Title
Type-Driven Repair for Information Flow Security

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

Authors
Polikarpova, N
Yang, J
Itzhaky, S
et al.

Publication Date
2017-09-12

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  
Massachusetts Institute of Technology  
schachi,asolar@csail.mit.edu

Polikarpova, Yang, and Itzhaky

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 [49] 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.

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 \cite{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 \cite{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 \cite{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. 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.

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 alternately 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” \cite{2} 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 [User] 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] \(\lambda\nu.\nu = \text{chair} w\). This policy says that a

\[\text{showPaper } w \text{ pid } = \text{let } u = \text{getCurrentUser } w \]
\[\text{out } = \text{do } \text{title } \leftarrow \text{getPaperTitle } w \text{ pid} \]
\[\text{authors } \leftarrow \text{getPaperAuthors } w \text{ pid} \]
\[\text{return } \text{(title ++ \\
\]
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 $\text{msg}$, 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 $\text{print } w u x$, 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 $\{\text{User} | P \nu\}$ 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.\textsuperscript{[5]} 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 [15, 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 $\nu = \text{chair } w$ flows into the argument out, which is required to have policy $\nu = u$ (or weaker). Since $\nu = u \not\Rightarrow \nu = \text{chair } w$, 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 [\text{"??"}]. 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

---

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

```
createdAtUser :: World → Tagged User <True>
getPaperTitle ::
  w:World → PaperId → Tagged String <True>
getPaperAuthors ::
  w:World → PaperId → Tagged [User] <\nu = \text{chair } w>
```

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

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

**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
        title ← getPaperTitle w pid
        t₁ ← getChair w
        t₂ ← u
        authors ← if t₁ = t₂ then getPaperAuthors w pid
        else return ["??"]
    in
      return (title ++ ";" ++ show authors)
      in print w u out
```

**Figure 6.** An implicit flow example and the code injected to overcome it.
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. We show the code for this, along with the repairs, in Fig. 2. Because LIFTY regards the condition $st = \text{Accepted}$ 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 $\text{print}$, 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 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 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 $\text{bind}$ and $\text{print}$. 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 $\text{getPaperAuthors}$ function has the return type $\text{Tagged \{} \text{User} \\mid \nu = \text{chair} \ \& \ \nu = \text{Chair} \text{\omega} \\}$, 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 $\text{print}$ 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.

Note also that there is a key difference between LIFTY’s use of predicate parameters and the standard use for abstracting over functional properties. With the standard usage, we can define a data type $\text{IntPair}$ with a constructor such as $\text{c :: } x \text{Int} \rightarrow y \text{(Int | P x y) \rightarrow IntPair \langle P \rangle}$, 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. 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 $\text{out :: Tagged String \ nu = w}$ for the program in Fig. 2 by collecting type constraints and solving them (as explained in Sec. 3.1). Through the application of monadic $\text{bind}$, LIFTY also infers the type authors :: $\text{Tagged \{User\} \ nu = w}$, at this point, LIFTY determines that $\text{getPaperAuthors w pid}$ will not type-check, since its expected type is $\text{Tagged \{User\} \ nu = w}$, while its actual type is $\text{Tagged \{User\} \ nu = \text{Chair} \ w}$, according to the type signature of $\text{getPaperAuthors}$ (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 $\text{Tagged} < P > < \text{Tagged} > Q > \iff Q \Rightarrow P$. 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 $\nu = u \nRightarrow \nu = \text{Chair}$, LIFTY will not only mark this as a type error, but it will also identify $\text{getPaperAuthors w pid}$ 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 $\text{getChair}$ and $\text{eq}$ to construct a program term equivalent to the logical formula $u = \text{Chair} w$. For efficiency purposes we do not use SYNQUID off-the-shelf; we explain our modifications in Sec. 3.

https://wiki.haskell.org/Phantom_type
4. Formal Semantics and Guarantees

We present the semantics and guarantees of LIFTY in two steps. First, we present the static semantics of L, a simple pure functional language that extends λP, 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 λP's decidable type-checking and automatic type inference.

Polymorphic data types allow us to encode tagging values with information flow policies directly in L, rather than extending the language. We first show how to implement tagging in L as the information flow monad Tagged. We then use a new proof technique we have developed to prove non-interference, introducing the Tagged monad that relates pairs of executions and showing that the type-checking with Tagged 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 L

We now present L. Like λP, L'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 λP [49]. L 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 L 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 L programs are in A-normal form [26]: application only appears in let-bindings and are built out of values, not arbitrary expressions (this is important for refinement type checking).

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

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

Types and Schemata. A L 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. Schemata are obtained by universally quantifying types over type and predicate variables. We explicitly label each quantification with its variance: covariant (□), contravariant (◇), or invariant (○). □ is the default variance and may be omitted.

5. Core Language of Abstract Policies

In Fig. 8 we show the relevant subset of well-formedness, subtyping, and type checking rules for L. These rules deviate from the standard semantics in the way we track variances of type and predicate parameters of polymorphic schemata; 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 λP are standard, they are important for deriving our safety property.

In our semantics, a typing environment Γ maps variables to types (x : S), bound type variables to their variances (α : □), and bound predicate variables to their types and variances (π : T(π)). 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 ∀α,π.∀T.∀S. In contrast, the operation of an appropriate type. The operation of an appropriate type.
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 BC

Now that we have data types, we can track information flow by wrapping sensitive values inside a data type Tagged $\alpha$ ($\lambda u.r$). The predicate parameter $\lambda u.r$, 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. 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

```
module Tagged where

private ct:x:U -- current context

private Tagged: \forall \alpha, \forall \beta: U \rightarrow Bool.
val: \alpha \rightarrow Tagged \alpha \beta

return: \forall \alpha, \forall \beta: U \rightarrow Bool. \alpha \rightarrow Tagged \alpha \beta

bind: \forall \alpha, \forall \beta: U \rightarrow Bool. \forall \alpha: \beta \rightarrow \beta \rightarrow Bool.
×: Tagged \alpha \beta \rightarrow (\alpha \rightarrow Tagged (\beta | \nu \rightarrow \nu) \beta) \rightarrow Tagged (\beta | \nu \rightarrow \nu) \beta

print: \forall \alpha, \forall \beta: U \rightarrow Bool.
0 \rightarrow u: Tagged (U | p r) \beta \rightarrow x: Tagged \alpha \beta \rightarrow 0
```

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

```
bindBool: \forall \beta, \forall \beta: U \rightarrow Bool.
∀ \beta: U \rightarrow Bool. \forall \beta: \beta \rightarrow Bool.
ν: Tagged (Bool | \nu \rightarrow \nu) \beta \rightarrow Tagged (\beta | \nu \rightarrow \nu) \beta

return: \forall \alpha, \forall \beta: U \rightarrow Bool.
\nu: Tagged (Bool | \nu \rightarrow \nu) \beta \rightarrow Tagged (\beta | \nu \rightarrow \nu) \beta
```

**Figure 10.** The type signature of bindBool.
a result that is at least as secret as either of them. The additional predicate parameter \( p \) of \( \text{bind} \) 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 \( \text{print} \) 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 \( \text{ctx} : U \), and when a \( \mathcal{BL} \) program is executed, it is executed with all possible values of \( \text{ctx} \) at the same time. This allows us to define \( \text{print} \) as follows:

\[
\begin{align*}
\text{print} &= \lambda o. \lambda u. \lambda x. \\
\text{match } u \text{ with Tagged } u' &\rightarrow \\
\text{ if } u' \neq \text{ctx} \text{ then } o &\text{ else match } x \text{ with Tagged } x' \rightarrow o ++ \text{ show } x'
\end{align*}
\]

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 \( \text{bind} \).** While the signature of \( \text{bind} \) 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 \( \text{bind} \) 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 \( \text{bindBool} \) (shown in Fig. 10), where the first argument is a (tagged) Boolean. According to the type of \( \text{bindBool} \), 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\(^2\) 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\(^2\) monad. We introduce the Tagged\(^2\) monad with new implementations of \( \text{bind}, \text{return}, \) and \( \text{print} \) 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 \( \text{ctx} \).
ul = ctx ∨ ur = ctx from the path condition, thus p ctx holds, which gives ul = ur guaranteed by the Tagged² constructor.

**Contextual noninterference.** We now show that type-checking with Tagged² implies contextual noninterference with Tagged. Because the Tagged² functions type-check and because the type system of BL is sound [49], we know that no type-correct program that manipulates Tagged² 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 Tagged² values, and show how type safety of the latter implies non-interference for the former.

We first show that replacing a Tagged² 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 \([e]_j\), for \( j = \{l, r\}\)) is an expression where every occurrence of Tagged² \( x_i \), \( x_r \) in \( e \) is replaced by Tagged \( x_j \).

**Lemma 1** (Projection). If \( e \rightarrow^* e' \) then \([e]_j \rightarrow^* [e']_j\), 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 bind² it is easy to see that it applies the function projection-wise, and thus preserves the property of the lemma. In case of print², 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 \( \Gamma; x : \text{Tagged} \; \alpha \; (p) \vdash e :: O \), and \( \rightarrow(p \; \text{ctx}) \). Let for \( j \in \{l, r\} \), \( \Gamma \vdash \nu_j :: \alpha \) and \([\text{Tagged} \; \nu_j]/x[e] \rightarrow o_j \). Then \( o_l = o_r \).

**Proof outline.** Since \( \rightarrow(p \; \text{ctx}) \), we know \( \Gamma \vdash \text{Tagged}^2 \; \nu_l \; \nu_r \; \nu_l : \text{id} :: \text{Tagged} \; \alpha \; (p) \) for any \( \nu_l \; \nu_r \). Let \( e'' \) be \([\text{Tagged}^2 \; \nu_l \; \nu_r \; \nu_l : \text{id}]/x[e]; \) note that \([e'']_j = ([\text{Tagged} \; \nu_j]/x[e]. By inspection of typing rules of BL, substitution of a subterm with the same type does not change the type of the term, so \( \Gamma \vdash e'' :: O \). By soundness of the type system, \( e'' \) either diverges or reduces to a value \( o \) of type \( O \). Note that the execution of \( e'' \) differs from the executions of either \([\text{Tagged} \; \nu_j]/x[e] \) only in the bodies of bind and print functions; since none of them introduces divergence, \( e'' \) cannot diverge either.

By **Lemma 1** \([e'']_j \rightarrow^* [a]_j\), that is \( o_j = [a]_j \), but \([a]_r = [a]_l \), 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 Tagged², the predicates are no longer phantom, but appear negatively in the type of prog, 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 the **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 \([\text{leaks}]\) 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.

**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 \( P \), 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 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, \( \text{print} \; w \; u \; x \) where \( u : \{\text{user} | \nu = \text{sessionuser} \; w\} \) from our first introductory example (Fig. 2) produces (among others) a Horn clause labeled with:

\[ u : \nu = \text{sessionuser} \; w \Rightarrow P_i \]

where \( P_i \) is the (as yet unknown) policy parameter of this \( \text{print} \); 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 \( \psi \land P \Rightarrow P \) (like the one above) or **goal clauses** of the form \( \psi \land P \Rightarrow \phi \), where \( \phi \) is a known formula. Whereas the Liquid Haskell type checker looks for the first offending term, we want all offending terms. Thus

<table>
<thead>
<tr>
<th>Algorithm 1 Repair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: REPAIR(( \Gamma, e, T ))</td>
</tr>
<tr>
<td>2: ( \text{leaks} \leftarrow \text{VERIFY}(\Gamma, e, T) )</td>
</tr>
<tr>
<td>3: for ( (x, T') \leftarrow \text{leaks} ) do</td>
</tr>
<tr>
<td>4: ( e \leftarrow \text{FIX}(\Gamma, x, T', e) )</td>
</tr>
<tr>
<td>5: ( \text{leaks'} \leftarrow \text{VERIFY}(\Gamma, e, T) )</td>
</tr>
<tr>
<td>6: if ( \text{leaks'} = [] ) then return ( e )</td>
</tr>
<tr>
<td>7: else fail</td>
</tr>
</tbody>
</table>

8. \( \text{FIX}(\Gamma, x, T, \text{let } x = f \; \text{in } e) \) |
9. \( \psi \leftarrow \text{ABDUCE}(\Gamma; \psi \; f : :: T) \) |
10. \( e \leftarrow \text{SYNTHESIZE}(\Gamma' \vdash c : \{\text{bool } | \nu \equiv \psi\}) \) |
11. \( c' \leftarrow \text{LIFT}(\Gamma, e) \) |
12. \( \text{return let } x = \text{bindBool } c' (\lambda c. \text{if } c \text{ then } f \; \text{else } f_{\text{adj}}) \; \text{in } e \) |
13. \( \text{FIX}(\Gamma, x, T, e) \) |
14. recursively call \( \text{FIX} \) on subterms of \( e \) |
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 \( T \) 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 a confidentiality policy: \( P_b \Rightarrow \nu = \text{chair} w \) (where \( P_b \) 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 \( P_b \Rightarrow \nu = \text{sessionUser} w \) 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 \( P_b \).

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 \( \text{let } x = f \bar{v}, \) which it has to replace with some \( \text{let } x = e'. \) Since we only need a specific kind of repair, finding \( e' \) reduces to solving the following local synthesis problem:

\[
\Gamma \vdash \text{bindBool (?t)} \ (\text{if } f \bar{v} \text{ then } e \text{ else } f_{\text{def}}) :: T
\]

Here \( f_{\text{def}} \) is the user-defined default alternative for the source \( f \); we require that for every component \( f : U \to \text{Tagged } T(p) \), the user designate, through a special annotation, a component \( f_{\text{def}} : \text{Tagged } T'(\top) \) 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 [19], 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 2). 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 abduce 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 \( \text{Boo}(\nu \leftrightarrow \psi) \); 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 [19]. 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 [3 6 8 12 20 33] 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][37][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
getPaperStatus w pid = getPaperScore w pid → do

  return 15

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 infer 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 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 ν → paperConflicts w pid. 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
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Manual</th>
<th>Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send paper status to authors</td>
<td>53</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>Send paper status to PC chair</td>
<td>49</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Display paper authors to user</td>
<td>59</td>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>Display paper status to user</td>
<td>23</td>
<td>50</td>
<td>71</td>
</tr>
<tr>
<td>Display status and session number</td>
<td>65</td>
<td>24</td>
<td>71</td>
</tr>
<tr>
<td>Display title and list of conflicts</td>
<td>37</td>
<td>30</td>
<td>56</td>
</tr>
<tr>
<td>(auxiliary function for next three benchmarks)</td>
<td>51</td>
<td>40</td>
<td>54</td>
</tr>
<tr>
<td>Display information about a list of papers</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Display information about all papers</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Display all papers belonging to session user</td>
<td>85</td>
<td>29</td>
<td>47</td>
</tr>
<tr>
<td>Display a list of all unconflicted papers</td>
<td>51</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>Bid on a paper for the review phase</td>
<td>55</td>
<td>38</td>
<td>50</td>
</tr>
<tr>
<td>Send decisions to all authors of papers</td>
<td>105</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Totals</td>
<td>727</td>
<td>261</td>
<td>488</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Manual</th>
<th>Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verify</td>
<td>2.15s</td>
<td>0.36s</td>
</tr>
<tr>
<td>Repair</td>
<td>0.39s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Recheck</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Total</td>
<td>2.79s</td>
<td>0.39s</td>
</tr>
</tbody>
</table>

Table 2. Case study: conference management system.
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 $\times 3$. 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 $\text{ifM} t_1 e_1 (\text{ifM} t_2 e_2) \text{ instead }$ of $\text{ifM} (\text{ifM} t_1 e_1 t_2 e_2) e_2$, essentially duplicating $e_1$ and causing some bloat. However, this affects only the size of the code and neither its functionality nor its performance. The manual and automatic checks were semantically equivalent across our benchmarks: the checks are not more conservative than needed.

As an anecdote, the human programmer was, at times, more conservatively 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”, but takes longer because of the extra code in scope.) As mentioned throughout, LIFTY relies on capabilities provided by LIFTY’s policy language and guarantees build on work from history or file system). FINE [10] and F∗ [38] 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 LIFTY’s type inference [11] [39] 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. Syndiquid, like other prior approaches for program synthesis [2–4, 17, 21, 23, 29, 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] rewrites 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


