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
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

Luke thinks

Someone wants

a house in

TriBeCa
Information flow

Luke thinks
Someone wants a house in TriBeCa
Luke thinks

Someone wants a house in TriBeCa
Luke thinks Someone wants a cheap house in TriBeCa
Abstracting away

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

1 Introduction
   - Motivation
   - Abstracting Away

2 Secure Information Flow
   - Tracking Effects
   - A Monadic Security Calculus
   - Informativeness

3 Non-interference proof

4 Related and Future Work
Tracking Information Flow

Key Idea

Only two things matter:

- Communication channels
- Computations with side-effects
### Key Idea

Only two things matter:
- Communication channels

### Channels

#### Memory
Mark each cell with security level:
- zipfileField
- 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 $\mathcal{L} = (\bot, T, \sqsubseteq, \sqcup, \sqcap)$

Example

\[\sqsubseteq\]
Operation Levels

Definition
An operation level \( o \) is a pair \((In, Out)\) of security levels with \( In \subseteq Out \).

Operation levels form a meet-semilattice \( O = (\bot, \leq, \lor) \) given by

\[
(r_1, w_1) \leq (r_2, w_2) \iff (r_1 \sqsubseteq r_2 \text{ and } w_2 \sqsubseteq w_1)
\]

Example

Security Levels

(\(\), \(\) ) \(\rightarrow\) (\(\), \(\) ) \(\rightarrow\) (\(\), \(\) )
Tracking inputs and outputs
First try at LHH.com

Example (A very imperative example)

```plaintext
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)

```plaintext
proc processForm () {
    houses :=
        fetchHouses (zipcodeField);  \textit{In: \textbullet{}}, \textit{Out: \textbullet{}, \textbullet{}}. ✔
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses,
        priceRange);
}
```
Tracking inputs and outputs
First try at LHH.com

Example (A very imperative example)

```plaintext
proc processForm () {
    houses :=
        fetchHouses (zipcodeField);
    priceRange :=
        calcPriceRange (incomeField);  In: 🔒, Out: 🔒. ✓
    showJustAffordable (houses,
                      priceRange);
}
```
Example (A very imperative example)

```plaintext
proc processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
    In:  Out: .
}
```
Example (A very imperative example)

```
proc processForm () {
    houses := 
        fetchHouses (zipcodeField);
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses,
        priceRange);
    In: 📂, 锁 Out: 锁. ?
}
```
Example (A very imperative example)

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

To Combine Inputs: Join

In:  🗞️ Out: 🗞️ ✔️
Example (What if houses is secret)

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

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

To Combine Outputs: Meet

```plaintext
[ ] ⊓ = [ ]
```
Example (What if houses is secret)

```javascript
proc processForm () {
    houses := fetchHouses (zipcodeField);
    priceRange := calcPriceRange (incomeField);  // In: 🔒, Out: 🔒. ✔
    showJustAffordable (houses, priceRange);
}
```
Example (What if \texttt{houses} is secret)

\begin{verbatim}
proc processForm () {
    houses :=
        fetchHouses (zipcodeField);
    priceRange :=
        calcPriceRange (incomeField);
    showJustAffordable (houses, priceRange);
    \texttt{In: \text{\textbullet}, \text{\textbullet} Out: \text{\textbullet\textbullet}. \checkmark}
}
\end{verbatim}
### A monadic language

#### Syntax (Types)

| Types          | $A ::= \cdots | A \rightarrow B | \cdots $ |
|----------------|--------------------------------------------------|
|                | $\mid \text{Ptr } A \mid \bigcirc A$            |
| Contexts       | $\Gamma ::= \cdot | \Gamma, x:A$            |
A monadic language

Syntax (Types)

Types
\[ A ::= \cdots | A \rightarrow B | \cdots \]
\[ \quad | \textbf{Ptr} A | \odot A \]

Contexts
\[ \Gamma ::= \cdot | \Gamma, x:A \]

Syntax (Terms)

Pure terms
\[ M, N ::= x \mid \cdots \mid \textbf{lam} x.M \mid M \; N \mid \cdots \]
\[ \quad | \textbf{do} \; E \]

Effectful Expressions
\[ E, F ::= \textbf{return} \; M \]
\[ \quad | x \leftarrow M \; ; \; E \]
\[ \quad | M \; ; \; E \]
\[ \quad | M \]
Example (increment)

\[
inc : (\text{Ptr Int, Int}) \rightarrow \text{Int}
\]

\[
inc (\text{ptr, amt}) = \\
do \\
\text{oldVal} \leftarrow \text{deref} \text{ ptr} \\
\text{ptr} := \text{oldVal} + \text{amt} \\
\text{return} \text{ oldVal}
\]
Example

\textit{repeatUntil} :
\begin{align*}
(A \rightarrow \mathbf{Bool}, \bigcirc A) &\rightarrow \bigcirc A \\
\text{repeatUntil} &\quad (\text{test, comp}) = \\
\text{do} &\quad \text{x} \leftarrow \text{comp} \\
\text{if} &\quad (\text{test} (\text{x})) \\
\text{then} &\quad \text{return} \; \text{x} \\
\text{else} &\quad \text{repeatUntil} \\
&\quad (\text{test, comp})
\end{align*}
Writing monadic programs

Control flow

Example

\[
\text{repeatUntil} : (A \rightarrow \text{Bool}, \circ A) \rightarrow \circ A
\]

\[
= \text{do}
\]

\[
x \leftarrow \text{comp}
\]

\[
\text{if} (\text{test} (x))
\]

\[
\text{then return } x
\]

\[
\text{else repeatUntil}
\]

\[
\text{(test, comp)}
\]

Example (Sample Output)

State = \{ptr \mapsto 0\}

\[
\text{repeatUntil} (\text{above100}, \text{inc} (\text{ptr}, 1))
\]
Writing monadic programs
Control flow

Example

\[ \text{repeatUntil} : \]
\[ (A \rightarrow \text{Bool}, \bigcirc A) \rightarrow \bigcirc A \]

\[ \text{repeatUntil} \]
\[ (\text{test, comp}) = \]
\[ \text{do} \]
\[ x \leftarrow \text{comp} \]
\[ \text{if (test (x))} \]
\[ \text{then return } x \]
\[ \text{else } \text{repeatUntil} \]
\[ (\text{test, comp}) \]

Example (Sample Output)

State = \{ptr \mapsto 0\}

do
\[ x \leftarrow \text{inc (ptr, 1)} \]
if (\text{above100 (x)})
\[ \text{then return } x \]
else \text{repeatUntil}
\[ (\text{above100, inc(ptr, 1)}) \]
Writing monadic programs

Control flow

Example

repeatUntil : 
  (A → Bool, ○A) → ○A
repeatUntil
  (test, comp) =
do
  x ← comp
  if (test (x))
    then return x
  else repeatUntil
    (test, comp)

Example (Sample Output)

State = {ptr ↦ 0}
do
⇒ x ← inc (ptr, 1)
  if (above100 (x))
    then return x
  else repeatUntil
    (above100, inc(ptr, 1))
Writing monadic programs

Control flow

Example

\[ \text{repeatUntil} : \]
\[ (A \rightarrow \text{Bool}, \, \bigcirc A) \rightarrow \bigcirc A \]
\[ \text{repeatUntil} \]
\[ (\text{test}, \, \text{comp}) = \]
\[ \text{do} \]
\[ x \leftarrow \text{comp} \]
\[ \text{if } (\text{test} (x)) \]
\[ \text{then return } x \]
\[ \text{else } \text{repeatUntil} \]
\[ (\text{test}, \, \text{comp}) \]

Example (Sample Output)

State = \{\text{ptr} \mapsto 1\}
\[ \text{do} \]
\[ x \leftarrow \text{inc} (\text{ptr}, \, 1) \]
\[ \Rightarrow \text{if } (\text{above100} (0)) \]
\[ \text{then return } 0 \]
\[ \text{else } \text{repeatUntil} \]
\[ (\text{above100}, \]
\[ \text{inc}(\text{ptr}, \, 1)) \]
Writing monadic programs

Control flow

Example

\( \text{repeatUntil} : \quad (A \to \text{Bool}, \ A) \to \ A \)

\( \text{repeatUntil} \)

\( (\text{test}, \ \text{comp}) = \)

do

\( x \leftarrow \text{comp} \)

if \( (\text{test} (x)) \)

then return \( x \)

else \( \text{repeatUntil} \)

(test, comp)

Example (Sample Output)

State = \{ptr \leftrightarrow 1\}

if \( \text{above100} (0) \)

then return \( 0 \)

else \( \text{repeatUntil} \)

(above100, \inc (ptr, 1))
Writing monadic programs

Control flow

Example

\[ \text{repeatUntil} : (A \to \text{Bool}, \bigcirc A) \to \bigcirc A \]

\[
\text{repeatUntil} (\text{test}, \text{comp}) =
\]

\[
\begin{align*}
\text{do} & \quad x \leftarrow \text{comp} \\
\text{if} & \quad (\text{test} (x)) \\
& \quad \text{then return } x \\
\text{else} & \quad \text{repeatUntil} (\text{test}, \text{comp})
\end{align*}
\]

Example (Sample Output)

State = \{ptr \mapsto 1\}

\[ \text{repeatUntil (above100, inc (ptr, 1))} \]
Writing monadic programs

Control flow

Example

```
repeatUntil : 
    (A → Bool, ⌣A) → ⌣A

repeatUntil
    (test, comp) =
do
    x ← comp
    if (test (x))
        then return x
    else repeatUntil
        (test, comp)
```

Example (Sample Output)

```
State = \{ptr ↦ 1\}

repeatUntil
    (above100,
     inc (ptr, 1))
```
The Language with security monads

Syntax (Types)

| Security Levels     | a, $ln$, $Out$ ::= 🔴 | 🔨 |
|---------------------|-------------------------|
| Types               | $A ::= \cdots | A \rightarrow B | \cdots$ | $\text{Ptr}_a A | \bigcirc (ln, Out) A$ |
| Contexts            | $\Gamma ::= \cdot | \Gamma, x:A$ |

Syntax (Terms)

| Pure terms          | $M, N ::= x | \cdots | \text{lam} x.M | M \ N | \cdots$ | $\text{do} \ E$ |
| Effectful Expressions | $E, F ::= \text{return} \ M$ |
|                     | $| x \leftarrow M ; E$ |
|                     | $| M ; E$ |
|                     | $| M$ |
The Language with security monads

Syntax (Types)

Security Levels  \( a, In, Out \) ::=

Types  \( A ::= \cdots | A \rightarrow B | \cdots | \text{Ptr}_a A | \circ (In,Out) A \)

Contexts  \( \Gamma ::= \cdot | \Gamma, x:A \)

Syntax (Terms)

Pure terms  \( M, N ::= x | \cdots | \text{lam}x.M | M \ N | \cdots | \text{do} E \)

Effectful Expressions  \( E, F ::= \text{return} \ M \ \ | \ x \leftarrow M ; E \ \ | \ M ; E \ \ | \ M \)
LHH.com with security monads

Types

Monadic code

\[
\text{processForm} = \\
\begin{align*}
\text{do} & \\
& \text{h} \leftarrow \text{fetchHouses} \text{ (zipcodeField)} \\
& \text{do} \\
& \quad i \leftarrow \text{deref} \text{ incomeField} \\
& \quad \text{showJustAffordable} \text{ (h, calcPriceRange (i))}
\end{align*}
\]
LHH.com with security monads

Types

\[ \text{fetchHouses} : \text{Ptr ZipCode} \rightarrow \circ \text{List House} \]

Monadic code

\[
\text{processForm} =
\begin{align*}
do & \quad \text{h} \leftarrow \text{fetchHouses} (\text{zipcodeField}) \\
do & \quad \text{i} \leftarrow \text{deref incomeField} \\
& \quad \text{showJustAffordable} (\text{h}, \text{calcPriceRange} (\text{i}))
\end{align*}
\]
LHH.com with security monads

**Types**

\[ \text{deref incomeField : } \bigcirc \text{Int} \]

**Monadic code**

\[
\text{processForm = do}
\]

\[
\text{h } \leftarrow \text{fetchHouses (zipcodeField)}
\]

\[
\text{do}
\]

\[
i \leftarrow \text{deref incomeField}
\]

\[
\text{showJustAffordable (h, calcPriceRange (i))}
\]
LHH.com with security monads

**Types**

calcPriceRange : Int → 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) → ○idental, ()
```

Monadic code

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

Typing rule

\[
\Gamma \vdash M : \bigcirc_{o_1} A \\
\Gamma, x:A \vdash E : \circ_{o_2} C \\
\circ_1 \leq o, \circ_2 \leq o \\
\Gamma \vdash x \leftarrow M; E \vdash_{o} C
\]

Portion of process `Form`

```plaintext
do
  i ← deref incomeField : \bigcirc\textbf{Int}
  showJustAffordable
    (h, calcPriceRange (i)) \vdash_{\textbf{()}}
```

20 / 42
Composing suspensions

Typing rule

\[
\frac{\Gamma \vdash M : \bigcirc A}{\Gamma \vdash x : A \vdash E \div o C} \quad (o_1 \leq o, o_2 \leq o)
\]

\[
\frac{\Gamma \vdash x \leftarrow M; E \div o C}{\Gamma \vdash x \leftarrow M; E \div o C}
\]

Portion of processForm

\[
do\ 
i \leftarrow \text{deref incomeField} : \bigcirc \text{Int}
\]

\[
\text{showJustAffordable} (h, \text{calcPriceRange} (i)) \div (())
\]

\[
\div (())
\]
Composing suspensions, cont’d

```
processForm
  do
    h ← fetchHouses
      (zipcodeField) : ○ List House
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
    : ○ ()
```
Composing suspensions, cont’d

processForm

\[
\text{do}
\]

\[
h \leftarrow \text{fetchHouses} \\
\quad (\text{zipcodeField}) : \text{List House}
\]

\[
\text{do}
\]

\[
i \leftarrow \text{deref incomeField}
\]

\[
\text{showJustAffordable} \ (h, \ \text{calcPriceRange} \ (i)) \\
\quad : \text{List} \\
\quad \div
\]

\[
\text{Compose Inputs}
\]

\[
\text{Compose Outputs}
\]
What are the types telling us?

Suppose `showJustAffordable` had type

\[(\text{List House, Range}) \rightarrow \text{Int}\]

Consider

```haskell
do
  nHouses ← do
    i ← \text{deref incomeField}
    \text{showJustAffordable} (...) : \text{Int}
  sendL (nHouses) : ()
```
A type $A$ is **informative only at high-security** if no computation with low-security input can make use of it.

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

**Typing rule**

\[
M : \bigcirc \quad A \quad A \rightarrow
\]

\[\text{upcall} \quad M : \bigcirc \quad A\]
processForm is well-typed

Promote the inner do-block

```plaintext
processForm =
do
  h ← fetchHouses (zipcodeField) : List House
  upcall do
    i ← deref incomeField
    showJustAffordable (...) : ()
  : ()
```

Other informative types?

Rules for the $A \rightarrow$ judgment

Rule

$\text{Ptr} \ A \rightarrow$
Other informative types?

Rules for the $A \notightarrow$ judgment

Rule

\[
\frac{A \notightarrow}{\text{Ptr} \ A \notightarrow}
\]
Other informative types?

Rules for the $A \not\rightarrow$ judgment

Rule

$$\frac{A \not\rightarrow B}{(A, B) \not\rightarrow}$$
Other informative types?

Rules for the $A \rightarrow$ judgment

\[
\frac{A \rightarrow}{A + B \rightarrow} \quad \times \quad \frac{B \rightarrow}{A + B \rightarrow} \quad \times
\]
Other informative types?

Rules for the $A \rightarrow$ judgment

**Rule**

\[
\frac{B \rightarrow}{A \rightarrow B}
\]
Other informative types?

Rules for the $A \rightsquigarrow$ judgment

Rule

\[
\begin{array}{c}
\circ\quad A \\
\hline
\rightarrow
\end{array}
\]
Secure computation continuation passing

Example

\[
\text{do}
\begin{align*}
\text{cont} & \leftarrow \text{highComp} \\
\text{lowComp} & \\
\text{cont}
\end{align*}
\]
Secure computation continuation passing

Example

do
    cont ← upcall highComp
    lowComp
    upcall cont
1 Introduction
   Motivation
   Abstracting Away

2 Secure Information Flow
   Tracking Effects
   A Monadic Security Calculus
   Informativeness

3 Non-interference proof

4 Related and Future Work
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 $\vdash P_1 \approx P_2 \div (ln,-) A$
- Prove two lemmas:
  - **High Security Step** If $ln = \mathbf{\_}$ then $P_1$ and $P_2$ can do whatever they want and stay related.
  - **Hexagon Lemma (relative confluence)** If $ln = \mathbf{\_}$ 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

\[
\frac{\Sigma_1; \Gamma \vdash V_1 : A \quad \Sigma_2; \Gamma \vdash V_2 : A}{\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx V_2 : A}
\]

Lemma (Functionality)

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

Lemma (Single High Security Step)

If \( \vdash P \div A \) and \( P \rightarrow P' \) then “memory of \( P \)” \( \approx \) “memory of \( P' \)”
Hexagon Lemma

\[ \vdash P_1 \approx P_2 \div A \]
Hexagon Lemma

\[ \vdash P_1 \approx P_2 \div A, \quad P_1' \leadsto P_1'' \quad P_2' \leadsto P_2'' \div A \]
Outline

1 Introduction
   Motivation
   Abstracting Away

2 Secure Information Flow
   Tracking Effects
   A Monadic Security Calculus
   Informativeness

3 Non-interference proof

4 Related and Future Work
Related work

Monads in language design

Foundations

- Semantics [Moggi 1989]

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]
- and much more...
Related work
Language-based security

<table>
<thead>
<tr>
<th>Non-interference:</th>
<th>Extensions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• SLam [Heintze, et al. 1998], DCC [Abadi et al. 1999]</td>
<td>• Timing channels [Agat 2000]</td>
</tr>
<tr>
<td>• and many others ...</td>
<td>• and many others...</td>
</tr>
<tr>
<td>[Sabelfeld, et al. 2003]</td>
<td></td>
</tr>
</tbody>
</table>
Our contributions

- Informativeness
- Monads for tracking information flow
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

fetchHouses : \textbf{Ptr} ZipCode → ∅, List House

calcPriceRange : Int → Range

New!

For a fixed security lattice, a Haskell library of security monads (as long as you don't use \texttt{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 $\lozenge \Box A$
Questions?
The three monad laws
Program equivalences

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

1. do x ← return expr
   func (x)

2. do x ← comp
   return x

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

1. do comp
   func (expr)

2. do comp
   comp

3. do
   x ← comp1
   x ← comp2 (x)
   x ← comp3 (y)
Differences in the paper

<table>
<thead>
<tr>
<th>Additions in the paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Full lattice of security levels</td>
</tr>
<tr>
<td>• Subtypes of $\text{Ptr}_aA$ type</td>
</tr>
<tr>
<td>• A proof of non-interference</td>
</tr>
<tr>
<td>• Discussion of allocation</td>
</tr>
<tr>
<td>• Encoding of a prior work</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stylistic differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Pfenning-style expressions vs. do-notation</td>
</tr>
</tbody>
</table>
Allocation

**Rule**

\[
\Gamma \vdash M : A \\
\Gamma \vdash \text{alloc } M : \odot \text{Ptr}_{\alpha} A
\]

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