Enforcing Declarative Policies with Targeted Program Synthesis

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

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

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

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

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

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

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

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

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

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

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

Type system. The main technical challenge in making targeted synthesis work is the design of a type system that, on
the one hand, is expressive enough to reason about the policies of interest, and on the other hand, produces appropriate type errors for the kinds of patches we want to synthesize.

For our policy language, we draw inspiration from the Jeeves language [6, 49], which supports rich, context-dependent policies, where the visibility of data might depend both on the identity of the viewer and the state of the system. For example, in a social network application, a user’s location can be designated as visible only to the user’s friends who are within a certain distance of that location. In Jeeves, these policies are expressed directly as predicates over users and states.

Our technical insight is that static reasoning about Jeeves-style policies can be encoded in a decidable refinement type system by indexing types with policy predicates. Moreover, we show how to instantiate the Liquid Types framework [36] to infer appropriate expected types at the error locations.

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

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

2 Lifty by Example

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

2.1 The EDAS Leak

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

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

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

Draft, 2017, USA

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

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

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

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

2.2 Programming with Lifty

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

The Lifty type system is equipped with a special type constructor \((T)^\pi\) (\(T\) tagged with policy \(\pi\)), where \(\pi : (\Sigma, User) \rightarrow \text{Bool}\) is a predicate on contexts, i.e. pairs of states and users. The type \((T)^\pi\) denotes values of type \(T\) that are only visible to a user \(u\) in a state \(s\) such that \(\pi (s, u)\) holds. For example, to express that a paper’s status is only visible when the conference phase is done, the programmer defines its type as a reference to Status tagged with policy \(\lambda (s, u).\text{phase} = \text{Done}\). Hereafter, we elide the binders \((s, u)\) from policy predicates for brevity, and simply write \(\lambda.\). The predicate \(\text{any} = \lambda.\text{True}\) annotates fields as public (i.e. visible in any context).

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

2.3 Targeted Program Synthesis

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

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

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

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

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

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

\[
\begin{align*}
B & \Rightarrow s[\text{phase}] = \text{Done} \quad (1) \\
P & \Rightarrow B \quad (2) \\
u = \text{client} \land s = \sigma & \Rightarrow P \quad (3)
\end{align*}
\]

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

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

2.3.2 Fault Localization

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

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

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

2.3.3 Patch Synthesis

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

\[
\Gamma \vdash ?? :: \langle\text{status}\rangle^\lambda.u = \text{client} \land s = \sigma
\]

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

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

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

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

\[
\begin{align*}
C & \land u = c_{\text{client}} \land s = \sigma \implies s[\text{phase}] = \text{Done}
\end{align*}
\]

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

2.4 Scaling Up to Real-World Policies

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

2.4.1 Multiple Leaks

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

2.4.2 Complex Policies

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

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

\[
\begin{align*}
\text{authors :: p : PaperId } & \rightarrow \text{Ref } ([\text{User}]^{\lambda . s}[\text{phase}] = \text{Done} \lor \text{u} \in \text{authors } p) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{false } \leftarrow \text{let } x0 = \text{get } (\text{authors } p) \text{ in do} \\
& \quad \text{c1 } \leftarrow \text{do } x1 \leftarrow \text{get phase}; \ x2 \leftarrow x0 \\
& \quad x1 = \text{Done} \lor \text{elem client } x2 \\
& \quad \text{if c1 then x0 else [ ]}
\end{align*}
\]

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

2.4.3 Nontrivial Patches

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

\[
\begin{align*}
\text{bid :: User } \rightarrow \text{Ref } ([\text{Maybe } (\text{Int})^{\lambda . s}[\text{phase}] = \text{Done}] \land \text{u} \neq \emptyset)
\end{align*}
\]

Draft, 2017, USA
With this definition, \textsc{lifty} generates the following patch inside a function \texttt{showBid} \texttt{client p}, which displays participant \( p \)'s bid to \texttt{client}:

\begin{verbatim}
1  b ← let x0 = get (bid p) in do
2  x1 ← get phase; x2 ← get (bid client); x3 ← x0
3  if x1 = Done
4    then x0
5  else if isJust x2
6    then mapJust (λ _ . 0) x3
7  else Nothing
\end{verbatim}

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

\[
\text{mapJust} :: (α → β) → \text{Maybe} α → \text{Maybe} β
\]

### 2.4.4 State Updates

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

### 3 The \( \lambda^L \) Type System

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

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

This section introduces the syntax of \( \lambda^L \) (Sec. 3.1) and its typing rules (Sec. 3.2), and shows that well-typed \( \lambda^L \) programs satisfy contextual non-interference (Theorem 3.2).

The runtime behavior of \( \lambda^L \) programs is straightforward; we provide an operational semantics in Appendix A.2.

#### 3.1 Syntax of \( \lambda^L \)

\( \lambda^L \) is a simple core language with references, extended with several information-flow specific constructs. We summarize the syntax of \( \lambda^L \) in Fig. 6.

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

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

Draft, 2017, USA

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

Other types of \(\lambda^L\) include primitive types, references, function types, and refinement types, which are standard \([26, 36]\).

In a refined base types \(\{B \mid r\}\), \(r\) is a refinement predicate over the program variables and a special value variable \(v\), which denotes the bound variable of the type.

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

### 3.2 Typing rules for \(\lambda^L\)

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

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

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

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

**Downgrading.** The safe downgrading construct, \([e]\), is a novel feature of \(\lambda^L\), which we introduced specifically to support static verification of programs with self-referential policies (Sec. 2.4.2). Informally, the idea is that we can we can safely downgrade a tagged term (i.e. weaken its policy), whenever we can prove that the term is constant, since constants cannot leak information. Whereas this property
is hard to check automatically in the general case, a special case where \( e \) is a tagged boolean turns out to be both amenable to automatic verification and particularly useful for self-referential policies. The rule \( T:\lambda \) allows tagging \( e \) with \( \lambda(s,u)p \) as long as there exists a refinement predicate \( r \) over program variables, such that \( e \) can be tagged with \( \lambda(s,u)p \wedge r \) and the value of \( e \) implies \( r \). This operation is safe because in any execution where \( r \) holds, the two policies are the same; while any execution where \( r \) does not hold, the value of \( e \) is guaranteed to be \text{false} (a constant).

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

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

### 3.3 Contextual Noninterference in \( \lambda^L \)

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

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

1. **ENFORCE \((\Gamma, e, T)\)**
2. \( \epsilon \leftarrow \text{LOCALIZE} \((\Gamma + \epsilon :: T)\) \)
3. **return Patch(\( \epsilon \))**
4. **Patch(let \( x = \langle T \epsilon \rangle d \) in \( e \))**
5. \( d' \leftarrow \text{Generate} \((x0 : T_d), \Gamma, T_e)\)
6. **return let \( x = d \) in \( d' \) in Patch(\( e \))**
7. **Patch(\( e \))**
8. recursively call Patch on subterms of \( e \)
9. **Generate \((\Gamma_B, \Gamma_G, \langle T \rangle)\)**
10. \( \Gamma_B \leftarrow \Gamma_B \cup \text{redaction functions for} \ T \)
11. \( \text{branches} \leftarrow \text{SYNTHESIZE} \((\Gamma_B + ?? :: \langle T \rangle)\) \)
12. \( \text{if} (\text{Check}(\Gamma + \text{defl} = \langle T \rangle) \text{ then} \)
13. \( \text{patch} \leftarrow \text{defl} \)
14. **else fail**
15. for \( b \leftarrow \text{guarded do} \)
16. \( \psi \leftarrow \text{ABDUCE} \((\Gamma_G, ?? + b :: \langle T \rangle)\) \)
17. \( T_g \leftarrow \langle \psi : \text{Bool} \mid \psi \Rightarrow \psi \rangle \)
18. \( \text{guard} \leftarrow \text{SYNTHESIZE} \((\Gamma + ?? :: T_g)\) \)
19. **patch \leftarrow \text{bind}(\text{guard}(\lambda g . \text{if} \ g \ \text{then} \ b \ \text{else patch}) \)
20. **return patch**

**Figure 8.** Policy enforcement algorithm

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

\[
\forall l. \text{ty}(l) = \text{Ref}(\langle T \rangle)^o \wedge (\pi[\sigma_1, o] \lor \pi[\sigma_2, o]) \Rightarrow \sigma_1[l] = \sigma_2[l]
\]

**Theorem 3.2** (contextual noninterference). Let \( s \) be a \( \lambda^L \) program and let \( \sigma_1, \sigma_2 \) be two stores. Assume that running \( s \) on \( \sigma_1, \sigma_2 \) produces outputs \( (o_{i,j}, v_{i,j}) \), for \( i \in 1..2, j \in 1..k \), where \( o_{i,j} \) is the viewer of output \( v_{i,j} \).

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

### 4 Targeted Synthesis for \( \lambda^L \)

We now turn to the heart of our system: the algorithm **ENFORCE** (shown in Fig. 8), which takes as input a type environment \( \Gamma \), a program \( e \) (in A-normal form), and a top-level type annotation \( T \), and determines whether policy-enforcing code can be injected into \( e \) to produce \( e' \), such that \( \Gamma \vdash e' :: T \). The algorithm proceeds in two steps. First, procedure **LOCALIZE** identifies unsafe terms (line 2), replacing them with type casts to produce a “program with holes” \( \hat{e} \) (Sec. 4.1). Second, procedure **PATCH** traverses \( \hat{e} \) (line 3) replacing each type cast with an appropriate patch, generated by the procedure **GENERATE** (Sec. 4.2).
4.1 Fault Localization

Type casts. For the purpose of fault localization, we extend the values of $\lambda^L$ with type casts:

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

Statically, our casts are similar to those in prior work [26]; in particular, the cast $\langle T < T' \rangle$ has type $T' \rightarrow T$. However, the dynamic semantics of casts in $\lambda^L$ is undefined. The idea is that casts are inserted solely for the purpose of targeting synthesis, and, if synthesis succeeds, are completely eliminated. We restrict the notion of type-safe $\lambda^L$ programs to those that are well-typed are free of type casts.

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

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

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

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

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

\[\text{In this context, the definition of a let-bound variable is considered a subterm of the let body}\]

4.2 Patch Generation

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

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

Synthesis of branches. In line 11, GENERATE uses SYNQUID to synthesize the set of all terms up to certain size with the right content type, but with no restriction on the policy (here, $\text{none} = \lambda . \text{false} \cdot \text{use}$). Note that branches are generated in a restricted environment $\Gamma_B$, which contains only the original faulty term, the “default” value of type $T$, and a small set of reduction functions for this type (such as $\text{mapJust}$ for $\text{Maybe}$ in Sec. 2.4.3). This restriction makes the branch synthesis both more predictable and more efficient.

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

Inferring the localization. Given an unsatisfiable system of Horn constraints, LOCALIZE first makes sure that all conflicting clauses have been generated by implication checks on policy predicate, and removes those clauses that were generated by the smallest term. It then re-runs the fixpoint solver [12, 36] on the remaining system, inferring strongest solutions for policy predicates, after which it reset the non-policy refinements of the removed terms to $\top$ and re-check the validity of the constraints. If the constraints are satisfied, we have obtained a sound and minimal localization (the expected types are the least restrictive because policies are strongest, and other refinements are $\top$). If the constraints are violated, it indicates that the program depends on some functional property of the unsafe term we want to replace. We consider such programs out of scope: if a programmer wants to benefit form automatic policy enforcement, they have to give up the ability to reason about functional properties of sensitive values, since our language reserves the right to substitute them with other values.
that the first branch can be used as the default branch, i.e. it satisfies the expected type conditionally. This property is always satisfied as long as $\Gamma_p$ contains a value $v$ of type $T$, since in our type system $T \ll (T)^\pi$ for any $\pi$. For this check, we use the original liquid type checking unmodified.

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

The main challenge of guard synthesis, however, is that the guard itself must be monadic, since it might need to retrieve and compute over some data from the store. Since the data it retrieves might itself be sensitive, we need to ensure that two conditions are satisfied (1) functional correctness: the guard returns a value equivalent to $\psi$, and (2) no leaky enforcement: the guard itself respects the expected policy $\pi$ of the patch. To ensure both conditions, we obtain the guard via type-driven synthesis, providing $\langle \{v : \text{Bool} \mid v \equiv \psi\}\rangle^\pi$ as the target type.

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

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

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

### 4.3 Guarantees and Limitations

In this section we summarize the soundness guarantee of targeted synthesis in $\lambda_b$ and then discuss the limitations on its completeness and minimality.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compilation time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Localize</td>
</tr>
<tr>
<td>Basic policy</td>
<td>0.04s</td>
</tr>
<tr>
<td>Self-referencing policy</td>
<td>0.00s</td>
</tr>
<tr>
<td>Implicit flow</td>
<td>0.01s</td>
</tr>
<tr>
<td>Filter by author</td>
<td>0.06s</td>
</tr>
<tr>
<td>Sort by score</td>
<td>0.03s</td>
</tr>
<tr>
<td>Send to multiple users</td>
<td>0.00s</td>
</tr>
<tr>
<td>Interleaved reads/writes</td>
<td>0.01s</td>
</tr>
<tr>
<td>Copy a private field</td>
<td>0.00s</td>
</tr>
</tbody>
</table>

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

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

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

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

### 5 Evaluation

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

**Expressiveness of policy language.** We use Lifty to implement a conference manager, a grading application, and a health portal. The two authors who developed the case study [46]. The health portal is based on the HealthWeb example from the Fine paper [41].
Enforcing Declarative Policies with Targeted Program Synthesis

Draft, 2017, USA

(a) Conference Management System

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program size (tokens)</th>
<th>Time</th>
<th>Policy size (tokens): 345</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Manual</td>
<td>Auto</td>
</tr>
<tr>
<td>Register user</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>View users</td>
<td>20</td>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>Paper submission</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Search papers</td>
<td>77</td>
<td>96</td>
<td>82</td>
</tr>
<tr>
<td>Show paper record</td>
<td>53</td>
<td>46</td>
<td>82</td>
</tr>
<tr>
<td>Show reviews for paper</td>
<td>57</td>
<td>54</td>
<td>45</td>
</tr>
<tr>
<td>User profile: GET</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>User profile: POST</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Submit review</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Assign reviewers</td>
<td>47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>432</td>
<td>205</td>
<td>233</td>
</tr>
</tbody>
</table>

(b) Gradr—Course Management and Interactive Grading System

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program size (tokens)</th>
<th>Time</th>
<th>Policy size (tokens): 141</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>(Lifty)</td>
<td>Localize</td>
</tr>
<tr>
<td>Display the home page</td>
<td>12</td>
<td>0</td>
<td>0.00s</td>
</tr>
<tr>
<td>Student: get classes</td>
<td>13</td>
<td>0</td>
<td>0.04s</td>
</tr>
<tr>
<td>Instructor: get classes</td>
<td>13</td>
<td>0</td>
<td>0.00s</td>
</tr>
<tr>
<td>Get class information for a user</td>
<td>29</td>
<td>0</td>
<td>0.01s</td>
</tr>
<tr>
<td>View a user’s profile (owner)</td>
<td>23</td>
<td>0</td>
<td>0.00s</td>
</tr>
<tr>
<td>View a user’s profile (any user)</td>
<td>25</td>
<td>19</td>
<td>0.01s</td>
</tr>
<tr>
<td>Instructor: view scores for an assignment</td>
<td>36</td>
<td>17</td>
<td>0.00s</td>
</tr>
<tr>
<td>Student: view all scores for user</td>
<td>36</td>
<td>17</td>
<td>0.00s</td>
</tr>
<tr>
<td>Instructor: view top scores for an assignment</td>
<td>62</td>
<td>17</td>
<td>0.02s</td>
</tr>
<tr>
<td>Totals</td>
<td>253</td>
<td>70</td>
<td>0.13s</td>
</tr>
</tbody>
</table>

(c) HealthWeb—Health Information Portal

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program size (tokens)</th>
<th>Time</th>
<th>Policy size (tokens): 194</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>(Lifty)</td>
<td>Localize</td>
</tr>
<tr>
<td>Search a record by id</td>
<td>23</td>
<td>293</td>
<td>0.00s</td>
</tr>
<tr>
<td>Search a record by patient</td>
<td>56</td>
<td>293</td>
<td>0.03s</td>
</tr>
<tr>
<td>Show authored records</td>
<td>45</td>
<td>0</td>
<td>0.02s</td>
</tr>
<tr>
<td>Update record</td>
<td>34</td>
<td>0</td>
<td>0.00s</td>
</tr>
<tr>
<td>List patients for a doctor</td>
<td>52</td>
<td>44</td>
<td>0.03s</td>
</tr>
<tr>
<td>Totals</td>
<td>216</td>
<td>630</td>
<td>0.10s</td>
</tr>
</tbody>
</table>

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

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

demonstrate that Lifty’s policy language supports the desired policies for these systems.

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

Quality of patches. We compare the code generated by Lifty to a version with manually written policy checks and show that not only does Lifty allow for policy descriptions to be centralized and concise, but also the compiler is able...
to recover all necessary checks, without reducing the functionality.

5.1 Microbenchmarks and Case Studies

We implemented the following code using Lifty.

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

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

Course manager. We implemented a system for sending grades to students based on their course enrollment and assignment status. An example policy is that a student can see their own scores, whereas instructors can see scores for all of their students.

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

5.2 Performance Statistics

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

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

5.3 Measuring the Quality of Patches

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

6 Related Work

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

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

Lifty takes a policy-agnostic approach [6, 48, 49] and factors information flow out from core program functionality, allowing programmers to implement policies as high-level predicates over the program state. Prior work uses runtime enforcement, which yields nontrivial runtime overheads and makes it difficult to reason about program behavior. Lifty addresses these issues by providing a synthesis-driven approach to support the Jeeves semantics statically, supporting an analogous security property.
Type systems. LIFTY’s type system, as well as the monadic encoding for information flow, are inspired by other value-dependent type systems [9, 10, 23, 41, 42]. The key difference is the decidability of both type inference, which yields automated verification and fault localization, crucial for targeted synthesis. LIFTY’s type inference engine is built on top of the Liquid Types framework [12, 36, 44–46], and extends it with a fault localization mechanism tailored towards security types. Our technique for using types for program rewriting resembles both hybrid type checking [26] and type-directed coercion insertion [43]. Both our types and rewriting capabilities are more advanced and thus can handle global, cross-cutting concerns such as information flow.

References

[1] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In CSF.
[33] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In PLDI.
[34] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In PLDI.
[37] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In PLDI.
[40] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In PLDI.


A.1 Desugaring do-notation

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

\[
\begin{align*}
&\text{showPaper client p =} \\
&\quad \text{let row =} \\
&\quad \quad \text{bind (get (title p)) (λ t .} \\
&\quad \quad \quad \quad \text{bind (get (status p)) (λ st .} \\
&\quad \quad \quad \quad \quad \text{bind (if st = Accepted} \\
&\quad \quad \quad \quad \quad \quad \quad \text{then (session p) else return ""}) (λ ses .} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{return (t + "\_" + ses)))} \\
&\quad \text{print client row}
\end{align*}
\]

A.2 Operational Semantics of \(\lambda^L\)

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

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

A.3 The \(\lambda^L\) Type System

We show the full typing rules in Fig. 11.

A.4 Contextual Noninterference with Store Updates

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

Definition A.1 (observational equivalence). For some observer \(o : \text{User}\) and a set of stores \(Δ = \{Δ_1, \cdots, Δ_n\}\), two stores \(σ_1, σ_2\) are \((o, Δ)-equivalent\) — written \(σ_1 \sim_{o, Δ} σ_2\) — if

\[
\forall l. \text{ty}(l) = \text{Ref } (T)^p \land (\bigvee_{Δ_i ∈ Δ} p(Δ_i, o)) \Rightarrow σ_1[l] = σ_2[l]
\]

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

When \(Δ\) is omitted, it means \(Δ = \{σ_1, σ_2\}\).

In order to discuss the privacy properties of the language, we need to add some annotation to program terms.
Expression Evaluation \( [\sigma, e \rightarrow e] \)

\[ e \beta \sigma, \lambda x : T. e_1 \rightarrow [x \mapsto e_2] e_1 \]

\[ e \text{-get} \sigma, \text{get} l \rightarrow \sigma[l] \]

\[ e \text{-true} \sigma, \text{if true then} e_1 \text{ else} e_2 \rightarrow e_1 \]

\[ e \text{-false} \sigma, \text{if false then} e_1 \text{ else} e_2 \rightarrow e_2 \]

\[ e \text{-bind} \sigma, \text{bind} e_x (\lambda x : T. e) \rightarrow [x \mapsto e_x] e \]

\[ e \text{-ctx} e \rightarrow e' \]

where \( C := \bullet | C e | e C | \lambda x : T. C | \text{get} C | \text{if} C \text{ then} e_1 \text{ else} e_2 | \text{bind} e C \)

Statement Execution \( [\sigma, s \rightarrow \sigma, s] \)

\[ \sigma, e \rightarrow^* v \]

\[ \sigma, \text{let} x = e \text{ in} S \rightarrow \sigma, [x \mapsto v] S \]

\[ \sigma, \text{set} l v ; S \rightarrow \sigma[l := v], S \]

\[ \sigma, \text{print} u v ; S \rightarrow \sigma[u + = v], S \]

Figure 10. \( \lambda^l \) operational semantics.

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

As in the \( \text{print} \) case, \( s'_j = t_j \) and the rest is the same.

**Lemma A.8** (contextual noninterference for “let” statements). Let \( \sigma_1 \sim_{o, \Lambda} \sigma_2 \),

\[ s_j = \text{let} x = e_j \text{ in} t_j \text{ for } j \in \{1, 2\}, \]

\[ \text{such that } s_1 \sim_{o, \Lambda} s_2 \text{ and } s_j, s_j \rightarrow s'_j ; s'_j \text{.} \]

Assume \( \sigma_{1, 2} ; \sigma'_{1, 2} \in \Delta \).

Then \( s'_1 \sim_{o, \Lambda} s'_2 \) and \( s'_1 \sim_{o, \Lambda} s'_2 \).

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

In both cases, \( \text{let} \) does not mutate the store, so \( s'_j = s_j \), and obviously \( s'_1 \sim_{o, \Lambda} s'_2 \).

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

With Theorem 3.2 we can be certain that if the permissions are set correctly, then no information flow can violate the policy throughout the execution of the program. The requirement is that any value that becomes public at any point, should be equal on the two initial stores. This is important because policies depend on the state of the store; so if a program grants permission to view a field that previously was
With Targeted Program Synthesis Draft, 2017, USA

Enforcing Declarative Policies

Figure 11. $\lambda^L$ static semantics.

<table>
<thead>
<tr>
<th>Well-formedness</th>
<th>$\Gamma \vdash r$</th>
<th>$\Gamma \vdash B$</th>
<th>$\Gamma \vdash S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$WF_r : \Gamma \vdash r : \text{Bool}$</td>
<td>$\Gamma \vdash T$</td>
<td>$\Gamma, y : \text{Store}, u : \text{User} \vdash r$</td>
<td></td>
</tr>
</tbody>
</table>

Subtyping

<table>
<thead>
<tr>
<th>Subtyping</th>
<th>$\Gamma \vdash T &lt; : T'$</th>
<th>$\Gamma \vdash B &lt; : B'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash B &lt; : B'$</td>
<td>$\vdash {B \mid r} &lt; : {B' \mid r'}$</td>
<td></td>
</tr>
</tbody>
</table>

Expression Typing

<table>
<thead>
<tr>
<th>Expression Typing</th>
<th>$\Gamma ; y \vdash e :: T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T-C$</td>
<td>$\Gamma ; y + e :: ty(e)$</td>
</tr>
<tr>
<td>$T-V AR$</td>
<td>$\Gamma ; y + x :: T$</td>
</tr>
<tr>
<td>$T-\lambda$</td>
<td>$\Gamma \vdash T_{\times} \Gamma, x : T_{\times}; y : e :: T$</td>
</tr>
<tr>
<td>$T-\text{get}$</td>
<td>$\Gamma ; y \vdash \text{get} x :: {B \mid r \land v = y[x]}$</td>
</tr>
<tr>
<td>$T-\text{If}$</td>
<td>$\Gamma ; y \vdash e :: {B \mid r}$</td>
</tr>
<tr>
<td>$T-\text{If} \Gamma, [v \mapsto T] r + e_1 :: T \Gamma, [v \mapsto \bot] r + e_2 :: T$</td>
<td>$\Gamma ; y \vdash e :: T$</td>
</tr>
<tr>
<td>$T-[\cdot]$</td>
<td>$\Gamma ; \sigma + e :: {\text{Bool} \mid v \mapsto r}^{\lambda(x, u) \cdot (x, u) \wedge r}$</td>
</tr>
<tr>
<td>$T-\text{Inst}$</td>
<td>$\Gamma ; y \vdash e :: \forall \alpha. S$</td>
</tr>
<tr>
<td>$T-\text{let}$</td>
<td>$\Gamma ; y \vdash e :: T$</td>
</tr>
<tr>
<td>$T-\text{print}$</td>
<td>$\Gamma ; y \vdash \text{print} x_1 x_2 :: s$</td>
</tr>
<tr>
<td>$T-\text{set}$</td>
<td>$\Gamma ; y \vdash \text{set} x_1 x_2 :: s$</td>
</tr>
</tbody>
</table>

Statement Typing

A.5 Health Portal Case Study

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

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

2017-11-29 11:22 page 17 (pp. 1-18)
The health portal code takes advantage of Lifty’s ability to generate checks as close to the data source as possible. One view function, showRecordsForPatientView, uses a filter over the list of all records to find the records that have a specified patient, and then outputs the result. The repair works as expected: the repaired version of the function generates a complex check (corresponding to the above) and runs it on each element of the list, so that only those records that pass the check will be shown.

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

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