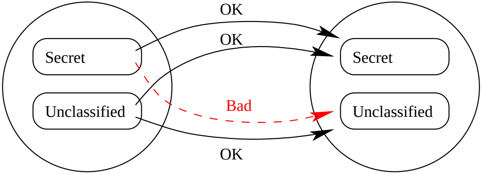
Mechanized Information Flow Analysis through Inductive Assertions

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu
http://www.cs.utexas.edu/users/sandip

(Joint Work with Warren A. Hunt, Jr., Robert B. Krug, and William D. Young)

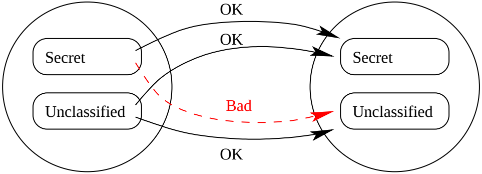
Goal

Information flow policies restrict inappropriate access to sensitive information.



Goal

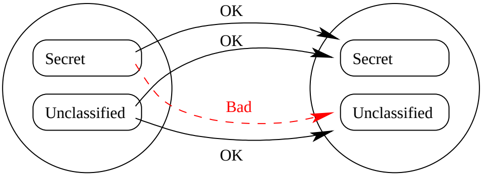
Information flow policies restrict inappropriate access to sensitive information.



Security of many systems depend on the system correctly implementing information flow policies.

Goal

Information flow policies restrict inappropriate access to sensitive information.



Security of many systems depend on the system correctly implementing information flow policies.

Our Contribution: A generic, compositional, mechanized infrastructure for verifying information flow properties of software implementations.

Formalizing Information Flow: Background

Information flow properties are naturally formalized by a statement of **noninterference** (Goguen and Meseguer, 1982).

Formalizing Information Flow: Background

Information flow properties are naturally formalized by a statement of **noninterference** (Goguen and Meseguer, 1982).

- Let s and s' be any two initial states that have the same values of unclassified variables.
- Any computation from s and s' leads to final states that have identical values of unclassified variables.

Formalizing Information Flow: Background

Information flow properties are naturally formalized by a statement of **noninterference** (Goguen and Meseguer, 1982).

- Let s and s' be any two initial states that have the same values of unclassified variables.
- Any computation from s and s' leads to final states that have identical values of unclassified variables.

There has been significant research on specification and verification of noninterference since the 1980s. (Rushby, 1982; Haigh and Young, 1987)

Formalizing Information Flow: Background

Information flow properties are naturally formalized by a statement of **noninterference** (Goguen and Meseguer, 1982).

- Let s and s' be any two initial states that have the same values of unclassified variables.
- Any computation from s and s' leads to final states that have identical values of unclassified variables.

There has been significant research on specification and verification of noninterference since the 1980s. (Rushby, 1982; Haigh and Young, 1987)

Nointerference naturally extends to a lattice of security levels.

Formalizing Information Flow: Definitions

Quick Preliminaries:

- A state is a valuation of variables.
- If l is a variable, $l(s)$ is the value of l in state s.
- $step(s)$ returns the state after one transition from s.

Formalizing Information Flow: Definitions

Quick Preliminaries:

- A state is a valuation of variables.
- If l is a variable, $l(s)$ is the value of l in state s.
- $step(s)$ returns the state after one transition from s.

Some Definitions:

$$pre(s, s') \triangleq poised(s) \wedge poised(s') \wedge (\textstyle\bigwedge_{l \in L} l(s) = l(s'))$$
$$post(s, s') \triangleq (\textstyle\bigwedge_{l \in L} l(s) = l(s'))$$

Formalizing Information Flow: Definitions

Quick Preliminaries:

- A state is a valuation of variables.
- If l is a variable, $l(s)$ is the value of l in state s.
- $step(s)$ returns the state after one transition from s.

Some Definitions:

$$pre(s, s') \triangleq poised(s) \wedge poised(s') \wedge (\textstyle\bigwedge_{l \in L} l(s) = l(s'))$$
$$post(s, s') \triangleq (\textstyle\bigwedge_{l \in L} l(s) = l(s'))$$

Noninterference Condition:

If s and s' satisfy *pre*, and a final state t is reached from s, then a corresponding final state t' is reached from s and t and t' satisfy *post*.

Approach

Our approach is based on **inductive assertions**.

Noninterference Condition:
If s and s' satisfy *pre*, and a final state t is reached from s, then a corresponding final state t' is reached from s and t and t' satisfy *post*.

Approach

Our approach is based on **inductive assertions**.

Noninterference Condition:
If s and s' satisfy *pre*, and a final state t is reached from s, then a corresponding final state t' is reached from s and t and t' satisfy *post*.

Key observation: Noninterference involves proving certain binary relation is preserved by the code along the computations from s and s'.

This property can be proven by proving the following:

- The relation is preserved along each straight-line code fragment.
- A loop invariant (on pairs of states) preserves the relation along each loop iteration.
- The loops along the two computation paths are always in sync.

Approach

Our approach is based on **inductive assertions**.

Noninterference Condition:
If s and s' satisfy *pre*, and a final state t is reached from s, then a corresponding final state t' is reached from s and t and t' satisfy *post*.

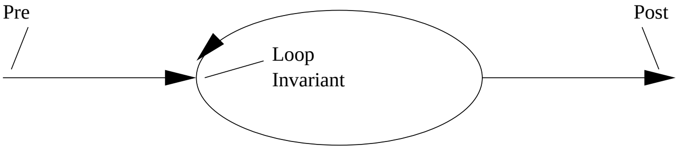
Key observation: Noninterference involves proving certain binary relation is preserved by the code along the computations from s and s'.

This property can be proven by proving the following:

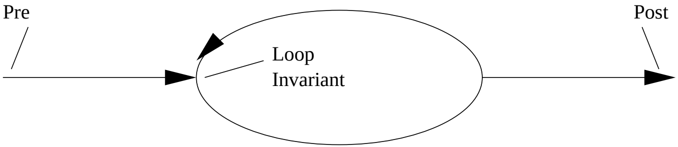
- The relation is preserved along each straight-line code fragment.
- A loop invariant (on pairs of states) preserves the relation along each loop iteration.
- The loops along the two computation paths are always in sync.

This is the essence of inductive assertions.

Inductive Assertions by Symbolic Simulation

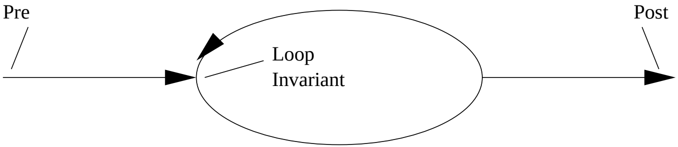


Inductive Assertions by Symbolic Simulation



Previous work showed how to do inductive assertion proofs of functional correctness by configuring the theorem prover as a symbolic simulator.
(Matthews, Moore, **Ray**, Vroon, 2006)

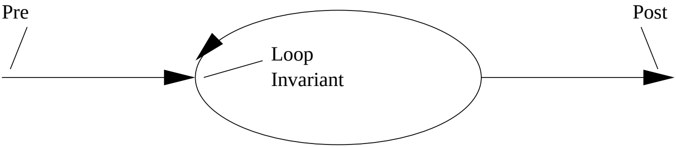
Inductive Assertions by Symbolic Simulation



Previous work showed how to do inductive assertion proofs of functional correctness by configuring the theorem prover as a symbolic simulator. (Matthews, Moore, **Ray**, Vroon, 2006)

The key contribution of the current work is to show how this can be extended for noninterference properties.

Inductive Assertions by Symbolic Simulation



Previous work showed how to do inductive assertion proofs of functional correctness by configuring the theorem prover as a symbolic simulator. (Matthews, Moore, **Ray**, Vroon, 2006)

The key contribution of the current work is to show how this can be extended for noninterference properties.

The symbolic simulation framework now has to guarantee that the pair of computations is in sync.

Verification Conditions for Noninterference

1. $pre(s, s') \Rightarrow C(s, s') \wedge cut(s) \wedge cut(s') \wedge assert(s, s')$
2. $exit(s) \Rightarrow cut(s)$
3. $cut(s) \wedge cut(s') \wedge assert(s, s') \wedge C(s, s')$
   $\wedge \neg exit(s) \wedge exit(run(s, n))$
   $\Rightarrow assert(nextc(step(s)), nextc(step(s')))$
4. $cut(s) \wedge cut(s') \wedge assert(s, s') \wedge C(s, s')$
   $\wedge \neg exit(s) \wedge exit(run(s, n))$
   $\Rightarrow C(nextc(step(s)), nextc(step(s')))$
5. $assert(s, s') \wedge exit(s) \wedge C(s, s') \Rightarrow exit(s')$
6. $assert(s, s') \wedge exit(s) \wedge C(s, s') \Rightarrow post(s, s')$

Noninterference follows from 1-6.

Verification Conditions for Noninterference

1. $pre(s, s') \Rightarrow C(s, s') \wedge cut(s) \wedge cut(s') \wedge assert(s, s')$
2. $exit(s) \Rightarrow cut(s)$
3. $cut(s) \wedge cut(s') \wedge assert(s, s') \wedge C(s, s')$
   $\wedge \neg exit(s) \wedge exit(run(s, n))$
   $\Rightarrow assert(nextc(step(s)), nextc(step(s')))$
4. $cut(s) \wedge cut(s') \wedge assert(s, s') \wedge C(s, s')$
   $\wedge \neg exit(s) \wedge exit(run(s, n))$
   $\Rightarrow C(nextc(step(s)), nextc(step(s')))$
5. $assert(s, s') \wedge exit(s) \wedge C(s, s') \Rightarrow exit(s')$
6. $assert(s, s') \wedge exit(s) \wedge C(s, s') \Rightarrow post(s, s')$

Noninterference follows from 1-6.

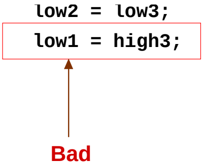
Each condition can be discharged by symbolic simulation using an operational semantics.

SSR1: $\neg cut(s) \Rightarrow nextc(s) = nextc(step(s))$

SSR2: $cut(s) \Rightarrow nextc(s) = s$

Type-based Approaches

- Classify program variables into different security types.
- Check that a low variable is not assigned the value of a high variable.

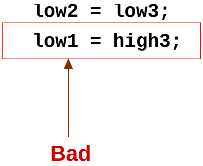


```
low2 = low3;
low1 = high3;
```

Bad

Type-based Approaches

- Classify program variables into different security types.
- Check that a low variable is not assigned the value of a high variable.



```
low2 = low3;
low1 = high3;
```

Bad

But information flow properties are often conflated with functional correctness.

```
 <big hairy code>;
if (result !=1) then {
 low = high;
}
```

Extending Axiomatic Semantics

There has been work done on using inductive assertions by extending Hoare logic to capture information flow.

Extending Axiomatic Semantics

There has been work done on using inductive assertions by extending Hoare logic to capture information flow.

A representative effort by Amtoft and Banerjee (2007).

- $\bowtie$ operator to specify agreement assertions between state pairs
- Axiomatic semantics for “loop flow” and “object flow”.

Extending Axiomatic Semantics

There has been work done on using inductive assertions by extending Hoare logic to capture information flow.

A representative effort by Amtoft and Banerjee (2007).

- $\bowtie$ operator to specify agreement assertions between state pairs
- Axiomatic semantics for "loop flow" and "object flow".

But capturing noninterference through axiomatic semantics is complicated.

The approach also needs a Verification Condition Generator for information flow.

Extending Axiomatic Semantics

There has been work done on using inductive assertions by extending Hoare logic to capture information flow.

A representative effort by Amtoft and Banerjee (2007).

- $\bowtie$ operator to specify agreement assertions between state pairs
- Axiomatic semantics for “loop flow” and “object flow”.

But capturing noninterference through axiomatic semantics is complicated.

The approach also needs a Verification Condition Generator for information flow.

Our approach makes use of the same operational semantics framework as used for functional correctness.

An Illustrative Example

This example is taken from Amtoft and Banerjee's paper.

```
Procedure tricky1 (int high, low, n) {
  int temp = low;
  for i = 0 to n do {
    if even(i) {
      out = out + temp;
      temp = high;
    } else {
      temp = low;
    }
  }
  out = out + 7;
  return out;
}
```

Our approach requires no more creative insight than Amtoft and Banerjee, but does not require additional information flow axioms or infrastructure.

An Illustrative Example

This example is taken from Amtoft and Banerjee's paper.

```
Procedure tricky1 (int high, low, n) {
  int temp = low;
  for i = 0 to n do {
    if even(i) {
      out = out + temp;
      temp = high;
    } else {
      temp = low;
    }
  }
  out = out + 7;
  return out;
}
```

Our approach requires no more creative insight than Amtoft and Banerjee, but does not require additional information flow axioms or infrastructure.

We could easily verify this code with respect to a pre-existing JVM model.

Compositionality

Our approach is compositional.

- Verify subroutines and other program components separately.

Compositionality

Our approach is compositional.

- Verify subroutines and other program components separately.

Compositional verification requires handling frame condition.

- When a subroutine exits, the caller can continue execution.

This is typically handled by characterizing the program components that are modified by the subroutine.

Compositionality

Our approach is compositional.

- Verify subroutines and other program components separately.

Compositional verification requires handling frame condition.

- When a subroutine exits, the caller can continue execution.

This is typically handled by characterizing the program components that are modified by the subroutine.

But for information flow verification, we do not want to develop full functional characterization!

Compositionality

Our approach is compositional.

- Verify subroutines and other program components separately.

Compositional verification requires handling frame condition.

- When a subroutine exits, the caller can continue execution.

This is typically handled by characterizing the program components that are modified by the subroutine.

But for information flow verification, we do not want to develop full functional characterization!

We can handle frame conditions by an additional symbolic simulation that produces fake functional characterization.

Details in the paper.

Concluding Observations

To our knowledge, this is the first framework for information flow analysis through inductive assertions directly on an operational semantics.

- No VCG or axiomatic semantics for information flow is necessary.
- Can handle information flow properties that depend on functional invariants.

Concluding Observations

To our knowledge, this is the first framework for information flow analysis through inductive assertions directly on an operational semantics.

- No VCG or axiomatic semantics for information flow is necessary.
- Can handle information flow properties that depend on functional invariants.

Of course, this work is in very early stages.

We are planning to extend this to handle:

Concluding Observations

To our knowledge, this is the first framework for information flow analysis through inductive assertions directly on an operational semantics.

- No VCG or axiomatic semantics for information flow is necessary.
- Can handle information flow properties that depend on functional invariants.

Of course, this work is in very early stages.

We are planning to extend this to handle:

- dynamic and declassification policies

Concluding Observations

To our knowledge, this is the first framework for information flow analysis through inductive assertions directly on an operational semantics.

- No VCG or axiomatic semantics for information flow is necessary.
- Can handle information flow properties that depend on functional invariants.

Of course, this work is in very early stages.

We are planning to extend this to handle:

- dynamic and declassification policies
- automated static analysis of data structure shapes

Concluding Observations

To our knowledge, this is the first framework for information flow analysis through inductive assertions directly on an operational semantics.

- No VCG or axiomatic semantics for information flow is necessary.
- Can handle information flow properties that depend on functional invariants.

Of course, this work is in very early stages.

We are planning to extend this to handle:

- dynamic and declassification policies
- automated static analysis of data structure shapes
- multithreaded programs