UNIVERSITY OF CALIFORNIA SAN DIEGO

Encoding Security Policies as Refinement Types for Database Applications

A Thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science

in

Computer Science

by

Sourav Anand

Committee in charge:

Professor Nadia Polikarpova, Chair
Professor Deian Stefan
Professor Ranjit Jhala

2019

The Thesis of Sourav Anand is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Nadia Polikarpova for her support as the chair of my committee. Her expertise in the domain and insightful discussions about the research were instrumental in helping me complete this thesis. I am deeply grateful to her for providing an enriching environment for the past two years which has helped me gain valuable research experience.

I would also like to acknowledge Professor Ranjit Jhala for his guidance throughout the project. His insights during our research meetings and classroom discussions have been incredibly helpful towards this thesis.

I am also grateful to Rose Kunkel for her invaluable contribution towards this thesis without which it would have taken a lot more time to finish.

I would like to thank all my family members and friends for their incredible support throughout my graduate studies. Aditionally, I would like to express my gratitude to Professor Subhajit Roy (CSE, IIT Kanpur) who kindled my interest in the field of Programming Languages.

Finally, I would like to thank Harshini, Kaustubh, Sumit, and Tanvi for reviewing my thesis and helping it reach its final shape.

Chapter 4 is coauthored with Anand, Sourav and Kunkel, Rose. The thesis author was the primary author of this chapter.

ABSTRACT OF THE THESIS

Encoding Security Policies as Refinement Types for Database Applications

by

Sourav Anand

Master of Science in Computer Science

University of California San Diego, 2019

Professor Nadia Polikarpova, Chair

This thesis presents an encoding of security policies for database-backed web applications as refinement types. We add refinement types to the function calls which interact directly with the database in order to attach a label to the output data. Any computation over the data retrieved from the database also requires computing a new label as per the refinement types added to the computation to enforce the security policies. The label on the data is verified against the security policies at the point where data is being displayed to a viewer. The refinement types are verified statically with LiquidHaskell which allows programmers to reason about the enforcement of the security policies across the whole application.

Chapter 1

Introduction

Recent years have seen an exponential increase in the number of web applications. Broad availability of the internet and the growing popularity for cloud computing has been an important aspect towards this growth. In order to provide a seamless and rich experience, these applications store and perform computations on user data, including private and sensitive information. Such information should not be leaked to unauthorized users because of bugs in the application. Programmers decide specific policies regarding the access and visibility of data and add various checks in the application to enforce these policies.

Programmers write test cases along with their desired output to test the functionality of web application. This also includes the tests required for verifying the enforcement of the security policy across the web application. This is a manual method and is often infeasible to write test cases for all of the possible data flow paths across the whole application. It is often the case that applications, especially the large ones have a control flow path which doesn't have the policies enforced. An attacker can use this to access private data of the users.

Static information flow control techniques [1, 2, 3, 4, 5, 6] attempt to verify the applications against the security policies decided by the programmers. However, these techniques are not expressive enough and do not support many real-world policies. Lifty [7] introduces a static information flow control technique which allows the programmers to verify expressive security policies across a whole application written in the Lifty toy language.

However, data handling in the Lifty toy language is different than the real world databases used by web applications. The main difference is that Lifty accesses each data fields individually, whereas, databases use query language to return all of the resulting data at once, which presents additional challenges when attempting to write refinement types for the data retrieval API. We look into the encoding of security policies used by Lifty and define a similar encoding for applications which use databases. Our database API is based on the Persistent library, a part of the Yesod Web Framework, which provides a query style API.

This thesis presents an API library which wraps the database access functions used by Yesod and adds refinement types to those functions which contain the security policies. We also adopt the `Tagged` monad based computation used in Lifty and all of the functions in the API library performs computation in the `Tagged` monad and appropriately labels the monad. The label is checked against the privileges of the users to which the data is being displayed. The main contribution of this thesis is the refinement types for the database API which correctly assign security label to values read from the database and enforce the security policy.

Chapter 2

Motivating Example

In this section, we discuss an example web application named *Todo*. The application is a simple todo list which allows users to add tasks and also allows them to share their entire list with other users. The security policy on the visibility of tasks can be broken down into the following two distinct policies.

P1. Users should be able to see all of the tasks created by them.

P2. User `a` should be able to see the tasks of User `b` only if User `b` has shared their tasks with User `a`.

The above policies are formalized with respect to the eventual *viewer*, which depends on how the data is displayed. Todo has two different methods of displaying data to users. The first method is to display the tasks to the logged-in user and the second method is a daily email update which is automatically sent to users showing the new tasks shared with them during the previous day. Enforcing security policies for these display actions means that the logged in user should only see their own tasks and the tasks shared with them, and the email update should only contain information which is visible to the receiver of the email.

2.1 Database Structure

The database design of the the Todo application consists of three tables: *User*, *TodoItem* and *SharedItem*, represented in Figure 2.1. The application is architected such that users are

added to the User table when they log in for the first time, tasks are added to the TodoItem table when a logged-in user adds a task and an entry is added to the SharedItem table when a user shares their list with another user.

The Todo application has a security policy on the Task field of the TodoItem table which allows the Task field to be displayed to a viewer only if they are the owner of that task or the owner has shared their list with them. A security policy is a function of two arguments, the row `r` and the viewer `v`. The policy has to be enforced for each row `r` present in the TodoItem table with respect to the viewer `v`. The policy annotated for the Task field in Figure 2.1 is a disjunction of two subexpressions, the first one stating that UserId of `v` is the same as the Owner field of `r` (Policy P1) and the second one stating that there exists a row in the SharedItem table indicating that the Owner of `r` has shared their list with the viewer `v` (Policy P2).

We will discuss the security policies with respect to the database snapshot of the application shown in Figure 2.2. The snapshot contains 3 users: Alice, Bob and Eve, along with the tasks added by them. The SharedItem table shows that Alice has shared their list with Bob and Bob has shared theirs with both Alice and Eve. The security policies, if enforced correctly, should allow Alice and Bob to access their own tasks and the tasks of each other and it should allow Eve to access their own tasks along with the tasks of Bob. No other accesses should be allowed by the application.

2.2 Security Policies

We categorize the policies on a field into two types: *local policies* and *global policies*. A policy on a field of row `r` is local if it depends only on the data from `r`. Such policies can be easily verified when a row and viewer are provided. Global policies, on the other hand, depend on more information than the current row and the viewer. The dependencies of global policies can also span across multiple tables of the databases. It is more difficult to verify such policies as a global context is required.

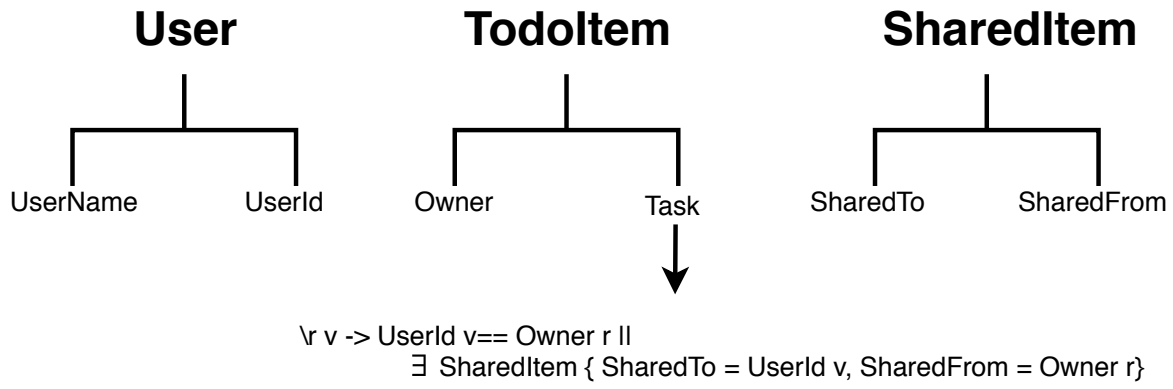

Figure 2.1. The database model along with the policy annotation for the Task field of the TodoItem table

User	
UserName	UserId
Alice	1
Bob	2
Eve	3

TodoItem	
Task	Owner
Write Thesis	1
Cook	2
Book Flights	1

SharedItem	
SharedTo	SharedFrom
2	1
1	2
3	2

Figure 2.2. The database Snapshot containing 3 users and their tasks. Alice shares their list with Bob and Bob shares it with both Alice and Eve

We can see from the database model for the Todo application from Figure 2.1 that Task field has a policy consisting of a disjunction whose subexpressions can be categorized into local policy and global policy. Policy P1 depends on the row and the viewer and hence it is a local policy. Policy P2 has a requirement on the SharedItem table and hence it is a global policy. We will discuss how to handle these two types of policies separately with example client code written in Yesod.

2.3 Yesod API example

We will follow the database API used in Yesod to discuss the client code. Figure 2.3 shows a sample function call which takes a UserId `u` as input and displays the tasks of rows from TodoItem table which have the Owner set to `u`. Yesod API provides the `selectList` function which takes a list of filter conditions as input and computes an output list of rows which

```
displayOwnedBy u = renderHTML (do
    rows <- selectList [TodoItemOwner ==. userId u]
    tasks <- projectTask rows
    return (show tasks)
    )

projectTask [] = []
projectTask x:xs = (task x):(projectTask xs)
```

Figure 2.3. Sample function call in Yesod which selects the rows from TodoItem table for which the Owner is same as the input u, and then displays the task from the selected rows to the logged-in user

```
searchTask t = renderHTML (do
    rows <- selectList [TodoItemTask ==. t]
    owners <- projectOwners rows
    return (show owners)
    )

projectOwners [] = []
projectOwners x:xs = (owner x):(projectOwners xs)
```

Figure 2.4. Sample function call in Yesod which selects the rows from TodoItem table for which the Task is same as the task t, and displays the owner from the selected rows to the logged-in user.

satisfy the given filter conditions. `projectTask` is a simple function which takes a list of rows and generates a new list containing only the Task field from the input. Finally, `renderHTML` function displays the tasks to the logged-in user.

2.4 Local Policies

Local policies are easier to enforce as they usually only require the programmer to add simple *if* checks in the code. We discussed that Policy P1 is a local policy and now we will see how we can enforce it.

```
displayOwnedBy u = renderHTML (do
    rows <- selectList [TodoItemOwner ==. userId u]
    tasks <- projectTask rows
    return (show tasks)
    )

projectTask [] = []
projectTask x:xs = if owner x == userId loggedInUser
                       then (task x):(projectTask xs)
                       else projectTask xs
```

Figure 2.5. The client code containing the if condition in the projectTask function which matches the userId of the logged-in user to the Owner of the task before adding the task from a row to the output list

2.4.1 Projecting a sensitive field

We will first examine the sample client code shown in Figure 2.3. The client code retrieves the rows owned by user `u` and displays their tasks to the logged-in user. Owner is a public field and hence filtering on it does not reveal any sensitive information. However, Task is a sensitive field and the client code projects the Task field and displays it to the logged-in user. The code does not place any checks on the input provided to the function `displayOwnedBy` and hence does not satisfy the policy P1 (we will discuss about P2 in later sections). The input `u` can be anything, so it might display the logged-in user the tasks from other users which would violate policy P1. In order to fix this, the programmer needs to add an *if* condition in the `projectTask` function which compares the owner of the row with the logged-in user before projecting the task field of that row.

Figure 2.5 shows the client code with additional check in the function `projectTask` which makes sure that the function returns Tasks of only those rows whose Owner is the same as `loggedInUser`. This forces the input `u` to `displyOwnedBy` function to be the userId of the `loggedInUser` in order to view any task. This will enforce policy P1 as it will allow only those tasks to be displayed which are owned by the `loggedInUser`.

```
1 searchTask t = renderHTML (do
2     rows <- selectList [TodoItemTask ==. t,
3                         TodoItemOwner ==. userId loggedInUser]
4     owners <- projectOwners rows
5     return (show owners)
6     )
```

Figure 2.6. The client code containing the extra filter in the selectList function call which selects the rows from TodoItem table for which the userId of the logged-in user is the same as the Owner of the task and the Task is the same as the input task t, and displays the owners from the selected rows

2.4.2 Filtering on a sensitive field

Figure 2.4 shows client code where the filter is done directly on a sensitive field and a public field is projected which is eventually displayed to the logged-in user. Even though the code is displaying a public field, it violates the policy P1 because it is indirectly leaking information of the users who have a specific task in their todo list. Policy P1 states that only the owner should be able to view the Task field and hence the programmer needs to add checks in the client code to enforce this indirect leak of the task information.

P1 can again be enforced by adding checks in the `projectOwners` function but it is slightly counter-intuitive since owner is a public field. We will look into another way to enforce this policy where the programmers add an extra filter in the `selectList` API call to contain only those rows in output which satisfy the policy P1. Figure 2.6 shows the fixed client code which has an extra filter in line 2 forcing the API to select only those rows whose Owner is same as the userId of `loggedInUser`. This additional filter will enforce the policy P1 as it will select only those tasks which are owned by the `loggedInUser`.

2.5 Global Policies

Global policies are harder to enforce as compared to local policies as they are dependent on data beyond what is being selected. In the Todo app, we saw that policy P2 is a global policy.

In previous section, the client code allowed the `loggedInUser` to display the tasks owned by them and we saw how various checks helped enforce policy P1. Todo also allows users to display the tasks owned by a different user but shared with them. A programmer can again use the function shown in Figure 2.3 to display the tasks owned by user `u`, but now we would like to see the tasks shared with the `loggedInUser`, which is allowed by P2. Enforcing policy P2 can be tricky as it requires checking the existence of a particular row in a different table. The programmer must first select the data from the SharedItem table and then select the data from TodoItem table based on the output of previous query, to satisfy the policy.

Figure 2.7 shows the sample client code which enforces policy P2. The function `displaySharedTasks` first select the rows from SharedItem table which have the SharedItemTo field equal to the UserId of the `loggedInUser` and is stored into `sharedItems`. The next step calls `projectSharedFrom` on `sharedItems` which generates a list of users who have shared their tasks with the `loggedInUser` and stores that in `sharedFromList`. This list is then used as the parameter of `selectList` in line 4 which selects a row from TodoItem table only if the Owner of a row is in `sharedFromList`. Finally, the Task field of the rows returned from the `selectList` function is projected using the function projectTasks and displayed to the viewer. This client code enforces policy P2 as only those Tasks are retrieved from the database which are shared with the `loggedInUser`.

```
1  displaySharedTasks = renderHTML (do
2      sharedItems <- selectList [SharedItemSharedTo ==.
          userId loggedInUser]
3      sharedFromList <- projectSharedFrom sharedItems
4      todoItems <- selectList [TodoItemOwner <-.
          sharedFromList]
5      tasks <- projectTasks todoItems
6      return (show tasks)
7      )
8
9  projectSharedFrom [] = []
10 projectSharedFrom (x:xs) = (sharedFrom x):(
      projectSharedFrom xs)
11
12 projectTask [] = []
13 projectTask x:xs = (task x):(projectTask xs)
```

Figure 2.7. Function call to display todoItems owned by the users who have shared it with the logged-in user

Chapter 3

Background

3.1 Yesod

Yesod [8] is a web application framework which allows programmers to write database-backed web applications in Haskell. It uses the Persistent library to interact with the database of the application via SQL style queries. Writing the web application in a strongly typed language provides a lot of benefits and helps avoid many common bugs. However, type safety only guarantees that the functions written in the application are well formed and does not provide security guarantees.

3.2 Liquid Haskell

Liquid Haskell [9] (LH) is a refinement type checker for Haskell. It allows programmers to define refinement types for various functions which allows them to define the exact requirements on the inputs and outputs of the functions. LH checks the complete source code statically and verifies it with respect to the predicates defined in refinement types. LH allows programmers to precisely encode the predicates on the inputs and outputs of a function which allows them to verify even complex properties attached to the functions.

```
1 {-@ data Tagged a <p :: User -> Bool> = Tagged { content ::
      a } @-}
2 data Tagged a = Tagged { content :: a }
3   deriving Eq
4
5 {-@ data variance Tagged covariant contravariant @-}
```

Figure 3.1. The data definition of Tagged along with its refinement type

3.3 Lifty

Lifty is a toy language which allows programmers to write secure web applications. It encodes security policies of the database and statically verifies the application against the policies. The application uses a `Tagged` monad which carries a label specifying the accessibility information for the data inside the monad and verifies the application based on the label and the viewer accessing the data. Lifty also provides the ability to automatically synthesize patches for security loopholes in the programs.

3.3.1 Tagged monad

Lifty combined the idea of security monads [10, 11] with *dependent labels*, which are predicates parameterized by the viewer derived from the security policies on the data stored inside the monad. The monad is initialized with the data retrieved from the database and assigned the label based on the policy associated with the database call. Any further computation on that data will bind the policy based on the computation being performed and the label will be updated accordingly.

We implemented the `Tagged` monad in a similar way as it is done in Lifty. Figure 3.1 is the definition is the `Tagged` type. The refinement type of `Tagged` contains an abstract refinement `p` which acts as the label for the `Tagged` monad. This label is a predicate parameterized by the viewer. The label determines if a viewer `v` is allowed to view the data inside the monad. For example, the label `v-> True` means that everyone is allowed to view the data,

```
1  {-@ instance Monad Tagged where
2        >>= :: forall <p :: User -> Bool>.
3               x:Tagged <p> a
4               -> (u:a -> Tagged <p> b
5               -> Tagged <p> b;
6        >>  :: forall <p :: User -> Bool>.
7               x:Tagged<{\v -> false}> a
8               -> Tagged<p> b
9               -> Tagged<p> b;
10       return :: a -> Tagged <{\v -> true}> a
11 @-}
```

Figure 3.2. Refinement type for the functions of Tagged which which help in the mainintaing the security policy by updating the label

`v->UserId v == 1` means that only the user with UserId equal to 1 is allowed to see the data and finally, and `v->False` means that no one is allowed to see the data inside the monad.

The variance property of the `Tagged` monad as shown in line number 5 states that the last parameter, i.e. the abstract refinement `p`, is contravariant. This is because having the label set to *True* means that every user is able to see the data, and as we move away from *True* to *False*, the policy gets more restrictive. Logically, this means that `Tagged<True> a <: Tagged <False> a`. Since this property of the label `p` is opposite from the general notion that *False* is a subtype of *True*, we mark `Tagged` as contravariant in `p`.

Figure 3.2 is the definitions of the monad instance for `Tagged`. The return functions takes a value of type `a` and returns a `Tagged a` with the label set to *True*. This is because `a` is just a constant value which has no security policy and hence it is safe to make it visible to everyone.

The bind operator (`>>=`) is the most important function for the `Tagged` monad as it is responsible for combining sensitive computations and properly propagating their labels. It takes two inputs, a `Tagged a` and a function `a -> Tagged b` and outputs a `Tagged b`. The label of `Tagged` is the same abstract refinement term `p` in each of `Tagged` monads. This

means that the final label `p` on the output should be at least as strong as both the label on the input `Tagged a` and the label added by the computation done by the input using the function `a -> Tagged b`.

The refinement type for the function `>>` states that it takes a `Tagged a` and `Tagged b` as input and outputs the second parameter. This function discards the result of previous computation and hence the output label of `>>` is the same as that of the second parameter. The first input has a label `v-> false` attached to it and it makes sure that any label attached to the first input is valid as all of the types are a sub-type of *False* as `Tagged` is contravariant in `p` .

Chapter 4

Encodings of security policies

In this chapter, we will discuss the database API along with the encoding of their refinement types which allows LH to verify the application with respect to the security policies. We will follow the database model of the Todo app (discussed in Chapter 2) to explain the refinement types added to the database API.

We had to write the `selectList` API for each table separately along with separate filter functions for each field. This was done to correctly encode the refinements which can be verified, keeping in mind LH's limitations. We will first discuss how local policies can be encoded and then will extend it to add support for global policies. Also, we will focus on just the encoding of the refinement types for the API and will not go into the implementation details of the API functions.

The main goal of refinement types is to infer the appropriate label regarding the user who can access the data and attach it to the `Tagged` monad containing the output data. The advantage of using the `Tagged` monad is that we can write the functions which displays the data from the `Tagged` monad to a viewer along with its refinement type that contains the predicate describing the viewer. LH will be able to verify if the viewer has the required permission when compared to the label attached to the `Tagged` monad passed as input.

Figure 4.1 shows an example function which takes a `Tagged` monad as input and displays its content to the currently logged in user. The refinement type added to the `renderHTML`

```
1 {-@ renderHTML :: Tagged <{\v-> UserId v == UserId
     loggedInUser}> a-> ()@-}
2 renderHTML :: Tagged a -> ()
```

Figure 4.1. A sample function which takes a Tagged monad as input and renders an HTML page for the logged in User

function states that the input to the function should be a sub-type of `Tagged <{\v->UserId v == UserId loggedInUser}>`. LH will verify the application only when all of the calls to the `renderHTML` function have the label of the `Tagged` monad as a subtype of the required label.

We saw how the `Tagged` monad and refinement types can be helpful in enforcing the policy with respect to the eventual viewer of the data. We just need to make sure that the `Tagged` monad is assigned an appropriate label, and we can rely on LH to verify that the policies are being enforced. We will now look into the encoding of the API functions and focus on how the refinement types assign a correct label to the `Tagged` monad which satisfies the security policies.

4.1 Filters

Yesod allows programmers to add filter conditions as the parameter to the `selectList` function. It is hard to define a generic refinement type which can encompass all possible filters. We can see from Figure 2.1 that the policies are attached to the specific fields of a table and also that the filter added to the database call in Figure 2.3 also depends on an individual field. This motivated us to encode the policies in the filter functions for various fields.

We implemented a new data type `RefinedFilter`, which allows us to attach the filter along with the policy for a field. Figure 4.2 shows the refinement type for `RefinedFilter` which is parametric over `record` defining the type of the item on which the filter will be applied (example, TodoItem for Todo app). It also defines two predicates (abstract refinements) `r` and `q`. The predicate `r` is of the type `record -> Bool` which corresponds to the post-condition of

```
{-@ data RefinedFilter record <r :: record -> Bool,
    q :: record -> User -> Bool> = _ @-}

{-@ data variance RefinedFilter covariant covariant
    contravariant@-}
```

Figure 4.2. Refinement type for the data type RefinedFilter

```

{-@ filterTodoItemOwner_EQ :: val: UserId -> RefinedFilter
  <{\row -> Owner row == val}, {\row v -> True }>
  TodoItem
@-}

{-@ filterTodoItemTask_EQ :: val: String -> RefinedFilter
  <{\row -> Task row == val}, {\row v -> Owner row ==
     UserId v }>
  TodoItem
@-}
```

Figure 4.3. Refinement type for the filter functions on the Owner and the Task field of TodoItem table.

the filter. The predicate `q` has the type `record -> User -> Bool`, and corresponds to the security policy on the field and is responsible for deciding the policy `p` for the `Tagged` monad. We can also see that the variance property of `RefinedFilter` marks the last parameter (the predicate `q`) contravariant. This is required because `q` represents the policy and hence follows the contravariance property of the label `p` attached to the `Tagged` monad.

We use the `RefinedFilter` data type to define various filters for the different fields of the tables. We use the predicates associated with the data type to encode the post-condition of the filter and the policy on the field. Figure 4.3 shows `filterTodoItemOwner_EQ` and `filterTodoItemTask_EQ` which are filters over the Owner and the Task field of the TodoItem table respectively. `filterTodoItemOwner_EQ` defines a filter over the Owner field which takes a UserId `val` as input and outputs a `RefinedFilter` where the

```
{-@selectTodoItem :: forall <q :: TodoItem -> User -> Bool,
  r :: TodoItem -> Bool, p :: User -> Bool>.
  {row :: TodoItem<r> |- User<p> <: User<q row>}
  RefinedFilter<r, q> TodoItem -> Tagged<p> [TodoItem<r>]
@-}
```

Figure 4.4. Refinement type for selectTodoItem which uses RefinedFilter TodoItem to retrieve TodoItems from the database

predicate `r` states that the filter select the rows whose Owner field is the same as `val` and `q` states that the policy on viewing the Owner field allows it to be displayed to everyone. `filterTodoItemTask_EQ` defines a filter over the Task field which takes a String `val` as input and outputs a `RefinedFilter` where the predicate `r` states that the filter selects the rows whose Task field is the same as `val` and `q` states that the policy on viewing the Task field restricts it to be displayed only to the owners of the task. This predicate `q` attaches the local policy P1 to the filter on the Task field.

The filters shown in Figure 4.3 are two of the many filters which are added in the API. We need to add filters for all of the fields with appropriate definitions of the predicates `r` and `q`. The filter function do not perform any select action from the database and are meant to be used along with the select functions. We will see in the refinement type of the select functions how the policy `q` helps in attaching the appropriate label to the `Tagged` monad.

4.2 Select Function

We discussed the data type `RefinedFilter` which contain the refinements for both filter property and the policy on the field. The goal is to compute a label over the data which is selected from the database using the filter and annotate the data with the computed label. The label on the `Tagged` monad should be based on the security policy on the field on which the data was filtered upon.

Figure 4.4 shows the encoding of the function `selectTodoItem` which takes a

```
1 {-@ todoItemSelect1 :: Tagged<{\v -> True}>
2     [TodoItem] @-}
3 todoItemSelect1 :: Tagged [TodoItem]
4 todoItemSelect1 = selectTodoItem (filterTodoItemOwner_EQ 1)
```

Figure 4.5. An example client code which selects the tasks owned by User with UserId 1.

`RefinedFilter` defined for TodoItem as input and outputs a list of TodoItems. The function is parameterized by three predicates, the label `p` for the output `Tagged` monad, the security policy `q` of accessing the field and the filter condition `r`. The predicates `r` and `q` are instantiated from the `RefinedFilter` passed as input and are used to compute the label `p`. The label `p` is computed from the constraint added in line 3 of Figure 4.4. The constraint states that if `row` is a TodoItem that satisfies the filter `r`, then User refined by `p` is a subtype of the User refined by `q` when `row` is passed as the input to `q`. We do a subtype here because `q` is the policy which must always be satisfied in order to read a `row` and hence the set of users who are allowed to view the data should be a sub set of the set of users who were allowed to access the data as described in the security policy `q`.

We will discuss how the label for `Tagged` monad is computed using client code. Figure 4.5 shows client code which selects TodoItems whose Owner field is equal to 1. The `todoItemSelect1` function selects the TodoItems using `filterTodoItemOwner_EQ` and passes the parameter 1 for UserId. The `selectTodoItem` function takes this `RefinedFilter` as input and instatiates the predicate `r` to `\row -> owner row == 1` and predicate `q` to `\ row v -> True`. The label `p` for the `Tagged` monad is computed based on the constraint `row::TodoItem<r> |- User<p> <: User<q row>`. Since `row` satisfies `r`, the owner of the `row` must be equal to 1. When such a row is passed as input to the policy `q`, the equation to compute `p` becomes `User<p> <: User <{\v -> True}>`. The predicate `p` is computed as `\v -> True` as LH uses a maximum fix-point solver and hence we get the value `p` with the largest possible subtype. Finally, the select function will return a list of TodoItems whose Owner

```
1 {-@ projectTask :: forall <r::TodoItem -> Bool,
2                            p::User->Bool>.
3     {row::TodoItem<r> |- User<p> <: User<{\v -> Owner row
        == UserId v}>}
4     [TodoItem<r>] -> Tagged<p> [String]
5 @-}
6 projectTask :: [TodoItem] -> Tagged [String]
7 projectTask input = Tagged {content = fmap (Task) input}
```

Figure 4.6. Refinement type of a function which takes a list of TodoItems as inputs and outputs a list with the Task field of the input

field is set to 1 and attach a policy which allows everyone to access the data inside the monad. This computed label is correct with the expect label for `Tagged` described in the refinement type for `todoItemSelect1` and LH verifies it successfully.

The select example discussed above filters on a public field and hence the label associated with the data allows any viewer to access it. We will now look into how projecting the sensitive field Task from the selected rows affect the tag associated with the `Tagged` monad. This is the same situation as discussed in Figure 2.3.

4.3 Project Function

We discussed the refinement type for the function which selects rows from the database based on a `RefinedFilter`. To correctly assign the label to the `Tagged` monad on sensitive field accesses, we add a restriction requiring the use of project functions to access a field from the data retrieved from the database. There can be a security policy attached to the field being projected and hence we need to update the label of the `Tagged` monad appropriately. We will discuss how project functions can be written which enforces the local policies of the field.

Project functions are field specific and hence we need to add a project function for each field in the database. Figure 4.6 shows the `projectTask` function which takes a list of TodoItems as input and outputs a `Tagged` list of String containing the Task field of the input

```
1 {-@ todoProjectTasks1 :: Tagged<{\v -> userId v == 1}>
2     [String] @-}
3 todoProjectTasks1 :: Tagged [String]
4 todoProjectTasks1 = do
5     todoItems <- selectTodoItem (filterTodoItemOwner_EQ 1)
6     projectTask todoItems
```

Figure 4.7. An example client code which projects the tasks owned by User with UserId 1.

rows. The function defines two predicates, a filter `r` on TodoItems and a label `p`. The filter `r` is instantiated with the predicate attached to the input list of TodoItem rows.

The label `p` is computed with respect to the constraint shown on line number 3. It states that if `row` satisfies the refinement `r` (i.e. all of the rows provided as input to `projectTask`), `User<p>` is a subtype of `User <{\v -> Owner row == UserId v}>`. This constraint restricts the label `p` to be able to display the data only to the viewer who is the owner of the input rows and helps in enforcing the local policy P1 for the field Tasks.

Figure 4.7 contains the function `todoProjectTask1` which performs both the action of selecting rows owned by user with UserId 1 from the TodoItem table and projecting out the Task field. We will discuss how the final label on the output is computed and compare it with the expected label based on the security policy. The first step of the function selects the TodoItem by filtering the rows whose Owner is set to 1. This output is stored in the variable `todoItems`. This step is the same as the example function discussed in Figure 4.5. We can see from the previous discussion that `todoItems` will contains the list of rows filtered from the table and will bind the label `v->True` with the output `Tagged` monad. Now, we use the data stored in `todoItems` and pass it to the function `projectTask`. The predicate `r` for `projectTask` will be instantiated with the predicate `r` defined by the `RefinedFilter` for the `selectTodoItem` step on line number 5. Hence, the TodoItem `row` satisfying the predicate `r` in line number 3 of Figure 4.6 will have the property `Owner row == 1`. When this row is substituted on the right side of the implies statement in the constraint on label `p`, we

get `User <p> <: User<{\v -> 1 == UserId v}>`. LH will infer the type of `p` as `\v -> UserId v == 1` and hence this would be the label assigned to the output of projectTask. Finally, we can see that the output of the function is the output of the projectTask call inside the do block, however, the label on the `Tagged` of the output for the function will bind the labels computed at line number 5 and 6 because we used the output of line number 5 to compute the value on line number 6. The bind function (`>>=`) will have to compute a new label which is a sub-type of both `\v -> True` and `\v -> UserId v == 1`. The type `\v -> UserId v == 1` is a subtype of both of the input types and hence LH's maximum fix-point solver will compute this as the final label on the `Tagged` monad.

We can see that the refinement type added to the `todoProjectTasks1` states the output to have the label `\v-> UserId v == 1` which corresponds to the label computed by us and hence it is successfully verified by LH. This label also follows policy P1 as it marks the task projected from the TodoItems owned by user with UserId 1 visible only to them.

4.4 Multiple Filters

We saw the use of abstract refinement over the filters and select functions to compute the label of the viewer on the data retrieved from the database. The select function is parameterized by three abstract refinements two of which are instantiated from the `RefinedFilter` passed as parameter and are used to compute the label `p` for `Tagged` monad. In order to increase the number of filters which can be used in one select query, we would need to define the predicates `r` and `q` for each of the `RefinedFilter`. We solve this issue by defining a new data type `FilterList` which takes multiple filters and computes a single `r` and `p` satisfying each one of them. This allows us to use the previously defined select and project functions without any changes.

Figure 4.8 shows the definition of `FilterList` along with the refinement encoding of the data type. `FilterList` has two constructors, `Empty` and `Cons`. `Empty` is the

```
{-@
data FilterList a <r :: a -> Bool,
    q :: a -> User -> Bool>. where
    Empty :: FilterList<{\_-> True}, {\_ _-> True}> a
    Cons  :: forall <r  :: a -> Bool,
                r1 :: a -> Bool,
                r2 :: a -> Bool,
                q  :: a -> User -> Bool,
                q1 :: a -> User -> Bool,
                q2 :: a -> User -> Bool>.
              {a_r1 :: a<r1>, a_r2 :: a<r2> |- {v:a | v ==
                 a_r1 && v == a_r2} <: a<r>}
              {row :: a<r> |- User<q row> <: User<q1 row>}
              {row :: a<r> |- User<q row> <: User<q2 row>}
              RefinedFilter<r1, q1> a ->
              FilterList<r2,q2> a ->
              FilterList<r,q> a
@-}
{-@ data variance FilterList covariant
    covariant contravariant@-}
data FilterList a = Empty
                  | Cons (RefinedFilter a) (FilterList a)
```

Figure 4.8. Definition of the data type FilterList along with its the refinement encoding.

`FilterList` when no filters are added and `Cons` works like the list constructor *cons* which joins a `RefinedFilter` with the `FilterList`. The refinement type encoding of the `Empty` constructor is set to `True` for both `r` and `q`. This is done because `Empty` means that there is no filter used for the select function and hence the complete table would be selected and the policy should be that it is visible to everyone. This is valid because there was no specific field on which the data was filtered with. The data will be stored inside the `Tagged` monad with label true but any attempt made to access a sensitive field of the database would update the label accordingly.

The data type `FilterList` stores multiple `RefinedFilter` functions using the `Cons` constructor and computes a common `r` and `q` satisfying all of them. The refinement encoding of `Cons` consists of six predicates, three for the filters and three for the policies. The predicates `r1`, `q1` are instantiated with the `RefinedFilter` passed as input and `r2`, `q2` are instantiated with the `FilterList` to which the input `RefinedFilter` is being added. The predicates `r` and `q` are computed based on the inputs and are used as the filter and the policy for the final `FilterList`. The predicate `r` is computed using the constraints placed on line number 11 which states that the set of elements satisfying `a<r>` should at least contain the set of common elements satisfying both `a<r1>` and `a<r2>`. The common policy `q` for the `FilterList` should individually satisfy the policy attached to each `RefinedFilter` in the `FilterList`. The constraints added on line number 12 and 13 are added to compute the policy `q` which satisfies both `q1` and `q2`. The first constraint states that if a `row` satisfies `a<r>`, `User<q row>` should be a sub type of the `User<q1 row>`. A similar constraint is placed for `q2`.

The select function defined previously can now be updated to take `FilterList` instead of the `RefinedFilter` as input and use the predicates `r` and `q` attached with the `FilterList`. We also define an infix operator '? :' which is used to build a `FilterList` using the `Cons` constructor whose definition is shown in figure 4.9.

```
(?:) :: RefinedFilter a -> FilterList a -> FilterList a
f ?: fs = f `Cons` fs
```

Figure 4.9. Definition of the infix operator ?: which takes a RefinedFilter and FilterList and generates a new FilterList

4.5 Global Policy

In the previous sections we saw the use of refinement types to compute the label for the `Tagged` monad based on the the security policies required to view the data inside it. We focused only on the local policies in the previous section to keep the discussion simple. We will now extend the encoding to support global policies for different fields.

Global policies depend on more data than the current `row` being filtered and hence we need to establish the validity of the extra requirements in the policy. These requirements can even span over multiple tables and hence it would become a difficult task to verify the policy for each `row` individually. We saw in Figure 2.7 that in order to satisfy the policy, the client code initially retrieved all the requirements and then ran the final query in the database. This motivated us write encoding to support the global policies in a similar way which requires the programmers to first retrieve all the requirements from the database.

We handle global policies by attaching various properties to the data being retrieved from the database and update the refinement types for filters and projection functions to require those properties. For each global policy, we create a property which is attached to the output of all of the select functions calls in our API library. We will discuss an example with respect to enforcing policy P2 in the Todo app which requires viewer `v` to be able to view the Task field of a row owned by User `u` only if there is a row in the the sharedItem table which states that the User `u` has shared their tasks with the viewer `v`.

Figure 4.10 shows the property which is attached to the rows retrieved from the Shared-Item table. It takes the SharedTo field as its first parameter and SharedFrom field as its second

```
{-@ measure sharedItemProp :: Int->Int-> Bool@-}
```

Figure 4.10. The measure which is attached to the output of selecting from SharedItem table

```
{-@selectSharedItem :: forall <q :: SharedItem -> User ->
    Bool, r :: SharedItem -> Bool, p :: User -> Bool>.
  {row :: SharedItem<r> |- User<p> <: User<q row>}
  FilterList<r, q> SharedItem -> Tagged<p> [{v:SharedItem<r
     >| sharedItemProp (sharedTo v) (sharedFrom v)}]
@-}
```

Figure 4.11. Updated definition of selectSharedItem which attaches the measure to the output list

parameter. We further use this to encode the refinement on the output of the select function. Figure 4.11 shows the updated select function for SharedItem table which takes a `FilterList` as input and outputs list of SharedItems. `selectSharedItem` updates the refined type of output to `Tagged<p> [{v:SharedItem<r>| sharedItemProp (sharedTo v)(sharedFrom v )}]` which states that the items of the output list satisfies the predicate `r` and each element `v` of the output satisfies the property `sharedItemProp (sharedFrom v)(sharedTo v)`. This property will be preserved by LH and can be used to satisfy the policies to filter and project sensitive fields.

```
{-@
projectTasks :: forall <r1::TodoItem-> Bool,
                        p::User->Bool>.
    {row::TodoItem<r1> |- User<p> <: User<{\v -> Owner row
       == UserId v || sharedItemProp  (UserId v) (Owner row
       )}>}
    xs:[v:TodoItem<r1>] -> Tagged<p> [String]
    @-}
```

Figure 4.12. Updated refinement type of projectTasks function which encodes both policy P1 and policy P2

Figure 4.12 shows the updated definition of `projectTasks` function which now allows it to display the Tasks shared with the viewer. The constraint on `User <p>` requires it to be a sub-type of `User<{\v -> Owner row == UserId v || sharedItemProp (UserId v)(Owner row)}>`. This either requires the rows to be owned by viewer `v` or by user `u` who satisfies `sharedItemProp (UserId v) (UserId u)` in order to be displayed to viewer `v`. The second disjunct can be satisfied only if the TodoItem rows were filtered by owners who were retrieved from the SharedItem table whose SharedTo field were set to `UserId v`. This allows LH to infer the label `p` which allows the viewer `v` to access the data in the `Tagged` monad.

We attached the `sharedItemProp` property to the output of the select function which contains information about each field of the row. The policies discussed also contained the same information and hence we were able to infer a proper label for the `Tagged` monad. There might be situations where there is extra information in the row which is not needed to enforce a particular policy. This will lead to an existential constraint which is not supported by LH. This situation can be handled by creating new properties which contain only the information required to verify policies instead of one single policy for the complete row of the table. Figure 4.13 shows an example where we are attaching 2 different properties to the output of the select function from the TodoItem table. These properties can now be individually used in different policies without having to deal with the existential term.

Chapter 4 is coauthored with Anand, Sourav and Kunkel, Rose. The thesis author was the primary author of this chapter.

```
1 {-@ measure taskOwnerProp :: Int->String-> Bool@-}
2 {-@ measure taskDeadlineProp :: String->Int-> Bool@-}
3
4 {-@selectTodoItem :: forall <q :: TodoItem -> User -> Bool,
      r :: TodoItem -> Bool, p :: User -> Bool>.
5   {row :: TodoItem<r> |- User<p> <: User<q row>}
6   FilterList<r, q> TodoItem -> Tagged<p> [{v:TodoItem<r>| (
       TaskOwnerProp (Owner v) (Task v)) && (taskDeadlineProp
        (Task v) (Deadline v)) }]
7 @-}
```

Figure 4.13. Updated definition of selectTodoItem which attaches multiple measure to the output list

Chapter 5

Evaluation

We evaluated our encoding on the Todo application discussed in previous sections. We modified the model of the database by adding an additional field and security policies. Figure 5.1 shows the updated model of the database along with the annotated security policies which was used for evaluation.

Our evaluation focused on retrieving data from the database and verifying the visibility of the data under four different types of viewer consisting of viewer A with UserId 1, viewer B with UserId 2, visible to everyone and not visible to anyone. We wrote multiple test cases corresponding to different scenarios where data is retrieved from the database and is passed to the different display functions. Our display functions are similar to the function shown in Figure 4.1 and have a refinement label for the input Tagged monad corresponding to the type of the viewer. We verified each of the test case with LH and reported the output as Safe or Unsafe.

5.1 Test Cases

We wrote eight test cases which displays different types of data which was computed using various queries to the database. All of the test cases retrive data with respect to user A. The following are the description of the test cases:

T1. Selected TodoItems owned by User A

```
x = selectTodoItem (filterTodoItemOwner_EQ 1 ?: Empty)
```

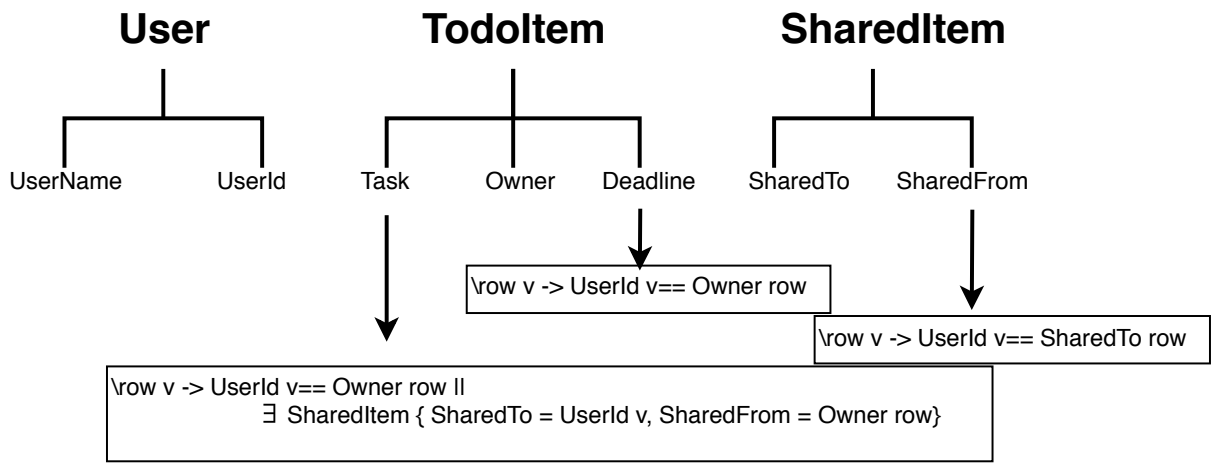

Figure 5.1. The database model along with the policy annotation for the TodoItem table

Table 5.1. LH output for the different test cases using the four different sets of viewers. Each output contains the time taken by LH to check the file (in s).

Test	Viewer = A	Viewer = B	Viewer = All	Viewer = None
T1	Safe(1.72)	Safe(1.82)	Safe(1.57)	Safe(1.56)
T2	Safe(1.68)	Unsafe(1.65)	Unsafe(1.67)	Safe(1.63)
T3	Safe(1.56)	Unsafe(1.88)	Unsafe(1.99)	Safe(1.58)
T4	Safe(1.52)	Safe(1.5)	Safe(1.53)	Safe(1.56)
T5	Safe(1.62)	Unsafe(1.77)	Unsafe(1.75)	Safe(1.81)
T6	Safe(1.7)	Unsafe(2.15)	Unsafe(2.39)	Safe(1.98)
T7	Safe(2.2)	Unsafe(2.57)	Unsafe(2.13)	Safe(1.97)
T8	Unsafe(2.91)	Unsafe(2.54)	Unsafe(2.82)	Safe(2.46)

Expected Viewers : All

T2. Projected Task field of TodoItems owned by User A

```
x = do
  rows<-selectTodoItem (filterTodoItemOwner_EQ 1 ?:
     Empty)
  projectTodoItemTasks rows
```

Expected Viewers : User A

T3. Projected Deadline field of the TodoItems owned by User A

```
x=do
  rows<-selectTodoItem (filterTodoItemOwner_EQ 1 ?:
     Empty)
  projectTodoItemDeadline rows
```

Expected Viewers : User A

T4. Selected SharedItems which are shared to User A

```
x=selectSharedItem (filterSharedItemSharedTo 1 ?:
    Empty)
```

Expected Viewers : All

T5. Projected the SharedFrom field for data selected in T4

```
x =do
  sharedItems <- selectSharedItem (
     filterSharedItemSharedTo 1 ?: Empty)
  projectSharedItemSharedFrom sharedItems
```

Expected Viewers : User A

T6. Used the SharedFrom data from T5 to select TodoItems shared with User A

```
1 x =do
2   sharedItems <- selectSharedItem (
      filterSharedItemSharedTo 1 ?: Empty)
3   sharedFromList<- projectSharedItemSharedFrom
      sharedItems
4   selectTodoItem (filterTodoItemowner_In sharedFromList
       ?: Empty)
```

Expected Viewers : User A

T7. Projected the Task field from the TodoItems shared with User A (T6)

```
1 x =do
2   sharedItems <- selectSharedItem (
      filterSharedItemSharedTo 1 ?: Empty)
3   sharedFromList<- projectSharedItemSharedFrom
      sharedItems
4   todoItems <- selectTodoItem (filterTodoItemowner_In
      sharedFromList ?: Empty)
5   projectTodoItemTasks todoItems
```

Expected Viewers : User A

T8. Projected the Deadline field from the TodoItems shared with User A (T6)

```
1 x =do
2   sharedItems <- selectSharedItem (
      filterSharedItemSharedTo 1 ?: Empty)
3   sharedFromList<- projectSharedItemSharedFrom
      sharedItems
```

```
4   todoItems <- selectTodoItem (filterTodoItemowner_In
       sharedFromList ?: Empty)
5   projectTodoItemDeadline todoItems
```

Expected Viewers : None

We show the output of verifying the above test cases using LH in Table 5.1. LH verifies the label attached to the input Tagged monad with that of the refinement type added to the display function. Test case T1 and T4 retrieves rows from the database by filtering on a public field and hence every user is able to view the data. Test cases T2-T3 projects a sensitive field of rows owned by User A and hence it is visible only to User A. T5-T7 retrieves the data which is shared with User A and hence the label on the Tagged monad gets updated to allow only User A to view the data. Finally, test case T8 retrieves the rows shared with User A but projects the Deadline field of those rows. We can see from the policy on the Deadline field that only owners are allowed to see the data and hence even User A is unable to view the data. We can see that the column corresponding to viewer = None is always marked safe. This is expected as no one is viewing the data retrieved from the database and hence it is always safe to write such code.

The test cases show that the encoding of the refinement types added to the database API is correctly assigning the labels to the `Tagged` monad. The client code which was marked as Unsafe by LH tells the programmer that the display function is trying to display data to a user in a way which does not follows the security policies of the application. We verified the client code with respect to just User A but since this is symmetric for each user, we can expect that the API will be able to correctly assign the labels to the `Tagged` monads.

5.2 Verification Time

We ran the test cases on Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz processor with 16 GB RAM. The tests were performed on Windows Subsystem for Linux (Ubuntu). Each of the timing data reported in this section is an average of 5 runs.

Table 5.1 shows the time to run individual client code for each test case. These timings do not include the time to verify the API Library. We can see that the each client code takes approximately *2-3* seconds to verify.

The time taken to verify the API library is *3.1* seconds. We also measured the time taken by LH to verify a single file containing multiple client codes. We created this file by copying all of the client codes which displayed data to User A. The time taken to verify this file is *3.83* seconds.

Chapter 6

Future Work

We discussed the encoding of refinement types on the database API which allows us to verify the application with respect to the security policies. The programmers need to label the displayed functions with policy pertaining to the viewer to whom the data is being displayed and LH will statically verify if the viewer is allowed to view the data which is being passed as input to the display function, with respect to the security policies.

We only discussed the refinement types in this thesis and did not discuss the implementation. We are currently implementing a Yesod application which uses the refinement types in the Database API. We have completed the implementation of the API which conforms to the refinement types in Yesod. We are currently working towards implementing the functions in Yesod which displays the data to the user. We hope to verify the complete application written in Yesod with respect to the security policies.

We also saw that the API is slightly different than the Yesod examples discussed in Chapter 2. We had to define separate functions for each action which adds burden onto the programmer to write the application appropriately. This design decision was taken because of the limitations of LH. We are also examining other approaches where we can write the functions in the same way as actual Yesod functions work.

Bibliography

[1] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 105–118, USENIX Association, 2010.

[2] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, (Berlin, Heidelberg), pp. 529–549, Springer-Verlag, 2010.

[3] L. Jia and S. Zdancewic, "Encoding information flow in aura," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, (New York, NY, USA), pp. 17–29, ACM, 2009.

[4] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *Int. J. Inf. Secur.*, vol. 6, pp. 67–84, Mar. 2007.

[5] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), pp. 158–170, ACM, 2005.

[6] A. C. Myers and A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 228–241, ACM, 1999.

[7] N. Polikarpova, J. Yang, S. Itzhaky, and A. Solar-Lezama, "Type-driven repair for information flow security," *CoRR*, vol. abs/1607.03445, 2016.

[8] M. Snoyman, *Yesod Web Framework for Haskell*. http://www.yesodweb.com/, 2018.

[9] N. Vazou, E. L. Seidel, and R. Jhala, "Liquidhaskell: Experience with refinement types in the real world," *SIGPLAN Not.*, vol. 49, pp. 39–51, Sept. 2014.

[10] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in haskell," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, (New York, NY, USA), pp. 13–24, ACM, 2008.

[11] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Programming Languages and Systems* (A. D. Gordon, ed.), (Berlin, Heidelberg), pp. 529–549, Springer Berlin Heidelberg, 2010.