Decentralized Information Flow Control with the LIO library

Pablo Buiras, Amit Levy, **David Mazières**, John Mitchell, Alejandro Russo, Deian Stefan, David Terei, and Edward Yang

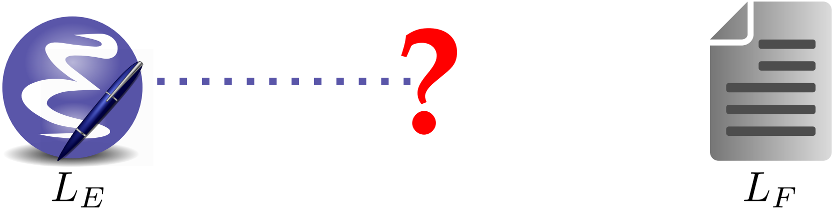
Stanford and Chalmers

October 18, 2013

Project goal

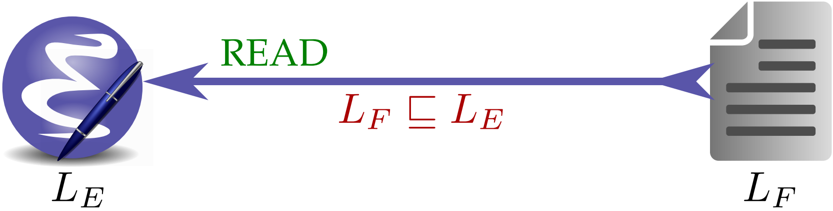
Make it possible to hire median-quality programmers to build secure systems.

What is DIFC?



- IFC originated with military applications and classified data
- Every piece of data in the system has a label
- Every process/thread has a label
- Labels are partially ordered by $\sqsubseteq$ ("can flow to")
- Example: Emacs (labeled L_E) accesses file (labeled L_F)

What is DIFC?



READ

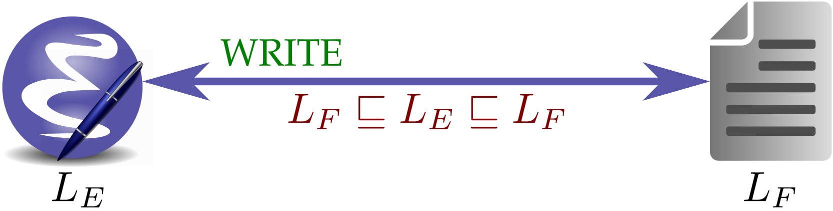
$L_F \sqsubseteq L_E$

L_E

L_F

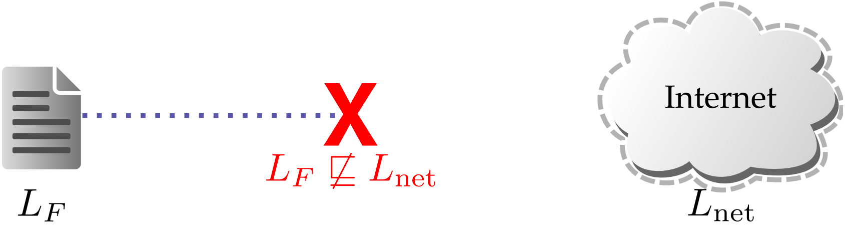
- IFC originated with military applications and classified data
- Every piece of data in the system has a label
- Every process/thread has a label
- Labels are partially ordered by $\sqsubseteq$ (”can flow to”)
- Example: Emacs (labeled L_E) accesses file (labeled L_F)
  - File read? Information flows from file to emacs. System requires $L_F \sqsubseteq L_E$.

What is DIFC?



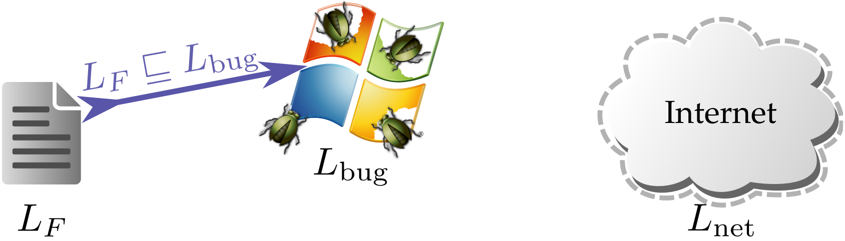
- **IFC originated with military applications and classified data**
- **Every piece of data in the system has a label**
- **Every process/thread has a label**
- **Labels are partially ordered by $\sqsubseteq$ ("can flow to")**
- **Example: Emacs (labeled L_E) accesses file (labeled L_F)**
  - File read? Information flows from file to emacs. System requires $L_F \sqsubseteq L_E$.
  - File write? Information flows in both directions. System enforces that $L_F \sqsubseteq L_E$ and $L_E \sqsubseteq L_F$.

Labels are transitive



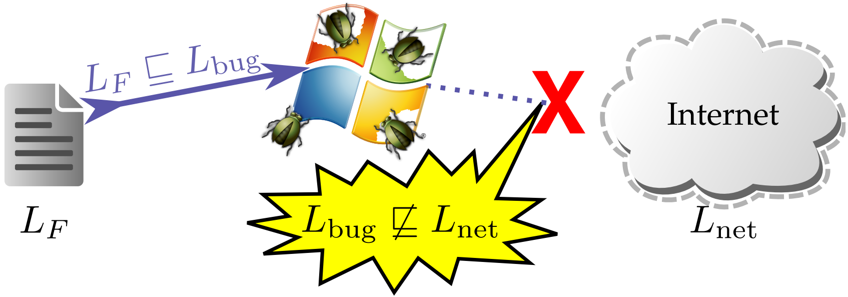
- **$\sqsubseteq$ is a transitive relation**
  - Transitivity makes it easier to reason about security
- **Example: Label file so it cannot flow to Internet:** $L_F \not\sqsubseteq L_{\text{net}}$
  - Policy holds regardless of what other software does

Labels are transitive



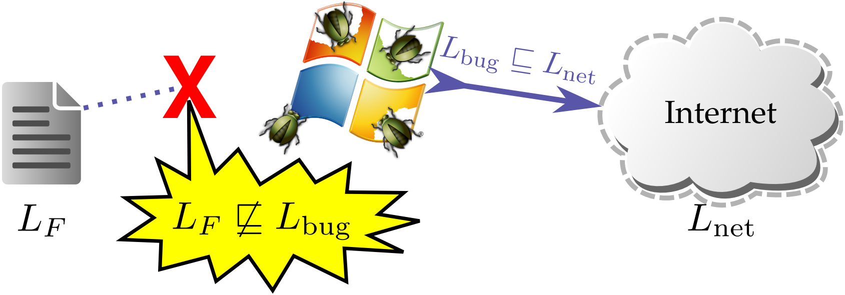
- **$\sqsubseteq$ is a transitive relation**
  - Transitivity makes it easier to reason about security
- **Example: Label file so it cannot flow to Internet:** $L_F \not\sqsubseteq L_{\text{net}}$
  - Policy holds regardless of what other software does
- **Suppose a buggy app reads file (e.g., desktop search)**

Labels are transitive


$L_F \sqsubseteq L_{\text{bug}}$
L_F
X
$L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$
Internet
L_{net}

- $\sqsubseteq$ **is a transitive relation**
  - Transitivity makes it easier to reason about security
- **Example: Label file so it cannot flow to Internet:** $L_F \not\sqsubseteq L_{\text{net}}$
  - Policy holds regardless of what other software does
- **Suppose a buggy app reads file (e.g., desktop search)**
  - Process labeled L_{bug} reads file, so must have $L_F \sqsubseteq L_{\text{bug}}$
  - But since $L_F \not\sqsubseteq L_{\text{net}}$, it must be the case that $L_F \sqsubseteq L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$

Labels are transitive



L_F
$L_F \not\sqsubseteq L_{\mathrm{bug}}$
$L_{\mathrm{bug}} \sqsubseteq L_{\mathrm{net}}$
Internet
L_{net}

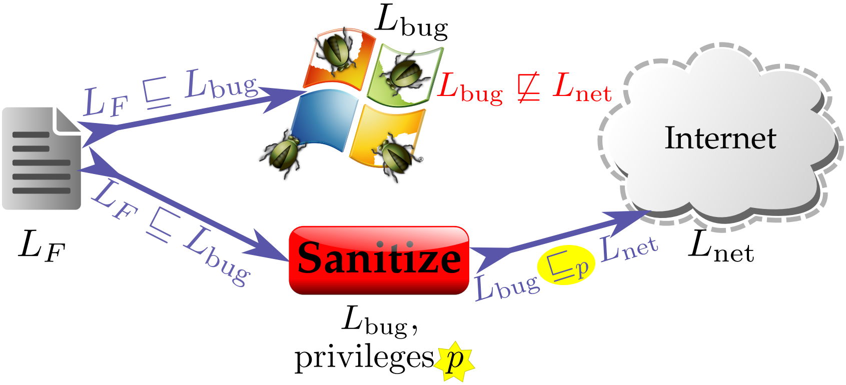
- **$\sqsubseteq$ is a transitive relation**
  - Transitivity makes it easier to reason about security
- **Example: Label file so it cannot flow to Internet:** $L_F \not\sqsubseteq L_{\mathrm{net}}$
  - Policy holds regardless of what other software does
- **Suppose a buggy app reads file (e.g., desktop search)**
  - Process labeled L_{bug} reads file, so must have $L_F \sqsubseteq L_{\mathrm{bug}}$
  - But since $L_F \not\sqsubseteq L_{\mathrm{net}}$, it must be the case that $L_F \sqsubseteq L_{\mathrm{bug}} \not\sqsubseteq L_{\mathrm{net}}$
- **Conversely, if app write to network have** $L_F \not\sqsubseteq L_{\mathrm{bug}} \sqsubseteq L_{\mathrm{net}}$

Labels form a lattice

$$\begin{tikzcd}
 & L_{\mathrm{AB}} & \\
L_{\mathrm{A}} \arrow[ur, "\sqsubseteq"] & & L_{\mathrm{B}} \arrow[ul, "\sqsubseteq"'] \\
 & L_{\emptyset} \arrow[ul, "\sqsubseteq"'] \arrow[ur, "\sqsubseteq"] &
\end{tikzcd}$$

- **Consider two users, A and B**
  - Label public data $L_{\emptyset}$, A's private data L_A, B's private data L_B
- **What if you mix A's and B's private data in a single document?**
  - Both A and B should be concerned about the release of such a document
  - Need a label at least as restrictive as both L_A and L_B
  - Use the least upper bound (a.k.a. *lub* or *join*) of L_A and L_B, written $L_A \sqcup L_B$

DIFC is Decentralized



L_{bug}
$L_F \sqsubseteq L_{\text{bug}}$
$L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$
Internet
L_F
$L_F \sqsubseteq L_{\text{bug}}$
Sanitize
$L_{\text{bug}} \sqsubseteq_p L_{\text{net}}$
L_{net}
L_{bug},
privileges p

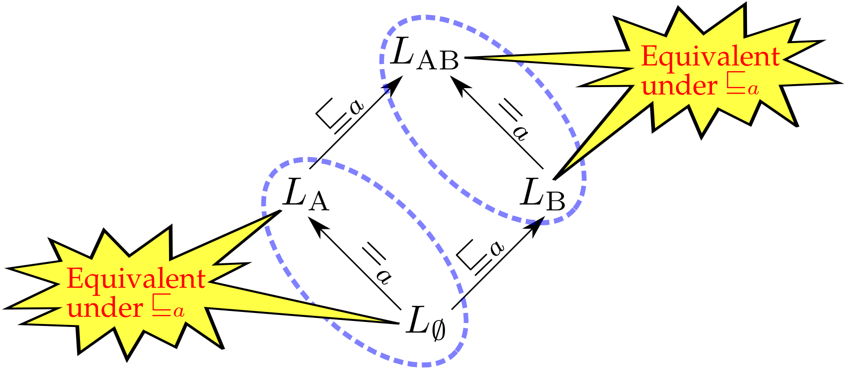
- **Different software has access to different privileges**
- **Exercising privilege p changes label requirements**
    - $\sqsubseteq_p$ ("can flow under privileges p") is more permissive than $\sqsubseteq$
    - $L_F \sqsubseteq_p L_{\text{proc}}$ to read, and additionally $L_{\text{proc}} \sqsubseteq_p L_F$ to write file
- **Idea: Set labels so you know who has relevant privs**

Example privileges

$$\begin{tikzcd}
 & L_{\mathrm{AB}} & \\
L_{\mathrm{A}} \arrow[ur, "\sqsubseteq", sloped] & & L_{\mathrm{B}} \arrow[ul, "\sqsubseteq"', sloped] \\
 & L_{\emptyset} \arrow[ul, "\sqsubseteq"', sloped] \arrow[ur, "\sqsubseteq", sloped] &
\end{tikzcd}$$

- **Consider again simple two user lattice**
- **Let a be user A's privileges, b be user B's privileges**
- **Clearly $L_A \sqsubseteq_a L_\emptyset$ and $L_B \sqsubseteq_b L_\emptyset$**
  - Users should be able to make public or *declassify* their own private data
- **Users should also be able to *partially declassify* data**
  - I.e., $L_{AB} \sqsubseteq_a L_B$ and $L_{AB} \sqsubseteq_b L_A$

Example privileges



- **Consider again simple two user lattice**
- **Let a be user A's privileges, b be user B's privileges**
- **Clearly $L_A \sqsubseteq_a L_\emptyset$ and $L_B \sqsubseteq_b L_\emptyset$**
  - Users should be able to make public or *declassify* their own private data
- **Users should also be able to *partially declassify* data**
  - I.e., $L_{AB} \sqsubseteq_a L_B$ and $L_{AB} \sqsubseteq_b L_A$

Labels in Haskell

- **Represent as type class to accommodate various lattices**

```
class (Eq l, Show l, Typeable l) => Label l where
  lub :: l -> l -> l            -- Least upper bound
  glb :: l -> l -> l            -- Greatest lower bound
  canFlowTo :: l -> l -> Bool -- "Can flow to" partial order
(⊑) = canFlowTo
```

- **We use *DC labels*, pairs of CNF formulas over principals**

$$\overbrace{\textit{reader-condition}}^{\text{secrecy component}} \;\%\%\; \overbrace{\textit{writer-condition}}^{\text{integrity component}}$$

  - Example: `("A" \/ "B") %% "X" /\ ("A" \/ "B")`
    A or B can read; one of A's or B's permissions *plus* X's required to write
  - Mixing data increases secrecy, decreases integrity

$$(S_1 \,\%\%\, I_1) \sqcup (S_2 \,\%\%\, I_2) = (S_1 \wedge S_2 \,\%\%\, I_1 \vee I_2)$$

  - Data can only flow to less secrecy or more integrity ($\Rightarrow$ is "implies")

$$(S_1 \,\%\%\, I_1) \sqsubseteq (S_2 \,\%\%\, I_2) \quad \textbf{iff} \quad (S_1 \Rightarrow S_2) \wedge (I_2 \Rightarrow I_1)$$

Enforcing IFC

- **Supply a "Labeled IO" monad `LIO` to be used in place of `IO`**

```
{-# LANGUAGE Unsafe #-}
data LIOState l = LIOState { lioLabel, lioClearance :: !l }

newtype LIO l a = LIOTCB (IORef (LIOState l) -> IO a)
instance Monad (LIO l) where
  return = LIOTCB . const . return
  (LIOTCB ma) >>= k = LIOTCB $ \s -> do
    a <- ma s
    case k a of LIOTCB mb -> mb s

ioTCB :: IO a -> LIO l a -- back door for privileged code
ioTCB = LIOTCB . const    -- to execute arbitrary IO actions
```

- **Note: constructor `LIOTCB` *not* exported to safe code**
  - Idea: Start with no side effects possible in safe `LIO` code
  - Build up library of label-respecting side effects in trustworthy code
  - By convention, all privileged, unsafe symbols end . . . `TCB`

Adjusting and checking labels

- **Privileged code must check labels before impure actions**
- **Before reading object** obj**, must ensure** $L_{\text{obj}} \sqsubseteq L_{\text{thread}}$

```
taint :: Label l => l -> LIO l ()
taint lobj = do
  LIOState { lioLabel = l, lioClearance = c } <- getLIOStateTCB
  let l’ = l ⊔ lobj
  unless (l’ ⊑ c) $ labelError "taint" [lobj]
  modifyLIOStateTCB $ \s -> s { lioLabel = l’ }
```

- **Before writing, must check** $L_{\text{thread}} \sqsubseteq L_{\text{obj}} \sqsubseteq C_{\text{thread}}$

```
guardWrite :: Label l => l -> LIO l ()
guardWrite lobj = do
  LIOState { lioLabel = l, lioClearance = c } <- getLIOStateTCB
  unless (l ⊑ lobj) $ labelError "guardWrite" [newl]
  taint lobj
```

Representing privileges

- **Privilege type `p` describes pre-orders $\sqsubseteq_p$ on labels of type `l`**

```
class (Label l) => PrivDesc l p where
  downgradeP :: p -> l -> l -- get least equivalent label under ⊑p
  canFlowToP :: p -> l -> l -> Bool
  canFlowToP p l1 l2 = downgradeP p l1 ⊑ l2
```

- **DC label privileges are just CNF formulas, so that**

$$(S_1 \mathbin{\%\%} I_1) \sqsubseteq_p (S_2 \mathbin{\%\%} I_2) \quad \textbf{iff} \quad (p \wedge S_1 \Rightarrow S_2) \wedge (p \wedge I_2 \Rightarrow I_1)$$

- **Note a `PrivDesc` instance merely *describes* privileges**
  - To *exercise* them, must wrap them in type `Priv`

```
newtype Priv p = PrivTCB p
```

  - Safe code cannot import unsafe `PrivTCB` symbol
  - But can bootstrap privileges in `IO` monad before entering `LIO`

```
privInit :: p -> IO (Priv p)
privInit p = return $ PrivTCB p
```

Using `Priv` objects

- **For convenience, `Priv`s are also `PrivDesc`s**

```
instance (PrivDesc l p) => PrivDesc l (Priv p) where
  downgradeP (PrivTCB p) = downgradeP p
  canFlowToP (PrivTCB p) = canFlowToP p
```

- **Most functions have `...P` variants taking a `Priv` argument, e.g.:**

```
taintP :: PrivDesc l p => Priv p -> l -> LIO l ()
taintP p lobj_high = do
  ... Same basic body as taint ...
  where lobj = downgradeP p lobj_high
        (⊑)= canFlowToP p
```

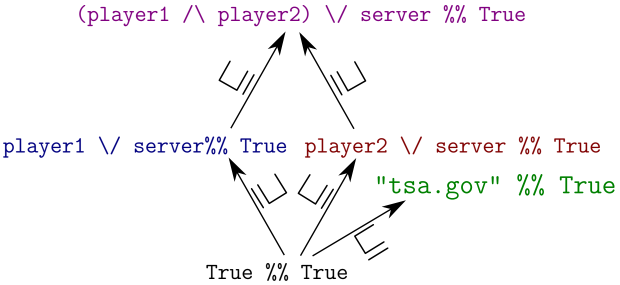
- **Can use one `Priv` object to obtain weaker ones it *speaks for***

```
delegate :: (SpeaksFor p) => Priv p -> p -> Priv p
delegate start_privs wanted_privs = ...
```

  - With DC labels: p_1 speaks for p_2 **iff** $p_1 \Rightarrow p_2$

Example: Rock-Paper-Scissors server

- **Allow untrusted third parties to improve/translate game**
- **Third-party code should *not* be able to cheat (look at opponent's move before playing) or report scissors to `tsa.gov`**
- **Approach:**
  - Give privileges "`server`" to main server loop
  - Delegates sub-privileges to each player, e.g., "`(player1 \/ server)`", ...
  - Use appropriately labeled MVars to record each player's move
- **Lattice:**



`(player1 /\ player2) \/ server %% True`

`player1 \/ server%% True`

`player2 \/ server %% True`

`"tsa.gov" %% True`

`True %% True`

Demo time

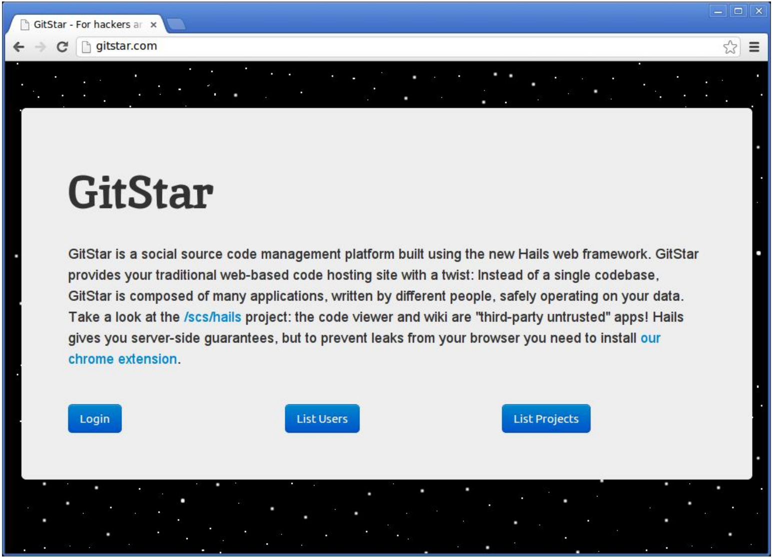
Get the code!

```
git clone http://tinyurl.com/liorock-git
cabal install --haddock-hyperlink-source lio
```

Hails: An LIO web framework

- **Introduces Model-Policy-View-Controller paradigm**
- **A Hails server comprises two types of software packages**
  - *VCs* contain view and controller logic
  - *MPs* contain model and policy logic
- **Policies enforced using LIO**
  - Also isolate spawned programs with Linux namespaces
- **Used for several web sites…**

GitStar

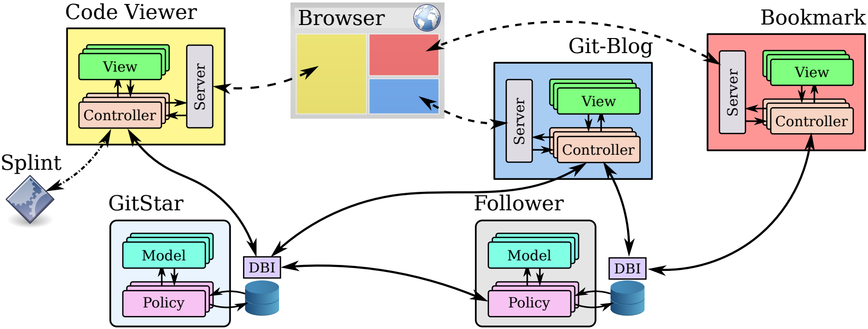


GitStar

GitStar is a social source code management platform built using the new Hails web framework. GitStar provides your traditional web-based code hosting site with a twist: Instead of a single codebase, GitStar is composed of many applications, written by different people, safely operating on your data. Take a look at the /scs/hails project: the code viewer and wiki are "third-party untrusted" apps! Hails gives you server-side guarantees, but to prevent leaks from your browser you need to install our chrome extension.

- **Public GitHub-like service supporting private projects**

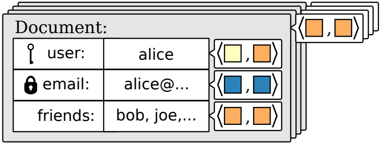
Simplified GitStar architecture



- **Two MPs: *GitStar* hosts git repos, *Follower* stores a relationship between users**
- **Three different VC apps make use of these MPs**
  - VCs can be written after the fact w/o permission of MP author
  - `LIO` ensures they cannot mis-use data

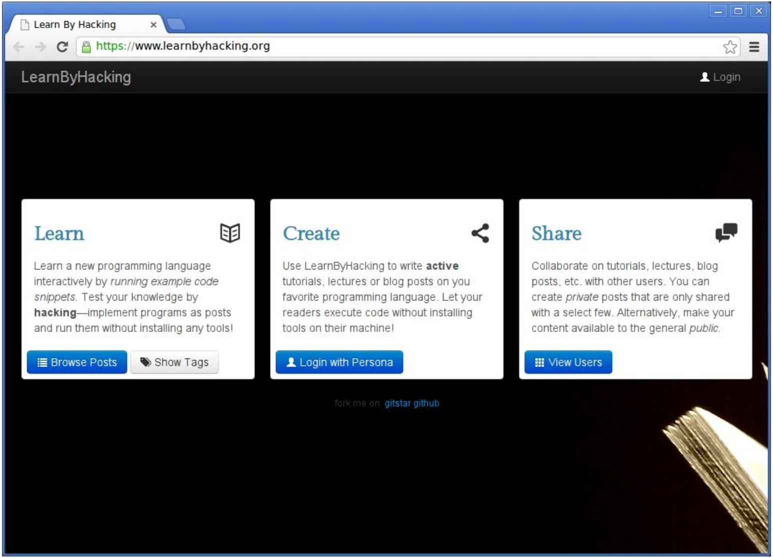
What policy looks like

```
-- Set policy for "users" collection:
collection "users" $ do
  -- Set collection label:
  access $ do
    readers ==> anybody
    writers ==> anybody
  -- Declare user field as a key:
  field "user" key
  -- Set document label, given document doc:
  document $ \doc -> do
    readers ==> anybody
    writers ==> ("user" ‘from‘ doc) \/ _Follower
  -- Set email field label, given document doc:
  field "email" $ labeled $ \doc -> do
    readers ==> ("user" ‘from‘ doc)
                \/ fromList ("friends" ‘from‘ doc)
                \/ _Follower
    writers ==> anybody
```



Labeled by: 🟨 Collection 🟧 Document 🟦 Field

LearnByHacking



Learn

Learn a new programming language interactively by *running example code snippets*. Test your knowledge by **hacking**—implement programs as posts and run them without installing any tools!

Create

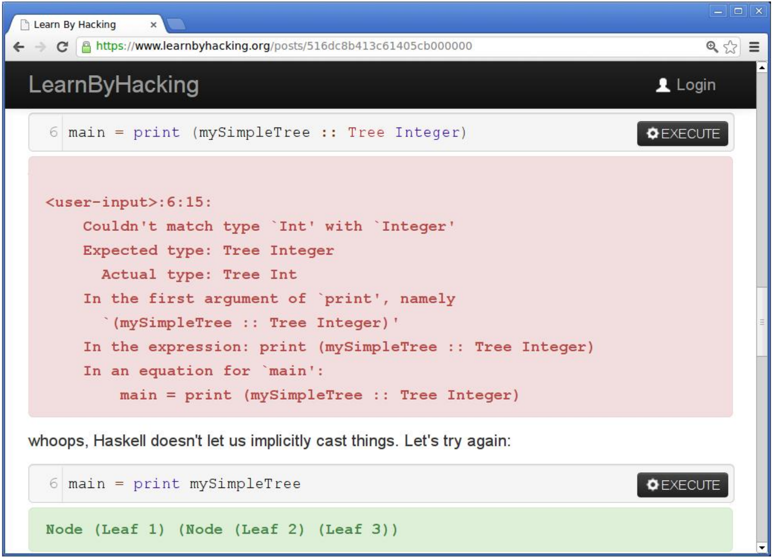
Use LearnByHacking to write **active** tutorials, lectures or blog posts on you favorite programming language. Let your readers execute code without installing tools on their machine!

Share

Collaborate on tutorials, lectures, blog posts, etc. with other users. You can create *private* posts that are only shared with a select few. Alternatively, make your content available to the general *public*.

fork me on: gitstar github

LearnByHacking



Learn By Hacking
https://www.learnbyhacking.org/posts/516dc8b413c61405cb000000
LearnByHacking
Login
EXECUTE

```
main = print (mySimpleTree :: Tree Integer)
```

```
<user-input>:6:15:
    Couldn't match type `Int' with `Integer'
    Expected type: Tree Integer
      Actual type: Tree Int
    In the first argument of `print', namely
      `(mySimpleTree :: Tree Integer)'
    In the expression: print (mySimpleTree :: Tree Integer)
    In an equation for `main':
        main = print (mySimpleTree :: Tree Integer)
```

whoops, Haskell doesn't let us implicitly cast things. Let's try again:

```
main = print mySimpleTree
```

```
Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
```

Questions

Secure
Computer
Systems

```
http://www.scs.stanford.edu/
git clone http://tinyurl.com/liorock-git
```