## Title

Liquid information flow control

## Permalink

https://escholarship.org/uc/item/0t20j69d

## Journal

Proceedings of the ACM on Programming Languages, 4(ICFP)

## ISSN

2475-1421

## Authors

Polikarpova, Nadia
Stefan, Deian
Yang, Jean
et al.

## Publication Date

2020-08-02

## DOI

10.1145/3408987

Peer reviewed

# 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

Programs that compute over sensitive data are becoming more complex. In addition to directly displaying sensitive values, applications often support functionality such as search and aggregation. To protect users' privacy and prevent information from being disclosed to unauthorized users, programmers must implement access checks across the program. Any missing check may result in an information leak.

Traditional approaches to information flow control can detect leaks that result from missing access checks, but the amount of programmer effort required to write secure code remains high. Techniques for language-based information flow security [43] statically verify the absence of such leaks, but they require the programmer to first correctly implement information flow checks across the program and then additionally provide specifications of permitted flows. While dynamic approaches [9, 27, 54] decrease the annotation burden, the programmer remains responsible for correctly implementing checks in order to avoid exceptions or silent failures.

We take a *policy-agnostic* approach [7, 52], factoring out information flow policies from the rest of the program to mitigate programmer burden. In prior work on policy-agnostic programming [7, 52, 53], the programmer implements each information flow policy once, associated with data definitions, rather than as checks across the program. The remainder of the program may be free of checks and the language runtime is responsible for steering dynamic behavior to adhere to policies. While this programming model makes it impossible to leak information through missing checks, the dynamic solution unfortunately involves potentially prohibitive overheads and unpredictable runtime behavior.

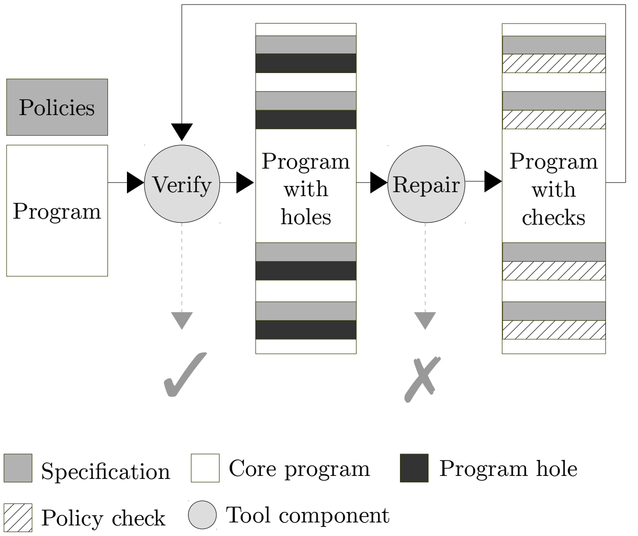

**Figure 1.** Type-driven repair for policy-agnostic programs.

We take a static approach to avoid the pitfalls of runtime techniques for policy-agnostic programming. Generating information flow checks seems like a perfect application for program synthesis, as what we want is to synthesize small code snippets corresponding to policy checks. To make the synthesis problem tractable, however, we need to be able to synthesize each check independently. Thus, we need a way to decompose the global repair problem into local synthesis problems. This is challenging because information flow checks may depend on both sources and sinks for sensitive data and there may be complex computations in between.

To address these challenges, we propose a type-driven solution for policy-agnostic programming. At the core of our technique is the insight that we can use refinement types—types decorated with decidable predicates [41, 49]—to statically enforce a policy-compliant semantics. Refinement types have the advantages that(1) they support expressive policies, (2) type-checking is decidable, and (3) type inference can provide error localization while reducing annotation burden. In our solution, the programmer implements each information flow policy only once, as a type annotation, rather than as repeated checks across the program; the type checker is responsible for identifying unsafe flows, and a subsequent repair phase prevents these flows by automatically inserting conditional expressions implementing the policy checks into the code. The approach is fully static and requires no runtime analysis.

In this paper, we present(1) a policy-agnostic, security-typed language called LIFTY (Liquid Information Flow TYpes) and (2) a compiler that repairs LIFTY programs by inserting information flow checks. In Fig. 1 we show an overview of LIFTY and its compilation process. In LIFTY, the programmer specifies information flow policies as refinement type associated with sources of sensitive data. LIFTY's verifier uses type inference techniques inspired by *liquid types* [41] to produce a program with holes, where each hole corresponds to an unsafe data access paired with a local policy specification. The repair phase builds on an existing type-based program synthesis technique [39] to produce a policy check for each hole. We formalize a core language for LIFTY, prove a non-interference property, and demonstrate the practical promise of our approach using an implementation of a LIFTY-to-Haskell compiler.

This paper makes the following contributions:

- **Static, type-based approach to policy-agnostic programming.** We present a programming model that supports the implementation of information flow policies as refinement types, separately from other functionality. The compiler, rather than the programmer, becomes responsible for generating code to implement the policies.
- **Verification of expressive type-based policies.** In order to support policy-agnostic programming, we developed a verification technique for information flow security based on liquid type inference. The technique is sound, has minimal annotation overhead, and supports expressive policies and program constructs (such as recursion and higher-order functions).
- **Formalization and proof of security guarantee.** We formalize our verification approach in terms of refinement types [41, 49] and prove a non-interference property. Our proof technique uses a *phantom encoding* to reduce the problem of proving non-interference to the problem of proving type safety, within the same language.
- **Multistage program synthesis for separating concerns.** We introduce a two-stage technique for synthesizing policy checks that handles the functional specification of the check separately from its information flow specification. The separation is based on the insight that the two concerns are orthogonal, and helps make the synthesis problem tractable.
- **Demonstration of practical promise.** We implement a LIFTY-to-Haskell compiler and demonstrate through micro-benchmarks and 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 (just over two minutes for the entire conference management system), and can repair programs to prevent reported real-world leaks.

## 2. Introductory Example

We introduce the programming model of LIFTY using code from our conference management case study. We first show how to implement core functionality in a policy-agnostic style and add specifications for information flow policies. Then we explain how the LIFTY compiler inserts access checks to ensure policy compliance.

### 2.1 Policy-Agnostic Core Functionality

The core functionality of a conference management system is to provide access to a persistent store of paper submissions to authors and reviewers. The function `showPaper` in Fig. 2 gets as argument a paper ID `pid`, retrieves the paper's title and author information from the persistent store, and displays it to the current user[1]. Note that the function has both read effects (retrieval from the store) and write effects (output to user). For simplicity, 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 persistent store and the observations made by the users.

```
showPaper w pid =
  let u = getCurrentUser w
      out = do title ← getPaperTitle w pid
               authors ← getPaperAuthors w pid
               return (title ++ ":␣" ++ show authors)
  in print w u out
```

**Figure 2.** Excerpt from a conference management server code.

The body of `showPaper` accesses potentially sensitive data using accessor functions `getCurrentUser`, `getPaperTitle`, and `getPaperAuthors`. For example, implementing double-blind review requires hiding the list of paper authors from ordinary reviewers and making it visible only to the program chair. One way to enforce this policy using conventional programming paradigms, is to guard the call to `getPaperAuthors` by a conditional that checks if the session user is allowed to see the author list, and alternatively returns a constant default value. Note that the check is specific not only to the data element being read, but also to the eventual viewer of the result. (In this case it happens to be the session user, but this is not always the case.) Because of this, the problem of checking cannot be solved simply by delegating the checks to accessor functions. Instead, a potentially different access check has to appear in *every* computation that involves `getPaperAuthors`, which quickly becomes tedious and error-prone.

LIFTY obviates the need for writing policy checks: as long as the programmer correctly annotates `getPaperAuthors` with the desired policy, the compiler will automatically guard each invocation of this function with an appropriate policy check. To this end, the compiler has to propagate the information about where a sensitive value is flowing backwards through the computation towards the source of the value. We achieve this by wrapping every sensitive computation in a "static taint tracking" monad we call `Tagged`. A computation in this monad has a refined type that keeps track of the policy associated with the result, while at runtime `Tagged` has no effect (*i.e.* it is equivalent to the identity monad). In Fig. 2, in the interest of readability, we wrap the sensitive computation of the `out` variable in a Haskell-like **`do`**-notation (in our implementation, this function is written without the **`do`**; instead, the string operations are lifted into the `Tagged` monad.)

### 2.2 Adding Policies Through Types

In Fig. 3 we show how to specify a policy that paper authors are only visible to the program chair. In LIFTY the programmer designates accessor functions as sensitive by wrapping their return type in a `Tagged` type constructor, which is parameterized by a *predicate* corresponding to the information flow policy. More precisely, the type `Tagged` $\alpha$ `<`$P$`>`, where $P$ is a unary predicate, stores a value (of type $\alpha$) that can only be seen by a user $u$ provided that $P\ u$ holds. In the example, we gave the function `getPaperAuthors` the type `Tagged [User] <`$\lambda\nu\,.\,\nu =$ `chair w>`[2]. This policy says that a

[1] In most web-based systems, there is a notion of a "session" and a session's "current user".

[2] For readability, in the rest of the paper we always use $\nu$ for the sole argument of the policy predicate and omit the binding.

```
getCurrentUser :: World → Tagged User <True>
getPaperTitle ::
    w: World → PaperId → Tagged String <True>
getPaperAuthors ::
    w: World → PaperId → Tagged [User] <ν = chair w>
```

**Figure 3.** A basic policy for double-blind review.

```
print :: <P: User → Bool> .
  World → viewer: Tagged {User | P ν} <P>
    → msg: Tagged String <P> → World
```

**Figure 4.** Output function from LIFTY standard library.

```
showPaper w pid =
  let u = getCurrentUser w
      out = do
        title ← getPaperTitle w pid
        t1 ← getChair w
        t2 ← u
        authors ← if t1 = t2 then getPaperAuthors w pid
                  else return ["??"]
        return (title ++ ":␣" ++ show authors)
  in print w u out
```

**Figure 5.** Example implementation with injected policy code.

```
getSessionNo :: World → PaperId → Tagged Int <True>
getPaperStatus ::
    w: World → PaperId
            → Tagged Status <currentPhase w = Done>

showPaper w pid =
  let u = getCurrentUser w
      out = do
        t1 ← getCurrentPhase w
        st ← if t1 = Done then getPaperStatus w pid
                     else return NoDecision
        if st = Accepted
          then liftM show (getSessionNo w pid)
          else return ""
  in print w u out
```

**Figure 6.** An implicit flow example and the code injected to overcome it.

viewer $\nu$ may see the return value of the function as long as it is equal to the chair field of the persistent store. Note that policy predicates can directly refer to the fields of the persistent store (such as chair above), while the executable program can only obtain their Tagged versions by means of accessor functions; this separation is important in order to support policies that themselves depend of sensitive values (see Sec. 6).

## 2.3 Output functions

Output functions, such as **print** in Fig. 2, are responsible for imposing the requirement that the sensitive values they consume are visible to the target of the output. In Fig. 4 we show how this is accomplished through the type signature of **print**. 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. (We show interesting cases of this in Sec. 6.) When checking an application **print** w u x, the LIFTY 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 u must hold (as expressed by the refinement {User | P $\nu$} in the type of viewer).

Even though in web applications the viewer of most output operations is the current session user, we cannot assume that this is always the case. Supporting functionality such as sending email has led to real-world information leaks: one such documented bug in the HotCRP conference management allowed users to send password reminders for any other user—to themselves [54]. By making the viewer explicit in output operations and enforcing policies with respect to an arbitrary viewer, LIFTY can prevent such leaks.

## 2.4 Inserting Policy Checks

Our goal is to get the policy-agnostic code from Fig. 2 to adhere to the policy we specified in Fig. 3. Prior tools for static verification of information flow properties [5, 10, 30, 40, 43] will alert the programmer that checks are missing, but they will not help the programmer produce the checks. Using LIFTY, the programmer may write policy-agnostic programs and rely on the *compiler* to generate the necessary checks.

The key innovation in the LIFTY compiler is a repair algorithm that generates policy-enforcing code. LIFTY first attempts to verify the code against the provided annotations. In our example, LIFTY detects that a value with policy $\nu =$ chair w flows into the argument out, which is required to have policy $\nu =$ u (or weaker). Since $\nu =$ u $\not\Rightarrow$ $\nu =$ chair w, LIFTY deems this flow unsafe. There are several ways to prevent this flow. One option is to wrap the **print** invocation itself in a conditional; this would fix the leak but will have an undesired side effect of hiding the paper title along with the author list. The goal of the LIFTY compiler is to preserve as much of the original program behavior as possible, thus it always chooses to guard the smallest possible subterm, which has the effect of inserting checks directly at the source of the sensitive value. In this example, LIFTY identifies getPaperAuthors w pid as the offending source.

Fig. 5 shows the repaired version of the code, in which the invocation of getPaperAuthors is guarded with an appropriate check. In the code, shaded areas indicate injected policy code evaluated in the Tagged monad. This code retrieves the value of the program chair using getChair (the accessor function for the field chair) and compares it to u. In case the policy is violated, the guarded access returns a default value ["??"]. LIFTY requires that the programmer designate a constant default value for every sensitive accessor method involved in repair.

Note that in a more realistic example, several sensitive values with different policies are likely to flow into the same **print** operation. This presents no problem for LIFTY: since the repair is performed at the source, every sensitive value access will be guarded with its own check, which complies with our goal of preserving maximum functionality.

## 2.5 Preventing Implicit Flows

Information leaks can also occur because viewers access values derived from sensitive data as a result of control flow. Suppose that in our conference management system, authors are not allowed to see whether a paper has been accepted prior to the notification date. Now consider a function that prints out, for an accepted paper, which conference session it has been assigned to, and prints an empty value if the paper has not been accepted. If this function is executed on behalf of an author, they might infer whether their paper has been accepted, based on the session value they observe; in fact, this is a documented leak in the EDAS conference management system [1]. We show the code for this, along with the repairs, in Fig. 6. Because Lifty regards the condition `st = Accepted` as secret, the result of the `if` expression is also secret. Since the result flows into an output, the compiler wraps the source of the sensitive data (`getPaperStatus`) with a check. (Small note: `show` converts a value to a string; `liftM` lifts an operation to `Tagged`.)

In summary, Lifty ensures that any value derived (either explicitly or implicitly) from a sensitive value will be shown to a viewer only if the policies allow. The only parts of the code that need to be trusted are the policy predicates in the types of the accessor functions and the output functions, such as `print`, which can be written once and reused across different systems. Note that if the programmer has already implemented the checks in the program, Lifty will leave the program as is. Lifty is also able to enhance existing checks by adding conditionals inside of the existing ones.

## 3. Solution Overview

We now describe the encoding and algorithm that enable Lifty to insert checks. The main insight is that we can reduce the problem of inserting information flow checks to the problem of local synthesis from refinement types. By extending the subtyping rules for refinement types to support *phantom predicates*, we can use liquid type checking [41] for decidable verification and error localization. Our encoding also allows us to use type inference both for propagating policies through a program and for associating each source of a policy violation with the specific policy that has been violated. We can then use the abduction technique of the tool Synquid [39] for synthesizing the necessary conditional checks. In this section we provide an overview of the solution. We present the detailed formalism in Sec. 4 and the detailed repair algorithm in Sec. 5.

### 3.1 Security Policies as Phantom Refinements

The key to understanding our solution is understanding the `Tagged` monad that we introduced in Sec. 2. `Tagged` is a data container that marks sensitive values and associates them with their policies for the purpose of static verification, error localization, and repair. It has a single constructor, which is *private* — it cannot be referenced in user code, which ensures that such code cannot accidentally deconstruct the value and discard its policy; instead, this is done in a safe manner inside core library routines such as `bind` and `print`. This encapsulation provides the following two properties:(1) all results of computations of a tagged value are tagged with the same policy (or one that is more strict), and (2) once a value is tagged, it remains tagged for all subsequent computations.

Lifty employs *refinement types* and allows the programmer to express policies as *refinement predicates*. Recall from Fig. 3 that the `getPaperAuthors` function has the return type `Tagged [User] <`$\nu =$ `chair w>`. The expression $\nu =$ `chair w` is the refinement predicate for the `Tagged` type of the returned value, where $\nu$ is a reserved name for that predicate's argument. Because the function has this type, programs will only compile if the type checker can statically prove that eventual `print` destinations of this value have (at most) the chair as the viewer. Once values become tagged, Lifty's type checker is responsible for correct propagation of policies through derived values.

Our encoding allows Lifty to take advantage of the main benefit of liquid types: automatic type inference. This alleviates the annotation burden, allowing programs to be policy-agnostic, associating policy-related annotations *only* with sources of sensitive data, rather than throughout the program. This level of automation is not achievable with the general value-dependent types previously used for security verification [47, 48].

Note also that there is a key difference between Lifty's use of predicate parameters and the standard use [49] for abstracting over functional properties. With the standard usage, we can define a data type `IntPair` with a constructor such as `C :: x:Int` $\rightarrow$ `y:{Int` $|$ $P$ `x` $\nu$`}` $\rightarrow$ `IntPair <`$P$`>`, such that $P$ denotes the relation between the two components of the pair. In Lifty, policy predicates carry information that has nothing to do with the runtime values of the type, but that describes which viewers are allowed to see each value. Thus policy predicates are *phantom*, *i.e.* parameters that do not appear in the arguments of the data constructor, a notion analogous to phantom types in Haskell.[3] Our use of phantom predicates not only supports our encoding of information flow policies, but it also simplifies our formalization and proof of our security property. In Sec. 4 we provide the formal details of phantom predicates and safety in Lifty.

### 3.2 Inserting Security Checks Through Program Repair

The stages of repair are as follows.

***Verification.*** The Lifty type-checker uses a variation of liquid type checking to verify the code against the provided annotations. From our example in Fig. 3, Lifty infers the type judgement `out :: Tagged String <`$\nu =$ `u>` for the program in Fig. 2 by collecting type constraints and solving them (as explained in Sec. 5.1). Through the application of monadic `bind`, Lifty also infers the type `authors :: Tagged [User] <`$\nu =$ `u>`. At this point, Lifty determines that `getPaperAuthors w pid` will not type-check, since its expected type is `Tagged [User] <`$\nu =$ `u>`, while its actual type is `Tagged [User] <`$\nu =$ `chair w>`, according to the type signature of `getPaperAuthors` (Fig. 3).

To reason about policies that may be stronger or weaker than the explicitly stated policy, we take advantage of a *subtyping* relationship between refinement types. The important subtyping rule to note is `Tagged<`$P$`>` $<:$ `Tagged<`$Q$`>` iff $Q \Rightarrow P$. For this reason it is essential for the phantom predicate parameter to be *contravariant*, so as to allow more public values to flow into more secret values, and not the other way around. We define the subtyping rules in Sec. 4. Note that this check is always decidable for the predicates allowed by the type system. Decidability is especially important for a verification procedure to be used for automated synthesis.

***Error localization.*** The information obtained from type inference is used to localize errors. Since $\nu =$ `u` $\not\Rightarrow$ $\nu =$ `chair w`, Lifty will not only mark this as a type error, but it will also identify `getPaperAuthors w pid` as the offending term. Whereas standard error localization for liquid types finds the first offending term, Lifty identifies *all instances* of subtyping violations. Lifty's use of contravariance, along with its implementation of type inference, allows Lifty to identify precise locations for expressions that need to be wrapped in conditional checks, as well as specifications for the check holes.

***Check synthesis.*** The final stage of compilation involves replacing the holes with checks implementing the policies. The problem of synthesizing information flow checks has the nice property that it is restricted to expressions of Boolean type, allowing us to use Synquid's *abduction* to infer a sufficient logical condition for the hole. We then use Synquid to translate the logical condition into a program term. In our example, Synquid combines the local variables `u` and `w` with the context components `getChair` and `eq` to construct a program term equivalent to the logical formula `u` $=$ `chair w`. For efficiency purposes we do not use Synquid off-the-shelf; we explain our modifications in Sec. 5.

[3] `https://wiki.haskell.org/Phantom_type`

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

**Figure 7.** Terms and types.

# 4. 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 $\lambda_P$, the core language of Abstract Refinement Types [49], 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 $\lambda_P$'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 `Tagged`$^2$ monad that relates pairs of executions and showing that type-checking with `Tagged`$^2$ implies non-interference with `Tagged`. Since the repair phase always generates type-correct programs, this is sufficient for verifying the correctness of LIFTY's repair.

## 4.1 Syntax and Types of $\mathcal{BL}$

We now present $\mathcal{BL}$. Like $\lambda_P$, $\mathcal{BL}$'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 $\lambda_P$ [49]. $\mathcal{BL}$ 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. 7.

***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 [20]: 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 $\psi$ of the refinement logic and applications of predicate variables $\pi$. 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.*** A $\mathcal{BL}$ type is either a scalar—a refined base type—or a dependent function type. 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.

## 4.2 $\mathcal{BL}$ Static Semantics

In Fig. 8 we show the relevant subset of well-formedness, subtyping, and type checking rules for $\mathcal{BL}$. 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 $\lambda_P$ are standard, they are important for deriving our safety property.

In our semantics, a *typing environment* $\Gamma$ 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 $\forall_\circ \bar{\alpha}.\forall_\circ \overline{\pi : T}.T_1 \to \ldots \to T_n \to \{D\ \bar{\alpha}\ \overline{\pi\ \bar{x}} \mid r\}$ and determines the type and predicate parameters of the type constructor $D$.

***Well-Formedness.*** A refinement $r$ is *well-formed* in the environment $\Gamma$, written $\Gamma \vdash r$, if it sort-checks to Boolean and none of its predicate variables are bound in a contravariant manner in $\Gamma$. We use a judgment $\Gamma \vdash r : T$ in the premises of rules WF-$\psi$ and WF-$\pi$ 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, $\Gamma^-$ 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. 8) 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. In the rule P-IF, 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

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

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

$$\text{WF-}\alpha\ \frac{\Gamma(\alpha) \neq \ominus}{\Gamma \vdash \alpha} \qquad \text{WF-}\textsc{Sc}\ \frac{\Gamma \vdash B \qquad \Gamma; \nu : B \vdash r}{\Gamma \vdash \{B \mid r\}}$$

$$\text{WF-}\textsc{Fun}\ \frac{\Gamma^{-} \vdash T_x \qquad \Gamma; x : T_x \vdash T}{\Gamma \vdash T_x \to T}$$

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

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

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

$$<:\text{-}\textsc{Sc}\ \frac{\Gamma \vdash B\ <:\ B' \qquad \mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge r \Rightarrow r')}{\Gamma \vdash \{B \mid r\}\ <:\ \{B' \mid r'\}}$$

$$<:\text{-}\textsc{Fun}\ \frac{\Gamma \vdash T_y\ <:\ T_x \qquad \Gamma; y : T_y \vdash [y/x]T\ <:\ T'}{\Gamma \vdash x\colon T_x \to T\ <:\ y\colon T_y \to T'}$$

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

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

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

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

$$\textsc{Var-Sc}\ \frac{\Gamma(x) = \{B \mid r\}}{\Gamma \vdash x :: \{B \mid \nu = x\}} \qquad \textsc{Var}\ \frac{\Gamma(x) = S \qquad S \text{ non-scalar}}{\Gamma \vdash x :: S}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash T_x \qquad \Gamma; x\colon T_x \vdash e :: T}{\Gamma \vdash \lambda x : T_x.e :: (x\colon T_x \to T)}$$

$$\textsc{Let}\ \frac{\begin{array}{c}\Gamma \vdash v_1 :: (y\colon T_y \to T') \qquad \Gamma \vdash v_2 :: T_2 \\ \Gamma \vdash T_2 <: T_y \qquad \Gamma; x : [v/y]T' \vdash e :: T\end{array}}{\Gamma \vdash \texttt{let}\ x = v_1\ v_2\ \texttt{in}\ e :: T}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash x :: \{D\,\bar{T_x}\,\langle \bar{p_x} \rangle \mid r_x\} \\ \Gamma(D) = \forall_{\circ}\bar{\alpha}.\forall_{\circ}\langle \bar{\pi} \rangle.T_1 \to \ldots \to T_n \to \{D\,\bar{\alpha}\,\langle \bar{\pi} \rangle \mid r\} \\ \Gamma; y_i : [\bar{T_x}/\bar{\alpha}][\bar{p_x} \rhd \bar{\pi}]T_i; x : \{D\,\bar{T_x}\,\langle \bar{p_x} \rangle \mid r_x \wedge r\} \vdash e :: T\end{array}}{\Gamma \vdash \texttt{match}\ x\ \texttt{with}\ D\,\bar{x} \to e :: T}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash x :: \{\texttt{Bool} \mid r\} \\ \Gamma; [\top/\nu]r \vdash e_1 :: T \qquad \Gamma; [\bot/\nu]r \vdash e_2 :: T\end{array}}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: T}$$

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

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

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

**Figure 8.** Static semantics of $\mathcal{BL}$: well-formedness, subtyping, and type-checking.

```
module Tagged where

private ctx: U -- Current context

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

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

bind: ∀α β . ∀⊖<p:  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>

print: ∀α . ∀⊖<p:  U → Bool> .
  O → u: Tagged {U | p ν}<p> → x: Tagged α <p> → O
```

**Figure 9.** The `Tagged` monad. $U$ denotes the type of principals; $O$ denotes the type of observations.

```
bindBool: ∀β . ∀⊖<p:  U → Bool> .
  ∀<f:  Bool → β → Bool> . ∀<c:  Bool> .
    x:  Tagged {Bool  |  ν ⇒ c} <λu . p u ∧ c>
    → (u:  {Bool | ν ⇒ c} → Tagged {β | f u ν} <p>)
    → Tagged {β | f (val x) ν} <p>
```

**Figure 10.** The type signature of `bindBool`.

understood as substituting the lambda-term $p$ for every occurrence of $\pi$ in $S$ and then "beta-reducing" the result using the actual arguments of $\pi$ (see [49] 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 by liquid type inference (see Sec. 5).

### 4.3 Encoding Information Flow in $\mathcal{BL}$

Now that we have data types, we can track information flow by wrapping sensitive values inside a data type `Tagged` $\alpha$ $\langle \lambda u.r \rangle$. The predicate parameter $\langle \lambda u.r \rangle$, which we refer to as *policy*, encodes which principals are allowed to see the wrapped value. We show the type of the corresponding data constructor together with the basic monadic operations in Fig. 9. With these functions, we can rely on the type checking from Fig. 8 to propagate policies through all computations involving sensitive values and to reject programs that call sink functions with arguments whose policies are too restrictive.

It is important that the policy parameter of the `Tagged` constructor is contravariant, since a value with a less restrictive tag (*i.e.* visible to more users, more public) should be allowed to flow into a variable with a more restrictive tag (more secret) and not the other way around. In addition, 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 Fine's [47] technique of using private data constructors.

***Manipulating tagged values.*** Policy-agnostic code manipulates tagged values using the monadic `return` and `bind` shown in Fig. 9. Their implementations are the same as for the identity monad and are not shown on the figure, while their type signatures ensure proper propagation of tags. In particular, 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 at sinks.*** For the sake of simplicity we define a single sink function called **print** as part of the Tagged module that enables output of a value tagged with a policy $p$ to a user $u$, as long as $u$ satisfies $p$. To simplify formalization of noninterference, we parameterize the semantics of $\mathcal{BL}$ by the context, *i.e.* the principal who is observing the execution. More concretely, we assume that the environment always contains a variable ctx : $U$, and when a $\mathcal{BL}$ program is executed, it is executed with all possible values of ctx at the same time. This allows us to define **print** as follows:

```
print = λo . λu . λx .
  match u with Tagged u' →
    if u' ≠ ctx then o
      else match x with Tagged x' → o ++ show x'
```

The notion of context allows us to define a concept of *contextual noninterference*: informally, if a program type-checks with a given context, then substituting different values for tagged variables that are not visible to the context will not influence the final result of the execution.

***Relaxing requirements on bind.*** While the signature of **bind** is safe, it 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 execute conditional checks that depend on sensitive values. We might want to, for instance, show results only to viewers who are in a sensitive authors list. We may still want the program to define behavior for when the check fails, but with our existing **bind** we are not able to perform checks on sensitive values that may fail.

In order to allow conditional checks on sensitive values that may fail, we provide a separate function bindBool (shown in Fig. 10), where the first argument is a (tagged) Boolean. According to the type of bindBool, the first argument $x$ is allowed to carry a policy with an additional conjunct $c$. This is allowed because whenever $c$ is violated, $x$ must be false (that is, any insufficiently exported $x$ has the same value). We are able to make a special case for binding Boolean values: liquid type checking supports this construct for Booleans, as it is possible to use predicate abstraction for automatic type inference in the case of Booleans.

### 4.4 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 [40]: 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 **bind**, **return**, and **print** yielding the property that if a program type-checks with Tagged$^2$, then it preserves contextual noninterference with Tagged.

***The Tagged$^2$ monad.*** 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 observer ctx. Mirroring what we want for our noninterference property, the two versions are only allowed to differ for those sensitive values that are *not visible* to ctx. The Tagged$^2$ constructor accepts two $\alpha$ values, l and r, which we call *projections*. Its third argument prop serves as a proof of the property $p$ ctx $\Rightarrow l = r$, that is, if the policy holds of the current observer, the two projections must be equal.

```
module Tagged^2 where

private ctx: U -- Current context

-- | Tagged data constructor
private Tagged^2: ∀α . ∀⊖<p: U → Bool> .
  l:α → r:α → prop: ({() | p ctx} → {() | l = r})
  → Tagged α <p>

return^2: ∀α . ∀⊖<p: U → Bool> . α → Tagged α <p>
return^2 = λx . Tagged^2 x x id

bind^2: ∀α β . ∀⊖<p: U → Bool> . ∀<f:α → β → Bool> .
  x: Tagged α <p> → (y: α → Tagged {β | f y ν} <p>)
  → Tagged {β | f (l x) ν} <p>
bind^2 = λx . λg . match x with Tagged^2 xl xr _ →
  match g xl with Tagged^2 yl _ _ →
    match g xr with Tagged^2 _ yr _ → Tagged^2 yl yr id

print^2: ∀α . ∀⊖<p: U → Bool> .
  O → u: Tagged {U | p ν}<p> → x: Tagged α <p> → O
print^2 = λo . λu . λx .
  match u with Tagged^2 ul ur _ →
    if ul ≠ ctx ∧ ur ≠ ctx then o
      else if ul ≠ ur then fail
        else match x with Tagged^2 xl xr _ →
          if xl ≠ xr then fail else o ++ show xl
```

**Figure 11.** The Tagged$^2$ monad, which keeps track of two projections.

A Tagged$^2$ value with different projections corresponds to Pottier and Simonet's "bracket value" in [40], 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 the implementation of the Tagged$^2$ in Fig. 11. The phantom encoding provides alternative implementations of the primitive policy combinators: **return**$^2$ gives the same value for both projections, while **bind**$^2$ applies the function projection-wise. The $\mathcal{BL}$ type checker can easily show both implementations type-safe.

The new module also provides a new implementation for **print** that is designed to fail when it detects interference. This is not a function designed for printing to allowed users, but for checking values across multiple executions. The main idea is that **print**$^2$ fails whenever the observer could notice a difference between the two executions, either because 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** is an untypable term, so the only way to type-check **print**$^2$ is to prove that both failing branches are unreachable, which $\mathcal{BL}$ successfully accomplishes. To understand why the first failing branch is unreachable, recall that from the type of u we know that p ul $\wedge$ p ur; we also know that

$\texttt{ul} = \texttt{ctx} \lor \texttt{ur} = \texttt{ctx}$ from the path condition, thus $\texttt{p ctx}$ holds, which gives $\texttt{ul} = \texttt{ur}$ guaranteed by the $\texttt{Tagged}^2$ constructor.

***Contextual noninterference.*** We now show that type-checking with $\texttt{Tagged}^2$ implies contextual noninterference with `Tagged`. Because the $\texttt{Tagged}^2$ functions type-check and because the type system of $\mathcal{BL}$ is sound [49], we know that no type-correct program that manipulates $\texttt{Tagged}^2$ values can go wrong, *i.e.* attempts to print the results of two executions that are different. Now we only have to formally connect computations with `Tagged` values and those with $\texttt{Tagged}^2$ values, and show how type safety of the latter implies noninterference for the former.

We first show that replacing a $\texttt{Tagged}^2$ 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 $\texttt{Tagged}^2\ x_l\ x_r\ \_$ in $e$ is replaced by $\texttt{Tagged}\ x_j$.

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

*Proof outline.* The only steps that are different in the evaluation of $e$ and its projections are those resulting from the bodies of `bind` and `print`. By inspection of $\texttt{bind}^2$ it is easy to see that it applies the function projection-wise, and thus preserves the property of the lemma. In case of $\texttt{print}^2$, since it does not fail, either it does not do any output, or the two projections are the same; in both cases, projections of its body will have the same behavior. □

**Theorem** (Contextual Noninterference)**.** Let $\Gamma; x : \texttt{Tagged}\ \alpha\ \langle p \rangle \vdash e :: O$, and $\neg(p\ \texttt{ctx})$. Let for $j \in \{l, r\}$, $\Gamma \vdash v_j :: \alpha$ and $[(\texttt{Tagged}\ v_j)/x]e \to^* o_j$. Then $o_l = o_r$.

*Proof outline.* Since $\neg(p\ \texttt{ctx})$, we know $\Gamma \vdash \texttt{Tagged}^2\ v_l\ v_r\ \texttt{id} :: \texttt{Tagged}\ \alpha\ \langle p \rangle$ for any $v_l, v_r$. Let $e^2$ be $[(\texttt{Tagged}^2\ v_l\ v_r\ \texttt{id})/x]e$; note that $\lfloor e^2 \rfloor_j = [(\texttt{Tagged}\ v_j)/x]e$. By inspection of typing rules of $\mathcal{BL}$, substitution of a subterm with the same type does not change the type of the term, so $\Gamma \vdash e^2 :: O$. By soundness of the type system, $e^2$ either diverges or reduces to a value $o$ of type $O$. Note that the execution of $e^2$ differs from the executions of either $[(\texttt{Tagged}\ v_j)/x]e$ only in the bodies of `bind` and `print` functions; since none of them introduces divergence, $e^2$ cannot diverge either. By Lemma 1, $\lfloor e^2 \rfloor_j \to^* \lfloor o \rfloor_j$, that is $o_j = \lfloor o \rfloor_j$, but $\lfloor o \rfloor_l = \lfloor o \rfloor_r$ since $o$ is a value and is not tagged. □

***A note on the proof technique.*** Being able to express tagged values as a data type with a phantom predicate parameter is not only simpler, but also allows us to prove non-interference over pairs of traces simply by grounding phantom predicates. In the information flow monad `Tagged`, policies are phantom predicates that do not appear in the arguments of data constructors. In $\texttt{Tagged}^2$, the predicates are no longer phantom, but appear negatively in the type of `prop`, consistent with its variance annotation. Using these predicates for explicitly relating multiple program executions helps simplify the formalization and proof of non-interference.

# 5. 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 $\Gamma$ 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 sub-expression of $e$ and needs to be enhanced by a conditional check.

**Algorithm 1** Repair

```
 1: $\textsc{Repair}(\Gamma, e, T)$
 2:     $\mathit{leaks} \leftarrow \textsc{Verify}(\Gamma, e, T)$
 3:     for $(x, T') \leftarrow \mathit{leaks}$ do
 4:         $e \leftarrow \textsc{Fix}(\Gamma, x, T', e)$
 5:     $\mathit{leaks}' \leftarrow \textsc{Verify}(\Gamma, e, T)$
 6:     if $\mathit{leaks}' = []$ then return $e$
 7:     else fail

 8: $\textsc{Fix}(\Gamma, x, T, \texttt{let}\ x = f\ \bar{v}\ \texttt{in}\ e)$
 9:     $\psi \leftarrow \textsc{Abduce}(\Gamma; \psi \vdash f\ \bar{v} :: T)$
10:     $c \leftarrow \textsc{Synthesize}(\Gamma' \vdash c :: \{\texttt{Bool} \mid \nu \Leftrightarrow \psi\})$
11:     $c' \leftarrow \textsc{Lift}(\Gamma, c)$
12:     return $\texttt{let}\ x = \texttt{bindBool}\ c'\ (\lambda c.\texttt{if}\ c\ \texttt{then}\ f\ \bar{v}\ \texttt{else}\ f_{def})\ \texttt{in}\ e$
13: $\textsc{Fix}(\Gamma, x, T, e)$
14:     recursively call $\textsc{Fix}$ on subterms of $e$
```

***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.

## 5.1 Verification and Error Localization

The LIFTY compiler uses a variation of the liquid type inference [41] with predicate polymorphism [49] 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. 8). 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 [19], 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, $\texttt{print}\ w\ u\ x$ where $u : \{\texttt{User} \mid \nu = \texttt{sessionUser}\ w\}$ from our first introductory example (Fig. 2) produces (among others) a Horn clause labeled with $u$:

$$u\colon \nu = \texttt{sessionUser}\ w \Rightarrow P_1$$

where $P_1$ is the (as yet unknown) policy parameter of this `print`; this clause corresponds to the precondition on $u$ that it satisfy the policy. All Horn clauses generated by the type checker are either *definite clauses* of the form $\psi \land \bar{P} \Rightarrow P$ (like the one above) or *goal clauses* of the form $\psi \land \bar{P} \Rightarrow \phi$, where $\phi$ 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 `authors` to rest of the `Tagged` computation produces the constraint `authors`: $P_0 \Rightarrow \nu = \texttt{chair}\ w$ (where $P_0$ is the policy parameter of the corresponding **`bind`**). 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 $P_0 \mapsto \nu = \texttt{sessionUser}\ w$ 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 `authors`, whose goal clause does not hold for the inferred solution to $P_0$.

### 5.2 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:

$$\Gamma \vdash \texttt{bindBool}\ (\texttt{??})\ (\lambda c.\textbf{if}\ c\ \textbf{then}\ f\ \bar{v}\ \textbf{else}\ f_{def}) :: T$$

Here $f_{def}$ is the user-defined default alternative for the source $f$: we require that for every component $f : \bar{U} \to \texttt{Tagged}\ T\langle p\rangle$, 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 [39], but with a key modification. While off-the-shelf Synquid can solve our problem in principle, the monadic code Lifty needs to synthesize is sub-optimal for Synquid's approach to specification decomposition. 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 $\psi$ that would make the first branch of the conditional above type check (line 9). Like type-checking, condition abduction relies on predicate abstraction, but uses the *greatest-fixpoint* algorithm instead of the least. This is necessary for obtaining the weakest precondition instead of the strongest, which would be $\bot$. This allows Lifty programs to retain the original functionality (*i.e.* get the real sensitive value) in as many executions as possible. There might be no unique conjunctive solution $\psi$: abduction may return multiple solutions, which we treat as a disjunction. If the weakest $\psi$ the solver can construct out of given qualifiers is $\bot$, 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 $\Gamma'$, 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.

### 5.3 Implementation

We have implemented Lifty in Haskell, using the same minimal Haskell dialect as Synquid and using infrastructure provided by the Synquid synthesizer [39]. 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 [15] 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.

## 6. Lifty Gallery

We now show that Lifty is able to handle cases useful for real-world programming. We use examples that build on the conference management example we introduce in Sec. 2.

***Policies that depend on sensitive values.*** A tricky corner case of policy enforcement occurs when policies may depend on sensitive values. For example, it makes sense to show the list of authors to any of the paper's other authors. This policy is self-referential: the policy depends on the sensitive value that it protects. Despite the cyclic reasoning, there is a clear solution: since authors are allowed to see the author list, they can also see that they are *on* the author list, so it is safe to display the list to them. Other users, who are not allowed to see the list, may be able to infer from observing the default value that they are not on the list — but no additional information about the list's contents.

Lifty is able to perform verification and repair in the presence of self-referential policies and policies that depend on other sensitive values, in a manner consistent with our definition of safety in Sec. 4.4. We show the implementation of the author list policy in Fig. 12, and apply it to the same `showPaper` routine from Fig. 2. The repaired code now contains two cases where the original term is used: when the current user is chair and when the user is in the list of authors. We can see this in the disjunction in line 12. In this code, `paperAuthors` is the logical counterpart of `getPaperAuthors`.

Note that policies that depend on sensitive values are not in scope for label-based approaches [5, 6, 8, 12, 30, 35] because these approaches trust the programmer to correctly encode policies in terms of labels. With these approaches, the programmer is responsible for correctly managing these dependencies. And while these policies arise quite frequently in our survey of real-world security policies, other verification-based approaches for security [10, 11, 47, 48] do not address such policies.

***Output to multiple users.*** In our Sec. 2 we said that the viewer is usually the session user, but it may also be a different user. For instance, a user may initiate a transaction that sends email to one or

```
getPaperAuthors ::
    w: World → PaperId
       → Tagged (List User) <ν ∈ paperAuthors w pid ∨
                              ν = chair w>

showPaper w u pid =
  let out = do
     title ← getPaperTitle w pid
     t1 ← getChair w
     t2 ← u
     t3 ← getPaperAuthors w pid
     authors ← if t2 = t1 ∨ t2 ∈ t3 then
                  getPaperAuthors w pid
               else defaultPaperAuthors
     return (title ++ ":␣" ++ show authors)
  in print w u out
```

**Figure 12.** A policy that itself depends on sensitive values, and injected code when applied to the example in Fig. 2.

```
printMany ::
  w: World → Tagged [{User | P ν}] <P>
          → Tagged a <P> → World

getPaperStatus ::
  w: World → pid: PaperId
          → Tagged Int <ν in paperAuthors w pid ∧
                        currentPhase w = Done>

notifyAuthors w pid =
  let status = do
    t1 ← getCurrentPhase w
    if t1 = Done then getPaperStatus w pid
                 else defaultPaperStatus
  let authors = getPaperAuthors w pid
  printMany w authors status
```

**Figure 13.** A program that produces output to multiple users.

more other users (e.g. all reviewers, all co-authors, etc.). Because of how LIFTY propagates information about both the policies and the viewer, it is able to(1) generate customized checks based on the identity of the viewer, (2) generate appropriate policies even when viewers may be sensitive values, (3) determine a sufficiently strong policy when outputting to multiple viewers at once.

We now show how LIFTY ensures that when there are multiple viewers for the result of a single transaction, the system shows a version of the result that each viewer is allowed to see. Note that the list of recipients is itself sensitive and LIFTY's verification algorithm ensure that the checks respect the policy on the list. In Fig. 13 we show a transaction that sends messages to authors of a paper, notifying them of the paper's status as decided by the committee. The policy on the status is that it should be hidden until the conference phase is `Done` (reviews have been finalized), and then visible only to authors. LIFTY inserts a check for the phase being `Done`. LIFTY is able to infers that a check for the viewer being an author is not necessary, since the list of viewers is exactly the list of authors. It also verifies that submitting a message to the authors does not leak sensitive information about the authors, since all the recipients have sufficient privilege to see the list.

***Policy-generic functions.*** We now show that by following the functional programming idiom of using higher-order functions that

```
sortM ::
  (a → a → Tagged Bool <P>) → [a] → Tagged [a] <P>

getPaperScore :: w: World → pid: PaperId
   → Tagged Int <ν ∉ paperConflicts w pid>

sortPapersByScore w =
  let u = getCurrentUser w
  let cmpScore pid1 pid2 = do
        t1 ← u
        t2 ← getPaperConflicts w pid1
        s1 ← if t1 ∉ t2 then getPaperScore w pid1
                        else defaultPaperScore
        t3 ← getPaperConflicts w pid2
        s2 ← if t1 ∉ t3 then getPaperScore w pid2
                        else defaultPaperScore
        return s1 ≤ s2
      out = do pids ← getAllPaperIds w
               sortM cmpScore pids
  in print w u out
```

**Figure 14.** A function that sorts papers by their score. Some scores may be hidden from the current viewer due to conflicts.

are generic with respect to computations, we can write LIFTY functions that are generic with respect to the policies. While higher-order reasoning is often out of reach of first-order reasoning techniques, LIFTY's type system does the heavy lifting and thus LIFTY is able to support an idiom where programmers write policy-generic code by annotating functor arguments.

The code in Fig. 14 shows the repaired result of code for sorting a list of paper IDs using a comparator `cmpScore`. Instead of implementing this functionality with a specialized sorting function customized to the policy on the score, we use the `sortM` function, which is generic with respect to both the comparison function and the policy on that function. Since `cmpScore` consumes the sensitive paper score, its result is also sensitive, so its type is `PaperId → PaperId → Tagged Bool <..>`. LIFTY infers the predicate from the context, causing LIFTY to insert check code into this function. Notice that since LIFTY injects the check around the sensitive access inside `cmpScore`; the code for `sortM` need not change. Notice also that the programmer does not need to annotate `cmpScore`. As with all other policies, the programmer only needs to provide a type annotation on the accessor `getPaperScore`.

## 7. Evaluation

We implemented a set of micro-benchmarks and a larger conference management system example, 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

| | Compilation time | | | |
|---|---|---|---|---|
| Benchmark | Verify | Repair | Recheck | Total |
| Basic policy (Fig. 3) | 0.03s | 0.18s | 0.19s | 0.41s |
| Self-referencing policy (Fig. 12) | 0.05s | 0.29s | 0.52s | 0.88s |
| Implicit flow (Fig. 6) | 0.07s | 0.37s | 0.70s | 1.15s |
| Filter by author | 1.28s | 1.05s | 3.32s | 5.65s |
| Sort by score (Fig. 14) | 0.22s | 2.53s | 1.27s | 4.03s |
| Send to multiple users (Fig. 13) | 0.02s | 0.41s | 0.64s | 1.08s |

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

Policy size (tokens): 105

| | Program size (tokens) | | | Time | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Security checks | | Manual | Auto | | | |
| Benchmark | Original | Manual | Auto | Verify | Verify | Repair | Recheck | Total |
| Send paper status to authors | 53 | 18 | 19 | 2.15s | 0.36s | 0.51s | 1.91s | 2.79s |
| Send paper status to PC chair | 49 | 0 | 0 | 0.37s | 0.39s | 0.00s | 0.00s | 0.39s |
| Display paper authors to user | 59 | 16 | 26 | 2.61s | 0.50s | 0.59s | 4.00s | 5.10s |
| Display paper status to user | 23 | 50 | 71 | 5.23s | 0.10s | 1.86s | 8.45s | 10.41s |
| Display status and session number | 65 | 24 | 71 | 3.56s | 0.54s | 1.87s | 9.19s | 11.61s |
| Display title and list of conflicts | 37 | 30 | 56 | 3.60s | 0.19s | 13.51s | 7.31s | 21.02s |
| (auxiliary function for next three benchmarks) | 51 | 40 | 54 | 7.97s | 0.69s | 6.91s | 11.67s | 19.28s |
| Display information about a list of papers | 27 | 0 | 0 | 0.07s | 0.09s | 0.00s | 0.00s | 0.09s |
| Display information about all papers | 30 | 0 | 0 | 0.06s | 0.08s | 0.00s | 0.00s | 0.08s |
| Display all papers belonging to session user | 85 | 29 | 47 | 6.42s | 4.34s | 1.84s | 10.60s | 16.79s |
| Display a list of all unconflicted papers | 51 | 34 | 28 | 6.38s | 2.47s | 5.58s | 7.39s | 15.45s |
| Bid on a paper for the review phase | 55 | 38 | 50 | 4.83s | 0.57s | 9.44s | 7.26s | 17.28s |
| Send decisions to all authors of papers | 105 | 19 | 19 | 2.48s | 0.92s | 0.57s | 2.80s | 4.30s |
| Totals | 727 | 261 | 488 | 45.78s | 11.47s | 43.96s | 75.29s | 130.74s |

**Table 2.** Case study: conference management system.

conference management system (391 lines of Lifty) in a little over two minutes.

### 7.1 Overview of Case Study

We implemented the following code using Lifty.

***Micro-benchmarks*** We implemented the following representative micro-benchmarks based on the examples we have shown so far:(1) policies that depend on sensitive values, (2) implicit flow, (3) higher-order functions that compute on sensitive values, and (4) functionality that outputs to multiple users. For these examples we implement information flow policies as described and rely on Lifty to insert policy checks into the programs.

***Conference management system*** We implemented a basic conference management system, using Lifty to implement all information policy checks. 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.

For the rest of this section, we break down the features of the system into *transactions*, which are different queries that the user can issue. These were implemented as a set of Lifty functions, and the underlying implementation of the accessors to the database was implemented in Haskell. On top of that, some non-security-critical UI code was also implemented in Haskell, but without being allowed to access the database directly — only invoke the transactions. The system contains 756 lines of code in total (391 Lifty + 365 Haskell) and provides a superset of the functionality shown in our micro-benchmarks. Essentially, our conference management is a superset of our micro-benchmarks, but it also exposes some cross-dependencies between software features. We show information about which transactions we implemented in Tab. 2.

### 7.2 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. 2. 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. 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 t₁ e₁ (ifM t₂ e₁ e₂)` instead of `ifM (liftM or t₁ t₂) e₁ e₂`, essentially duplicating `e₁` 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.

As an anecdote, the human programmer was, at times, more conservative than the automatic tool, in ways that led to unnecessarily restricting application functionality. In "Display status and session number", for instance, the human guarded the sensitive access with a check that the conference phase is `Done`, making it so that even the conference chair will not see the status in an earlier phase, although this flow is in fact permitted by the policy. LIFTY was able to come up with a check that correctly handles this case.

### 7.3 Performance Statistics

We show the performance of the LIFTY compiler for the micro-benchmarks, as well as for the conference system, in Tabs. 1 and 2. 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 three of our benchmarks required no checks at all: one because the information is sent to the chair, who has sufficient privilege as it is and two others because all the checks are already inserted in a subroutine they depend on. We show that LIFTY is able to handle all checks for the conference management system in a little over two minutes.

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. (The transaction "Send paper status to authors" has the same functionality as the micro-benchmark "Self referencing-policy (12)", but takes longer to compile due to having more policy type declarations and global functions visible to the synthesizer.) Other than the 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.

## 8. Related Work

Our work builds on ideas from information flow security, program synthesis, and program repair to develop the first technique for repairing programs to adhere to information flow policies.

While LIFTY's policy language and guarantees build on work in language-based information flow [43], the programming model that LIFTY supports differs from that of most prior work [5, 8, 10, 13, 16, 27, 30, 40, 42, 48, 54] in the following key way. Prior approaches detect leaks, through either static or dynamic analysis, in programs written in programming models that are not security-aware. In order to implement programs that do more than raise errors (either at compile-time or runtime) or silently fail, the programmer needs to implement the policy checks and filters correctly across the program. In contrast, our solution is to use a security-aware, *policy-agnostic* [7, 52] programming model.

LIFTY supports the first static solution for policy-agnostic programming. While it is relatively straightforward to factor out access control checks [18, 34, 36], addressing the implicit and indirect flows involved with information flow security requires deeper integration with the language semantics. The Jeeves language [7, 52] and Jacqueline web framework [53] support a programming model where the programmer implements information flow policies as program functions and runtime performs *faceted execution* [6], 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, but additionally removes unnecessary runtime overheads and makes runtime behavior explicit.

LIFTY's verification algorithm differs from prior work in that(1) it allows the programmer to implement policies using expressive predicates and (2) provides static guarantees even when policies depend on sensitive values. Rather than having to encode permissions as labels, as with decentralized information flow control [5, 6, 8, 12, 30, 35], the LIFTY programmer provides higher-level predicates. While a label-based system trusts the programmer to correctly assign labels, a predicate-based approach ensures that permissions are assigned correctly. Fine [10] and F$^*$ [48] similarly encode information flow policies as dependent types, but, as opposed to LIFTY's type-checking, verification is undecidable.

As mentioned throughout, LIFTY relies on capabilities provided by liquid type inference [41, 49–51] for verification and error localization. The localization problem we solve is easier than that of Haskell type error localization tools such as SHErrLoc [55], since it is meant for consumption by our synthesis algorithm rather than by a human developer.

We build on prior work in program synthesis to perform program repair. LIFTY's repair technique uses abduction technique of the SYNQUID tool [39] for type-based program synthesis. SYNQUID, like other prior approaches for program synthesis [2–4, 17, 21, 24, 28, 33, 37, 45], solves synthesis problems (1) based on full functional specifications and (2) for synthesizing self-contained pieces of functionality. Our work is the first to extend these techniques for program repair, based on specifications that are not fully functional (but rather for a cross-cutting concern), and for functionality that is not self-contained.

Our repair solution differs from general repair techniques 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. There is a body of prior work in unsound program repair that is unsound and not based on logical specification [14, 25, 29, 31, 32, 38, 44]. Kneuss *et al.* [26] provide a sound program repair solution, but it requires full functional specifications. Because our approach is type-based, it is also able to localize errors better.

While there has also been prior work on rewriting programs specifically based on security concerns, LIFTY's policies are more expressive and the analysis is deeply integrated with the program semantics. The SWIM tool [23] performs automatic instrumentation to insert label-manipulation code into programs. *Policy weaving* [22] rewrites programs to adhere to stateful access control policies (*e.g.* "an application may not send a package after reading from history or file system"). FIXMEUP [46] repairs access control checks, but does not detect or repair information flow checks. Because LIFTY changes the underlying programming model, it is also able to do more than halt or fail silently in cases where sensitive values may not be shown.

## 9. 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] S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *CSF*, 2016.

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

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

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

[5] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Symposium on Security and Privacy*, SP, 2012.

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

[7] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security - PLAS '13*, 2013.

[8] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *European Symposium on Programming, ESOP*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[9] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. *CoRR*, abs/1507.06189, 2015.

[10] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Conference on Programming Language Design and Implementation*, PLDI, 2010.

[11] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, 2010.

[12] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Symposium on USENIX Security*, SS'07, 2007.

[13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Conference on Programming Language Design and Implementation*, PLDI, 2009.

[14] Z. Coker, D. Garlan, and C. Le Goues. Sass: Self-adaptation using stochastic search. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, Piscataway, NJ, USA, 2015. IEEE Press.

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

[16] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), 1977.

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

[18] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering*, ICSE '05. ACM, 2005.

[19] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, 2001.

[20] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, New York, NY, USA, 1993. ACM.

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

[22] 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 *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012.

[23] W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, New York, NY, USA, 2010. ACM.

[24] 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.

[25] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing Programs with Semantic Code Search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, November 2015.

[26] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*. Springer, 2015.

[27] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, 2007. ACM.

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

[29] 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.

[30] 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 *Symposium on Operating Systems Principles*, SOSP. ACM, 2009.

[31] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, 2015. ACM.

[32] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1), Jan. 2016.

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

[34] A. Milicevic, D. Jackson, M. Gligoric, and D. Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!, 2013.

[35] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, POPL, 1999.

[36] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Symposium on the Foundations of Software Engineering*, SIGSOFT/FSE '12. ACM, 2012.

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

[38] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, 2009. ACM.

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

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

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

[42] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Conference on Programming Language Design and Implementation*, PLDI.

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

[44] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.*, 50(6), June 2015.

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

[46] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[47] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, 2010.

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

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

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

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

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

[53] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, New York, NY, USA, 2016. ACM.

[54] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. *ACM Symposium on Operating Systems Principles*, 2009.

[55] 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.