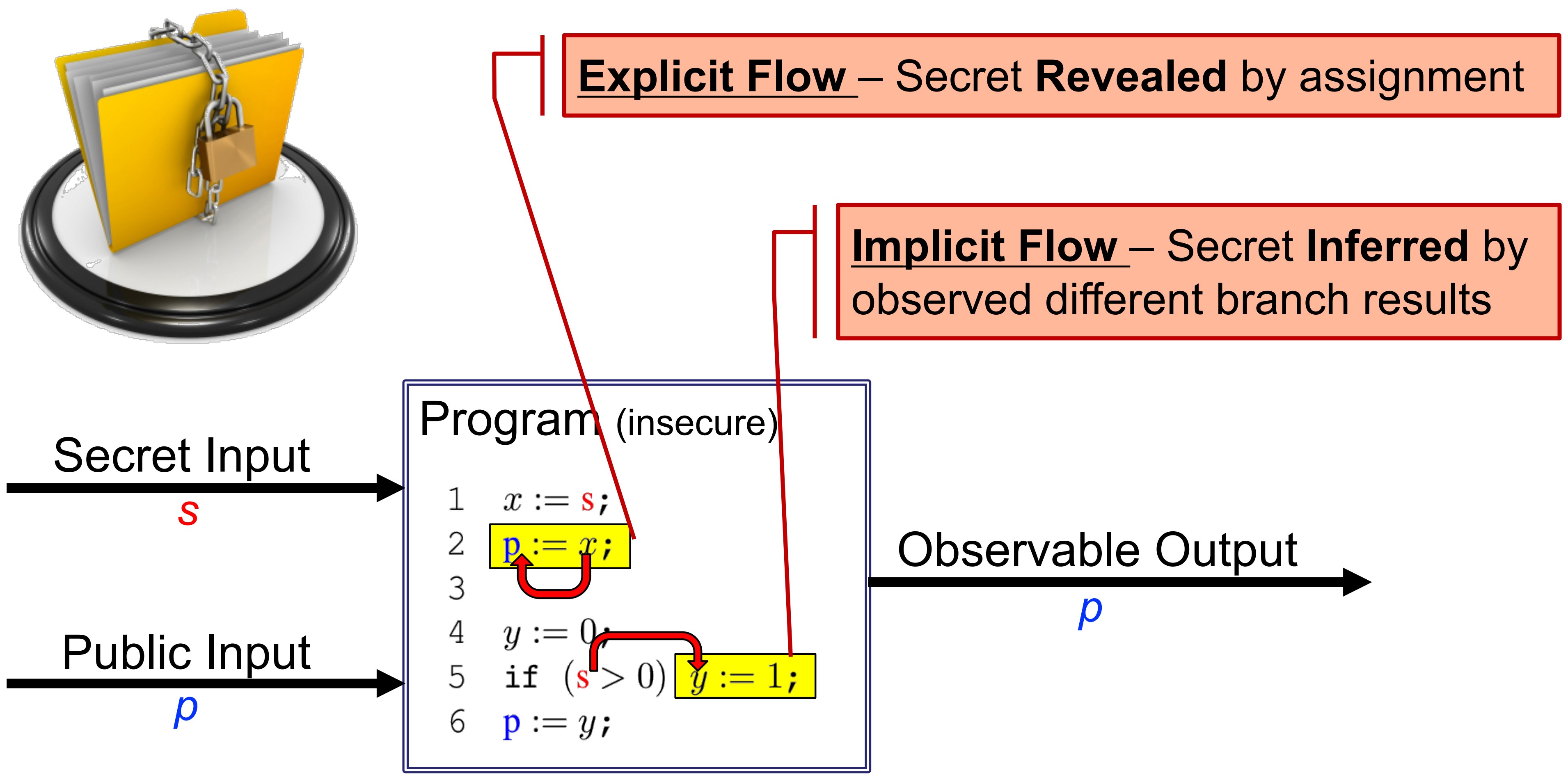
Towards a Flow- and Path-Sensitive Information Flow Analysis

Peixuan Li Danfeng Zhang

Abstract – This paper investigates a flow- and path-sensitive static information flow analysis. Compared with security type systems with fixed labels, it has been shown that flow-sensitive type systems allow accepting more secure programs. We show that an information flow analysis with fixed labels can be both flow- and path-sensitive. The novel analysis has two major components: 1) a general-purpose program transformation that removes false dataflow dependency in a program that confuses a fixed-label type system, and 2) a fixed-label type system that allows security type to depend on path conditions. We formally prove that the proposed analysis enforces a rigorous security property: noninterference. Moreover, we show that the analysis is strictly more permissive than a classical flow-sensitive type system, and it allows sound control of information flow in the presence of mutable variables without resorting to run-time mechanisms.

Information Flow Security



Explicit Flow – Secret **Revealed** by assignment

Implicit Flow – Secret **Inferred** by observed different branch results

Secret Input *s*

Public Input *p*

Program (insecure)

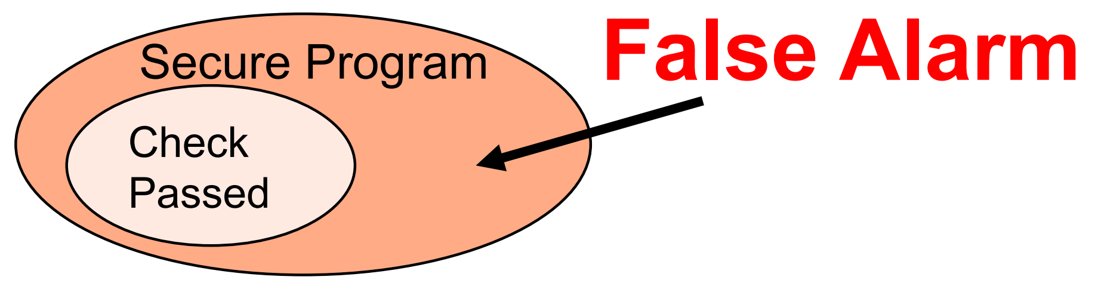
```
x := s;
p := x;

y := 0;
if (s > 0) y := 1;
p := y;
```

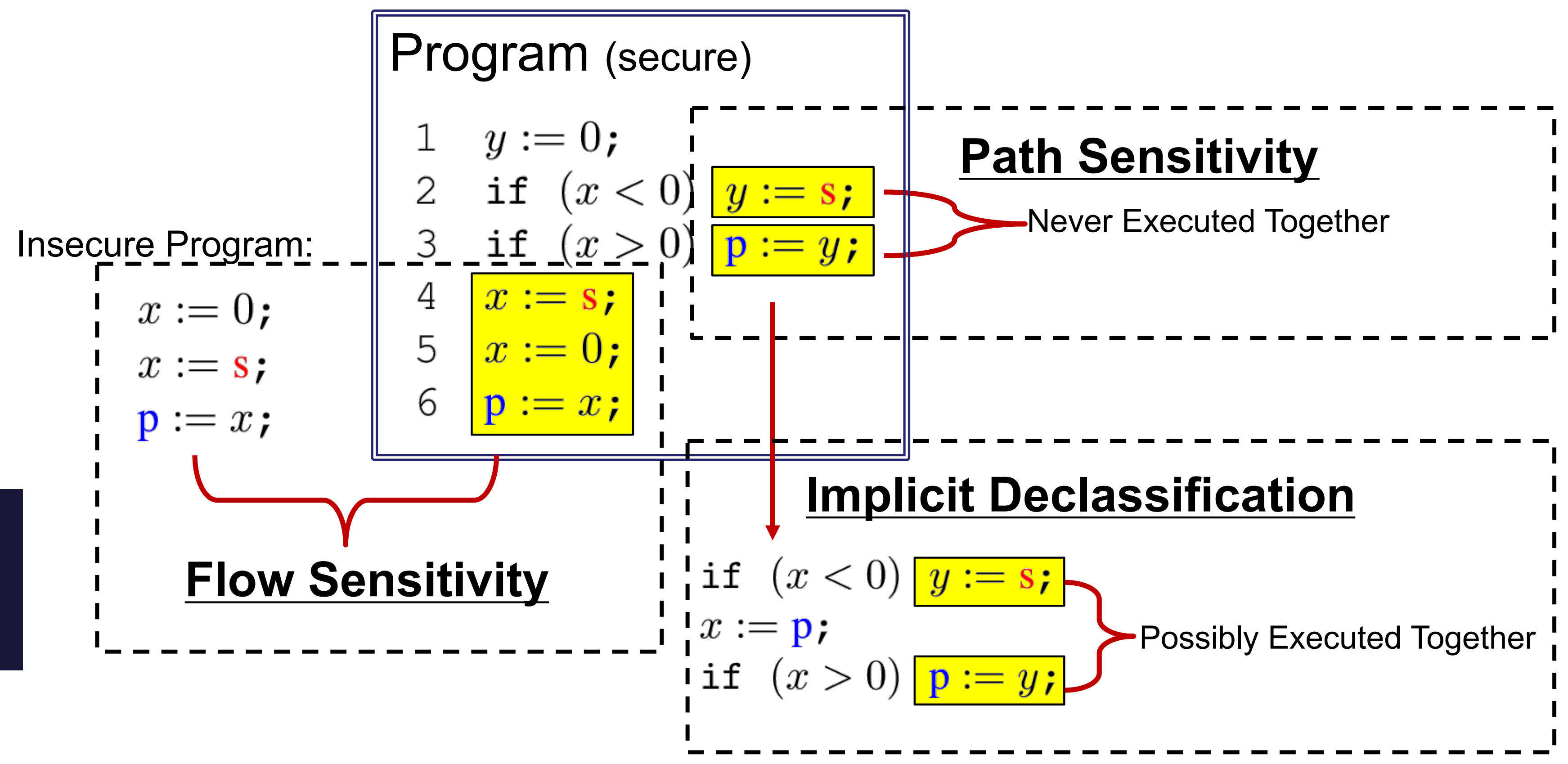
Observable Output *p*

Challenges

- ❑ Conservativeness
  - ❑ Checked → Secure



- ❑ Flow Sensitivity: Differentiate for the order of the execution
- ❑ Path Sensitivity: Consider the predicates at conditional branches
  - ❑ Implicit Declassification: mutable branch variables



Program (secure)

```
y := 0;
if (x < 0) y := s;
if (x > 0) p := y;
x := s;
x := 0;
p := x;
```

Insecure Program:

```
x := 0;
x := s;
p := x;
```

Flow Sensitivity

Path Sensitivity

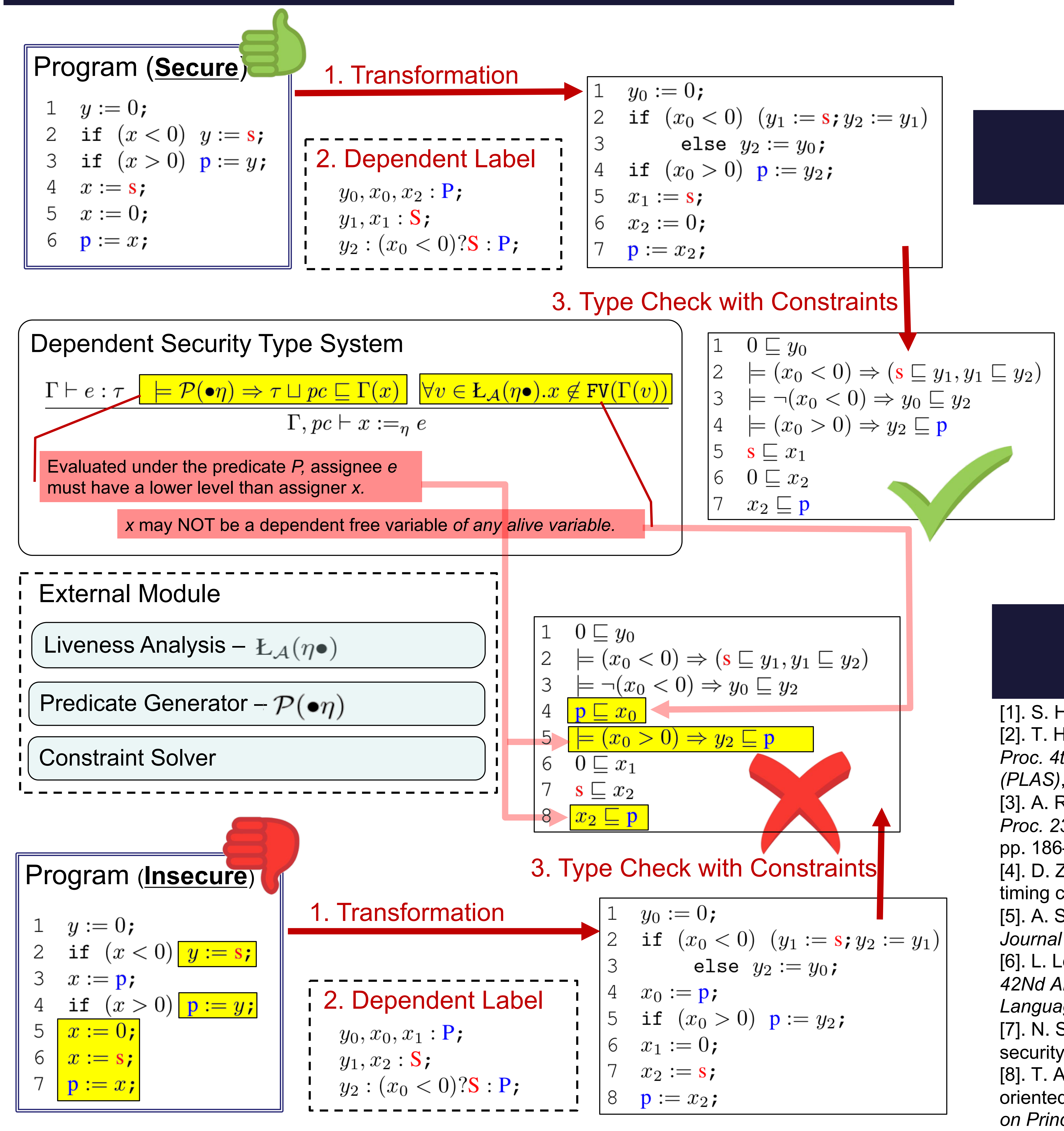
Never Executed Together

Implicit Declassification

```
if (x < 0) y := s;
x := p;
if (x > 0) p := y;
```

Possibly Executed Together

Flow- & Path- Sensitive Type System



Program (**Secure**)

```
y := 0;
if (x < 0) y := s;
if (x > 0) p := y;
x := s;
x := 0;
p := x;
```

1. Transformation

```
y0 := 0;
if (x0 < 0) (y1 := s; y2 := y1)
      else y2 := y0;
if (x0 > 0) p := y2;
x1 := s;
x2 := 0;
p := x2;
```

2. Dependent Label

$y_0, x_0, x_2 : \mathrm{P}$;
$y_1, x_1 : \mathrm{S}$;
$y_2 : (x_0 < 0)?\mathrm{S} : \mathrm{P}$;

3. Type Check with Constraints

$$
\begin{array}{l}
0 \sqsubseteq y_0 \\
\models (x_0 < 0) \Rightarrow (\mathrm{s} \sqsubseteq y_1, y_1 \sqsubseteq y_2) \\
\models \neg(x_0 < 0) \Rightarrow y_0 \sqsubseteq y_2 \\
\models (x_0 > 0) \Rightarrow y_2 \sqsubseteq \mathrm{p} \\
\mathrm{s} \sqsubseteq x_1 \\
0 \sqsubseteq x_2 \\
x_2 \sqsubseteq \mathrm{p}
\end{array}
$$

Dependent Security Type System

$$
\frac{\Gamma \vdash e : \tau \quad \models \mathcal{P}(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(x) \quad \forall v \in \text{Ł}_{\mathcal{A}}(\eta\bullet).x \notin \mathtt{FV}(\Gamma(v))}{\Gamma, pc \vdash x :=_{\eta} e}
$$

Evaluated under the predicate *P,* assignee *e* must have a lower level than assigner *x.*

x may NOT be a dependent free variable *of any alive variable.*

External Module

Liveness Analysis – $\text{Ł}_{\mathcal{A}}(\eta\bullet)$

Predicate Generator – $\mathcal{P}(\bullet\eta)$

Constraint Solver

$$
\begin{array}{l}
0 \sqsubseteq y_0 \\
\models (x_0 < 0) \Rightarrow (\mathrm{s} \sqsubseteq y_1, y_1 \sqsubseteq y_2) \\
\models \neg(x_0 < 0) \Rightarrow y_0 \sqsubseteq y_2 \\
\mathrm{p} \sqsubseteq x_0 \\
\models (x_0 > 0) \Rightarrow y_2 \sqsubseteq \mathrm{p} \\
0 \sqsubseteq x_1 \\
\mathrm{s} \sqsubseteq x_2 \\
x_2 \sqsubseteq \mathrm{p}
\end{array}
$$

3. Type Check with Constraints

Program (**Insecure**)

```
y := 0;
if (x < 0) y := s;
x := p;
if (x > 0) p := y;
x := 0;
x := s;
p := x;
```

1. Transformation

```
y0 := 0;
if (x0 < 0) (y1 := s; y2 := y1)
      else y2 := y0;
x0 := p;
if (x0 > 0) p := y2;
x1 := 0;
x2 := s;
p := x2;
```

2. Dependent Label

$y_0, x_0, x_1 : \mathrm{P}$;
$y_1, x_2 : \mathrm{S}$;
$y_2 : (x_0 < 0)?\mathrm{S} : \mathrm{P}$;

Conclusions

- ❑ Novel information flow analysis
  - ❑ Path-sensitive
  - ❑ Flow-sensitive
  - ❑ Purely Static method
  - ❑ Formalized soundness prove for
    - ❑ Terminate-insensitive Non-interference
- ❑ On-Going Work
  - ❑ Implementation
    - ❑ Java Polyglot
  - ❑ Type Inference – lower annotation burden

Related Publications

[1]. S. Hunt and D. Sands, “On flow-sensitive security types,” in *POPL 33*, 2006, pp. 79–90.

[2]. T. H. Austin and C. Flanagan, “Efficient purely-dynamic information flow analysis,” in *Proc. 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 113–124.

[3]. A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF)*, ser. CSF ’10, 2010, pp. 186–199.

[4]. D. Zhang, A. Askarov, and A. C. Myers, “Language-based control and mitigation of timing channels” in ACM SIGPLAN Notices, 2012,. 47(6), 99-110.

[5]. A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[6]. L. Lourenc ̧o and L. Caires, “Dependent information flow types,” in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 317–328.

[7]. N. Swamy, B. J. Corcoran, and M. Hicks, “Fable: A language for enforcing user-defined security policies,” in *Proc. IEEE Symp. on Security and Privacy*, 2008, pp. 369–383.

[8]. T. Amtoft, S. Bandhakavi, and A. Banerjee, “A logic for information flow in object-oriented programs,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 91–102.

This work was supported by NSF grant CCF-156641