A Logic for Information Flow

in Object-oriented Programs

A Logic for Information Flow

Joint work with Torben Amtoft and Shruthi Bandhakavi

Kansas State University

http://www.cis.ksu.edu/~ab

Anindya Banerjee

http://www.cis.ksu.edu/~ab
Permits JML-style programmer assertions.

Flow-sensitive specs.

Uses alias information ([Jif, Banerjee/Naumann] don't).

Uses local reasoning about state [O'Hearn/Reynolds/Tang/...]

Specialization for interprocedural information flow
Information flow regulates confidentiality.

- Data is secret (High) or public/observable (Low).
- Confidentiality: High inputs do not influence Low output.
- Channelless (end-to-end property).
- Typical analyses based on security types, e.g.,
- Flow sensitive [Volpano/Smith/Irvine, Myers,...]
- Flow insensitive [Hunt/Sands].
Noninterference property [Goguen-Meseguer]: For any two runs of program, $L_{\text{out}}$-indistinguishable input states yield $L_{\text{out}}$-indistinguishable output states.

Equivalently [Cohen]: $L_{\text{out}}$ independent of initial $H_{\text{in}}$. 

Noninterference concept:

- Classified channels
- Unclassified channels
- Program

\[
\begin{align*}
\{ & \text{classified channels} \} & \{ & \text{out} \} \\
\{ & \text{unclassified channels} \} & \{ & \text{in} \} \\
\{ & \text{out} \} & \{ & \text{in} \} \\
\end{align*}
\]
Noninterference property

Equivalently [Cohen]: Low-indistinguishable input states yield Low-indistinguishable output states.

For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.

Noninterference: L := l; l := h; | := h - h; | := h; | := h; | := h.

Insecure:

\[\text{h := l; if h then l := 7 else l := 8 (indirect flow)}\]

Secure:

\[\text{h := l; l := h; h := l} \]
Noninterference

Security types: well-typed programs are noninterferent.

Noninterference property [Goguen-Meseguer]: For any two runs of program, \( L \)-indistinguishable input states yield \( L \)-indistinguishable output states.

Equivalently [Cohen]: \( L \) out independent of initial \( H \) in.

Secure: \( h := \lfloor h \rfloor \)
\[ l := h \]
\[ l := h - h \]
\[ l := h \]
\[ l := l \]
\[ l := h \]
\[ l := l - h \]
\[ l := h \]
\[ l := 7 \]

Insecure: \( h := \lfloor h \rfloor \)
\[ l := h \]
\[ l := 7 \]
\[ l := h \]
\[ l := h \]
\[ l := h \]
\[ l := l \]
\[ l := l \]
\[ l := 8 \]

(indirect flow)
Security types: well-typed programs are noninterferent.

Insecure:

```plaintext
h := l
```

Equiv-alently [Cohen]: L out independent of initial H in.

Secure:

```plaintext
h := l; l := h
```

Noninterference property [Goguen-Meseguer]: For any two runs of program, L-indistinguishable input states yield L-indistinguishable output states.

Noninterference channels

Classified channels

Unclassified channels
Object Examples
Object Examples

\[ x_1.b : \text{secret} = \text{secret} \]

\[ x_2.b : \text{secret} = \text{secret} \]

\[ x_2 = : z \]

\[ x_1.b : \text{secret} = \text{secret} \]
Object Examples

x_1 \cdot b := \text{secret} \quad \text{OK}

x_2 := z \quad \text{OK}

z := x_2 \cdot b \quad \text{OK}
Object Examples
Object Examples

\[ x_1.\ b := \text{secret}; \quad // \text{OK} \]
\[ z := x_2.\ b; \quad // \text{OK} \]

\[ x_1 := x_2; \quad // \text{Reject!} \]

\[ x_1 . b := \text{secret}; \quad // \text{OK} \]
\[ z := x_2.\ b \]
Aliasing distinguishes these examples.

\[ b \cdot x =: z \]

// Reject

\[ x_1 := b \cdot x \]

\[ x_1 := x_2' := \text{secret} \]

// OK

\[ x_2 := x \cdot x \]

// Reject

\[ z := x_2 \cdot b := \text{secret} \]

// OK

\[ x_1 := \text{secret} := x \cdot x \]

// OK

Object Examples
Checking Noninterference

Check (Hoare-style) triple
\[
\{ x_1 \Downarrow \, \ldots \, \Downarrow x_n \} P \{ y_1 \Downarrow \, \ldots \, \Downarrow y_m \}
\]

Independence Assertions

Given any two runs of \( P \):

- If observable inputs \( u_{x_1}, \ldots, u_{x_n} \) agree (precondition), then observable outputs \( u_{y_1}, \ldots, u_{y_m} \) agree in the same two runs (postcondition).

Check (Hoare-style) triple
\[
\{ x_1 \Downarrow \, \ldots \, \Downarrow x_n \} P \{ y_1 \Downarrow \, \ldots \, \Downarrow y_m \}
\]
Checking Noninterference

Check (Hoare-style) triple

\( \{ x_1, \ldots, x_n \} P \{ y_1, \ldots, y_m \} \)

Independence Assertions

... Independence Assertions ...

\( \{ x_1, \ldots, x_n \} \parallel \{ y_1, \ldots, y_m \} \)

Check (Hoare-style) triple

Two-state semantics of assertions correspond to two runs of program:

\( \begin{align*}
\text{If observable inputs } x_1, \ldots, x_n \text{ agree (precondition)} & \implies \\
\text{Then observable outputs } y_1, \ldots, y_m \text{ agree in the same two runs (postcondition).} & \implies \\
\text{If observable inputs } x_1, \ldots, x_n \text{ agree (precondition)} & \implies \\
\text{Then any two runs of } P & \implies
\end{align*} \)

POPL, 2006-1-11 / 6-a
Program secure. ⊢

\[\{l \triangleright \text{lost}\} \quad \{l \triangleright \text{recovered}\}\]

\[0 =: l\]

\[\eta =: l\]

\[\{l \triangleright \} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\}

\text{Does}\}

\[0 =: l, \eta =: l \quad \{l \triangleright \} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\} \quad \{\}

\text{Example:}\]
Proof rules:

\[ \{ \phi \} \ \Box \ C \ \{ \phi' \} \]

\[ \begin{array}{ll}
\phi & \text{assertions that hold in precondition.} \\
\phi' & \text{assertions that hold in postcondition.} \\
X & \text{is set of variables that may be modified by command } C. \\
\end{array} \]

Then \( s_1 \models \phi \) and \( [\ [C]\ ] \ s_1 = \ s_1' \) and \( [\ [C]\ ] \ s_2 = s_2' \) and \( \phi \models s_2 \).

Suppose \( s_1 \models \phi \) and \( s_2 \models \phi' \).

Meaning:

\[ [X] \{ \phi \} \ C \ \{ \phi' \} \]
\[
\begin{array}{c}
\mathbb{F}\{x\} \{\times x\} =: x \{\times u_z \cdot \cdots \cdot \times l_z\}
\end{array}
\]

Assignment rule
Frame rule, because these variables not modified.

In larger context, can add extra variables (except $x$) by

Small specification: provides bare essence of reasoning.

Local reasoning: Only $z_1, \cdots, z_n$ and $x$ relevant to $x := E$.

\[
\begin{align*}
\{\{x\}\} \{\times x\} \mathcal{E} := \ x \ {\times \ u \ \ldots \ \times \ l z} \\
(\mathcal{E}) \ fresh \ = \ \{u \ z, \ \ldots, l z\}
\end{align*}
\]
Frame rule

Frame rule permits move from local to non-local specs. Crucial for modular analysis.

\[ \phi \}
\{ \phi' \}
\{ X \}

\{ \phi \land \phi_1 \}
\{ \phi' \land \phi_1 \}
\{ X \}

if \phi_1 \circ X.

\phi_1 \circ X means variables mentioned in \phi_1 disjoint from X (not modified by C).

\phi_1 \circ X means variables mentioned in \phi_1 same before and after execution of C.

\phi_1 \circ X is invariant for C.

\phi_1 \circ X means variables mentioned in \phi_1 disjoint from X (not modified by C).

Frame rule permits move from local to non-local specs. Crucial for modular analysis.
Can’t compose because \( x \times l \) don’t match!

\[
\begin{align*}
\text{[[h},x] \{\times l\} l &=: h \{\times l\} x \\
\{h\} \{\times h\} l &=: h \{\times l\} \\
\{x\} \{\times x\} l &=: x \{\times l\}
\end{align*}
\]

Example: \( l =: h \{\times l\} =: x \)
\[
[\{\text{i}, x\}] \ {\times} x \ {\times} \text{i} \ {\vdash} \text{i}, x =: x \ {\times} 1 \\
[\{\text{i}\}] \ {\times} x \ {\times} \text{i} \ {\vdash} \text{i} \ {\times} x \ {\times} 1 \\
[\{x\}] \ {\times} 1 \ {\times} x \ {\vdash} x \ {\times} 1
\]

\text{(} 1 \ {\vdash} \text{i} \text{) not modified in } x; \ \text{i} \ {\vdash} x \text{ not modified in } 1.

\text{Frame to rescue!}

\text{Can't compose because } x \ {\times} 1 \text{ don't match!}

\[
[\{\text{i}, x\}] \ {\times} \text{i} \ {\vdash} \text{i}, x =: x \ {\times} 1 \\
[\{\text{i}\}] \ {\times} \text{i} \ {\vdash} \text{i} \ {\times} 1 \\
[\{x\}] \ {\times} x \ {\vdash} x \ {\times} 1
\]

\text{Example:}
Alias analysis (in logical form)

- Not performed by previous approaches for info. flow.
- Want local reasoning about aliasing: use small specs.
  - Use abstract locations $L$, which abstract sets of concrete locations.
  - Abstract addresses are variables or $f$ (abstracting heap-allocated value, e.g., $x.f$)
  - $L_1 \land L_2$ holds provided $L_1$ and $L_2$ abstract disjoint sets of concrete locs.

Not performed by previous approaches for info. flow.

Alias analyses (in logical form)
alias. Otherwise, \( x \) may alias \( y \). If \( x \bowtie L_1 \) and \( y \bowtie L_2 \) and \( L_1 \cong L_2 \) then \( x \), \( y \) must not alias. Otherwise, \( x \) may alias \( y \). If \( L_1.f \) contains \( L_2 \), then \( L_2 \) is abstracted by \( L_1 \). If \( L_1 \bowtie L_2 \), then any concrete loc. \( L_1 \) abstracted by \( L_1 \), if \( L_1 \bowtie L_2 \), then \( x \). Otherwise, \( L_1 \bowtie L_2 \).
Region assertions

\( \text{alias. Otherwise, } x, y \text{ may alias.} \)

\( x \xRightarrow{I \cap} x \) is another popular notation.

\( \text{If } x \xRightarrow{I_1} \text{ and } y \xRightarrow{I_2} \text{ then } x, y \text{ must not alias. Otherwise, } x, y \text{ may alias.} \)

\( f: I_1 \xRightarrow{f} I_2: \text{ for any concrete loc. } I_1 \text{ abstracted by } I_1, I_2. \)

\( I_1 \xRightarrow{I} I_2 \) abstracts concrete loc. denoted by \( x \).
Some small specs. for alias analysis
Need independences on abstract addresses; e.g.

\( x \ltimes \top \). Have

Back to independences
\[
\{x\} \\
\text{x} \Rightarrow x \\
f \cdot \tilde{h} =: x \\
\{ \times f \cdot \tilde{I}, \times \tilde{I} \} \Leftarrow f \cdot \tilde{I}, \tilde{I} \Leftarrow \tilde{h} \\
\text{FieldAccess}
\]
 establishing no aliasing

Aliasing examples revisited

\[ x_1, x_2 \]
must be in same abs. loc.

\[ x_1 \rightarrow L_1, x_2 \rightarrow L_2, L_1 \not\sqsubset L_2 \]

\[ x_1 \gets secret; \]

\[ x_1 \cdot b = secret; \]

\[ x_1 \cdot b : secret \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]

\[ x_1 \cdot b : secret; \]
Observational purity

Typically use pure functions in specifications.

Also, can use methods with "benevolent side-effects" [Hoare].

Typically use pure functions in specifications.
(i) Show \( m \) modifies only locations not visible to caller.

(ii) Show result depends only on \( x \).

```java
class C {
    private Hashtable t = new Hashtable();
    // cache with key, val fields
    public U m(T x) {
        // memo function
        if (!t.contains(x)) {
            U y = costly(x);
            t.put(x, y);
        }
        U res = t.get(x);
        assert res = costly(x);
        return res;
    }
}
```

(i) Show \( m \) modifies only locations not visible to caller.

(ii) Show result depends only on \( x \).
Example class C {
1. private Hashtable t := new Hashtable; // cache with key, val fields
2. private Hashtable t := new Hashtable; // new HashTable
3. if (t.containsKey(x)) {'
4. y := costly(x); t.put(x, y);'
5. res := t.get(x);'
6. assert res = costly(x);'
7. Result := res;
8. return;
'}

(i) Show m modifies only locations not visible to caller.
(ii) Show result depends only on x. Assume x ⋈. Show x ⋈.

Example

```java
Example class C {

1. private Hashtable t := new Hashtable;
   // cache with key, val fields

2. public U m(T x) {
   // memo function
   if (!t.contains(x)) {
   x ⋉
   t.put(x, costly(x));
   }
   else {x ⋉
   }

3. {∀x
   if (t.contains(x)) {
   y := costly(x);
   t.put(x, y);
   }
   else 
   module
   }

4. {∀x
   result := res;
   }

5. {∀res
   res := costly(res);
   assert res = costly(res);
   }

6. {∀res
   res := costly(res);
   }

7. {∀res
   res := costly(res);
   assert res = costly(res);
   }

8. {∀result
   result := res;
   assert res = costly(res);
   }

(ii) Show m modifies only locations not visible to caller.
(i) Show result depends only on x. Assume x ⋉.
Assume x ⋉. Show result ⋉.

```
Example

```java
public C {
private Hashtable t := new Hashtable;
// cache with key, val fields

1. private Hashtable t := new Hashtable;
//

2. public U m(T x) {
  // memo function

3. if (!t.contains(x)) {

4. y := costly(x);
  t.put(x, y);

5. U res := (U) t.get(x);

6. assert res := costly(x);

7. result := res;

8. result depends only on x. Assume x ⋉.
}

} }

(ii) Show m modifies only locations not visible to caller.

(i) Show result depends only on x. Assume x ⋉.

♦ Assume L0 disjoint from all abstract locations used outside of m.

♦ Assume t = L0. Only t = key, L0 = val modified (by put).
```

(i) Show m modifies only locations not visible to caller.

(ii) Show result depends only on x. Assume x ⋉.
```

- Reason about observational purity, selective dependency.
- Postcondition can be computed.
- With region and independence assertions, strongest postcondition can be computed.
- There exists a sound algorithm to compute postconditions.
- Given method environment, precondition, precondition and command,
- Considered sequentinal Java-like language with programmer assertions (as in JML).
- Crucial: Interprocedural alias analysis; uses local reasoning.
- Spec. for interproc. into flow analysis; uses local reasoning.
Future Work

• In general, interested in using local reasoning for program analysis (small specs., disjointness, reasoning via Frame).

• Build a modular verifier for info. flow (or other) properties – maybe extend JML? Specify other analyses on top of alias analysis.

• Declassification: use richer assertion language, e.g., FOL, e.g., $\theta \Rightarrow x \in \theta$, where $\theta$ are assertions on events?

• Use, e.g., $\theta \subseteq x \times \theta$, where $\theta$ are assertions on events?

• Completeness of logic wrt underlying abstract interpretation.

• Support local reasoning for concurrency.

• Maybe extend JML? Specify other analyses on top of alias analysis.

• Build a modular verifier for info. flow (or other) properties – analyses (small specs., disjointness, reasoning via Frame).

• In general, interested in using local reasoning for program analysis.
Some references


