A Monadic Analysis of Information-Flow Security with Mutable State

Enforcing Secrecy with Monadic Types

Aleksey Kliger
joint work with
Karl Crary and Frank Pfenning

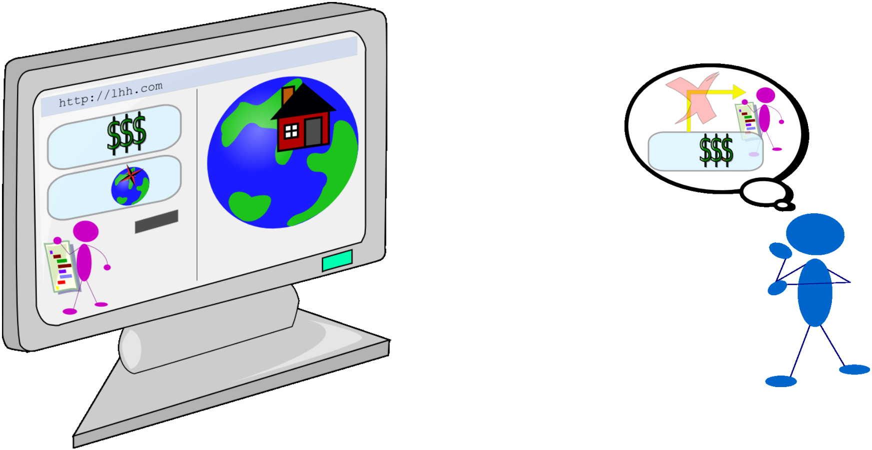
Carnegie Mellon University

April 21, 2006 / Stevens Institute of Technology

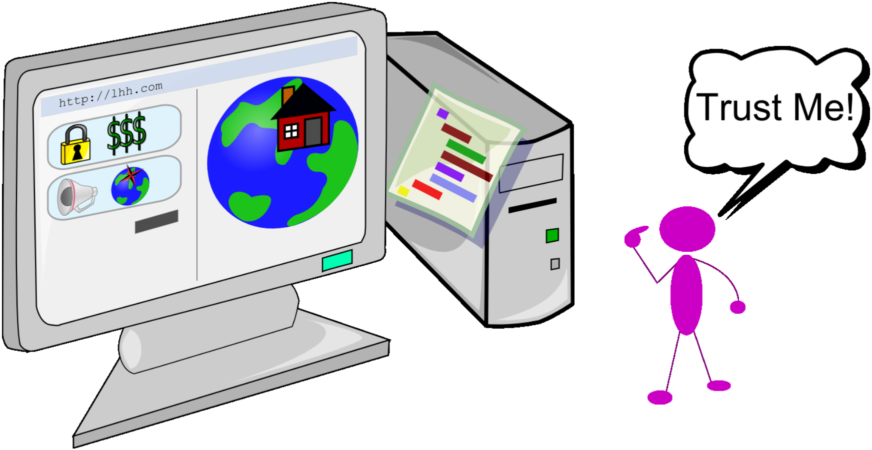
Helen shops for a house

LHH.com

LHH.com



LHH.com



Running untrusted code

Helen wants

A guarantee that Luke cannot learn income

Luke wants

Reasonable burden of proof

Running untrusted code

Language-based information flow security.

Type system tracks flow of information.
Well typed programs do not leak secrets. □

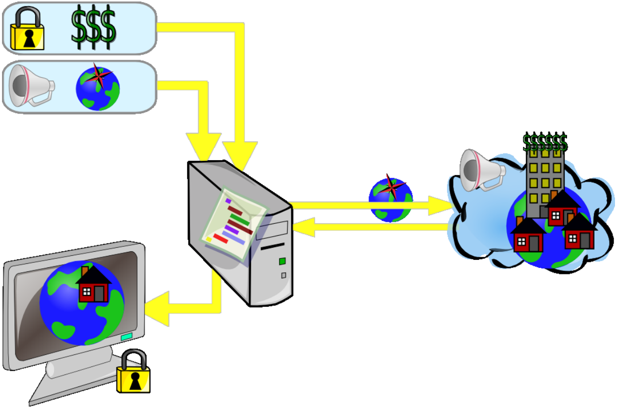
Helen wants

A guarantee that Luke cannot learn income

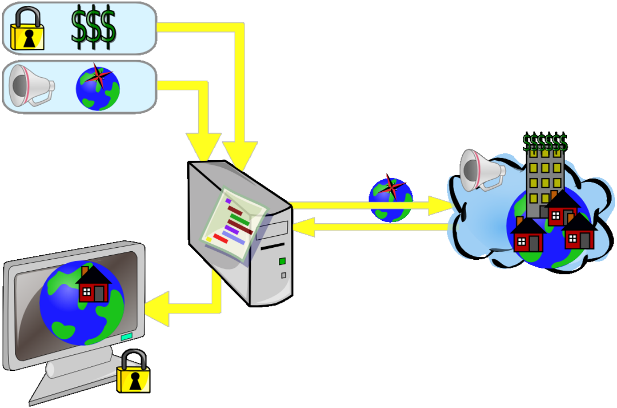
Luke wants

Reasonable burden of proof

Information flow



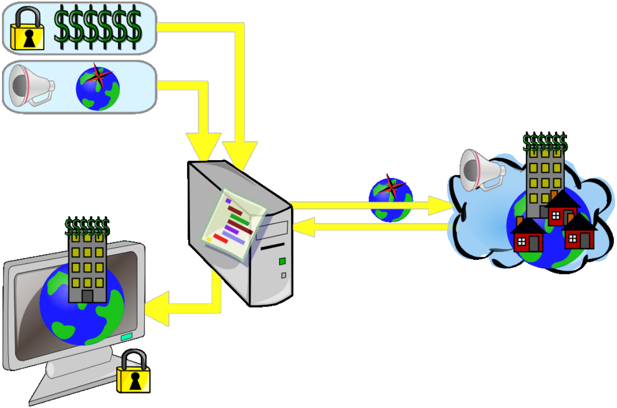
Information flow



Luke thinks

Someone wants a house in TriBeCa

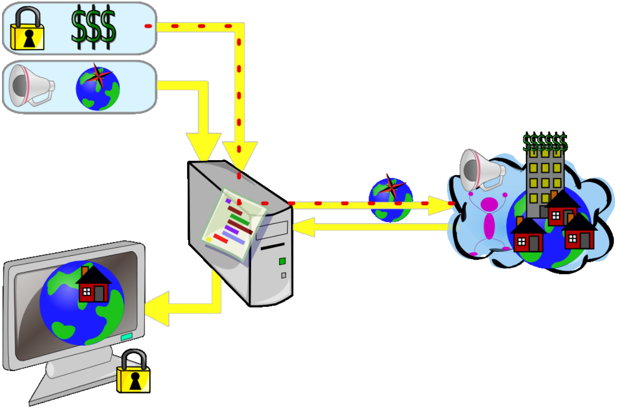
Information flow



Luke thinks

Someone wants a house in TriBeCa

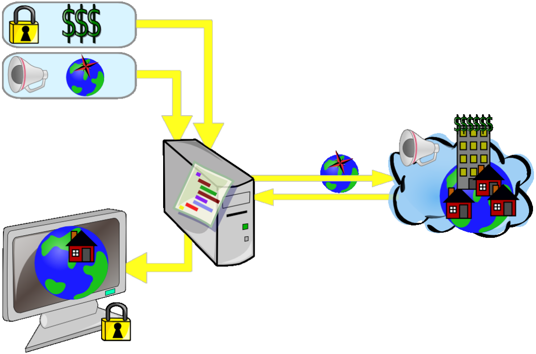
Information flow



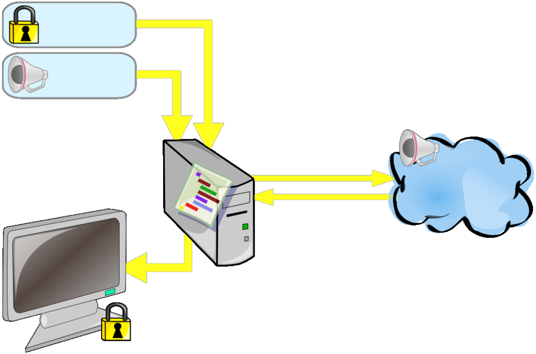
Luke thinks

Someone wants a cheap house in TriBeCa

Abstracting away



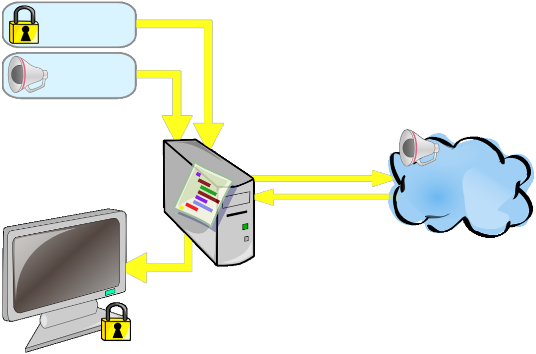
Abstracting away



Situation

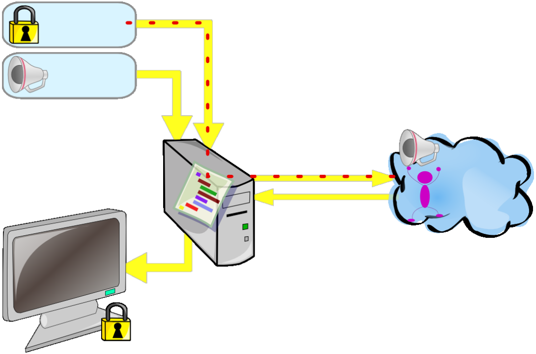
- Untrusted program
- Private data
- Public data
- Must access both

Abstracting away



Guarantee

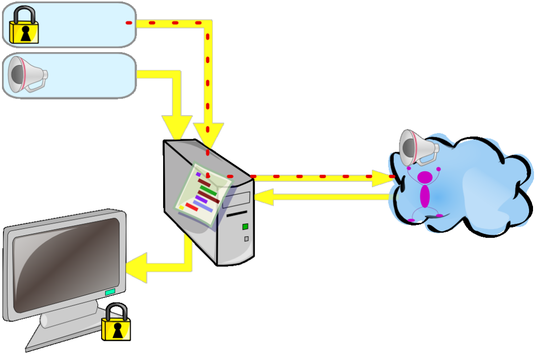
Abstracting away



Guarantee

No information flow of *high*-security data to *low*-security observer.

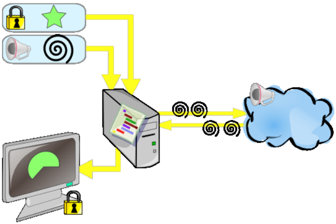
Abstracting away

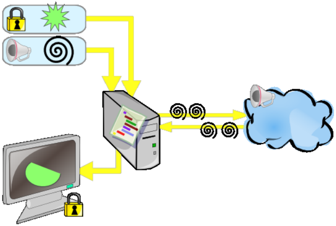


Guarantee

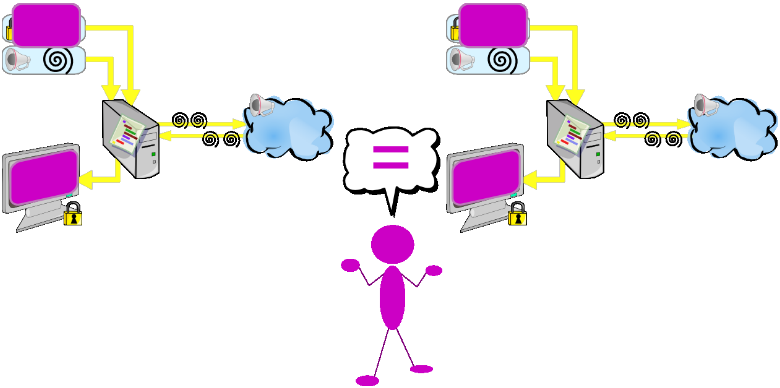
The high-security data does not interfere with the low-security behavior of the program.

Non-interference





Non-interference



Outline

Tracking Information Flow

Key Idea

Only two things matter:

Tracking Information Flow

Key Idea

Only two things matter:

- Communication channels

Channels

Memory Mark each cell with security level:

- zipcodeField : 
- incomeField : 

I/O

- Network (Luke) : 
- Display (Helen) : 

Tracking Information Flow

Key Idea

Only two things matter:

- Communication channels
- Computations with side-effects

Channels

Memory Mark each cell with security level:

- zipcodeField : 
- incomeField : 

I/O

- Network (Luke) : 
- Display (Helen) : 

Computations

Track inputs and outputs

Security Levels

Definition

Security levels form a lattice $\mathcal{L} = (\bot, \top, \sqsubseteq, \sqcup, \sqcap)$

Example

 $\sqsubseteq$ 

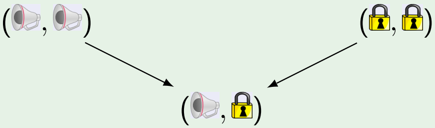
Operation Levels

Definition

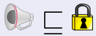
An operation level o is a pair (In, Out) of security levels with $In \sqsubseteq Out$.
Operation levels form a meet-semilatice $\mathcal{O} = (\bot, \preceq, \curlywedge)$ given by

$$(r_1, w_1) \preceq (r_2, w_2) \iff (r_1 \sqsubseteq r_2 \text{ and } w_2 \sqsubseteq w_1)$$

Example



Security Levels



Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);   In: , , Out: , . ✔
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);
  priceRange :=
    calcPriceRange (incomeField);   In: , Out: . ✔
  showJustAffordable (houses,
                      priceRange);
}
```

Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);   In:  Out: .
}
```

Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);   In: , Out: . ?
}
```

Tracking inputs and outputs

First try at LHH.com

Example (A very imperative example)

proc *processForm* () {
  houses📢
    fetchHo
  priceRan
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                priceRange);   *In:* 🔒 *Out:* 🔒. ✔
}

To Combine Inputs: Join

📢 $\sqcup$ 🔒 $=$ 🔒

Example (What if `houses` is secret)

```
proc processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Tracking inputs and outputs

Another try at LHH.com

Example (What if houses is secret)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);  In: , , Out: , . ?
  priceRange :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Tracking inputs and outputs

Another try at LHH.com

Example (What if `houses` is secret)

proc *processForm* () {

  houses :=

    fetchHouses (zipcodeField);   *In:* , *Out:* . ✔

  priceRange :=

    calcPric

  showJustA

}

To Combine Outputs: Meet

$\sqcap$ =

Tracking inputs and outputs

Another try at LHH.com

Example (What if `houses` is secret)

```
proc processForm () {
  houses :=
    fetchHouses (zipcodeField);
  priceRange :=
    calcPriceRange (incomeField);   In: , Out: . ✔
  showJustAffordable (houses,
                      priceRange);
}
```

Example (What if `houses` is secret)

proc *processForm* () {
  houses🔒 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                priceRange);   *In:* 🔒, 🔒 *Out:* 🔒. ✔
}

A monadic language

Syntax (Types)

$$\begin{array}{llcl} \text{Types} & A & ::= & \cdots \mid A \to B \mid \cdots \\ & & \mid & \textbf{Ptr}\ A \mid \bigcirc A \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

A monadic language

Syntax (Types)

$$
\begin{array}{llll}
\text{Types} & A & ::= & \cdots \mid A \to B \mid \cdots \\
 & & \mid & \textbf{Ptr}\ A \mid \bigcirc A \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}
$$

Syntax (Terms)

$$
\begin{array}{llll}
\text{Pure terms} & M, N & ::= & x \mid \cdots \mid \textbf{lam}x.M \mid M\ \ N \mid \cdots \\
 & & \mid & \textbf{do}\ E \\
\text{Effectful Expressions} & E, F & ::= & \textbf{return}\ M \\
 & & \mid & x \leftarrow M\ ;\ E \\
 & & \mid & M\ ;\ E \\
 & & \mid & M
\end{array}
$$

Example (increment)

inc : (**Ptr** **Int**, **Int**) $\rightarrow \bigcirc$**Int**
inc (ptr, amt) =
do
  oldVal $\leftarrow$ **deref** ptr
  ptr := oldVal + amt
  return oldVal

Example

$$
\begin{array}{l}
\mathit{repeatUntil}\ :\\
\qquad (A \rightarrow \mathbf{Bool},\ \bigcirc A) \rightarrow \bigcirc A\\
\mathit{repeatUntil}\\
\qquad \texttt{(test, comp) =}\\
\mathbf{do}\\
\quad \texttt{x} \leftarrow \texttt{comp}\\
\quad \mathbf{if}\ \texttt{(}\mathit{test}\ \texttt{(x))}\\
\qquad \mathbf{then}\ \mathbf{return}\ \texttt{x}\\
\qquad \mathbf{else}\ \mathit{repeatUntil}\\
\qquad\qquad \texttt{(test, comp)}
\end{array}
$$

Example

$$\mathit{repeatUntil} : (A \rightarrow \mathbf{Bool},\ \bigcirc A) \rightarrow \bigcirc A$$

$\mathit{repeatUntil}$ $(\mathtt{test},\ \mathtt{comp})$ =

do

  $\mathtt{x} \leftarrow \mathtt{comp}$

  if $(\mathit{test}\ (\mathtt{x}))$

    then return $\mathtt{x}$

    else $\mathit{repeatUntil}$ $(\mathtt{test},\ \mathtt{comp})$

Example (Sample Output)

$$\text{State} = \{\mathtt{ptr} \mapsto 0\}$$

$$\mathit{repeatUntil}\ (\mathit{above100},\ \mathit{inc}\ (\mathtt{ptr},\ 1))$$

Example

$\mathit{repeatUntil} :$
$\quad (A \rightarrow \mathbf{Bool},\ \bigcirc A) \rightarrow \bigcirc A$

$\mathit{repeatUntil}$ `(test, comp) =`

do
$\quad$ `x` $\leftarrow$ `comp`
$\quad$ **if** `(`*test* `(x))`
$\quad\quad$ **then** **return** `x`
$\quad\quad$ **else** *repeatUntil*
$\quad\quad\quad$ `(test, comp)`

Example (Sample Output)

State $= \{\mathtt{ptr} \mapsto 0\}$

do
$\quad$ `x` $\leftarrow$ *inc* `(ptr, 1)`
$\quad$ **if** `(above100 (x))`
$\quad\quad$ **then** **return** `x`
$\quad\quad$ **else** *repeatUntil*
$\quad\quad\quad$ `(`*above100*`,`
$\quad\quad\quad$ *inc*`(ptr, 1))`

Example

$$
\begin{aligned}
&\mathit{repeatUntil} \;: \\
&\quad (A \rightarrow \mathbf{Bool},\ \bigcirc A) \rightarrow \bigcirc A \\
&\mathit{repeatUntil} \\
&\quad\quad (\mathtt{test},\ \mathtt{comp})\ = \\
&\mathbf{do} \\
&\quad \mathtt{x} \leftarrow \mathtt{comp} \\
&\quad \mathbf{if}\ (\mathit{test}\ (\mathtt{x})) \\
&\quad\quad \mathbf{then}\ \mathbf{return}\ \mathtt{x} \\
&\quad\quad \mathbf{else}\ \mathit{repeatUntil} \\
&\quad\quad\quad\quad (\mathtt{test},\ \mathtt{comp})
\end{aligned}
$$

Example (Sample Output)

$$
\begin{aligned}
&\text{State} = \{\mathtt{ptr} \mapsto 0\} \\
&\mathbf{do} \\
&\Rightarrow \mathtt{x} \leftarrow \mathit{inc}\ (\mathtt{ptr},\ \mathtt{1}) \\
&\quad \mathbf{if}\ (\mathtt{above100}\ (\mathtt{x})) \\
&\quad\quad \mathbf{then}\ \mathbf{return}\ \mathtt{x} \\
&\quad\quad \mathbf{else}\ \mathit{repeatUntil} \\
&\quad\quad\quad\quad (\mathit{above100}, \\
&\quad\quad\quad\quad\ \mathit{inc}(\mathtt{ptr},\ \mathtt{1}))
\end{aligned}
$$

Example

$$\textit{repeatUntil} : (A \rightarrow \textbf{Bool}, \bigcirc A) \rightarrow \bigcirc A$$

$\textit{repeatUntil}$ `(test, comp) =`

do

  $\texttt{x} \leftarrow \texttt{comp}$

  if (*test* (x))

    then return x

    else *repeatUntil* (test, comp)

Example (Sample Output)

$\text{State} = \{\texttt{ptr} \mapsto 1\}$

do

  $\texttt{x} \leftarrow \textit{inc}\ (\texttt{ptr, 1})$

$\Rightarrow$ **if** (above100 (0))

    then return 0

    else *repeatUntil* (*above100*, *inc*(ptr, 1))

Example

$$
\begin{array}{l}
\mathit{repeatUntil} \;: \\
\qquad (A \to \mathbf{Bool},\;\; \bigcirc A) \to \bigcirc A \\
\mathit{repeatUntil} \\
\qquad\; \mathtt{(test,\ comp)} \;= \\
\mathbf{do} \\
\quad \mathtt{x} \leftarrow \mathtt{comp} \\
\quad \mathbf{if}\;\; (\mathit{test}\;\; \mathtt{(x)}) \\
\qquad \mathbf{then}\;\; \mathbf{return}\;\; \mathtt{x} \\
\qquad \mathbf{else}\;\; \mathit{repeatUntil} \\
\qquad\qquad\quad \mathtt{(test,\ comp)}
\end{array}
$$

Example (Sample Output)

$$
\mathrm{State} = \{\mathtt{ptr} \mapsto 1\}
$$

$$
\begin{array}{l}
\mathbf{if}\;\; \mathtt{(above100\ \ (0))} \\
\quad \mathbf{then}\;\; \mathbf{return}\;\; \mathtt{0} \\
\quad \mathbf{else}\;\; \mathit{repeatUntil} \\
\qquad\quad (\mathit{above100}, \\
\qquad\quad\; \mathit{inc}\;\; \mathtt{(ptr,\ 1))}
\end{array}
$$

Example

$$\mathit{repeatUntil}\ :\ (A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$$

$\mathit{repeatUntil}$ (test, comp) =

do

  x $\leftarrow$ comp

  if (test (x))

    then return x

    else $\mathit{repeatUntil}$ (test, comp)

Example (Sample Output)

State $= \{\mathtt{ptr} \mapsto 1\}$

$\mathit{repeatUntil}$ ($\mathit{above100}$, inc (ptr, 1))

Example

$\mathit{repeatUntil}$:
  $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
$\mathit{repeatUntil}$
  `(test, comp) =`
do
  `x` $\leftarrow$ `comp`
  if (test `(x)`)
    then return `x`
    else $\mathit{repeatUntil}$
      `(test, comp)`

Example (Sample Output)

State $= \{\texttt{ptr} \mapsto 1\}$

$\mathit{repeatUntil}$
  ($\mathit{above100}$,
  inc `(ptr, 1))`

The Language with security monads

Syntax (Types)

Security Levels	a, In, Out	$::=$	 $\mid$ 
Types	A	$::=$	$\cdots \mid A \to B \mid \cdots$
		$\mid$	$\mathbf{Ptr}_a A \mid \bigcirc_{(In,Out)}\, A$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$

Syntax (Terms)

Pure terms	M, N	$::=$	$x \mid \cdots \mid \mathbf{lam} x.M \mid M\ N \mid \cdots$
		$\mid$	$\mathbf{do}\ E$
Effectful Expressions	E, F	$::=$	$\mathbf{return}\ M$
		$\mid$	$x \leftarrow M\ ;\ E$
		$\mid$	$M\ ;\ E$
		$\mid$	M

The Language with security monads

Syntax (Types)

Security Levels	a, In, Out	$::=$	 $\mid$ 
Types	A	$::=$	$\cdots \mid A \to B \mid \cdots$
		$\mid$	$\mathbf{Ptr}_{a} A \mid \bigcirc_{(In,Out)}\ A$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$

Syntax (Terms)

Pure terms	M, N	$::=$	$x \mid \cdots \mid \mathbf{lam} x.M \mid M\ N \mid \cdots$
		$\mid$	$\mathbf{do}\ E$
Effectful Expressions	E, F	$::=$	$\mathbf{return}\ M$
		$\mid$	$x \leftarrow M\ ;\ E$
		$\mid$	$M\ ;\ E$
		$\mid$	M

LHH.com with security monads

Types

Monadic code

```
processForm =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

LHH.com with security monads

Types

$\mathit{fetchHouses} : \mathbf{Ptr}\ \mathit{ZipCode} \rightarrow \bigcirc_{,}\ \mathbf{List}\ \mathit{House}$

Monadic code

processForm =

do

```
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

LHH.com with security monads

Types

deref `incomeField` : $\bigcirc_{\text{🔒},\text{🔒}}$ **Int**

Monadic code

processForm =
do
  h ← *fetchHouses* (zipcodeField)
  do
    i ← **deref** incomeField
    showJustAffordable (h, *calcPriceRange* (i))

LHH.com with security monads

Types

$calcPriceRange : \mathbf{Int} \rightarrow Range$

Monadic code

$processForm$ =
do
  h ← *fetchHouses* (zipcodeField)
  do
    i ← **deref** incomeField
    showJustAffordable (h, *calcPriceRange* (i))

LHH.com with security monads

Types

showJustAffordable : (**List** *House*, *Range*) $\rightarrow \bigcirc_{,}()$

Monadic code

```
processForm =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

Composing suspensions

Typing rule

$$\frac{\Gamma \vdash M : \bigcirc_{o_1} A \quad \Gamma, x{:}A \vdash E \div_{o_2} C \quad o_1 \preceq o \quad o_2 \preceq o}{\Gamma \vdash x \leftarrow M; E \div_o C}$$

Portion of `processForm`

do

`i` $\leftarrow$ **deref** `incomeField` : $\bigcirc$ **Int**

showJustAffordable

`(h,` *calcPriceRange* `(i))` $\div$ `()`

Composing suspensions

Typing rule

$$\frac{\Gamma \vdash M : \bigcirc_{o_1} A \quad \Gamma, x{:}A \vdash E \div_{o_2} C \quad o_1 \preceq o \quad o_2 \preceq o}{\Gamma \vdash x \leftarrow M; E \div_o C}$$

Portion of `processForm`

do

  `i` $\leftarrow$ **deref** `incomeField` : $\bigcirc_{🔒,🔒}$ **Int**

  showJustAffordable

    `(h,` *calcPriceRange* `(i))` $\div_{📢,🔒}$ `()`

$\div_{🔒,🔒}$ `()`

Composing suspensions, cont'd

processForm

do
  h ← *fetchHouses*
              (zipcodeField) : ◯ , **List** *House*
  do
    i ← **deref** incomeField
    showJustAffordable (h, *calcPriceRange* (i))
  : ◯ , ()

Composing suspensions, cont'd

Compose Inputs

$\sqcup =$

Compose Outputs

$\sqcap =$

processForm

do
  h ← *fetchHouses*
      (zipcodeField) : $\bigcirc$ **List** *House*
  do
    i ← **deref** incomeField
    showJustAffordable (h, *calcPriceRange* (i))
  : $\bigcirc$ ()
  $\div$ ()

What are the types telling us?

Suppose `showJustAffordable` had type

$(\textbf{List}\ \textit{House}, \textit{Range}) \rightarrow \bigcirc_{,}\textbf{Int}$

Consider

$$
\begin{array}{l}
\textbf{do} \\
\quad \texttt{nHouses} \leftarrow \textbf{do} \\
\qquad\qquad\qquad \texttt{i} \leftarrow \textbf{deref}\ \texttt{incomeField} \\
\qquad\qquad\qquad \textit{showJustAffordable}\ (\ldots) \\
\qquad\qquad\quad : \bigcirc_{,}\ \textbf{Int} \\
\quad \textit{sendL}\ (\texttt{nHouses}) \qquad : \bigcirc_{,}\ ()
\end{array}
$$

() is not informative

Definition ($A \nearrow$ judgment)

A type A is informative only at high-security
if no computation with low-security input can make use of it.

Observation

Any computation that could make use of A can read
the same secrets as the computation that produced A

Typing rule

$$\frac{M : \bigcirc_{\text{🔒},\text{🔒}} A \qquad A \nearrow}{\textbf{upcall}\ M : \bigcirc_{\text{📢},\text{🔒}} A}$$

`processForm` is well-typed

Promote the inner do-block

processForm =

do

  h ← *fetchHouses* (zipcodeField) : ◯ **List** *House*

  upcall do

    i ← **deref** incomeField

    showJustAffordable (. . .)

  : ◯ ()

: ◯ ()

Other informative types?

Rules for the $A \nearrow$ judgment

Rule

$$\frac{}{\mathbf{Ptr}_{\text{🔒}} A \nearrow}$$

Rule

$$\frac{A \nearrow}{\mathbf{Ptr}\ A \nearrow}$$

Other informative types?

Rules for the $A \nearrow$ judgment

Rule

$$\frac{A \nearrow \quad B \nearrow}{(A, B) \nearrow}$$

Rule

$$\frac{A \nearrow}{A + B \nearrow} \text{✘} \quad \frac{B \nearrow}{A + B \nearrow} \text{✘}$$

Rule

$$\frac{B \nearrow}{A \to B \nearrow}$$

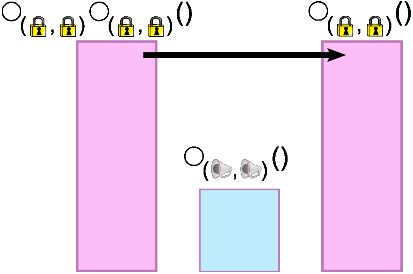
Other informative types?

Rules for the $A \nearrow$ judgment

Rule

$$\frac{A \nearrow}{\bigcirc_{\_,\text{🔒}} A \nearrow}$$

Secure computation continuation passing

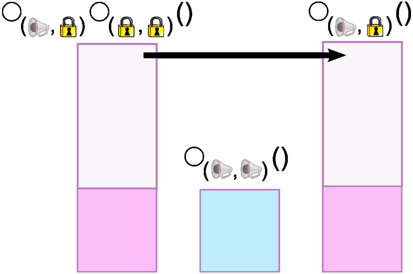


Example

do

```
    cont ← highComp
    lowComp
    cont
```

Secure computation continuation passing

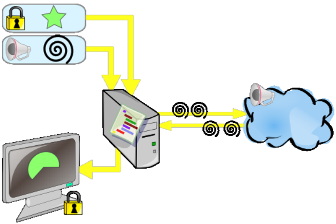


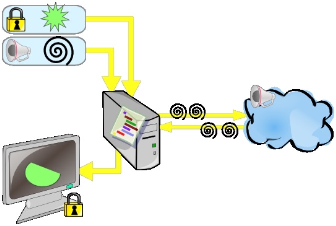
Example

```
do
    cont ← upcall highComp
    lowComp
    upcall cont
```

Outline

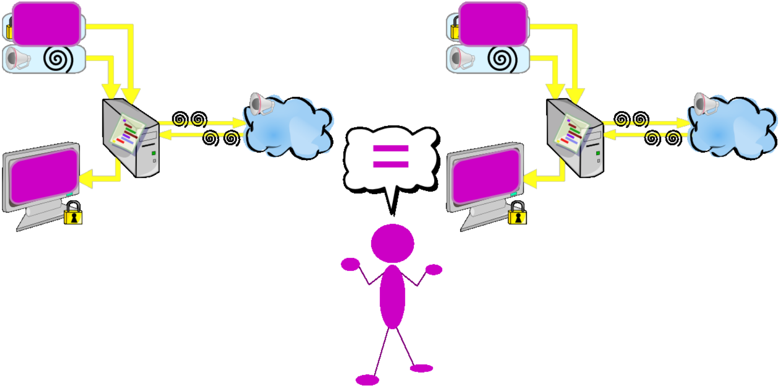
Is it secure?





Is it secure?

Yes



Proof outline

Definition (Program)

A program P is a memory $H : \Sigma$ along with a (closed) effectful expression E

- Formalize equivalence relation $\vdash P_1 \approx P_2 \div_{(In,-)} A$
- Prove two lemmas:

  High Security Step If $In =$ then P_1 and P_2 can do whatever they want and stay related.

  Hexagon Lemma (relative confluence) If $In =$ then if P_1, P_2 execute in lock step.
- Put everything together: If you run P_1, P_2 to completion, the final states are related.

Equivalence Relation

The only non-trivial rule

$$\frac{\Sigma_1; \Gamma \vdash V_1 : A \quad \Sigma_2; \Gamma \vdash V_2 : A \quad A \nearrow}{\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx V_2 : A}$$

Lemma (Functionality)

If $\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx V_2 : A$
and $\Sigma_1; \Sigma_2; \Gamma, x{:}A \vdash E_1 \approx E_2 \div_{(In,Out)} B$
then $\Sigma_1; \Sigma_2; \Gamma \vdash E_1[V_1/x] \approx E_2[V_2/x] \div_{(In,Out)} B$

Lemma (Single High Security Step)

If $\vdash P \div_{\_,} A$ *and* $P \longrightarrow P'$
then "memory of P" $\approx$ "memory of P'"

Hexagon Lemma

$$
\begin{tikzcd}
 & \vdash & P_1 \arrow[dll] & \approx & P_2 \arrow[drr] & \div_{,-} A & \\
P_1' & & & & & & P_2'
\end{tikzcd}
$$

Hexagon Lemma

$$
\begin{tikzcd}
 & \vdash P_1 & \approx & P_2 \div_{,-} A \arrow[dr] & \\
P_1' \arrow[dr, dotted, "*"] & & & & P_2' \arrow[dl, dotted, "*"'] \\
 & \vdash P_1'' & \approx & P_2'' \div_{,-} A &
\end{tikzcd}
$$

Outline

Foundations

- Semantics [Moggi 1989]
- Lax Logic [Fairtlough, *et al.* 1994] [Pfenning, *et al.* 1999]

Have been used for:

- I/O in Haskell [Peyton-Jones, *et al.* 1993]
- Parsing Combinators [Wadler 1992]
- Composable Transactional Memory [Peyton-Jones, *et al.* 2005]
- Probabilistic Computation [Park, *et al.* 2005]
- and much more...

Non-interference:

- [Volpano, *et al.* 1996]
- SLam [Heintze, *et al.* 1998], DCC [Abadi *et al.* 1999]
- CoreML^2 [Pottier, *et al.* 2003]
- and many others ... [Sabelfeld, *et al.* 2003]

Extensions:

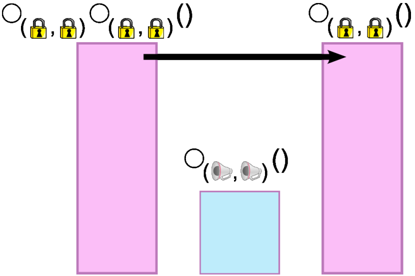
- Concurrency [Honda, *et al.* 2002]
- Timing channels [Agat 2000]
- Robust declassification [Zdancewic, *et al.* 2001], [Myers, *et al.* 2004]
- and many others...

Our contributions

- Informativeness
- Monads for tracking information flow

Informativeness: $A \nearrow$

Allow high-security computations to pass temporary results through low-security computations



Monads for tracking information flow

Isolate security concerns: simplify reasoning
Only monads and channels are tagged

Example

$\mathit{fetchHouses} : \mathbf{Ptr}\ \mathit{ZipCode} \to \bigcirc_{,}\ \mathbf{List}\ \mathit{House}$

$\mathit{calcPriceRange} : \mathbf{Int} \to \mathit{Range}$

New!

For a fixed security lattice, a Haskell library of security monads
(as long as you don't use `unsafePerformIO`, etc)

Future work

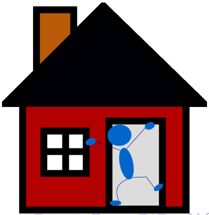
Richer languages: Concurrency (*e.g.* transactional memory), robust declassification

Trustless computing: Formalize all proofs in Twelf

Logical foundations: Decomposition of lax modality $\bigcirc A$ as $\Diamond\Box A$

Thanks

Questions?



The three monad laws

Within each security level, we obey the monad laws
(Upcalls are an additional relationship between monad families)

1. ```
   do x ← return expr
      func (x)
   ```
   ```
   do
      func (expr)
   ```

2. ```
   do x ← comp
      return x
   ```
   ```
   do
      comp
   ```

3. ```
   do y ← do
            x ← comp1
            comp2 (x)
      comp3 (y)
   ```
   ```
   do
      x ← comp1
      y ← comp2 (x)
      comp3 (y)
   ```
   ```

Differences in the paper

Additions in the paper

- Full lattice of security levels
- Subtypes of $\mathbf{Ptr}_a A$ type
- A proof of non-interference
- Discussion of allocation
- Encoding of a prior work

Stylistic differences

- Pfenning-style expressions *vs.* **do**-notation

Allocation

Rule

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{alloc}\ M : \bigcirc_{\,,}\textbf{Ptr}_a A}$$

Since no pointer comparison operators, only read/writes with aliasing to observe a pointer.
New memory is unaliased, so effectively no Input or Output.