Towards a Flow- and Path-Sensitive Information Flow Analysis

Peixuan Li, Danfeng Zhang
Pennsylvania State University
University Park, PA, USA
{pzl129,zhang}@cse.psu.edu
Background: Information Flow Analysis

- Security enforcement to prevent leakage of sensitive data
  - Non-interference: no dependence of public outputs on secret inputs
  - Lattice model: Information has labels that form a lattice

- Information Flow
  - Explicit flow – assignment
  - Implicit flow – branch

```
Program (secure)
1  y := 0;  x := -1;
2  if (x < 0) y := s;
3  if (x > 0) p := y;
4  x := s;
5  x := 0;
6  p := x;
```

S: secret  P: public
Problem of Interest

- **Conservative**: sound but not complete
  - **Sound**: Checked $\rightarrow$ Secure
  - **Complete**: Secure $\rightarrow$ Checked

- **Source of Conservativeness**
  - **Flow-Sensitivity** – to differentiate for the order of execution
  - **Path-Sensitivity** – to differentiate for the execution paths
Source of Conservativeness

- **Flow-Sensitivity** – to differentiate for the order of execution
- **Path-Sensitivity** – to differentiate for execution path

Program (secure)

1. \(y := 0;\) \(x := 1;\)
2. if \((x < 0)\) \(y := s;\)
3. if \((x > 0)\) \(p := y;\)
4. \(x := s;\)
5. \(x := 0;\)
6. \(p := x;\)

Insecure Program:

- if \((x < 0)\) \(y := s;\)
- if \((x < 0)\) \(p := y;\)

Flow-Sensitivity

Path-Sensitivity
Overview

Flow- & Path- Sensitive Analysis

Program

Program Transformation

Flow-sensitivity

Transformed

Dependent Type System

Path-sensitivity

Soundness Proof

Comparison with a flow-sensitive system
Overview – Sensitivity Knob

- Sensitivity Tuner
  - Flow-sensitivity – **Bracketed Assignments**
  - Path-sensitivity – **Dependent Type Labels**

- Less type annotation
- Simple analysis
- Program readability
- Accept more secure programs
Overview

Flow- & Path- Sensitive Analysis

Program

Program Transformation

Flow-sensitivity

Transformed

Dependent Type System

Path-sensitivity

Proof of soundness on non-interference

Comparison with a flow-sensitive system
Program Transformation

- Goal – To gain flow-sensitivity

Program (secure)

1. \( y := 0; \ x := -1; \)
2. \( \text{if } (x < 0) \ y := s; \)
3. \( \text{if } (x > 0) \ p := y; \)
4. \( x := s; \)
5. \( x := 0; \)
6. \( p := x; \)

Flow-Sensitive Type System:

\[
\Gamma \vdash_{\text{HS}} e : \tau
\]

\[
p_{c} \vdash_{\text{HS}} \Gamma\{x := e\} \Gamma\{x \mapsto p_{c} \sqcup \tau\}
\]

- Update & Record types at each program point
- Complicates the design of the type system
Program Transformation

- Goal – To gain **flow-sensitivity**

Program (secure)

1. \( y := 0; \ x := -1; \)
2. \( \text{if } (x < 0) \ y := s; \)
3. \( \text{if } (x > 0) \ p := y; \)
4. \( x := s; \)
5. \( x := 0; \)
6. \( p := x; \)

Secure:

- \( x := s; \)
- \( x_1 := 0; \)
- \( p := x_1; \)

Type checked by **flow-insensitive** type systems

- Renaming gains flow-sensitivity
Program Transformation

- **Goal** – To gain **flow-sensitivity**

Program (secure)

```
1  y := 0; x := -1;
2  if (x < 0) y := s;
3  if (x > 0) p := y;
4  x := s;
5  x := 0;
6  p := x;
```

**Bracketed Assignment**

Active Copy

Active Set

\[
\text{TRSF-ASSIGN-CREATE} \\
\langle e, \mathcal{A} \rangle \Rightarrow e \\
\langle [x := e], \mathcal{A} \rangle \Rightarrow x_i := e, \mathcal{A}\{x \mapsto x_i\}
\]

\( \hat{e} \) is a fresh index for \( x \)
Program Transformation

- **Goal** – To gain **flow-sensitivity**

```
Program (secure)
1   y := 0;  x := −1;
2   if (x < 0) y := s;
3   if (x > 0) p := y;
4   x := s;
5   x := 0;
6   p := x;
```

**Bracketed Assignment**

```
if (x < 0) [y := s];                   y → y
                      ▼
Rename
if (x < 0) y1 := s; (else skip;)
                    ▼
Merge
if (x < 0) (y1 := s;
                ▼
y2 := y1)
else (y2 := y;
           ▼
y2 := y)
```

**TRSIF**

\[
\langle e, A \rangle \Rightarrow e \quad \langle c_1, A \rangle \Rightarrow \langle c_1, A_1 \rangle \quad \langle c_2, A \rangle \Rightarrow \langle c_2, A_2 \rangle \quad \Phi(A_1, A_2) \Rightarrow A_3
\]

\[
\langle \text{if (e) then } c_1 \text{ else } c_2, A \rangle \Rightarrow \langle \text{if (e) then } (c_1; A_3 := A_1) \text{ else } (c_2; A_3 := A_2), A_3 \rangle
\]
Program Transformation

- Difference from SSA
  - Tunable bracketed assignments – not all assignment need renaming
  - No phi-function – simplify the analysis and soundness proof
    - Details are discussed in the paper
Program Transformation

- Transformation Correctness
  - memory projection on active set
    \[ m^A(x) = m(A(x)) \]

\[ \forall c, c, m, m', m', A, A'. \]
\[ \langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \langle c, m \rangle \rightarrow^{*} \langle \text{skip}, m' \rangle \land \langle c, m \rangle \rightarrow^{*} \langle \text{skip}, m' \rangle \land m = m^A \]
\[ \Rightarrow m' = (m')^{A'}. \]
Review

Flow- & Path- Sensitive Analysis

Program Transformation
Flow-sensitivity

Dependent Type System
Path-sensitivity

Transformed

Soundness Proof

Comparison with a flow-sensitive system
Dependent Type System

- Goal – To gain **Path-Sensitivity**
  - Dependent Security Label
    
    \[
    y : (x < 0?S : P);
    \]
  - Predicates Generator
    
    Line 2: \[\vdash (x < 0) \Rightarrow (S \sqsubseteq (x < 0?S : P))\]
    Line 3: \[\vdash (x > 0) \Rightarrow ((x < 0?S : P) \subseteq P)\]

Program (secure)

```
1 y := 0; x := -1;
2 if (x < 0) y := s;
3 if (x > 0) p := y;
4 x := s;
5 x := 0;
6 p := x;
```
Dependent Type System

- Challenge

Program (secure)

1. \( y := 0; \ x := -1; \)
2. if \((x < 0)\) \(y := s;\)
3. if \((x > 0)\) \(p := y;\)
4. \(x := s;\)
5. \(x := 0;\)
6. \(p := x;\)

\(y : (x < 0?S : P);\)

Line 2: \(\models (x < 0) \Rightarrow (S \sqsubseteq (x < 0?S : P))\)

Line 4: \(\models (x > 0) \Rightarrow ((x < 0?S : P) \sqsubseteq P)\)

Insecure Program:

1. \(y := 0;\ x := -1;\)
2. if \((x < 0)\) \(y := s;\)
3. \(x := 1;\)
4. if \((x > 0)\) \(p := y;\)

\(x = -1\)

\(y: S\)

\(x := 1;\)

\(y: P\)

\(x = 1\)

Implicit Declassification
Dependent Type System

Solution to Implicit Declassification

- A. Reject if program contains *any* mutable dependency
- B. Dynamically *erase* variable content for mutable dependency
  - Runtime overhead
  - Changing program behavior
- C. Reject if program contains mutable dependency on *live* variable

```plaintext
1  y := 0;  x := -1;
2  if (x < 0) y := s;
3  if (x > 0) p := y;
4  x := 1;
5  // y never used later
```
Dependent Type System

- Soundness

**Theorem 3 (Soundness of Transformed Program):**

\[
\forall c, m_1, m_2, m_3, m_4, \ell, \Gamma, A, A' .
\]

\[
\langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \Gamma \vdash c \land m_1^A \approx^\ell_{\Gamma, A} m_2^A
\]

\[
\langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m_3 \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m_4 \rangle
\]

\[
\implies m_3^{A'} \approx^\ell_{\Gamma, A'} m_4^{A'}
\]
Proof of non-interference

- Soundness of non-interference

---

Initial State

Execution 1: \(<c, m_2>\) → \(<\text{skip}, m_2'\>\)

Execution 2: \(<c, m_1>\) → \(<\text{skip}, m_1'\>\)

Final State

Soundness (Original)

Soundness (Transformed)
Review

Flow- & Path- Sensitive Analysis

Program Transformation
Flow-sensitivity
Transformed

Dependent Type System
Path-sensitivity

Soundness Proof

Comparison with a flow-sensitive system
Comparison

- Comparison with a classic flow-sensitive type system

- Flow-Sensitivity
  - HS System
  - Our system

- Path-Sensitivity
  - HS System
  - Our system

⚠️ Strictly more precise than HS system
Comparison

- Strictly more precise than HS system
- Subsumes the HS system
- Accepts more secure program

∀c, Γ, A, Γ', Γ''.

\[
\langle [c], A \rangle \implies \langle c, A' \rangle \land pc \vdash_{HS} \Gamma \{c\}\Gamma' \Rightarrow \Gamma'' \vdash_{c}
\]
Conclusion

Flow- & Path Sensitive Analysis

Program Transformation
- Bracketed Assignment
- Correctness

Dependent Type System
- Dependent Labels
- Predicates Generator
- Implicit Declassification
- Liveness Analysis

Program

Transformed

Soundness Proof

Comparison with classic flow-sensitive system
Thank you!