A Type-based Approach to Program Security

Dennis Volpano & Geoffrey Smith, TAPSOFT 1997

Professor William L. Harrison
CS8440 Fall 2015

September 28, 2016

Overview

- Retrofits noninterference to programming languages

Overview

- Retrofits noninterference to programming languages
- Ex: Consider the following procedure:

  $proc\ P(inout\ x\ :\ low,\ inout\ y\ :\ high)$

Overview

- Retrofits noninterference to programming languages
- Ex: Consider the following procedure:

  $proc\ P(inout\ x\ :\ low,\ inout\ y\ :\ high)$

- Suppose $P(u : low,\ v : high)$ and $P(u : low, w : high)$ terminate with values u, v and w.

Overview

- Retrofits noninterference to programming languages
- Ex: Consider the following procedure:

  $proc\ P(inout\ x\ :\ low,\ inout\ y\ :\ high)$

- Suppose $P(u : low,\ v : high)$ and $P(u : low, w : high)$ terminate with values u, v and w.
- The final values for v and w may differ, **but**, if P is noninterfering, then the final values for u must be identical.

Overview

- Retrofits noninterference to programming languages
- Ex: Consider the following procedure:

  $proc\ P(inout\ x\ :\ low,\ inout\ y\ :\ high)$

- Suppose $P(u : low,\ v : high)$ and $P(u : low, w : high)$ terminate with values u, v and w.
- The final values for v and w may differ, **but**, if P is noninterfering, then the final values for u must be identical.
- Smith & Volpano's type system enforces noninterference—P is well-typed means it's noninterfering.

Explicit Flows

- For l : *low var* and h : *high var*, an explicit flow $l := h$ must be rejected.

Explicit Flows

- For l : *low var* and h : *high var*, an explicit flow $l := h$ must be rejected.
- In Denning & Denning, this was performed by the certification mechanism.

Explicit Flows

- For l : *low var* and h : *high var*, an explicit flow $l := h$ must be rejected.
- In Denning & Denning, this was performed by the certification mechanism.
- In Smith and Volpano, this is accomplished via a typing rule:

$$\frac{\gamma \vdash e : \tau\ acc \qquad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau\ cmd}$$

Explicit Flows

- For l : *low var* and h : *high var*, an explicit flow $l := h$ must be rejected.
- In Denning & Denning, this was performed by the certification mechanism.
- In Smith and Volpano, this is accomplished via a typing rule:

$$\frac{\gamma \vdash e : \tau\ acc \qquad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau\ cmd}$$

- This rule insists that h and l be typed on the same level. How?

Running Example

```
while h > 0 do
    l := l + 1;
    h := h - 1
od
```

Running Example

```
while h > 0 do
    l := l + 1;
    h := h - 1
od
```

- Q: What kind of flows exist in this program?

Running Example

```
while h > 0 do
    l := l + 1;
    h := h - 1
od
```

Running Example

```
while h > 0 do
    l := l + 1;
    h := h - 1
od
```

- The typing rule for *while* insists that the test and body of the loop be typed at the same level:

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau\ cmd}{\gamma \vdash \mathit{while}\ e\ \mathit{do}\ c : \tau\ cmd}$$

Programming Language Syntax

$$
\begin{array}{lll}
(\mathit{Phrase}) & p ::= & e \mid c \\
(\mathit{Expr}) & e ::= & x \mid n \mid l \mid e + e' \mid e - e' \mid e = e' \mid \\
 & & e < e' \mid \mathbf{proc}\ (\mathbf{in}\ x_1, \mathbf{inout}\ x_2, \mathbf{out}\ x_3)\ c \\
(\mathit{Comm}) & c ::= & e := e' \mid c; c' \mid e(e_1, e_2, e_3) \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid \\
 & & \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \mid \mathbf{letvar}\ x := e\ \mathbf{in}\ c \mid \\
 & & \mathbf{letproc}\ x(\mathbf{in}\ x_1, \mathbf{inout}\ x_2, \mathbf{out}\ x_3)\ c\ \mathbf{in}\ c'
\end{array}
$$

Type Syntax

$$\tau ::= s$$
$$\pi ::= \tau \mid \tau\ proc(\tau_1, \tau_2\ var, \tau_3\ acc) \mid \tau\ cmd$$
$$\rho ::= \pi \mid \tau\ var \mid \tau\ acc$$

N.b., s is a *security level*. It is assumed that all security levels form a lattice ordered by $\leq$.

Type Judgments

$$\lambda; \gamma \vdash e : \rho$$

Type Judgments

$$\lambda; \gamma \vdash e : \rho$$

- e is a program phrase (i.e., expression, command, etc.).

Type Judgments

$$\lambda; \gamma \vdash e : \rho$$

- e is a program phrase (i.e., expression, command, etc.).
- ρ is a type.

Type Judgments

$$\lambda; \gamma \vdash e : \rho$$

- e is a program phrase (i.e., expression, command, etc.).
- ρ is a type.
- γ is the *identifier typing environment*.
  N.b., "$\gamma(i) = \rho$" means i has type ρ in γ.

Type Judgments

$$\lambda; \gamma \vdash e : \rho$$

- e is a program phrase (i.e., expression, command, etc.).
- ρ is a type.
- γ is the *identifier typing environment*.
  N.b., "$\gamma(i) = \rho$" means i has type ρ in γ.
- λ is the *location typing environment*.
  - Locations are used for input-output in the semantics.
  - Locations are, in effect, global.
  - λ largely irrelevant to the type system; only occurs in one rule (VARLOC).

$$(\textsc{ident}) \quad \lambda;\gamma \vdash x : \tau \qquad \gamma(x) = \tau$$

$$(\textsc{var}) \quad \lambda;\gamma \vdash x : \tau\ var \qquad \gamma(x) = \tau\ var$$

$$(\textsc{acceptor}) \quad \lambda;\gamma \vdash x : \tau\ acc \qquad \gamma(x) = \tau\ acc$$

$$(\textsc{varloc}) \quad \lambda;\gamma \vdash l : \tau\ var \qquad \lambda(l) = \tau$$

$$(\textsc{int}) \quad \lambda;\gamma \vdash n : \tau$$

$$(\textsc{r-val}) \qquad \frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau}$$

$$(\textsc{l-val}) \qquad \frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau\ acc}$$

$$(\textsc{sum}) \qquad \frac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e + e' : \tau}$$

$$(\textsc{compose}) \qquad \frac{\lambda;\gamma \vdash c : \tau\ cmd,\ \ \lambda;\gamma \vdash c' : \tau\ cmd}{\lambda;\gamma \vdash c\,;\, c' : \tau\ cmd}$$

$$(\textsc{assign}) \qquad \frac{\lambda;\gamma \vdash e : \tau\ acc,\ \ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e := e' : \tau\ cmd}$$

$$(\textsc{if}) \qquad \frac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash c : \tau\ cmd,\ \ \lambda;\gamma \vdash c' : \tau\ cmd,}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ cmd}$$

$$(\textsc{while}) \qquad \frac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash c : \tau\ cmd}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}$$

Next time

- Natural semantics for language
- Noninterference as Type soundness argument:
  - Argue that well-typed programs do not interfere.