Flexible Dynamic Information Flow Control in Haskell

Deian Stefan[1] Alejandro Russo[2]
John C. Mitchell[1] David Mazières[1]

[1] STANFORD UNIVERSITY [2] CHALMERS

Haskell'11

www.scs.stanford.edu/∼deian/lio

Motivation

- Complex systems are composed of many different modules
- Generally, difficult to asses quality of modules ⇒ bugs and malware are pervasive
- Current approaches to execute untrusted code are very limited

Motivation: A paper review system

Integrating untrusted plugins

Administrator functionality

- Add papers and users
- Assign reviewers
- Specify conflict of interest relationships

User functionality

- Read papers and read/write reviews
- Provide and execute (untrusted) plugins

Security Policy: *User in conflict with a paper should not be able to read the corresponding review.*

Motivation: A paper review system

Integrating untrusted plugins

Administrator functionality

- Add papers and users
- Assign reviewers
- Specify conflict of interest relationships

User functionality

- Read papers and read/write reviews
- Provide and execute (untrusted) plugins

Security Policy: *User in conflict with a paper should not be able to read the corresponding review.*

Motivation: A paper review system

Integrating untrusted plugins

Example third-party plugins

1. Online chat for discussing common reviews
2. Alternative user interface
3. PDF viewer with review annotations
4. . . .

Motivation: A paper review system

Integrating untrusted plugins

Challenge: *How do we safely integrate plugins?*

1. Limit plugins to pure computations

✗ Inflexible: may want to use references, file-system, etc.

2. Allow plugins to use IO library

✗ Insecure: can easily violate security policies

Motivation: A paper review system

Integrating untrusted plugins

Challenge: *How do we safely integrate plugins?*

Solution: New Labeled IO (LIO) library

- ✓ Secure: security policies enforced in end-to-end fashion
- ✓ Flexible: can access references, file-system, etc., using policy-enforcing API

Enforcing Security Policies

Common approach: policy specifies *what code can be executed*

- ✗ Requires reasoning about *every* line of code

Enforcing Security Policies

Common approach: policy specifies *what code can be executed*

- ✗ Requires reasoning about *every* line of code

Information flow control approach: policy specifies *where data can flow*

- ✓ *No* reasoning about plugin code necessary
  - ➥ Well- suited for executing untrusted code

Enforcing Security Policies

Common approach: policy specifies *what code can be executed*

- ✗ Requires reasoning about *every* line of code

Information flow control approach: policy specifies *where data can flow*

- ✓ *No* reasoning about plugin code necessary
  - ➥ Well- suited for executing untrusted code
- ✓ Natural way to specify policies
  - ▷ e.g., if Bob is in conflict with review R:
    policy ≡ *information from R cannot flow to Bob*

Enforcing Security Policies

Common approach: policy specifies *what code can be executed*

- ✗ Requires reasoning about *every* line of code

Information flow control approach: policy specifies *where data can flow*

- ✓ *No* reasoning about plugin code necessary
  - ➥ Well- suited for executing untrusted code
- ✓ Natural way to specify policies
  - ▷ e.g., if Bob is in conflict with review R: policy ≡ *information from* R *cannot flow to Bob*

➠ LIO is an IFC library!

Enforcing IFC With Labels

How do we track and control the flow of information?

- Every piece of data in the system has a label
  - e.g., review has label L_R
- Every computation has a labels $\sim$ behavior
  - e.g., plugin has label L_P
- Labels are partially ordered by $\sqsubseteq$ (*can flow to*) relation $\Rightarrow$ determines allowable flows

E.g., Plugin accesses a review.

Enforcing IFC With Labels

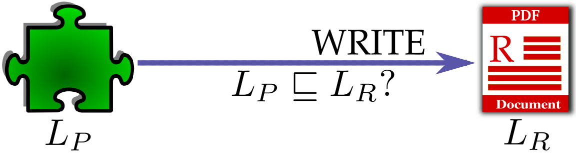
How do we track and control the flow of information?

- Every piece of data in the system has a label
  - e.g., review has label L_R
- Every computation has a labels $\sim$ behavior
  - e.g., plugin has label L_P
- Labels are partially ordered by $\sqsubseteq$ (*can flow to*) relation $\Rightarrow$ determines allowable flows

E.g., READ is a flow from review to plugin.

Enforcing IFC With Labels

How do we track and control the flow of information?



- Every piece of data in the system has a label
  - e.g., review has label L_R
- Every computation has a labels ∼ behavior
  - e.g., plugin has label L_P
- Labels are partially ordered by $\sqsubseteq$ (*can flow to*) relation ⇒ determines allowable flows

E.g., WRITE is a flow from plugin to review.

Reasoning About Policy Enforcement

Transitivity of $\sqsubseteq$ relation

How do labels help enforce security policies?

Reasoning About Policy Enforcement

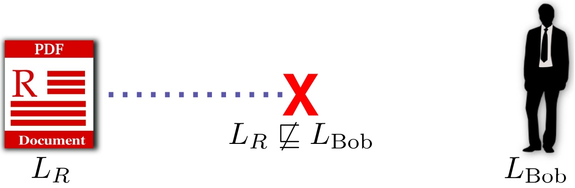
Transitivity of $\sqsubseteq$ relation

How do labels help enforce security policies?

➥ Labels impose restrictions on flow of data.

Reasoning About Policy Enforcement

Transitivity of $\sqsubseteq$ relation

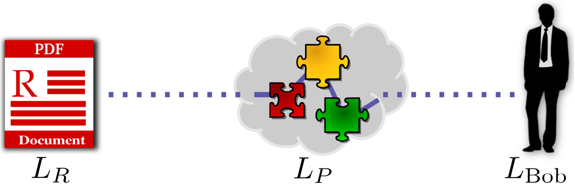


E.g., Label review so it cannot flow to Bob

➥ Label policy enforced end-to-end

Reasoning About Policy Enforcement

Transitivity of $\sqsubseteq$ relation

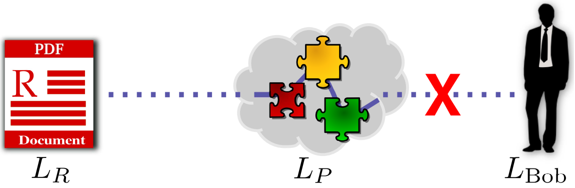


E.g., Even if there are many paths from R to Bob

➥ There is no label L_P such that $L_R \sqsubseteq L_P \sqsubseteq L_{\text{Bob}}$

Reasoning About Policy Enforcement

Transitivity of $\sqsubseteq$ relation

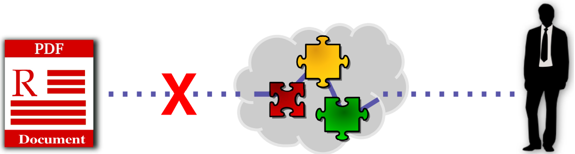


E.g., Even if there are many paths from R to Bob

➥ There is no label L_P such that $L_R \sqsubseteq L_P \sqsubseteq L_{\text{Bob}}$

Reasoning About Policy Enforcement

Transitivity of $\sqsubseteq$ relation



L_R L_P L_{Bob}

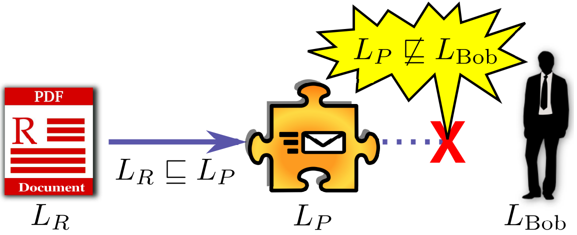
E.g., Even if there are many paths from R to Bob

➥ There is no label L_P such that $L_R \sqsubseteq L_P \sqsubseteq L_{\text{Bob}}$

Decentralized IFC

E.g., Suppose program chair wants to send *results*, once the review process is over

➥ He cannot send result to Bob: $\sqsubseteq$ is too strict

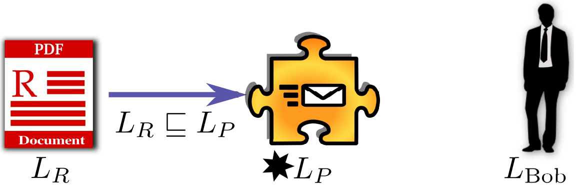


- A computation may employ privileges ($\star$) to bypass certain flow restrictions with $\sqsubseteq_\star$

Decentralized IFC

E.g., Suppose program chair wants to send *results*, once the review process is over

➥ He cannot send result to Bob: $\sqsubseteq$ is too strict

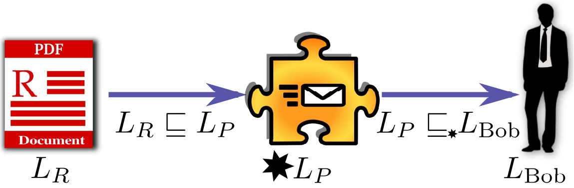


- A computation may employ privileges ($\star$) to bypass certain flow restrictions with $\sqsubseteq_\star$

Decentralized IFC

E.g., Suppose program chair wants to send *results*, once the review process is over

➥ He cannot send result to Bob: $\sqsubseteq$ is too strict



- A computation may employ privileges ($\star$) to bypass certain flow restrictions with $\sqsubseteq_\star$

The Right Language for DIFC

- Difficult to do DIFC as a library
  - ➥ Usually requires modifying language
- Haskell is a *natural* fit for IFC
  - Type-level distinction between pure and side-effecting code $\Rightarrow$ can control side-effects
  - Monad transformers $\Rightarrow$ can associate labels with computations

The Right Language for DIFC

- Difficult to do DIFC as a library
  - ➥ Usually requires modifying language
- Haskell is a *natural* fit for IFC
  - Type-level distinction between pure and side-effecting code ⇒ can control side-effects
  - Monad transformers ⇒ can associate labels with computations
- Haskell is *almost* perfect
  - ✗ Issue: `unsafe*` to break type system

The Right Language for DIFC

- Difficult to do DIFC as a library
  - ➥ Usually requires modifying language
- Haskell is a *natural* fit for IFC
  - Type-level distinction between pure and side-effecting code ⇒ can control side-effects
  - Monad transformers ⇒ can associate labels with computations
- Haskell is *almost* perfect
  - ✗ Issue: `unsafe`$*$ to break type system
  - ✓ Addressed by SafeHaskell (see D. Terei's talk)

LIO Overview

How do we implement an IFC library in Haskell?

Idea: Taint computation when reading sensitive data, and prevent it writing to public channels

- LIO monad used in enforcing IFC:

```haskell
newtype LIO l a = LIO (StateT l IO a)
```

- Monad keeps track of a *floating* label L_{cur}
  - can read object O if $L_O \sqsubseteq L_{\text{cur}}$
  - can raise L_{cur} to join $L_{\text{cur}} \sqcup L_O$ if $L_O \not\sqsubseteq L_{\text{cur}}$
  - can write/create object O if $L_{\text{cur}} \sqsubseteq L_O$
- Primitives enforce IFC & adjust L_{cur}

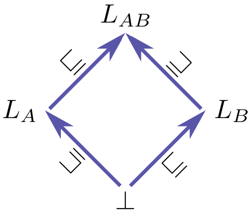
LIO Overview

An example: plugin reading reviews

```
R_A ← newLIORef L_A "..."
```

⋮

```
myPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  return (a,b)
```



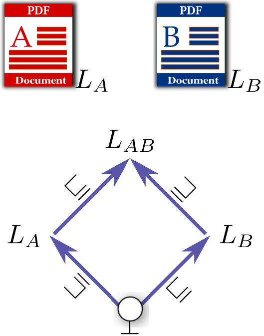
LIO Overview

An example: plugin reading reviews

$R_A \leftarrow$ `newLIORef` L_A "..."

⋮

myPlugin = **do**
  $a \leftarrow$ `readLIORef` R_A
  $b \leftarrow$ `readLIORef` R_B
  `return` (a, b)



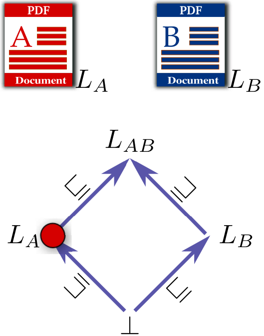
LIO Overview

An example: plugin reading reviews

$R_A \leftarrow$ `newLIORef` L_A "..."

⋮

```
myPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  return (a,b)
```



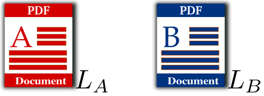
LIO Overview

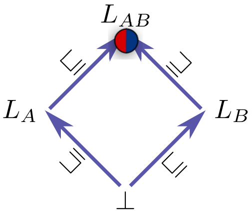
An example: plugin reading reviews

$R_A \leftarrow$ `newLIORef` L_A "..."

⋮

myPlugin = **do**
  $a \leftarrow$ `readLIORef` R_A
  $b \leftarrow$ `readLIORef` R_B
  `return` (a, b)





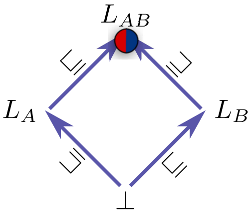
LIO Overview

An example: plugin reading reviews

```
R_A ← newLIORef L_A "..."
```

⋮

```
myPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  return (a,b)
```



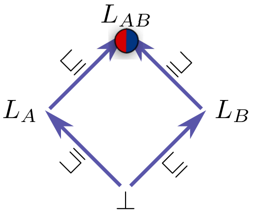
LIO Overview

An example: plugin reading reviews

$R_A \leftarrow$ `newLIORef` L_A "..."

$\vdots$

```
myPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  return (a,b)
```



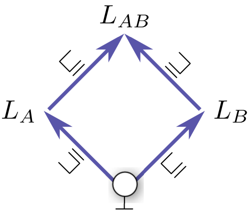
How does LIO differ from other language-level systems?

LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B


```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



LIO Overview

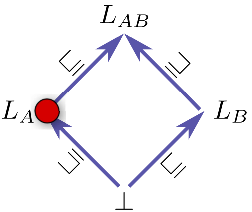
An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B

L_A

L_B

```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```

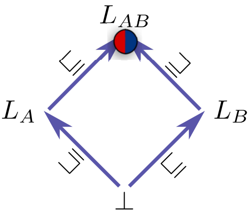


LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B

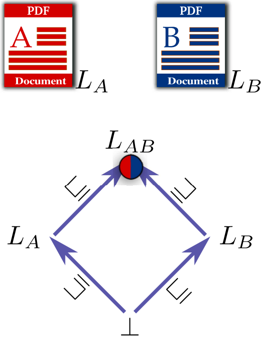

```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B



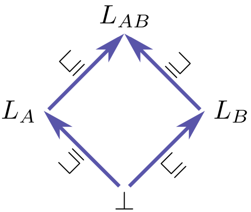

```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```

LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B


```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



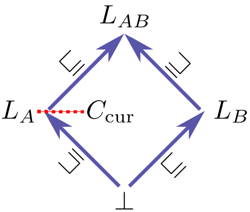
LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B

➥ limit L_{cur} with clearance C_{cur}


```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



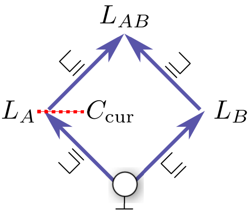
LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B

- ➥ limit L_{cur} with clearance C_{cur}

```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



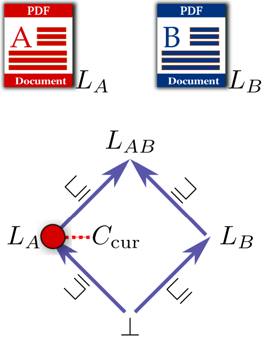
LIO Overview

An example: malicious plugin leaking review information

E.g., Suppose want to prevent plugins from accessing R_B

➥ limit L_{cur} with clearance C_{cur}

```
evilPlugin = do
  a ← readLIORef R_A
  b ← readLIORef R_B
  if b == "..."
    then forever $ return (a,b)
    else return (a,b)
```



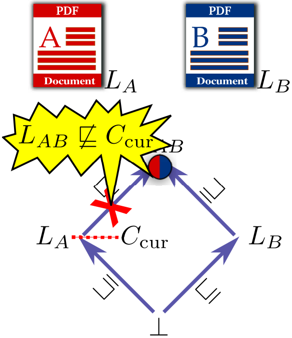
LIO Overview

An example: malicious plugin leaking review information

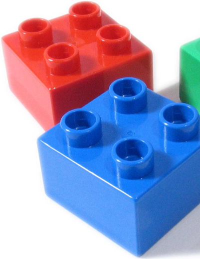
E.g., Suppose want to prevent plugins from accessing R_B

➥ limit L_{cur} with clearance C_{cur}

```
evilPlugin = do
 a ← readLIORef R_A
✗b ← readLIORef R_B
 if b == "..."
  then forever $ return (a,b)
  else return (a,b)
```



What constructs does LIO provide?



Overview of LIO Primitives

Pure labeled values: `Labeled l a`

- Create labeled values:

```
label :: Label l ⇒
             l → a → LIO l (Labeled l a)
```

- Inspect labeled values, affecting L_{cur}:

```
unlabel :: Label l ⇒
             Labeled l a → LIO l a
```

Overview of LIO Primitives

- Primitives for computing on secret data
- Privilege-exercising constructs
- Labeled references
- Labeled file-system support
  - Like references, but write also implies read
- Labeled exceptions

Why trust the LIO approach?

Security Guarantees

Non-interference

Publicly observable results are not affected by secret values in a program, through data or control flow.

Confinement

Program bounded by L_{cur} and C_{cur} cannot:

- Create/write values below L_{cur}
- Create/write/read values above C_{cur}

Semantics of Core LIO + References

A short overview

- Extended $\lambda^{\rightarrow}$ calculus
  - Bool, Labeled, LIORef, etc.
- Dynamics: small step SOS using evaluation contexts
- Runtime environment Σ:
  - $\Sigma.\mathtt{lbl}$: current label
  - $\Sigma.\mathtt{clr}$: current clearance
  - $\Sigma.\phi$: memory store

Step: $\langle \Sigma, e \rangle \longrightarrow \langle \Sigma', e' \rangle$

$$
\begin{aligned}
v &::= \cdots \mid l \mid a \mid (e)^{\mathtt{LIO}} \\
&\quad \mid \mathtt{Lb}\ v\ e \mid \bullet \\
e &::= \cdots \mid \mathtt{label}\ l\ e \\
&\quad \mid \mathtt{unlabel}\ e \\
&\quad \mid \mathtt{toLabeled}\ l\ e \\
&\quad \mid \mathtt{newRef}\ l\ e \\
&\quad \mid \mathtt{readRef}\ a \\
&\quad \mid \mathtt{writeRef}\ a\ e
\end{aligned}
$$

Semantics of Core LIO + References

A short overview

Example (Evaluation rule for newRef)

$$\frac{\Sigma.\phi(a) = \mathtt{Lb}\ l\ e \qquad l' = \Sigma.\mathtt{lbl} \sqcup l \qquad l' \sqsubseteq \Sigma.\mathtt{clr} \qquad \Sigma' = \Sigma[\mathtt{lbl} \mapsto l']}{\langle \Sigma, E[\mathtt{readRef}\ a] \rangle \longrightarrow \langle \Sigma', E[\mathtt{return}\ e] \rangle}$$

Non-Interference: Proof Idea

Idea: No observable difference between

1. Normal program
2. Program with all secret values erased to •

Approach: Simulation with erasure function ε_L

$$
\begin{array}{ccc}
\langle \Sigma, e \rangle & \longrightarrow & \langle \Sigma', e' \rangle \\
\Big\downarrow \varepsilon_L & & \Big\downarrow \varepsilon_L \\
\varepsilon_L(\langle \Sigma, e \rangle) & \longrightarrow_L & \varepsilon_L(\langle \Sigma', e' \rangle)
\end{array}
$$

Non-Interference: Proof Idea

Idea: No observable difference between

1. Normal program
2. Program with all secret values erased to $\bullet$

Approach: Simulation with erasure function ε_L

$$\begin{array}{ccc}
\langle \Sigma, e \rangle & \longrightarrow & \langle \Sigma', e' \rangle \\
\big\downarrow \varepsilon_L & & \big\downarrow \varepsilon_L \\
\varepsilon_L(\langle \Sigma, e \rangle) & \longrightarrow_L & \varepsilon_L(\langle \Sigma', e' \rangle)
\end{array}$$

Details available in paper.

Related Work

Much existing work on static IFC

- DCC[1], DLM[28], FlowCaml[30], SecIO[31], etc.

  Pro: Little/no runtime overhead

  Con: Not very permissive or flexible

Related Work

Existing work on dynamic IFC in Haskell

- Li and Zdancewic[25], Tsai et. al.[7], Devriese and Piessens[12]

Pro: Flexible, support multi-threading

Con: Little means for declassification or mitigation covert channels

Summary & Future Work

- Labeled IO library approach to IFC
  - Flexible and permissive dynamic system
  - Addresses covert channels (with clearance)
- Formal security proofs
  - Non-interference property
  - Containment property
- Ongoing work
  - Improve analysis of extensions (files, etc.)
  - Distributed systems support (DStar, etc.)
  - Termination-sensitive non-interference
  - Web framework for executing untrusted code

Thank you!

cabal install dclabel lio