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)
Information flow policies restrict inappropriate access to sensitive information.
**Motivation**

**Goal**

Information flow policies restrict inappropriate access to sensitive information.

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

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

Noninterference naturally extends to a lattice of security levels.
Quick Preliminaries:

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

Noninterference Condition:

If $s$ and $s'$ satisfy $\text{pre}(s, s')$, and a final state $t$ is reached from $s$, then a corresponding final state $t'$ is reached from $s$ and $t$ and $t'$ satisfy $\text{post}(s, s')$. 
**Quick Preliminaries:**
- A state is a valuation of variables.
- If \( l \) is a variable, \( l(s) \) is the value of \( l \) in state \( s \).
- \( \text{step}(s) \) returns the state after one transition from \( s \).

**Some Definitions:**
\[
\begin{align*}
\text{pre}(s, s') & \triangleq \text{poised}(s) \land \text{poised}(s') \land (\land_{l \in L} l(s) = l(s')) \\
\text{post}(s, s') & \triangleq (\land_{l \in L} l(s) = l(s'))
\end{align*}
\]
Quick Preliminaries:
- A state is a valuation of variables.
- If $l$ is a variable, $l(s)$ is the value of $l$ in state $s$.
- $\text{step}(s)$ returns the state after one transition from $s$.

Some Definitions:

$$\text{pre}(s, s') \triangleq \text{poised}(s) \land \text{poised}(s') \land (\bigwedge_{l \in L} l(s) = l(s'))$$

$$\text{post}(s, s') \triangleq (\bigwedge_{l \in L} l(s) = l(s'))$$

Noninterference Condition:
If $s$ and $s'$ satisfy $\text{pre}$, and a final state $t$ is reached from $s$, then a corresponding final state $t'$ is reached from $s$ and $t$ and $t'$ satisfy $\text{post}$.
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 $post$. 
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.
Our approach is based on **inductive assertions**.

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

**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.**
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.
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)*
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.
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. $\text{pre}(s, s') \Rightarrow C(s, s') \land \text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s')$
2. $\text{exit}(s) \Rightarrow \text{cut}(s)$
3. $\text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s') \land C(s, s')$
   $\land \neg \text{exit}(s) \land \text{exit}(\text{run}(s, n))$
   $\Rightarrow \text{assert}(\text{nextc}(\text{step}(s)), \text{nextc}(\text{step}(s')))$
4. $\text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s') \land C(s, s')$
   $\land \neg \text{exit}(s) \land \text{exit}(\text{run}(s, n))$
   $\Rightarrow C(\text{nextc}(\text{step}(s)), \text{nextc}(\text{step}(s')))$
5. $\text{assert}(s, s') \land \text{exit}(s) \land C(s, s') \Rightarrow \text{exit}(s')$
6. $\text{assert}(s, s') \land \text{exit}(s) \land C(s, s') \Rightarrow \text{post}(s, s')$

**Noninterference follows from 1-6.**
Verification Conditions for Noninterference

1. $\text{pre}(s, s') \Rightarrow C(s, s') \land \text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s')$

2. $\text{exit}(s) \Rightarrow \text{cut}(s)$

3. $\text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s') \land C(s, s')$
   $\land \neg \text{exit}(s) \land \text{exit}(\text{run}(s, n))$
   $\Rightarrow \text{assert}(\text{nextc}(\text{step}(s)), \text{nextc}(\text{step}(s'))))$

4. $\text{cut}(s) \land \text{cut}(s') \land \text{assert}(s, s') \land C(s, s')$
   $\land \neg \text{exit}(s) \land \text{exit}(\text{run}(s, n))$
   $\Rightarrow C(\text{nextc}(\text{step}(s)), \text{nextc}(\text{step}(s'))))$

5. $\text{assert}(s, s') \land \text{exit}(s) \land C(s, s') \Rightarrow \text{exit}(s')$

6. $\text{assert}(s, s') \land \text{exit}(s) \land C(s, s') \Rightarrow \text{post}(s, s')$

**Noninterference follows from 1-6.**

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

**SSR1:** $\neg \text{cut}(s) \Rightarrow \text{nextc}(s) = \text{nextc}(\text{step}(s))$

**SSR2:** $\text{cut}(s) \Rightarrow \text{nextc}(s) = s$
Type-based Approaches

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

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

But information flow properties are often conflated with functional correctness.
Type-based Approaches

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

But information flow properties are often conflated with functional correctness.

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

```
<big hairy code>;
if (result != 1) then {
  low = high;
}
```
There has been work done on using inductive assertions by extending Hoare logic to capture information flow.
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).

- \( \circ \) 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).

- ⊻ 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.
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).
- $\times$ 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.

```c
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.
Our approach is compositional.

- Verify subroutines and other program components separately.
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.
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!
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.
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.
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:
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
Conclusion

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
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