A Haskell and Information Flow Control Approach to Safe Execution of Untrusted Web Applications

Deian Stefan
Stanford University
April 11, 2011

Joint work with David Mazières, Alejandro Russo, Daniel B. Giffin, David Terei, and John C. Mitchell
Outline

1 Untrusted code integration
- Motivation
- Example: the Facebook case
- An alternative approach
- Our approach: HAILS web framework
Motivation

- Increasing number of web sites support third-party apps
  
  Google Apps
  LinkedIn
  Facebook
  Salesforce
  Tumblr

- Apps require access to private information: email, contacts, profile information, documents, etc.

- Several issues arise when wishing to use such apps:
  - Users grant access blindly
  - No guarantees about the app’s use of data
  - Difficult for privacy-conscious providers to enforce policies with traditional access control
Example

The Facebook case

- Curry installs a social app:
Example
The Facebook case

- Curry installs a social app:

![](image1)

- Uses the app:

![](image2)
Example
The Facebook case

- Curry installs a social app:

  ![Social app installation]

- Howard receives notification:

  ![Notification]

- Uses the app:

  ![App usage]

Example: the Facebook case
Example

The Facebook case

- Curry installs a social app:

- Howard receives notification:

- Tries to read the message:

- Uses the app:
Example

The Facebook case

- Curry installs a social app:

- Howard receives notification:

- Tries to read the message:

- Uses the app:
Example
The Facebook case: design issues

- To install apps
  - must grant (excessive) access to app
  - must grant unnecessary access to app author
Example

The Facebook case: design issues

- To install apps
  → must grant (excessive) access to app
  → must grant unnecessary access to app author

- To read message generated through the app, must install it
  → no way to safeguard profile content
Example

The Facebook case: design issues

- To install apps
  - must grant (excessive) access to app
  - must grant unnecessary access to app author
- To read message generated through the app, must install it
  - no way to safeguard profile content
- Separate privacy settings already specify sharing rules:
Example

The Facebook case: design issues

- To install apps
  - must grant (excessive) access to app
  - must grant unnecessary access to app author

- To read message generated through the app, must install it
  - no way to safeguard profile content

- Separate privacy settings already specify sharing rules:

![Privacy settings screenshot]

Need a better approach that leverages these settings!
An alternative approach

- Reduce trust placed of individual apps
  - limit trust to service provider, e.g. Facebook
  - third-party apps are always untrusted
An alternative approach

- Reduce trust placed of individual apps
  - limit trust to service provider, e.g. Facebook
  - third-party apps are always untrusted

- Force apps to abide by privacy constraints of profile data
An alternative approach

- Reduce trust placed of individual apps
  - limit trust to service provider, e.g. Facebook
  - third-party apps are always untrusted
- Force apps to abide by privacy constraints of profile data

Example

If Howard specifies that only his friend may read his profile data:
- An app can disclose Howard’s profile information to friends
- An app cannot disclose the data to others (inc. app author)
An alternative approach

- Reduce trust placed of individual apps
  → limit trust to service provider, e.g. Facebook
  → third-party apps are always untrusted
- Force apps to abide by privacy constraints of profile data

**Example**

If Howard specifies that only his friend may read his profile data:
- An app can disclose Howard’s profile information to friends
- An app cannot disclose the data to others (inc. app author)

**HAILS bases security on information flow control!**
Our approach: HAILS web framework

**Haskell Automatic Information Labeling System**

- **Information flow control library**
  - Tracks and controls information propagation by providing apps an environment alternative to IO

- **Type-safe IFC-aware relational DB interface**
  - Infers database scheme and labeling rules from data model
  - Enforces IFC on database accesses

- **Simple IFC-aware web server**
  - Enforces IFC on app/browser communication
Access control through labels

An introduction

- Every piece of data in the system has a label
- Every running app has a label
- Labels are partially ordered by \( \sqsubseteq \) (“can flow to”)
- Example: App (labeled \( L_A \)) accesses profile table (labeled \( L_T \))
Access control through labels

An introduction

- Every piece of data in the system has a label
- Every running app has a label
- Labels are partially ordered by $\sqsubseteq$ ("can flow to")
- Example: App (labeled $L_A$) accesses profile table (labeled $L_T$)
  - Is it a query? Information flows from table to the app.
  - System enforces that $L_T \sqsubseteq L_A$. 
Access control through labels

An introduction

- Every piece of data in the system has a label
- Every running app has a label
- Labels are partially ordered by \( \sqsubseteq \) (“can flow to”)
- Example: App (labeled \( L_A \)) accesses profile table (labeled \( L_T \))
  - Is it a query? Information flows from table to the app. System enforces that \( L_T \sqsubseteq L_A \).
  - Is it an update? Information flows in both directions. System enforces that \( L_T \sqsubseteq L_A \) and \( L_A \sqsubseteq L_T \).
Transitivity and decentralization

\[ L_T \not\sqsubseteq L_B \]

\[ \sqsubseteq \text{ is a transitive relation} \]
Transitivity and decentralization

\[ \sqsubseteq \text{ is a transitive relation} \]
Transitivity and decentralization

- ⊑ is a transitive relation
- Every app has a set of privileges
Transitivity and decentralization

- $\sqsubseteq$ is a transitive relation
- Every app has a set of privileges
- Exercising privilege $p$ changes label requirements
  - A more permissive $\sqsubseteq_p$ ("can flow to given privilege $p"$) is used
  - $L_A \sqsubseteq_p L_B$ to read and additionally $L_B \sqsubseteq_p L_A$ to write
Transitivity and decentralization

- \( \sqsubseteq \) is a transitive relation
- Every app has a set of privileges
- Exercising privilege \( p \) changes label requirements
  - A more permissive \( \sqsubseteq_p \) ("can flow to given privilege \( p \)"") is used
  - \( L_A \sqsubseteq_p L_B \) to read and additionally \( L_B \sqsubseteq_p L_A \) to write
- **Idea:** Set labels so you know who has relevant privileges
Implementing IFC in Haskell

A taste of Haskell

- Strongly typed, pure, lazy functional language
Implementing IFC in Haskell

A taste of Haskell

- Strongly typed, pure, lazy functional language
- Has type-level distinction between side-effecting and non-side-effecting computations
  → the former has type encapsulated by IO
Implementing IFC in Haskell

A taste of Haskell

- Strongly typed, pure, lazy functional language
- Has type-level distinction between side-effecting and non-side-effecting computations
  → the former has type encapsulated by IO
- The type IO is a monad, implementing
  - return: encapsulates a pure value by type IO
  - bind: sequences multiple IO actions
Implementing IFC in Haskell

A taste of Haskell

- Strongly typed, pure, lazy functional language
- Has type-level distinction between side-effecting and non-side-effecting computations
  → the former has type encapsulated by `IO`
- The type `IO` is a *monad*, implementing
  - `return`: encapsulates a pure value by type `IO`
  - `bind`: sequences multiple `IO` actions
- Haskell provides special `do`-syntax for monads

**Example: read and print character using `IO`**

```
main = do {c ← getChar ; putChar c ;}
```
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for any monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for any monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”

- Idea: Define a labeled IO (LIO) monad
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for *any* monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”

- **Idea:** Define a labeled IO (LIO) monad

- Untrusted code:
  - can only bind LIO actions to other LIO actions
  - cannot directly execute IO actions → cannot perform arbitrary IO
  - provides the trusted code an LIO action to execute
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for any monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”

- **Idea:** Define a labeled IO (LIO) monad

- Trusted code:
  - can safely execute an untrusted LIO action without violating label restrictions
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for *any* monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”

- **Idea**: Define a labeled IO (LI0) monad

- Trusted code:
  - can safely execute an untrusted LI0 action without violating label restrictions
  - IFC violation attempts → exceptions
Implementing IFC in Haskell

Why Haskell?

- Haskell’s do-syntax works for any monad
  - IO is just one kind of monad; we can define our own
  - Effectively allows us to reprogram the meaning of “;”

- **Idea:** Define a labeled IO (LIO) monad

- Trusted code:
  - can safely execute an untrusted LIO action without violating label restrictions
  - IFC violation attempts → exceptions
  - can bind IO actions to LIO ones, but must keep track of labels
Implementing IFC in Haskell

Overview of LIO library

- LIO monad keeps track of a “current label” $L_{\text{cur}}$
  - LIO operations can fail if not allowed by $L_{\text{cur}}$
    - e.g., writing to a sink with label below $L_{\text{cur}}$
  - $L_{\text{cur}}$ can be raised to satisfy restrictions (up to current clearance)
    - e.g., reading from a source with label above $L_{\text{cur}}$
Implementing IFC in Haskell

Overview of LIO library

- LIO monad keeps track of a “current label” $L_{\text{cur}}$
  - LIO operations can fail if not allowed by $L_{\text{cur}}$
    - e.g., writing to a sink with label below $L_{\text{cur}}$
  - $L_{\text{cur}}$ can be raised to satisfy restrictions (up to current clearance)
    - e.g., reading from a source with label above $L_{\text{cur}}$

- Structures may contain heterogeneously labeled data
  - $L_{\text{refs}}$ are labeled values (with label $\not\equiv L_{\text{cur}}$)
  - $\text{openR}$ is used to get value from $L_{\text{ref}}$, raising $L_{\text{cur}}$ accordingly
  - $\text{closeR}$ is used to ‘seal’ and LIO computation
Implementing IFC in Haskell

Overview of LIO library

- LIO monad keeps track of a “current label” $L_{\text{cur}}$
  - LIO operations can fail if not allowed by $L_{\text{cur}}$
    - e.g., writing to a sink with label below $L_{\text{cur}}$
  - $L_{\text{cur}}$ can be raised to satisfy restrictions (up to current clearance)
    - e.g., reading from a source with label above $L_{\text{cur}}$

- Structures may contain heterogeneously labeled data
  - $\text{LRef}$s are labeled values (with label $\neq L_{\text{cur}}$)
  - $\text{openR}$ is used to get value from $\text{LRef}$, raising $L_{\text{cur}}$ accordingly
  - $\text{closeR}$ is used to ‘seal’ and LIO computation

- Library provides LIO versions of standard library functions
  - e.g., labeled mutable cells, labeled file system
Labeled type-safe DB interface

Motivation

- Relational DBs are a key component to many app designs
- A labeled DB interface enforces IFC when performing a DB access, e.g., query, insert, delete, etc.
- By construction, a type-safe DB interface prevents SQL-injections
Labeling database information

Automatic labeling of data

- Each table $T$ has a label $L_T$ and clearance $C_T$
- Each column $C$ has a labeling policy $f_C : \text{Record} \rightarrow \text{Label}$
- Label a DB cell by applying corresponding column policy to record

Example: policy of phoneNr column

Label: A set of secrecy and integrity categories
Category: A set of principals, e.g., “alice”
Flow: $\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle$ iff:
- secrecy categories of $S_2 \subseteq S_1$
- integrity categories of $I_1 \subseteq I_2$
Labeling database information

Automatic labeling of data

- Each table $T$ has a label $L_T$ and clearance $C_T$
- Each column $C$ has a labeling policy $f_C : \text{Record} \rightarrow \text{Label}$
- Label a DB cell by applying corresponding column policy to record

Example: policy of phoneNr column

Only user "owning" the row may update her phone number:

$$f_P(r) = \langle \{\}, \{[\sigma_{\text{username}}(r)]\} \rangle$$

For $r = (\text{username: "alice"}, \text{phoneNr: +441...})$

$$f_P(r) = \langle \{\}, \{[\text{"alice"}]\} \rangle$$
Implementing DB interface

SHaskellDB

- Extend *existing* HaskellDB library
  - Provides EDSL that leverages Haskell type-checker
  - Supports arbitrary back-end drivers, e.g., to MySQL, SQLite
  - Already used by the community
Implementing DB interface

SHaskellDB

- Extend *existing* HaskellDB library
  - Provides EDSL that leverages Haskell type-checker
  - Supports arbitrary back-end drivers, e.g., to MySQL, SQLite
  - Already used by the community

- Enforce IFC when executing a LIO DB action
  - E.g., when inserting, check: i) current label ⋑ table label, ii) current label ⋑ record label, and iii) record label ⋑ table clearance
Implementing DB interface

**SHaskellDB**

- **Extend existing** HaskellDB library
  - Provides EDSL that leverages Haskell type-checker
  - Supports arbitrary back-end drivers, e.g., to MySQL, SQLite
  - Already used by the community
- **Enforce IFC when executing a LIO DB action**
  - E.g., when inserting, check: i) current label $\sqsubseteq$ table label, ii) current label $\sqsubseteq$ record label, and iii) record label $\sqsubseteq$ table clearance
- **Allow for persistent storage of Lrefs into tables**
  - $\rightarrow$ can use label of Lref to create very fine-grained policies
Implementing DB interface

SHaskellDB

- Extend *existing* HaskellDB library
  - Provides EDSL that leverages Haskell type-checker
  - Supports arbitrary back-end drivers, e.g., to MySQL, SQLite
  - Already used by the community
- Enforce IFC when executing a LIO DB action
  - E.g., when inserting, check: i) current label $\subseteq$ table label, ii) current label $\subseteq$ record label, and iii) record label $\subseteq$ table clearance
- Allow for persistent storage of Lrefs into tables → can use label of Lref to create very fine-grained policies
- Provide LIO versions of HaskellDB’s functions
Safely executing untrusted apps with HAILS

\( \lambda \text{book: An example web site} \)

- **Role of \( \lambda \text{book} \)**
  - Provide app writers a simple API
    - Restricted versions of SHaskellDB functions
  - Authenticate users
  - Launch third-party apps in a container
    - Sanitize app content
    - Notify users of app IFC violations

- **Role of apps**
  - Run on behalf of a user
  - Have a set of principals and privileges
  - May access DB (e.g., profile table, or relationship table)
  - May access labeled form values and cookies
  - Provide \( \lambda \text{book} \) an LIO action that returns page content
Safely executing untrusted apps with HAILS

\( \lambda \) book container

- Apps may display content in 1 of 2 container locations
Safely executing untrusted apps with HAILS

λbook container

- Apps may display content in 1 of 2 container locations
- On behalf of *alice*:
  - A blogging app is running in left container location
  - A friend list app is running in right container location
Safely executing untrusted apps with HAILS

λbook app interaction design & example attacks

- Alice launches BlogApp
Safely executing untrusted apps with HAILS

\(\lambda\)book app interaction design & example attacks

- Alice launches \textit{BlogApp}
- BlogApp access the DB and sends response to Alice
Safely executing untrusted apps with HAILS

\( \lambda \) book app interaction design & example attacks

- Alice launches *BlogApp*
- *BlogApp* access the DB and sends response to Alice
- *BlogApp* attempts to leak Alice’s data to Bob’s browser
Safely executing untrusted apps with HAILS

\(\lambda\)book app interaction design & example attacks

- Alice launches *BlogApp*
- *BlogApp* access the DB and sends response to Alice
- *BlogApp* attempts to leak Alice’s data to Bob’s browser
- *BlogApp* attempts to insert forged data on behalf of Alice
Performance measurements

Microbenchmark and λBook app

- **Microbenchmarks:**

- **λBook blog app:** HAILS is roughly 12% slower than Apache+PHP equivalent
Performance measurements

Microbenchmarks & λbook app

- Microbenchmarks:

- λbook blog app: HAILS is roughly 12% slower than Apache+PHP equivalent

You need not sacrifice (much) performance for security!
Thanks you!
Questions?