     c′ ← LIFT(Γ,c)
12:     return let x = ifM c′ (f v̄) f_def in e
13: FIX(Γ,x,T,e)
14:     recursively call FIX on subterms of e
```

4. Repair Algorithm

In this section we give more detail about how LIFTY inserts access checks into policy agnostic code. We outline the process in Algorithm 1. REPAIR takes as input a program term e (in A-normal form), its top-level type annotation T, as well as an environment Γ that includes all necessary components (such as the `Tagged` library and all sources of sensitive data). Repair proceeds as follows.

Type-checking (1) and error localization. Type-checking the program (line 2) will either succeed, result in a failure (if the e has a type-error unrelated to information flow), or return a list *leaks* of unsafe accesses. Each unsafe access is a pair of a variable name x and a type T', where x is bound to an unsafe subexpression of e and needs to be enhanced by a conditional check.

Repair. Error localization has reduced the repair problem to local synthesis. Function FIX replaces every violation (line 4).

Type-checking (2) . While repair is guaranteed to produce functionally correct checks, the checks themselves may leak information if they depend on sensitive values. For this reason we re-run type-checking the resulting program in line 5.

4.1 Verification and Error Localization

The LIFTY compiler uses a variation of the liquid type inference [38] with predicate polymorphism [47] to produce a list of typed leaks. We first provide an overview of liquid type inference and then describe how we extend it.

Liquid type inference with predicate polymorphism translates a type checking problem $\Gamma \vdash e :: T$ into a set of Horn constraints over *predicate unknowns* P_i, corresponding to unknown parametric refinements in the instantiations of predicate-polymorphic components (*i.e.* the p in the typing rule P-INST in Fig. 5). The inference algorithm solves Horn constraints using *predicate abstraction*: restricting the search space for each P_i to conjunctions of atomic predicates generated from a given set of templates called *qualifiers*. The algorithm efficiently

finds a solution to the set of Horn constraints using the Houdini algorithm [15], a a *least-fixpoint* algorithm that computes the strongest solution for each P_i (*i.e.* the largest subset of atomic predicates that satisfies the constraints).

The LIFTY compiler modifies standard liquid type inference to produce the list of leak signatures by (1) labeling Horn clauses and (2) using a version of the least fixed point algorithm that finds *all* violations, rather than the first violation we can find. LIFTY's type checker labels each Horn clause it generates with the name of the variable whose type is constrained by this clause. For example, `print w mem out` where `mem`: $\{$`User` $\mid \nu =$ `pcMember w i` $\wedge\ \nu \in$ `pc w`$\}$from our introductory example (Fig. 2) produces (among others) a Horn clause labeled with `mem`:

$$\texttt{mem:}\ \nu = \texttt{pcMember}\ w\ i\ \wedge \nu \in \texttt{pc}\ w \Rightarrow P_1$$

where P_1 is the (as yet unknown) policy parameter of this **print**; this clause corresponds to the precondition on `mem` that it satisfy the policy. All Horn clauses generated by the type checker are either *definite clauses* of the form $\psi \wedge \bar{P} \Rightarrow P$ (like the one above; $\bar{P}$ stands for a conjunction of multiple unknowns) or *goal clauses* of the form $\psi \wedge \bar{P} \Rightarrow \phi$, where ϕ is a known formula. Whereas the Liquid Haskell type checker looks for the first offending term, we want all offending terms. Thus our implementation of the least fixpoint algorithm first finds the strongest solution that satisfies *all* definite clauses and then checks which goal clauses are violated by this solution. (Note that finding the strongest solution is always possible since a definite clause can always be satisfied by assigning $\top$ to its right-hand side.) The labels of these goal clauses give us the list of variables to return as leaks.

It turns out that we can rely on type checking to determine, for an insufficiently protected sensitive value, both (1) the precise source access that is "too secret" for the sink it is flowing into, and (2) the most restrictive policy it must satisfy in order to be "public enough" for that sink (represented by the solution to definite subset of Horn clauses). Normally, when type checking functional properties, goal clauses arise from checking either preconditions of function calls or the top-level user-provided type annotation. Because the policy parameter of the `Tagged` type is contravariant, however, policy checks produce Horn clauses with the two sides flipped, so goal clauses correspond to the user-specified policies on the sources of the sensitive data. For instance, in the introductory example, binding the variable `s` to rest of the `Tagged` computation produces the constraint `s` : $P_0 \Rightarrow u' \in \texttt{pc}\ w' \wedge u' \notin \texttt{conflicts}\ w'\ pid$ (where P_0 is the policy parameter of the corresponding **bind** and w', u' are fresh variables). As a result, the first phase of the least-fixpoint algorithm has the effect of propagating the type of the sinks all the way backwards through a `Tagged` computation, resulting in the assignment $P_0 \mapsto u' = \texttt{pcMember}\ w\ i\ \wedge u' \in \texttt{pc}\ w'$ for this example. The second phase has the effect of identifying accesses to sources whose policies are too restrictive for the inferred sinks, such as `s`, whose goal clause does not hold for the inferred solution to P_0.

4.2 Fix generation

We now give details of the FIX procedure outlined on lines 8–14 of Algorithm 1. Given a leak signature (x, T), the function finds the violating binding **let** $x = f\ \bar{v}$, which it has to replace with some **let** $x = e'$. Since we only need a specific kind of repair, finding e' reduces to solving the following local synthesis problem:

$$\Gamma \vdash \texttt{ifM}\ (\texttt{??})\ (f\ \bar{v})\ f_{def} :: T$$

Here f_{def} is the user-defined default alternative for the source f: we require that for every component $f : \bar{U} \rightarrow \texttt{Tagged}\ T\langle p \rangle$, the user designate, through a special annotation, a component f_{def} : `Tagged` $T\langle \top \rangle$ to serve this purpose. Thus the only unknown term in the synthesis problem is the check. Note that this synthesis problem is completely local, *i.e.* can be solved independently from other violations.

LIFTY's synthesizer relies on procedures from the SYNQUID tool for synthesis from refinement types [36], but with a key modification. While off-the-shelf SYNQUID can solve our problem in principle, the monadic code LIFTY needs to synthesize is the worst-case scenario for SYNQUID's goal-directed approach. Our insight for efficient synthesis is that we can make use of the property that functional properties (*i.e.* compute a condition that is strong enough to make $f\ \bar{v}$ comply to the policy in T) are orthogonal to confidentiality policies (*i.e.* the check itself should not be too secret). Synthesis in LIFTY first tries to satisfy the functional specification and then checks if the result is too secret.

LIFTY performs synthesis in three steps.

Condition abduction. LIFTY infers the weakest precondition ψ that would make the first branch of the conditional above type check (line 9). To that end, LIFTY uses the *liquid abduction* technique from SYNQUID, which searches for a solution as a minimal conjunction of atomic predicates (qualifiers) from a given set. There might be no unique solution ψ: abduction may return multiple solutions, which we treat as a disjunction. If the weakest ψ the solver can construct out of given qualifiers is $\perp$, the system issues a warning that it failed to adbuce a nontrivial access check.

Check synthesis. In the next step (line 10), we use SYNQUID to synthesize from the abduced condition a pure version of the check, *i.e.* a program term c of type $\{\texttt{Bool} \mid \nu \Leftrightarrow \psi\}$; the synthesis is performed in a modified environment Γ', where all sensitive components are stripped of their tags. Since this is non-monadic code, SYNQUID can synthesize it efficiently.

Lifting. On line 11 we lift the pure term c into a `Tagged` computation c' through a simple syntactic transformation, inserting calls to **bind** and **return** where required. Since the lifting step is purely syntactic, if policies depend on sensitive values the resulting lifted check might end up being too private for the policies in T. For this reason, the REPAIR algorithm re-checks the solution on line 5.

4.3 Implementation

We have implemented LIFTY in Haskell, using the same minimal Haskell dialect as SYNQUID and using infrastructure pro-

vided by the SYNQUID synthesizer [36]. We implemented the least-fixpoint Horn solver required for VERIFY on top of SYNQUID's abduction and program synthesis mechanisms. We also enhanced SYNQUID's qualifier extraction procedure. Like SYNQUID, LIFTY uses the Z3 SMT solver [13] for solving Horn constraints. We also implemented a SYNQUID to Haskell compiler that enables executing the code repaired by LIFTY and linking it with non-security-critical modules written directly in Haskell.

5. Evaluation

We implemented a conference management system example comprising of several views and forms, measuring code size and compiler performance. We demonstrate the following:

- **Expressiveness of policy language.** We demonstrate that we can use LIFTY's policy language to implement realistic systems with nontrivial policies.
- **Support for policy-agnostic programming.** We compare LIFTY's output to checks that were written manually. We show that not only does our policy specifications allow for information checks to be centralized and concise, but also that the compiler is able to recover *all* necessary checks, without reducing the functionality.
- **Good performance.** We demonstrate that the LIFTY compiler is sufficiently efficient at verification, error localization, and repair to use for systems of reasonable size. We demonstrate that LIFTY is able to generate all necessary checks for our conference management system (421 lines of LIFTY) in about seven minutes.

5.1 Overview of Case Study

We implemented a basic conference management system, using LIFTY to implement all information policy checks. It consists of a rewrite, in LIFTY, of the system used as a case study in [51]. The system handles confidentiality policies for papers in different phases of the conference (`Submission`, `Review`, and `Done`) and different statuses of each paper (`NoDecision`, `Accepted`, and `Rejected`). Users of the system have the roles of author, PC member, and PC chair. Policies depend on this state, as well as additional properties such as conflicts with a particular paper. The system provides features for displaying (1) paper title and authors, (2) paper status, (3) list of conflicts, and (4) conference information conditional on acceptance. Information may be displayed to the user currently logged in ("session user") or sent via various means to different users.

The system contains 785 lines of code in total (421 LIFTY + 364 Haskell) and provides a superset of the functionality shown in our micro-benchmarks. This case study exposes some cross-dependencies between software features. We show statistics, broken down by the different components of the system, in Tab. 1.

5.2 Measuring the Quality of Repair

Towards quantitatively and qualitatively evaluating LIFTY's repair capabilities, we had a developer who was not involved with developing LIFTY build an alternate implementation of the conference management system with manual checks. For this benchmark we compare three versions of the code: (1) a policy-agnostic implementation with no checks at all, (2) an implementation with manually implemented checks, and (3) an implementation with automatically generated checks.

We show the results of the comparison in Tab. 1. The column "Original" shows the size of the code, in terms of number of tokens, without any security checks. Then we show the size of additional security checks, both those inserted manually by a human programmer and those automatically generated by the system. The size of the predicates specifying the policy is given as "Policy size". Note that the checks sometimes approach the size of the code, confirming our hypothesis that for many applications, much of the programming burden is in the security checking.

Our results reveal that while manual checks are more concise than LIFTY-generated checks, the tool generates checks that are the same order of magnitude. The most code overhead is 3×. We found that the bloat in the automatically generated code comes from redundancy and unnecessary verbosity, rather than from additional functionality; for example, LIFTY would typically generate an expression such as `ifM` t_1 e_1 (`ifM` t_2 e_1 e_2) instead of `ifM (liftM or` t_1 t_2) e_1 e_2, essentially duplicating e_1 and causing some bloat. However, this affects only the size of the code and neither its functionality nor its performance. The manual and automatic checks were semantically equivalent across our benchmarks: the checks are not more conservative than needed.

5.3 Performance Statistics

We show the performance of the LIFTY compiler for the different transactions implemented in our case study, in Tab. 1. We break down running time into verification, error localization, and synthesis of new checks. For the version that contains manual checks, we show only verification time, as LIFTY skips the other phases. Notice that the LIFTY is able to determine that six of our benchmarks required no checks at all; in particular, all the writes are safe.

It is important to explain that the repair of 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. Other than such cross effects, changing the body of one function does not require recompilation of other functions. This makes it possible to cache compilation artifacts to speed up development.

6. Related Work

We describe related work in language-based information flow security, program synthesis, and program repair.

Type-based information flow control. LIFTY builds on a long history of work in language-based information flow [40]. Label-based approaches [4, 7, 11, 29, 33, 37, 54] provide a basis for using value-dependent types for security. As the label-based approach trusts programmers to correctly express high-level policies in terms of label-manipulating code, people have re-

Policy size (tokens): 345

Benchmark	Program size (tokens) Original	Security checks Manual	Security checks Auto	Time Manual Verify	Auto Verify	Auto Repair	Auto Recheck	Auto Total
Register user	10	0	0	0.00s	0.00s	0.00s	0.00s	0.00s
View users	20	9	24	4.01s	2.72s	0.43s	9.21s	12.37s
Paper submission	45	0	0	5.15s	5.45s	0.00s	0.00s	5.45s
Search papers	73	120	82	62.34s	23.17s	16.68s	72.10s	111.96s
Show paper record	53	46	82	39.18s	13.08s	19.24s	62.13s	94.45s
Show reviews for paper	57	54	45	68.04s	23.11s	99.16s	63.29s	185.57s
User profile: GET	66	0	0	5.03s	5.84s	0.00s	0.00s	5.84s
User profile: POST	17	0	0	1.14s	1.24s	0.00s	0.00s	1.24s
Submit review	40	0	0	4.55s	4.88s	0.00s	0.00s	4.88s
Assign reviewers	47	0	0	16.22s	17.67s	0.00s	0.00s	17.67s
Totals	428	229	233	205.70s	97.19s	135.52s	206.74s	439.46s

Table 1. Case study: conference management system.

ferred to labels as providing an "assembly language" [20] for enforcing security policies.

Like other approaches using expressive value-dependent types [9, 10, 22, 45, 46], LIFTY additionally provides a policy language on top of the types. Unlike Fine [9, 45] and F* [46], however, type-checking for LIFTY is decidable, making it better suited for casual development. LIFTY additionally supports policies that may depend on sensitive values, and thus be self-referential. Type-checking information flow in AURA [22] is decidable, but as AURA uses authorization logic, programmers must explicitly pass proof terms, whereas inference in LIFTY automates this process.

Similar to the non-interference property dynamic policy-agnostic approaches [], our contextual non-interference property extends standard non-interference to be more like a declassification property [28] in order to take into account potential dependencies of policies on sensitive values, and the parameterization of policies by the viewer. Our parameterized policies are like those in UrFlow [10], but UrFlow does not support negative or self-referential policies.

Policy-agnostic programming. LIFTY supports the first static solution for policy-agnostic programming. The Jeeves language [6, 50] and Jacqueline web framework [51] support a programming model where the programmer implements information flow policies as program functions and runtime performs *faceted execution* [5], simulating simultaneous multiple executions in order to propagate sensitive values and policies. There are two main drawbacks: (1) nontrivial runtime overheads and (2) difficulty of reasoning about program behavior. Our static repair-based approach supports similarly expressive policies without these drawbacks.

Error Localization and Program Synthesis. LIFTY relies on capabilities provided by liquid type inference [38, 47–49] for verification and error localization. The localization problem we solve is easier than that of Haskell type error localization tools such as SHErrLoc [53], since it is meant for consumption by our synthesis algorithm rather than by a human developer. LIFTY's repair technique uses abduction technique of the SYNQUID tool [36], which, like other prior approaches for program synthesis [1–3, 14, 17, 21, 26, 32, 34, 42], solves synthesis problems (1) based on full functional specifications and (2) for synthesizing self-contained, rather than cross-cutting, functionality.

Program repair. Our repair solution differs from prior work on general repair techniques [12, 23, 24, 27, 30, 31, 35, 41] in that it is (1) sound, (2) based on specifications that are semantically intertwined with the rest of the program, and (3) based on specifications of a cross-cutting concern rather than on full functional specifications.

LIFTY also differs from prior work on rewriting programs based on security concerns because of the generality of its policies, and because it can rewrite the program to do more than halting or silently failing when checks are not satisfied. The SWIM tool [20] performs automatic instrumentation to insert process-level label-manipulation code for operating system-level decentralized information flow control. *Policy weaving* [18] inserts checks into programs based on stateful, temporal logic access policies, but does not track implicit flows. Similarly, there are also repair solutions for access control [19, 43] that do not reason about the interaction between sensitive values and the rest of the program.

7. Conclusions

We demonstrate that by encoding information flow policies as refinement types, we can develop a sound and automatic program repair technique to insert missing conditional policy checks across a program. This allows us to support a policy-agnostic programming model, where the compiler, rather than the programmer, is responsible for implementing policy checks. We show how, by decomposing a global synthesis problem into local synthesis problems, we can decrease the opportunity for programmer error to cause information leaks.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, 2013.

[2] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

[3] R. Alur, P. Černý, and A. Radhakrishna. Synthesis through unification. In *CAV*, 2015.

[4] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Oakland*, 2012.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.

[6] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *PLAS*, 2013.

[7] N. Broberg and D. Sands. Flow Locks: Towards a core calculus for dynamic flow policies. In *ESOP*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[8] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. In *CSF*, 2014.

[9] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*, 2010.

[10] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, 2010.

[11] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, 2009.

[12] Z. Coker, D. Garlan, and C. Le Goues. Sass: Self-adaptation using stochastic search. In *SEAMS*, 2015.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[14] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[15] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, 2001.

[16] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[17] J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

[18] M. Fredrikson, R. Joiner, S. Jha, T. W. Reps, P. A. Porras, H. Saïdi, and V. Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *CAV*, 2012.

[19] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *SP*, 2006.

[20] W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. In *CCS*, 2010.

[21] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015.

[22] L. Jia and S. Zdancewic. Encoding information flow in aura. In *PLAS*, 2009.

[23] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing Programs with Semantic Code Search. In *ASE*, 2015.

[24] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *CAV*, 2015.

[25] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[26] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38, 2012.

[28] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. 2005.

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

[30] F. Long and M. Rinard. Staged program repair with condition synthesis. In *FSE*, 2015.

[31] F. Long and M. Rinard. Automatic patch generation by learning correct code. 2016.

[32] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1), Jan. 1980.

[33] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[34] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[35] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[36] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

[37] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), Jan. 2003.

[38] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[39] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411289. URL `http://doi.acm.org/10.1145/1411286.1411289`.

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

[41] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. 2015.

[42] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[43] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *NDSS*. The Internet Society, 2013.

[44] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Oakland*, 2008.

[45] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*, 2010.

[46] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011.

[47] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[48] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In *Haskell*, 2014.

[49] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *ICFP*, 2014.

[50] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies, 2012.

[51] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, 2016.

[52] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. *SOSP*, 2009.

[53] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones. Diagnosing type errors with class. In *36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2015.

[54] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, 2007.