ABSTRACT

Title of thesis: LMONAD: INFORMATION FLOW CONTROL FOR HASKELL WEB APPLICATIONS

James Parker, Master of Science, 2014

Thesis directed by: Professor Michael Hicks
Department of Computer Science

Many web applications adhere to privacy policies for users and offer rich access control policies. It can be difficult to enforce these policies because applications can be complex, large, and involve multiple developers. Information Flow Control (IFC) can address this difficulty by guaranteeing that policies are enforced.

This thesis presents LMonad, an IFC system designed to enforce IFC policies in Haskell web applications. LMonad generalizes LIO, previous work that offers IFC for Haskell programs. Specifically, LMonad provides a monad transformer to enforce IFC, in LIO's style, over any existing computation. In addition, LMonad offers label annotations to specify policies, and it guarantees that database interactions adhere to the policies.

To evaluate LMonad, we developed an example website with various IFC policies and converted a large, existing web application to include LMonad policies. Results indicate that LMonad has low runtime overhead and is feasible to use in terms of programmer effort.

LMONAD: INFORMATION FLOW CONTROL FOR HASKELL WEB APPLICATIONS

by

James Lee Parker

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2014

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Professor Jeffrey S. Foster
Professor Elaine Shi

Acknowledgments

I would like to acknowledge my advisor, Professor Michael Hicks, for his guidance in completing this research. I would like to acknowledge my thesis committee, Professor Jeffrey S. Foster and Professor Elaine Shi, for taking the time to review this work. Finally, I would like to thank my labmates, friends, family, and loved ones for their support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

$\sqcup$	Join (Least upper bound)
$\sqcap$	Meet (Greatest lower bound)
$\sqsubseteq$	Flows to
$\bot$	Bottom
$\top$	Top

CSRF	Cross Site Request Forgery
DSL	Domain Specific Language
LOC	Lines of Code
IFC	Information Flow Control
SQL	Structured Query Language
XSS	Cross Site Scripting

Chapter 1: Introduction

As cloud services become ubiquitous, it is essential that web applications protect the privacy of their users and strengthen the security of their systems. These web applications can be complex since they are typically maintained by multiple developers and have large code bases. These realities make it extremely difficult to enforce privacy and access control policies through manual inspection of the code. One solution to this problem is to use Information Flow Control (IFC) to automatically guarantee that these policies are enforced.

LIO is an existing IFC framework designed for Haskell applications [1]. This system provides strong guarantees about confidentiality and integrity, which can be leveraged to enforce an application's policies. Unfortunately, LIO has some limitations that prevent it from being used for web applications written in Haskell. The first is that LIO only enforces IFC for the IO monad. This is insufficient since Yesod, a web framework for Haskell, runs in a special Handler monad. Therefore, LIO is unable to enforce IFC for Yesod applications. Another limitation is that LIO limits monadic lifting to trusted code, which prevents regular code from performing IO operations. This allows secure systems to be built from the ground up, but can be too constraining for developers who wish to use existing libraries like Yesod.

LMonad addresses these limitations by generalizing LIO. LMonad generalizes LIO by providing a monad transformer, so that IFC can be added to any existing computation. In addition, the developer may relax the condition that only trusted code can perform lifting.

Objects in LMonad have a security label associated with them, where labels have a partial ordering. LMonad enforces IFC in LIO's style by dynamically keeping track of two pieces of information. The first is the label of objects already read or written by a program. The second is the clearance label which restricts what the program is allowed to read or write. If the program ever tries to manipulate an object with a label below the current label or above the clearance label, a security exception occurs and the program fails safely.

LMonad integrates with Yesod by replacing functions that interact with the database. The database functions provided by LMonad guarantee IFC policies are enforced by inserting appropriate checks on database reads and writes. LMonad also supports richer database queries through a DSL called LSQL. LSQL queries can include outer joins, and they also enforce IFC policies with appropriate label checks. Haskell's type system statically enforces that LMonad's database functions are used. IFC policies are enforced dynamically by LMonad and fail gracefully when policy violations occur. To define IFC policies, LMonad offers label annotations, which allow the developer to define the label associated with objects stored in the database.

A key advantage of using LMonad is that it is much easier to guarantee that IFC policies are being enforced. In traditional applications, it is the developer's

job to ensure complete mediation of security-sensitive operations and to perform the right authorization checks — any failure to do so may result in a systemwide violation of security. The entire code base essentially becomes trusted, and it would be arduous to audit the entire system to ensure that all the checks are correct and satisfy IFC policies. With LMonad, the trusted computed base of an application shrinks dramatically. Typically, auditors would only need to verify the trusted code that sets the clearance works properly, while LMonad would guarantee that the rest of the code adheres to the IFC policies. For example, the original codebase of the Build it Break it Fix it web application consists of 7,077 trusted lines of code. It is almost imposible to verify this amount of code, and a previous version of the application had at least one security policy violation. By adding LMonad, the trusted code base of this site shrinks to 70 lines of code. It is almost trivial to verify that this code properly sets the clearance label.

LMonad's abstractions can be used to simplify code that performs permission checks. In many cases, manual access control checks can be eliminated since LMonad automatically provides these checks. For example, the Build it Break it Fix it website has a function, shown in Figure 1.1, that redirects unauthorized users before displaying a team's confidential information. This function first checks if the user is an administrator. It then does a database query to see if the user is on the given team. If both of these conditions are false, it redirects the user's browser to a permission denied page. With LMonad, this function can be removed because any database queries that retrieve the team's confidential information will automatically enforce IFC policies and redirect unauthorized users. LMonad gives the developer

```
redirectUnauthorizedTeam tId = do
  (Entity uId u) <- requireAuth
  if userAdmin u then
    return ()
  else do
    teams <- runDB $ select $ from $
      \(t `LeftOuterJoin` tm) -> do
        on (t ^. TeamId ==. tm ^. TeamMemberTeam)
        where_ ( t ^. TeamId ==. val tId &&.
        (t ^. TeamLeader ==. val uId
          ||. tm ^. TeamMemberUser ==. val uId))
        limit 1
        return t
    if (List.length teams) == 1 then
      return ()
    else
      permissionDenied "Permission␣denied."
```

Figure 1.1: Function that redirects unauthorized users from viewing a team's confidential information on the Build it Break it Fix it website. This function is unneeded with LMonad.

this type of access control check for free, thereby reducing the developer's workload.

Another way LMonad simplifies permission checks is that developers can ask the floating label system whether the program has permission to take an action. For example, Figure 1.2 shows a website without LMonad that needs to manually perform access control checks. This process is tedious as the program first needs to check whether the user is logged in. Then it needs to check whether the user is viewing its own profile or if the user is an administrator. If all of these conditions are true, then the site can display the confidential information. The details of this are later discussed in Section 4.3, but LMonad can simplify and reduce the code associated with this access control check. LMonad does this by looking at the current label and clearance to determine whether to display the confidential

```
getProfileR :: UserId -> Handler Html
getProfileR uId = do
  userM <- maybeAuth
  let dispEmail =
    case userM of
      Nothing ->
        mempty
      Just (Entity uId' u) ->
        if uId == uId' || userAdmin u then
          [whamlet|
              <p>
                  Email: #{userEmail u}
          |]
        else
          mempty
  ...
```

Figure 1.2: A manual access control check that decides whether to display a user's confidential information. LMonad can simplify and reduce the code associated with this check.

information or not.

LMonad's implementation is split into two libraries. The core library includes the generalization of LIO and enforces IFC for Haskell applications. The other library integrates with Yesod by providing label annotations, IFC safe database functions, and LSQL. The implementation of the former is 569 lines of code, while the latter is 4,028 lines of code. Both libraries are released as open source and are available on GitHub[1].

To evaluate LMonad, we developed an example website with access control policies designed to protect users' private information. We also converted Build it Break it Fix it, a large existing web application, to make use of LMonad. Our results indicate that it is feasible to use LMonad in Yesod applications. Programmers

[1]https://github.com/jprider63/LMonad

need to exert moderate effort to support LMonad, however most of the changes are straightforward and the type checker indicates where the changes need to be made. In terms of performance, experimental benchmarks only show a roundtrip latency overhead of up to 4 milliseconds or 4.82%.

The main contribution of this work is LMonad, which is a generalization of LIO that is more flexible and deployable. LMonad provides end-to-end security by enforcing IFC across database boundaries for the web development framework Yesod. LMonad supports a richer, policy-enforced query language, and it is particularly novel in that the query language allows outer joins. This work provides LMonad's implementation and an evalutation that demonstrates LMonad is effective at enforcing security policies with minimal overhead.

Chapter 2: Background

This chapter describes background information for this work, and previous work I built upon. I discuss some of the features from the Haskell programming language. Readers who are familiar with Haskell and its common abstractions may feel free to skip this section. Yesod is like the "Rails" of Haskell, and we mention some of its benefits. Then the chapter introduces the basics of IFC. Finally, we highlight LIO's floating label IFC system.

2.1 Haskell

Haskell is a lazy, functional programming language [2]. It has a strong, static type system that catches many bugs at compile time. As such, if a program compiles, the developer has more of an assurance that the program is correct [3].

Haskell has typeclasses, which are akin to interfaces in other languages and traits in Rust [4]. Typeclasses allow overloading of their functions. When a developer creates a new datatype, the developer can create an instance of a typeclass. Figure 2.1 illustrates how this would be done for the `Foo` data type and the `Eq` typeclass. The typeclass definition of `Eq` is also included. This definition requires that instances of `Eq` define two functions, `==` and `/=`, that implement equals and

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

data Foo = Foo | Fighters

instance Eq Foo where
    (==) Foo Foo = True
    (==) Fighters Fighters = True
    (==) _ _ = False
    (/=) a b = not (a == b)
```

Figure 2.1: `Eq` instance for the `Foo` data type.

not equals for the datatype. In the figure, type `Foo` has two constructors `Foo` and `Fighters`. `Foo`'s instance of `Eq` is implemented such that two `Foo`s are equal only if both arguments have the same constructors.

Monads are frequently used in Haskell to chain together special computations that can cause side effects [5]. For example, the `IO` monad allows programs to perform operations like interacting with the file system, reading from standard input, and writing to standard output. Monads can also be layered on top of one another. The mechanism to perform this layering is called a monad transformer. Monad transformers are important because they allows developers to combine the effects of computations that run in different monads. To run a computation of an inner monad in the outer monad, the programmer can simply lift the inner computation using a monad tranformer.

2.2 Yesod

Yesod is a web development framework for Haskell [6]. It is a very strong framework for developing web applications for many reasons. The first is that it has a strong type system that prevents many kinds of web-based attacks including cross site scripting (XSS), cross site request forgery (CSRF), and SQL injection. In addition, it provides a rich environment for web development by taking care of routing, database queries, and session cookies. Finally, Yesod is actively developed and has a responsive community that is willing to aid developers and fix issues.

2.3 Information Flow Control

Information Flow Control (IFC) systems restrict how information flows within a program's execution. Different channels are used to model the inputs and outputs of a program with different principals. IFC systems are used to enforce confidentiality models like non-interference and integrity models like confinement.

Non-inteference states that attackers cannot distinguish between the outputs of a program when that program is given different secret inputs [7]. This guarantees that non-interfering programs will not leak secret information. Confinement provides integrity in programs by preventing a principal from writing to channels other than its own [1].

Figure 2.2: The confidentiality security lattice representing classified and public data.

```
class (Eq l) => Label l where
    leq :: l -> l -> Bool
    lub :: l -> l -> l
    glb :: l -> l -> l
```

Figure 2.3: The `Label` typeclass, specifying an interface for the security lattice.

2.4 LIO

LIO is a dynamic, floating label IFC framework for Haskell [1]. In such a system, objects have labels associated with them that indicate policy properties like confidentiality and integrity. Labels allow the program to keep track of whether the code executing on behalf of a principal has permission to read or write to protected values. When a policy violation occurs, LIO halts the program's execution and handles the error.

Labels in LIO must create a security lattice. Security lattices can be used to model how information is allowed to flow within a system [8]. For example, Figure 2.2 shows a confidentiality lattice with two labels, high (H) and low (L). These can be thought of as classified and public information, respectively, so low information can flow to high information, but the reverse is not allowed. Labels in

```
printDocument handler = do          -- L0
    doc <- openDocument handler     -- L0 `join` LD
    print doc                       -- L0 `join` LD
```

Figure 2.4: An example program with a floating label system.

LIO programs must provide the least upper bound (Join, $\sqcup$), greatest lower bound (Meet, $\sqcap$), and flow to ($\sqsubseteq$) operators to create a security lattice. Specifically, labels must correctly implement the `Label` typeclass shown in Figure 2.3.

LIO is a floating label system because LIO dynamically keeps track of the current label and clearance as the program executes. Behind the scenes, LIO stores the current label and clearance label in a mutable tuple. Every operation performed in a LIO computation could cause the current label to be updated. For example, whenever a value with a label is manipulated, the current label rises to join of the value's label and itself. As a result, the current label monotonically increases. An example of a program with a floating label system is shown in Figure 2.4. The comments show what the current label is at every line in the function. The function `printDocument` take a handler to a document, opens it, and then prints the document to the screen. When the document is opened, `openDocument` taints the current label with the document's label `LD`. Since the current label is initially `L0`, the resulting current label rises to `L0` $\sqcup$ `LD`. From now on, the current label can only stay at its current level or increase.

Initially, LIO programs carefully set the clearance label. If the current label ever exceeds (or can not flow to) the clearance, then a policy violation has occurred and execution halts. This behavior is required so that LIO programs do not leak con-

```
evalLIO :: LIO l a -> LIOState l -> IO a

getLabel :: Label l => LIO l l
setLabel :: Label l => l -> LIO l ()

getClearance :: Label l => LIO l l
setClearance :: Label l => l -> LIO l ()
```

Figure 2.5: Highlights from LIO's API.

fidential data. LIO formally proves this statement with a proof of non-interference. LIO also proves confinement which guarantees the integrity of sensitive data.

Figure 2.5 highlights parts of LIO's API. The function `evalLIO` runs an LIO computation given an initial current label and clearance state. The first argument with type `LIO l a` is the LIO computation that will be evaluated. The polymorphic type `l` is the computation's label type, while type `a` is the type returned by the computation. The second argument with type `LIOState l` sets the initial current label and clearance label. `LIOState l` is basically a wrapper around a tuple of two labels of type `l`. The API also has getter and setter functions for the current label and clearance. It is important to note that the label passed into the setters must satisfy the condition, currentLabel $\sqsubseteq$ newLabel $\sqsubseteq$ clearance, in order to preserve the security of the label system.

LIO has a special mechanism which allows the program to perform a computation without raising the current value. The restriction is that the result of the computation cannot be directly returned. Instead, a `Labeled` object is returned in place of the result. The `Labeled` object remembers the result and the current label at the end of the computation. When the result is extracted from the `Labeled`

```
data Label l => Labeled l a

label :: Label l => l -> a -> LIO l (Labeled l a)
unlabel :: Label l => Labeled l a -> LIO l a
labelOf :: Label l => Labeled l a -> l

toLabeled ::Label l => l -> LIO l a -> LIO l (Labeled l a)
```

Figure 2.6: API for `Labeled` values.

object, the current label is tainted by the label of the computation. This allows a program to delay tainting of the current label.

The API associated with `Labeled` values is found in Figure 2.6. A key feature is that the constructor for `Labeled` values is not exported, so the contents of `Labeled` values cannot be read or modified. `label` and `unlabel` box and unbox `Labeled` values with appropriate IFC checks. `labelOf` retrieves the label of the boxed value. The interesting case is `toLabeled`, which allows the program to perform an LIO computation that delays raising the current label. This is safe since the result is boxed and cannot be read without calling `unlabel`, which would taint the current label appropriately.

Chapter 3: LMonad

The goal of this work is to support IFC in Yesod, and LIO already provides IFC for Haskell, so one might wonder why this work does not just use LIO. The primary reason is that LIO has limited compositionality. LIO runs on top of the IO monad, but Yesod mainly uses the Handler monad. As a result, one would have to rewrite all of Yesod's functionality on top of LIO.

LMonad provides a solution to the compositionality problem by creating a monad transformer. This monad transformer can be run over any existing monadic computation. Thus developers can utilize existing code and libraries, like Yesod, while enforcing IFC policies.

Another reason this work does not use LIO is that LIO only provides monadic lifting through the trusted function `ioTCB`. This means that only trusted code is able to call this function; this design decision prevents regular code from performing computations in the IO monad. As a result, programs using LIO are able to build secure systems from the ground up by selectively whitelisting code that does not circumvent IFC checks. Unfortunately, this whitelisting approach can be too restrictive and cumbersome for developers who wish to use existing libraries.

To address this issue, LMonad provides the option to allow more of a blacklist-

```
class Monad m => LMonad m where
    lFail :: m a
    lAllowLift :: m Bool

data (Label l, Monad m, LMonad m) =>
    LMonadT l m a = LMonadT (StateT (l, l) m a)
```

Figure 3.1: The `LMonad` typeclass, and the `LMonadT` transformer used to add IFC to any monad.

ing philosophy. When the developer allows it, arbitrary computations on the inner monad are allowed to be lifted. In this situation, the programmer is responsible for hiding code that could potentially circumvent IFC checks. This seems reasonable in Yesod applications since most code is imported from the `Import` module; the programmer should be able to replace imports of Yesod's database functions with LMonad's in a single location. Developers could also use automated tools to ensure that blacklisted code is not used. For instance, a simple script could use `grep` to detect when modules that go around IFC safety checks are imported.

3.1 Design and Implementation

Here we highlight some of the design and implementation of LMonad. LMonad itself is a generalization of LIO as it works for any monad that implements its typeclass, not just IO. Figure 3.1 shows this typeclass. `lFail` defines what to do when a policy violation occurs. `lAllowLift` tells LMonad whether it should allow arbitrary functions to be lifted, thereby making them available in LMonad computations. This function is how the programmer can choose either the whitelisting or blacklisting approach.

```
instance LMonad IO where
        lFail = exitFailure
        lAllowLift = return False

type LIO l a = LMonadT l IO a
```

Figure 3.2: LIO implemented in LMonad.

LMonad offers the transformer `LMonadT` to add IFC onto any monad. Its definition can be seen in Figure 3.1. The datatype `LMonadT` has type parameters `l`, `m`, and `a`. These types respectively correspond to the computation's label, the inner monad, and the computation's return type. The label type must be an instance of the `Label` typeclass, and the inner monad must be instances of the `Monad` and `LMonad` typeclasses. The `LMonadT` constructor is a wrapper around a state monad transformer (`StateT`), which keeps track of the current label and clearance label. `LMonadT`'s constructor is not exported so untrusted code cannot modify the current label and clearance label.

Figure 3.2 demonstrates how LIO can be implemented in LMonad, which shows that LMonad is a generalization of LIO. Specifically, IO is made an instance of `LMonad` where lifting is not allowed. Then LIO is defined as a type alias of `LMonadT` with the inner monad set to IO.

LMonad's API is very similar to LIO's, but there are a few differences that make LMonad easier to use. These changes still preserve the security properties demonstrated by LIO. As seen in Figure 3.3, the `Label` typeclass now requires instances to implement `bottom` ($\bot$) from the security lattice. This is now required due to a change in `runLMonad`. This change is a design choice that initially sets

```
class (Eq l) => Label l where
    leq :: l -> l -> Bool
    lub :: l -> l -> l
    glb :: l -> l -> l
    bottom :: l

runLMonad :: (Label l, LMonad m) =>
    LMonadT l m a -> m a

raiseClearanceTCB :: (Label l, LMonad m) =>
    l -> LMonadT l m ()

toLabeledTCB :: (Label l, LMonad m) =>
    l -> LMonadT l m a -> LMonadT l m (Labeled l a)
```

Figure 3.3: Differences in LMonad's API.

```
main = runLMonad $ do                    -- bottom
    handler <- getHandler "doc.txt"      -- bottom
    printDocument handler                -- bottom `join` LD
```

Figure 3.4: An example LMonad program that opens and prints a document. The current label is shown in the comments, and `runLMonad` initially sets the current label to `bottom`.

both the current label and clearance to bottom. Figure 3.4 shows an example of how to use `runLMonad` in an application to track IFC. The comments show the current label after every statement. Unlike LIO, this application does not need to explicitly specify an initial current label and clearance since `runLMonad` sets them both to bottom.

To set the clearance label, the trusted function `raiseClearanceTCB` is now provided. This function will lift the clearance to the join of the old clearance and the given label. This function is convenient when developers need to assign clearances multiple times, like when there are multiple types of principals. An example of how to use `raiseClearanceTCB` to set the clearance is found in Figure 3.5. The

```
setUserClearance = do          -- C0
    user <- getCurrentUser     -- C0
    raiseClearanceTCB
        (UserLabel user)       -- C0 `join` (UserLabel user)
```

Figure 3.5: An example trusted function which sets the clearance based on the current user. The clearance label is shown after every statement.

comments show the clearance label after every statement, and the initial clearance is `C0`. The trusted function `setUserClearance` sets the clearance label by calling `raiseClearanceTCB` with the current user's label. The resulting clearance is the join of the initial clearance and the current user's label.

`toLabeledTCB` is now a trusted function as it raises the clearance while computing `Labeled` values. Since we potentially allow lifting, the underlying monad could leak information to an external source. All trusted functions are in a separate module though, so trusted code must import this module explicitly. `toLabeledTCB` is used extensively by LMonad's database functions to delay reading results from the database and tainting the current label.

For convenience, LMonad implements two labeling systems. The first is called `PSLabel` and is the powerset of all principals. Another is `DCLabel`, which LIO also provides. Disjunction category labels were introduced by Stefan *et al.* [9]. They are represented as boolean formulas of principals in conjunctive normal form. In other words, they are *ands* of *ors* of principals. The security lattice is implemented for both of these labels. Now developers can use these labels in their LMonad programs by defining an appropriate principal datatype. An example of this is later shown in Figure 4.3.

Chapter 4: Integrating LMonad with Yesod

The high level goals of integrating LMonad with Yesod are to provide a simple mechanism to define IFC policies and to ensure that all database interactions respect these policies. Yesod has an existing DSL to define database models. IFC policies should be defined by making label annotations to the field definitions in this DSL. Database functionality provided by LMonad should also automatically insert IFC checks. This chapter details how these goals are accomplished.

4.1 Integrating an example website

Our efforts to integrate LMonad with Yesod are guided by a simple example website where users can create accounts, log in, view profiles, and update personal

```
getProfileR :: UserId -> Handler Html
getProfileR uId = do
    user <- runDB $ get uId
    defaultLayout $ do
        [whamlet|
            <p>
                Username: #{userIdent user}
            <p>
                Email: #{userEmail user}
        |]
```

Figure 4.1: An example website that violates policy.

```
import Yesod as Import hiding (
    ...
    runDB,          -- database functions
    get,
    update,
    ...,
    maybeAuthId,    -- authentication functions
    requireAuthId
    )
```

Figure 4.2: Hiding functions that could circumvent IFC checks.

information. In this example website, we want to enforce a policy where users can only read or modify their personal information (like their email address) and administrators can read, but not modify, anyones private information. Figure 4.1 depicts a simple webpage that violates this policy. This webpage takes a user's identifier as an argument, which Yesod provides by parsing the URL. It then does a database query to retrieve the user entity that corresponds to the given identifier. Finally, it generates HTML that displays the user's username and email address. This webpage violates the policy because it never checks whether the user is logged in before displaying the user's email address on the page. Any public visitor to the website will see the email address, which is a clear violation of the privacy policy. Integrating LMonad with Yesod can prevent this policy violation.

The first step to integrating LMonad in the application is to blacklist all functions that could circumvent IFC checks. This is simple to do in the Yesod application by hiding all the database functions from the `Import` module. This is demonstrated in Figure 4.2. Authentication functions are also hidden as a matter of convenience, as they can be replaced by trusted code that raises the current user's clearance.

```
data Principal =
      PrincipalAdmin
    | PrincipalUser UserId

type MyLabel = DCLabel Principal
```

Figure 4.3: Defining labels as DCLabels where principals are users and administrators.

The next step is to define an appropriate label to enforce the policy. For the example website, we use the DCLabels to keep track of confidentiality and integrity. Here, principals are administrators and all the users, as seen in Figure 4.3.

After choosing a labeling system, functionality should be added to set the clearance label when a user is logged into the site. It can be convient to add this functionality to the authentication functions since the authentication functions should already be used where privileged tasks occur. If setting the clearance is provided by another function, calls to this function would have to be added throughout the application's code, resulting in significantly more work for the developer. An example of these definitions are shown in Figure 4.4. The trusted function, `raiseUserLabel'`, creates a label corresponding to the given user. It also joins in the administrator label if the user is an adminstrator. Finally, it raises the clearance level to that of the computed label. Note that this function is not exported by the module for external use.

Entities are the Haskell data types that represent tables from the database, similar to an object relational mapping (ORM). For example, there is a `User` entity that is a record with fields for the user's username, password, and email address.

To continue integration, the developer needs to define how to derive labels

```
maybeAuth = ...
requireAuth = do
    user <- lLift Auth.requireAuth
    raiseUserLabel' user
    return user

raiseUserLabel' (Entity userId user) =
    let label' = dcSingleton $ PrincipalUser userId in
    let label = if userAdmin user then
            lub label' $ dcSingleton PrincipalAdmin
          else
            label'
    in
    raiseClearanceTCB label
```

Figure 4.4: The redefined authentication functions now also raise the clearance label when a user is logged in.

on reads, writes, and creations for every database entity. This is done by creating instances of the typeclass `LEntity`, which is displayed in Figure 4.5. The figure also shows an example instance implementation for the `User` table. Since either an administrator or the owner can read the user's data, the meet of the admin and user's confidentiality labels is returned. Similar functionality is implemented for writes and creations.

The final step in integration is to replace all database functions with ones provided by LMonad's Yesod library. We have already hidden Yesod's database functions, so we can easily replace them by exporting `Database.LPersist` in the `Import` module. These database function names are overloaded with Yesod's database functions so no more code changes are necessary. The provided database functions call the previously defined `LEntity` functions so that the current label rises on database reads, writes, and creations. An example of one of these library functions is shown

```
class Label l => LEntity l e where
    getLabelRead :: Entity e -> l
    getLabelWrite :: Entity e -> l
    getLabelCreate :: e -> l

instance LEntity MyLabel User where
    getLabelRead (Entity id user) = glb
        (dcConfidentialityLabel PrincipalAdmin)
        (dcConfidentialityLabel $ PrincipalUser id)
    getLabelWrite (Entity id user) = dcIntegrityLabel $
        PrincipalUser id
    getLabelCreate user = bottom
```

Figure 4.5: The `LEntity` typeclass which defines how to raise the current label on database interactions for an entity, and an instance of the `LEntity` typeclass for the `User` table.

```
import Database.LPersist as Import
```

Figure 4.6: Export of `Database.LPersist` to use LMonad's library functions that check IFC policies.

in Figure 4.7. The `get` function retrieves a result from the database and raises the current label when a result is returned.

Once integration is complete, we can see how to fix the policy violation from the original example in Figure 4.1 by using LMonad. Figure 4.8 shows the fixes. Specifically, the developer must indicate that the computation should run in LMonad by calling `runLMonad`; a call is made to `raiseUserLabel` to raise the logged in user's

```
get key = do
    res <- Persist.get key
    whenJust res $ lift .
        taintLabel . getLabelRead . (Entity key)
    return res
```

Figure 4.7: The replaced `get` function that now correctly raises the label by calling `getLabelRead` and `taintLabel`.

```
getProfileR :: UserId -> Handler Html
getProfileR uId = runLMonad $ do
    raiseUserLabel
    user <- runDB $ get uId
    defaultLayout $ do
        setTitle "Example␣webpage"
        [whamlet|
            <p>
                Username: #{userIdent user}
            <p>
                Email: #{userEmail user}
        |]
```

Figure 4.8: The fixed profile page example that no longer has a policy violation.

clearance; the redefined `runDB` and `get` are called which redirects the client's browser to an error page if the user does not have permission to view the entity returned by the database.

4.2 Simplifying Integration

At first glance, it appears as though it is a lot of work for developers to integrate LMonad with a Yesod application. Even worse is the fact that the developer could make a mistake during integration and fail to enforce an IFC policy. Ideally, the developer would only need to integrate a small, trusted amount of code, which could be easily verified. Fortunately, LMonad solves this problem because a lot of the integration can be automated. The library will generate code that enforces the IFC policies correctly. This leaves the developer to make cosmetic changes and focus on the small trusted computing base.

```
C = '<' L ',' L ',' L '>'
L = K | '_'
K = A '||' K | A
A = 'Id' | 'Const' name | 'Field' name
```

Figure 4.9: Grammar for label annotations used to define IFC policies. Each slot corresponds to the permissions required to read, write, and create a field, respectively. Keywords allow the expression of which permissions are required.

```
User
    ident Text
    password Text
    email Text < Const Admin || Id, Id, _ >
    admin Bool
```

Figure 4.10: An example of the modified DSL to specify IFC policies for the `email` field.

4.2.1 Label Annotations

Programmers can define IFC policies by manually creating LEntity instances for each table (or entity) in their database models. This presents an implementation burden to developers, and there is no assurance that they are accurately implementing the policies. LMonad addresses this with label annotations, which are a straightforward means to define IFC policies.

Yesod already provides a model DSL to define database schemas and each table's corresponding Haskell data type (called entities). LMonad modifies this DSL so that IFC policies can be defined in a simple manner. After each field definition in a database entity, the developer has the option of including a label annotation to indicate how the current label should rise on database reads, writes, and creations. Keywords are used to indicate different policy dependencies. `Id` means the current

label should be tainted by the current entity's identifier. `Const` means the current label should be tainted by a constant label. `Field` means the current label should be tainted by another field of a given name in that entity. An underscore, `_`, means that the current label should not change for the given action. Multiple label annotations can be combined with `||`, which means that either label has permission to do a specificied action. The grammar used to annotate labels is shown in Figure 4.9.

Figure 4.10 shows an example of how the developer can take advantage of the modified DSL to define IFC policies. After the `email` field's definition, optional angled brackets indicate a label annotation. Each slot corresponds to required permissions on read, write, and creation of the `email` field. This example policy states that the user must be an administrator or owner to read the `email` field. Users can only modify their own email addresses, and anyone can write to the `email` field while creating a new entity.

LMonad implements the IFC policies by automatically generating code that taints the current label for each field in the entity and creates a `LEntity` instance for the entity. This is done through the use of Template Haskell which manipulates the AST during compilation. This automates the process of writing the integration code previously seen in Figure 4.5.

To use label annotations, the developer only needs to define a mapping from a principal's type to its label. This can be done by creating an instance of `ToLabel` for each principal type. This is straightforward and is demonstrated in Figure 4.11. The mapping for `Const` labels is given by `String` instances.

Through the use of label annotations, developers can easily define IFC policies

```
class ToLabel t l where
    toConfidentialityLabel :: t -> l
    toIntegrityLabel :: t -> l

instance ToLabel (Key User) MyLabel where
    toConfidentialityLabel uId = dcConfidentialitySingleton $
        PrincipalUser uId
    toIntegrityLabel uId = dcIntegritySingleton $
        PrincipalUser uId
instance ToLabel String MyLabel where
    toConfidentialityLabel "Admin" =
        dcConfidentialitySingleton PrincipalAdmin
    toConfidentialityLabel _ = error "ToLabel: Invalid string"
    toIntegrityLabel "Admin" =
        dcIntegritySingleton PrincipalAdmin
    toIntegrityLabel _ = error "ToLabel: Invalid string"
```

Figure 4.11: The `ToLabel` typeclass creates a mapping from principals to labels. Instances of this typeclass are also shown for the user and administrator principals.

on their database models. These policies should be straighforward to understand, and the programmer can now have confidence that the policies are implemented correctly.

As a side note, the only restriction on label annotations is that `Id` cannot appear in the creation annotation. When the entity is being created, its identifier does not exist yet so you cannot have policies that depend on the identifier.

4.3 Protected Entities

The `LEntity` typeclass successfully enforces IFC policies, but it can be too conservative in certain cases. For example, consider a modification of the profile page that only shows public information to other users (Figure 4.12). The page will always redirect to the permission denied page if the user is not logged in as the user

```
getProfileR :: UserId -> Handler Html
getProfileR uId = runLMonad $ do
    authId <- maybeAuthId
    user <- runDB $ get uId
    let dispEmail = if (Just uId) == authId then
            [whamlet|
                <p>
                    Email: #{userEmail user}
            |]
          else
            mempty
    defaultLayout $ do
        setTitle "Example webpage"
        [whamlet|
            <p>
                Username: #{userIdent user}
            ^{dispEmail}
        |]
```

Figure 4.12: `get` is too eager in enforcing IFC so this page will always dispay a permission denied error for the public.

whose profile is being viewed. This happens since the call to `get` taints the current label by the join of all the fields in the returned entity. This is too conservative because the user's identity is public and anyone should be able to read it, but there is no way for them to read the identity. In this case, `get` is too eager to taint the `email` field's label, even though the program does not read that field for public users.

The solution to this problem is to introduce protected entities. Protected entities are a class of datatypes that allow a program to delay tainting the current label. They are similar to regular entities, but any field with an IFC policy is boxed into a `Labeled` value. Now the current label will only rise when `Labeled` fields are unboxed.

```
data ProtectedUser = ProtectedUser {
          pUserIdent :: Text
        , pUserPassword :: Text
        , pUserEmail :: Labeled MyLabel Text
    }

class ProtectedEntity e p | e -> p where
    toProtected :: LMonad m => Entity e -> LMonadT l m p
```

Figure 4.13: The protected entity version of `User`, and the `ProtectedEntity` typeclass that `ProtectedUser` must create an instance of.

In order to easily ensure that the security of IFC policies are maintained, each entity needs a corresponding protected entity data type. For example, Figure 4.13 shows the protected entity version of `User`. Protected entities must also be instances of the typeclass `ProtectedEntity` which maps each entity to its corresponding protected entity. These instances typically will make use of `toLabeledTCB` to box protected fields. To utilize protected entities, LMonad provides modified database functions, like `pGet` (Figure 4.14), to retrieve the protected entities. This is possible because the library functions can convert database results into protected entities by calling functions provided by the `ProtectedEntity` typeclass.

LMonad can autogenerate the code that defines protected entities and their `ProtectedEntity` instances by using Template Haskell and label annotations. Once again, developers do not need to manually define new data types and label checks. They can be confident that security policies are correctly enforced by the generated code.

With protected entities, we can fix the profile webpage example so that anyone can read the user's identity. The fixed version is found in Figure 4.15. As one can

```
pGet key = do
    res <- lLift $ Persist.runDB $ Persist.get key
    maybe (return Nothing) handler res
    where
        handler val = do
            protected <- toProtected $ Entity key val
            return $ Just protected
```

Figure 4.14: The `pGet` function to get the protected version of an entity from the database.

see, we now use `pGet` to retrieve the protected user entity. A nice feature of protected entities is that they enable the use of the floating label system itself to check whether the clearance gives us permission to unbox protected fields. An example of this can be seen in the protected profile page, where we use `canUnlabel` to determine whether to unbox and display the email address.

4.4 Expressive Database Queries

Many real world web applications need more expressive database queries than the simple ones we have seen so far. Haskell already has a nice library, called Esqueleto, which allows developers to perform richer queries in a type safe manner. For LMonad to be feasible for real world applications, these kinds of queries need to make IFC checks as well. This is problematic, though, because there can be many of these queries spread throughout a Yesod application. Each of these queries would need to be audited to make sure that they make the proper IFC checks. This would need to be redone everytime a policy changes as well. Furthermore, it is not always obvious how to uphold the IFC policies for complex database queries.

```
getProfileR :: UserId -> Handler Html
getProfileR uId = runLMonad $ do
    raiseUserLabel
    user <- runDB $ pGet uId
    canReadEmail <- canUnlabel $ pUserEmail user
    dispEmail <- if canReadEmail then do
            email' <- unlabel $ pUserEmail user
            return $ [whamlet|
                <p>
                    Email: #{email'}
                |]
        else
            return mempty
    defaultLayout $ do
        setTitle "Example webpage"
        [whamlet|
            <p>
                Username: #{userIdent user}
            ^{dispEmail}
        |]
```

Figure 4.15: The fixed version of the profile page making use of protected entities.

LMonad solves this problem by providing another DSL called LSQL, which is a subset of SQL. The grammar for LSQL is found in Figure 4.16. Developers can use LSQL to make more expressive SQL queries in a safe manner. A few notable features that it enables are inner joins, outer joins, conditionals, ordering, limits, offsets, and antiquotation of Haskell expressions. Under the hood, LSQL uses Esqueleto, which is a Haskell library that provides type safe SQL queries.

To ensure that all the IFC policies are enforced, the generation code extracts all of the `TERM`s in the LSQL statement. For all of the `TERM`s found, their label's dependencies are included in the requested fields of the select statement. The LSQL statement is converted into Esqueleto code, so the SQL queries generated are still type checked and protected against injection attacks. LMonad also generates IFC policy

```
STMT = CMD | CMD ';'
CMD = SELECT TERMS 'FROM' TABLES WHERE ORDERBY LIMIT OFFSET
SELECT = 'SELECT' | 'PSELECT'

TABLES = table | TABLES JOIN table 'ON' BEXPR
JOIN = 'INNER JOIN' | 'OUTER JOIN' | 'LEFT OUTER JOIN'
    | 'RIGHT OUTER JOIN' | 'FULL OUTER JOIN'

BEXPR = '(' BEXPR ')' | 'NOT' BEXPR | TERM 'IS NULL'
    | TERM 'IS NOT NULL' | B BBINOP B | BEXPR BEBINOP BEXPR
BEBINOP = 'AND' | 'OR'
BBINOP = '==' | '>=' | '>' | '<=' | '<'
B = '#{' antiquote '}' | CONST | TERM
CONST = 'TRUE' | 'FALSE' | int | double | '\'' string '\''

TERMS = '*' | TERMSS
TERMSS = TERM | TERM ',' TERMSS
TERM = table '.' FIELD | FIELD
FIELD = '*' | field

WHERE = 'WHERE' BEXPR | NULL
LIMIT = 'LIMIT' nat | NULL
OFFSET = 'OFFSET' nat | NULL

ORDERBY = 'ORDER BY' ORDER
ORDER = TERM | TERM 'ASC' | TERM 'DESC' | TERM ',' ORDER
```

Figure 4.16: The LSQL grammar used for expressive SQL queries. This language is used to perform IFC safe queries to the database. Some notable features are the ability to use outer joins and the use of antiquotation to include the results of Haskell expressions in queries.

checks for all of the `TERM`s previously found in the LSQL statement. If `PSELECT` was given, LMonad will wrap the values returned by the database into `Labeled` values.

To demonstrate how LSQL works, Figure 4.17 shows a LSQL statement from the Build it Break it Fix it web application. Tables `User` and `UserInformation` contain private information about contestants so the current label needs to be tainted with the labels of results from the database. The figure also shows the code generated from the LSQL statement. This code first runs the corresponding Esqueleto database query. Then it maps over every row returned by the query, calling `raiseLabelRead` to taint the label for every result. Figure 4.18 provides another example and shows the corresponding generated code. This LSQL statement protects the judges' email addresses.

```
-- LSQL
select User.*, UserInformation.* from User
left outer join UserInformation
    on User.id == UserInformation.user
where User.id == #{uId}
limit 1

-- Generate code
res_0 <- select $ from $
    \(LeftOuterJoin _user _userinformation) -> do
        on (just (_user ^. UserId) ==.
            _userinformation ?. UserInformationUser)
        where_ (_user ^. UserId ==. val uId)
        limit 1
        return (_user, _userinformation)
lift $ mapM (\(_e_user@(Entity _user_id _user),
    _userinformation_maybe) -> do
        raiseLabelRead _e_user
        maybe (return ())
            raiseLabelRead _userinformation_maybe
        return (_e_user,_userinformation_maybe)
  ) res_0
```

Figure 4.17: LSQL statement from the Build it Break it Fix it website, and the code that is automatically generated. A lot is going on here, but the main takeaway is that the appropriate IFC checks are made on all the results returned by the database through the calls to `raiseLabelRead`.

```
-- LSQL
select User.email from User inner join Judge
on User.id == Judge.judge
where Judge.contest == #{cId}

-- Generated code
res_0 <- select $ from $ \(InnerJoin _user _judge) -> do
    on (_user ^. UserId ==. _judge ^. JudgeJudge)
    where_ (_judge ^. JudgeContest ==. val cId)
    return (_judge ^. JudgeContest, _judge ^. JudgeJudge,
        _user ^. UserId, _user ^. UserEmail)
lift $ mapM (\(_judge_contest, _judge_judge,
    _user_id, _user_email) -> do
        taintLabel (readLabelUserEmail' _user_id)
        return (_user_email)
  ) res_0
```

Figure 4.18: LSQL statement retrieving judges' emails and the corresponding generated code. The main point here is that proper IFC checks are inserted via `taintLabel`.

Chapter 5: Evaluation

I developed two web applications to evaluate the integration of LMonad into Yesod. The first is the example website that is referenced throughout the previous chapter. The other is the Build it Break it Fix it website. Each web application has a LMonad implementation and a vanilla implementation that does not automatically provide IFC checks.

Two methods are used to evaluate LMonad's integration. The first is an approximation of developer effort in integrating LMonad. The measure we use to estimate this is the number of lines of code changed in handler functions that respond to web requests. This measure does not include integration changes that are automatically provided by LMonad's DSL and library functions.

The second method used is a measure of the performance difference between the vanilla and LMonad implementations of the web applications. We measure average roundtrip latency times in seconds over 1,000 trials for key page handlers. We also compute the standard deviation for these results and the overhead of the LMonad versus vanilla implementations. Most of the requests were simple GET requests that retrieve content. There are a few POST requests that update state in the web application. All requests were measured using a bash script that utilized

Handler	LOC Changed	Total LOC in File
/home	2	30
/profile	6	37
/protected_profile	19	49
/register	8	51
/update_email	13	47

Table 5.1: The number of LOC changed in handler files while integrating LMonad into the example Yesod application.

Handler	Verb	Vanilla Latency		LMonad Latency		Overhead
		Mean (s)	SD (s)	Mean (s)	SD (s)	
/home	GET	0.0488	0.0015	0.0483	0.0009	-1.02%
/register	GET	0.0481	0.0007	0.0483	0.0013	0.42%
/profile	GET	0.0487	0.0013	0.0485	0.0009	0.41%
/protected_profile	GET	0.0489	0.0014	0.0490	0.0014	0.20%
/update_email	GET	0.0489	0.0014	0.0484	0.0011	-1.02%
/update_email	POST	0.0556	0.0020	0.0557	0.0095	0.18%

Table 5.2: Experimental results comparing latency times between the example website with and without LMonad. The average and standard deviation of latency times in seconds for 1,000 trials are shown.

the unix `time` application. `curl` was used to send the HTTP requests. Cookies and CSRF tokens were explicitly defined so that a user was logged into the site, and the user had sufficient permissions for all of the handlers. The server used for benchmarks was running Red Hat Enterprise Linux Server 6.5 with 24 2.2 GHz CPUs and 32 GB of RAM. All measurements were performed locally to eliminate network effects. PostgreSQL 9.3.5 was also run locally as the database backend.

5.1 Example Website

As a reminder, the example website has a policy that only administrators and users can read their email addresses. Also, users can only edit their own email addresses. Table 5.1 shows the numbers of lines of code changed to support LMonad for various handlers. Experimental results measuring latency for the vanilla and the LMonad example sites are found in Table 5.2. This experiment was set up as described in the previous section. The handlers investigated correspond to various pages on the site like the home page, the registration page, and the profile page. The latency measurements also benchmark the time taken to update the user's email address via the `POST` request.

5.2 Build it Break it Fix it Web Application

The Build it Break it Fix it web application is used to help run a secure programming contest. The website allows contestants to provide personal demographic information, receive contest announcements, view scores, and examine their submissions. It is a relatively large application with 7,077 lines of code and 80 modules.

When adding LMonad to the site, there were various policies that needed to be enforced. The first is similar to the example website where only users can read or modify their personal information. The exception to this is that administrators can read this information. A similar policy is enforced for defining which git repository is used to make contest submissions for teams. Another policy is that while announce-

```
postAddAnnouncementR :: Handler Html
postAddAnnouncementR = do
  ((res, widget), enctype) <- runFormPost postForm
  case res of
    ...
    FormSuccess (FormData title contest markdown draft) -> do
      addAnnouncement title contest draft markdown
      redirect AdminAnnouncementsR
```

Figure 5.1: Security violation that allows anyone to post announcements on the contest website. LMonad prevents this type of IFC violation.

ments are public to read, only administrators can modify or create announcements. One limitation encountered during integration was that four modules use aliasing, which is currently not supported in LSQL statements. As a result, these modules import blacklisted libraries and check IFC manually.

Adding IFC actually does prevent security violations from occuring in the contest website. For example, one bug that previously existed in the code was a missing check to make sure a user was an administrator before posting a new announcement. This allowed anyone to make contest announcements! The offending code of this example is shown in Figure 5.1. This function parses `POST` data and then adds a new announcement via `addAnnouncement` upon success. The user is never authenticated, so anyone can post new announcements and potentially deface the website. In the converted version of the website, LMonad prevents this and similar bugs with its IFC checks. When `addAnnouncement` makes the database call to insert a new post, the insertion will fail and redirect to a permission denied page.

According to `git`, 1,149 insertions were made to enable LMonad in the contest site, with 524 of those insertions occuring in handler code. Table 5.3 displays the

Handler	**LOC Changed**	**Total LOC in File**
/announcements	22	145
/announcement/update	11	190
/profile	14	136
/buildsubmissions	3	213
/buildsubmission	1	213
/breaksubmissions	26	254
/breaksubmission	7	254

Table 5.3: The number of LOC changed while integrating LMonad into the Build it Break it Fix it application.

Handler	**Verb**	**Vanilla Latency**		**LMonad Latency**		**Overhead**
		Mean (s)	**SD (s)**	**Mean (s)**	**SD (s)**	
/announcements	GET	0.0577	0.0025	0.0584	0.0023	1.21%
/announcement/update	POST	0.0543	0.0014	0.0551	0.0020	1.47%
/profile	GET	0.0523	0.0017	0.0528	0.0018	0.96%
/buildsubmissions	GET	0.0677	0.0024	0.0694	0.0025	2.51%
/buildsubmission	GET	0.0706	0.0020	0.0740	0.0020	4.82%
/breaksubmissions	GET	0.0633	0.0022	0.0650	0.0022	2.69%
/breaksubmission	GET	0.0584	0.0014	0.0608	0.0017	4.11%

Table 5.4: Mean and standard deviation of latency in seconds of 1,000 trials for the vanilla and the LMonad Build it Break it Fix it web application.

```
getParticipationBreakSubmissionsR :: TeamContestId
  -> Handler Html
getParticipationBreakSubmissionsR tcId = runLMonad $
  Participation.layout Participation.BreakSubmissions tcId $ do
    submissions <- handlerToWidget $ runDB $ [lsql|
      select BreakSubmission.*, Team.name from BreakSubmission
      inner join TeamContest on
        BreakSubmission.targetTeam == TeamContest.id
      inner join Team on TeamContest.team == Team.id
      where BreakSubmission.team == #{tcId}
      order by BreakSubmission.timestamp desc
      |]
    case submissions of
      ...
      _ ->
        let row (Entity sId s, Value target) = do
          ...
          time <- lLift $ lift $ displayTime $
            breakSubmissionTimestamp s
          return [whamlet'|
            ...
          |]
        in
        ...
```

Figure 5.2: Excerpt of changes made to the break submission page to support LMonad. Changes are in orange and deletions are in red.

number of changed lines of code for select handlers. Note that the code for the buildsubmissions and buildsubmission pages is in the same file; here LOC changed is split according to which page was modified. The same is true for the breaksubmissions and breaksubmission pages. Figure 5.2 demonstrates some of the changes made to support LMonad on the break submissions page. Adding `runLMonad` runs the computation in the `LMonadT` transformer to track IFC. A previous Esqueleto expression is converted to use the LSQL DSL. The rest of the changes are small type fixes that need to be made due to the conversion.

Table 5.4 compares the latency of both versions of the site. The handlers benchmarked correspond to displaying the announcements, updating an announcement, retrieving the user's profile with personal information, getting the list of a team's submissions, and viewing the results of a specific submission. The same experimental setup was used as before.

5.3 Analysis

The total number of lines that have changed indicate that it is not trivial to integrate LMonad since every handler needs modification. This is mainly because computation is run in a different monad so certain functions need to be lifted. Also, Esqueleto database queries need to be rewritten in LSQL. That being said, most modules required relatively few changes compared to their overall size. The type checker also guides the programmer to where changes need to be made. Therefore, we conclude that it is feasible for a developer to properly intregrate LMonad into a Yesod application with moderate effort.

Most of the handlers show little to no overhead between the vanilla and LMonad versions of the website. In fact, the greatest latency occurs in the `buildsubmission` handler, and that is only 4 milliseconds of slowdown. This is only an overhead of 4.82%. These results indicate that LMonad incurs a negligible performance hit.

While LMonad's overhead is small, it still exists. This overhead most likely comes from the IFC checks that LMonad makes for every result returned by the database. This hypothesis implies that overhead increases with more results re-

turned by the database. The benchmark data from the Build it Break it Fix it web application seems to support this theory. For example, the buildsubmission handler, which has the largest overhead, returns 30 results from the database in the benchmark. On the other hand, the profile page has the lowest overhead and only returns two database results. It is possible that other factors could also affect performance. For instance, certain database operations, like writing to existing rows, could incur greater overhead. This seems possible given the benchmark results because updating an existing announcement has 1.47% overhead, despite the fact that this handler only makes four reads and one write.

Chapter 6: Related Work

This work directly builds upon LIO [1], DCLabels [9], Yesod [6], and security lattices [8], which have previously been discussed.

There are many works that attempt to control how information flows within a program. Sabelfeld and Myers present a comprehensive survey paper in this area [10]. This work distinguishes between various ways that secure information can leak out of a program. It also discusses different mitigation techniques like a program counter based static type system.

JIF by Myers provides IFC for Java programs using a static type system [11]. Pottier and Simonet present Flow Caml,which is similar to JIF except it is designed for ML [12]. Flow Caml also uses a static type system, and information flow can be infered since type inference for ML is decidable.

Other lines of work are more similar to LMonad since they attempt to track information flow accross database boundaries. Schoepe *et al.* offer SeLINQ to accomplish this. They use a DSL in a language like F# to express database queries. A main distinction between LMonad and their work is that SeLINQ uses a static type system. SeLINQ's quatation language is also less expressive since it does not support outer joins.

SELinks is a programming language by Corcoran *et al.* designed to build secure web applications that interact with databases [13]. An interesting feature of SELinks is that policy enforcement is moved to the PostgreSQL database server when possible to improve performance. The language also uses dependent types to make sure IFC policies are checked. Again, SELinks statically enforces IFC which differs from LMonad's dynamic floating label system. LMonad also provides richer database queries through the LSQL DSL.

Chlipala's UrFlow provides a unique method for implementing information flow policies [14]. UrFlow statically checks policies for Ur/Web applications using symbolic execution and automated theorem proving. While it is convenient that UrFlow checks policies statically, UrFlow applications can take a much longer time to compile than LMonad applications.

Fabric by Liu *et al.* is a language and decentralized system to enforce IFC [15]. Fabric is different from LMonad and previously mentioned works since it no longer focuses on the interaction between a web application and its database. Instead, Fabric provides guarantees about how information is distributed amongst nodes in a network.

There are various labeling schemes that can be use to track information flow. Montagu *et al.* survey various information flow labels [16]. This work compares different labels in terms of their expressivenes. The authors develop a theoretical abstraction called label algebras to perform this comparison. Developers should use this work to help decide which label system to use with their LMonad applications.

Chapter 7: Conclusion

This work has presented LMonad, a generalization of LIO that supports IFC for arbitrary monads. It provides additional functionality to integrate LMonad with web applications written in Yesod. In particular, a DSL for label annotations is given to specify IFC policies. LSQL and LMonad's library functions also enforce IFC when interacting with a backend's database by automatically inserting policy checks.

Integrating LMonad with Yesod applications is feasible since the provided functionality and the DSL reduce the developer's work to a reasonable effort. Furthermore, the runtime performance of the integrated version of the website is comparable to vanilla implemenations.

LMonad is currently limited by the fact that LSQL does not support aliases in queries. In future work, the author plans to address this issue by adding aliases to LSQL. In addition, the author aims to formally prove that LMonad's interactions with databases satisfy IFC models.

Bibliography

[1] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. *SIGPLAN Not.*, 46(12):95–106, September 2011.

[2] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992.

[3] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[4] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[5] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[6] Yesod web framework for haskell. http://www.yesodweb.com/.

[7] Joseph A. Goguen and Jos Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy'82*, pages 11–20, 1982.

[8] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[9] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, NordSec'11, pages 223–239, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.

[11] Andrew 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, pages 228–241, New York, NY, USA, 1999. ACM.

[12] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[13] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 269–282, June 2009.

[14] Adam 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, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[15] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 321–334, New York, NY, USA, 2009. ACM.

[16] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 3–17, Washington, DC, USA, 2013. IEEE Computer Society.