A Type-based Approach to Program Security

Dennis Volpano & Geoffrey Smith, TAPSOFT 1997

Professor William L. Harrison
CS8440 Fall 2016

September 30, 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:

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

- 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?

Example

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

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

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

Type Syntax

$$\begin{aligned}
\tau &::= s \\
\pi &::= \tau \mid \tau\ proc(\tau_1, \tau_2\ var, \tau_3\ acc) \mid \tau\ cmd \\
\rho &::= \pi \mid \tau\ var \mid \tau\ acc
\end{aligned}$$

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).

(IDENT)	$\lambda; \gamma \vdash x : \tau$	$\gamma(x) = \tau$
(VAR)	$\lambda; \gamma \vdash x : \tau \ var$	$\gamma(x) = \tau \ var$
(ACCEPTOR)	$\lambda; \gamma \vdash x : \tau \ acc$	$\gamma(x) = \tau \ acc$
(VARLOC)	$\lambda; \gamma \vdash l : \tau \ var$	$\lambda(l) = \tau$
(INT)	$\lambda; \gamma \vdash n : \tau$	

$$(\text{R-VAL}) \qquad \frac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau}$$

$$(\text{L-VAL}) \qquad \frac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau\ acc}$$

$$(\text{SUM}) \qquad \frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e + e' : \tau}$$

$$(\text{COMPOSE}) \qquad \frac{\lambda; \gamma \vdash c : \tau\ cmd, \quad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash c\, ; c' : \tau\ cmd}$$

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

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

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

Next time

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

GOALS

Extends Denning and Denning's Security Certification work – the paper we read.

- Uses a type system instead of attribute calculation
- Provides closer link between semantics of the language and secure flows.
  - static analysis was seen as too ad hoc
  - their approach amenable to formal verification

Approach was novel: "type soundness"

- Give a transition semantics
- Give a security type system with type inference
- Type soundness: any well-typed program never commits an insecure flow in the semantics.

$$
\begin{array}{rcll}
\text{Phrases} & p & ::= & e \mid c \\[1em]
\text{Expressions} & e & ::= & x \mid l \mid n \mid e + e' \mid e - e' \mid \\
 & & & e = e' \mid e < e' \\[1em]
\text{Commands} & c & ::= & e := e' \mid c; c' \mid \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c' \mid \\
 & & & \texttt{while}\, e\, \texttt{do}\, c \mid \texttt{letvar}\, x := e\, \texttt{in}\, c \\[1em]
\text{Security classes} & s & \in & SC \quad \text{(partially ordered by } \leq\text{)} \\[1em]
\text{Types} & \tau & ::= & s \\[1em]
\text{Phrase types} & \rho & ::= & \tau \mid \tau\ var \mid \tau\ cmd
\end{array}
$$

Type Assertions

$$\lambda\,;\gamma \vdash p : \rho$$

Heap: map l to ρ_l Stack: map x to ρ_x

- τ *cmd*: if $\lambda\,;\gamma \vdash c : \tau$ *cmd*, then for any l assigned to in c, $\tau \leq \lambda(l)$. (Lemma 6.4)
- τ *var*: a variable that can store values with type τ.

security level

EXAMPLE: TYPING COMMANDS

Say $l : low\ var$ and $h : high\ var$. To be more formal:

$$\gamma \vdash l : low\ var$$

$$\gamma \vdash h : high\ var$$

Clearly, the explicit flowing command, $l := h$, must be rejected

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

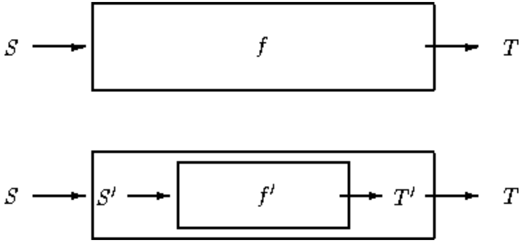
Allow type coercions "upwards" in security: e.g., consider $l : high\ var$

Subtypes

- A type S is a subtype of a type T (written $S \subseteq T$) if an expression of type S can be used in any context that expects an element of type T.
  - Another way of putting this is that any expression of type S can masquerade as an expression of type T.

Subtypes

- A type S is a subtype of a type T (written $S \subseteq T$) if an expression of type S can be used in any context that expects an element of type T.
  - Another way of putting this is that any expression of type S can masquerade as an expression of type T.
- Function subtyping is a bit non-intuitive at first:



Subtyping Rules

(BASE) $$\frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'}$$

(REFLEX) $$\vdash \rho \subseteq \rho$$

(TRANS) $$\frac{\vdash \rho \subseteq \rho', \ \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

(ACC$^-$) $$\frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \ acc \subseteq \tau \ acc}$$

(CMD$^-$) $$\frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \ cmd \subseteq \tau \ cmd}$$

(PROC) $$\frac{\vdash \tau_1' \subseteq \tau_1, \ \vdash \tau_3 \subseteq \tau_3', \ \vdash \tau' \subseteq \tau}{\vdash \tau \ proc(\tau_1, \ \tau_2 \ var, \ \tau_3 \ acc) \subseteq \tau' \ proc(\tau_1', \ \tau_2 \ var, \ \tau_3' \ acc)}$$

(SUBTYPE) $$\frac{\lambda; \gamma \vdash p : \rho, \ \vdash \rho \subseteq \rho'}{\lambda; \gamma \vdash p : \rho'}$$

Typing Rules with Subtyping

$$(\textsc{ident}') \quad \frac{\gamma(x) = \tau, \;\; \tau \leq \tau'}{\lambda; \gamma \vdash x : \tau'}$$

$$(\textsc{r-val}') \quad \frac{\lambda; \gamma \vdash e : \tau \; var, \;\; \tau \leq \tau'}{\lambda; \gamma \vdash e : \tau'}$$

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

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

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

Another Example

```
proc (in x, out y)
    let var a := x in
    let var b := y in
        while a > 0 do
            b := b + 1;
            a := a - 1;
        od
        y := b
```

Another Example

```
proc (in x, out y)
    let var a := x in
    let var b := y in
          while a > 0 do
              b := b + 1;
              a := a - 1;
          od
          y := b
```

- Has type: β **proc**(α, β **acc**) where α and β are the security levels for a and b, resp., and $\alpha \leq \beta$.

NATURAL SEMANTICS

A program is evaluated w.r.t. a **memory** (finite map from locations to integers)

expression

value

$$\mu \vdash e \Rightarrow n$$

input
memory

$$\mu \vdash c \Rightarrow \mu'$$

output memory

command

$\Rightarrow$ is the “evaluates to” relation; “**|-**” is overloaded

Evaluation Rules

(VAL) $\mu \vdash n \Rightarrow n$

(CONTENTS) $\mu \vdash l \Rightarrow \mu(l) \qquad l \in dom(\mu)$

(ADD) $$\frac{\mu \vdash e \Rightarrow n, \ \ \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}$$

(SEQUENCE) $$\frac{\mu \vdash c \Rightarrow \mu', \ \ \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c\,;\, c' \Rightarrow \mu''}$$

(BRANCH) $$\frac{\mu \vdash e \Rightarrow 1, \ \ \mu \vdash c \Rightarrow \mu'}{\mu \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0, \ \ \mu \vdash c' \Rightarrow \mu'}{\mu \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' \Rightarrow \mu'}$$

(CALL) $$\frac{\mu \vdash e \Rightarrow n, \ \ \mu \vdash [n, l, l'/x_1, x_2, x_3]c \Rightarrow \mu'}{\mu \vdash (\textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c)(e, l, l') \Rightarrow \mu'}$$

Evaluation Rules (cont'd)

(UPDATE)
$$\frac{\mu \vdash e \Rightarrow n, \;\; l \in dom(\mu)}{\mu \vdash l := e \Rightarrow \mu'[l := n]}$$

(BINDVAR)
$$\frac{\mu \vdash e \Rightarrow n, \;\; l \text{ is the first location not in } dom(\mu), \quad \mu[l := n] \vdash [l/x]c \Rightarrow \mu'}{\mu \vdash \mathbf{letvar}\ x := e\ \mathbf{in}\ c \Rightarrow \mu' - l}$$

(LOOP)
$$\frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu}$$

$$\frac{\mu \vdash e \Rightarrow 1, \;\; \mu \vdash c \Rightarrow \mu', \;\; \mu' \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu''}{\mu \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu''}$$

(BINDPROC)
$$\frac{\mu \vdash [\mathbf{proc}\ (\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c/x]c' \Rightarrow \mu'}{\mu \vdash \mathbf{letproc}\ x(\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c\ \mathbf{in}\ c' \Rightarrow \mu'}$$

Noninterference Theorem

Theorem 6.8 (*Type Soundness*) Suppose

(a) $\lambda \vdash c : \rho$ *c is well-typed*

(b) $\mu \vdash c \Rightarrow \mu'$ *execution one*

(c) $\upsilon \vdash c \Rightarrow \upsilon'$ *execution two*

(d) $dom(\mu) = dom(\upsilon) = dom(\lambda)$

(e) $\upsilon(l) = \mu(l)$ for all l such that $\lambda(l) \leq \tau$ *the same low inputs*

Then $\upsilon'(l) = \mu'(l)$ for all l such that $\lambda(l) \leq \tau$. *the same low outputs*