A Sound Type System for Secure Flow Analysis

Dennis Volpano, Geoffrey Smith, Cynthia Irvine

Presenter: Lantian Zheng
CS 711

September 29, 2003

Soundness of Dening's Program Certification Mechanism

- Define the soundness property: $S(P)$.
  - Noninterference
- Prove: $\texttt{certified}(P) \Rightarrow S(P)$.

Program Certification as Type Checking

$v := e$ `is certified if` $\underline{e} \rightarrow \underline{v}$.

$v := e$ `is welltyped if` $type(e) \leq type(v)$.

Program Certification as Type Checking

$v := e$ `is certified if` $\underline{e} \rightarrow \underline{v}$.

$v := e$ `is welltyped if` $type(e) \leq type(v)$.

- Security levels $\approx$ Types
- Lattice order on security levels $\approx$ Subtyping
- Program certification $\approx$ Type checking

Program Certification as Type Checking

$v := e$ `is certified if` $\underline{e} \rightarrow \underline{v}$.

$v := e$ `is welltyped if` $type(e) \leq type(v)$.

- Security levels $\approx$ Types
- Lattice order on security levels $\approx$ Subtyping
- Program certification $\approx$ Type checking

$$\boxed{\texttt{welltyped}(P) \Rightarrow \texttt{noninterference}(P)}$$

Background

- Greece and Rome
  - Program certification (76, Denings)
  - Noninterference (82, Goguen & Meseguer)
- Middle ages
  - The orange book (85)
  - More on security models
    - Nondeducibility (86 Sutherland)
    - Composibility of noninterference (87-88 McCullough)
  - Soundness of dynamic information-flow control
    - Proving noninterference using traces (92 McLean)

– Connect static and dynamic information-flow mechanisms
  * The operational semantics with labels is consistent with the abstract semantics on labels. (92 Mizuno&Schmidt, 95 Ørbæk)

- Renaissance
  – Soundness of compile-time analysis w.r.t. noninterference (94 Ban$\hat{a}$tre&M$\acute{e}$tayer&Beaulieu)

$$\text{“ } \forall S, P.\ \texttt{if}\ \vdash_1 \{Init\}S\{P\}\ \texttt{then}\ C(P,S) \text{ ”}$$

The Core Language

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

Typing Assertion

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

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.

Noninterference Theorem

Theorem 6.8 (*Type Soundness*) Suppose

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

Noninterference Theorem

Theorem 6.8 ($Type\ Soundness$) Suppose

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

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

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*

Noninterference Theorem

Theorem 6.8 ($\mathit{Type\ Soundness}$) 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) $\mathit{dom}(\mu) = \mathit{dom}(\upsilon) = \mathit{dom}(\lambda)$

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

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) $\mathit{dom}(\mu) = \mathit{dom}(\upsilon) = \mathit{dom}(\lambda)$

(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*

Typing Arithmetic Operations

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

- Example:

$$\frac{x\!:\!L, y\!:\!H \vdash x : H \qquad x\!:\!L, y\!:\!H \vdash y : H}{x\!:\!L, y\!:\!H \vdash x + y : H}$$

- Subsumption rule:

$$\frac{\lambda\,;\gamma \vdash e : \tau \qquad \vdash \tau \subseteq \tau'}{\lambda\,;\gamma \vdash e : \tau'}$$

- Lemma 6.3: if $\lambda \vdash e : \tau$, then for every l in e, $\lambda(l) \leq \tau$.

Subtyping Rules

$$\frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \qquad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \textit{ cmd} \subseteq \tau \textit{ cmd}}$$

$$\vdash \rho \subseteq \rho \qquad \frac{\vdash \rho \subseteq \rho' \qquad \vdash \rho' \subseteq \rho''}{\vdash \rho' \subseteq \rho''}$$

Corollary: τ *var* is invariant with respect to τ.

$$\frac{\tau = \tau'}{\vdash \tau \textit{ var} \subseteq \tau' \textit{ var}}$$

Typing Assignments

$$\frac{\lambda\,;\gamma \vdash e : \tau\ \textit{var} \qquad \lambda\,;\gamma \vdash e' : \tau}{\lambda\,;\gamma \vdash e := e' : \tau\ \textit{cmd}}$$

- The result of e' can be stored in e.
- The assignment command updates a location with type τ.
- Lemma 6.4: If $\lambda\,;\gamma \vdash c : \tau\,\textit{cmd}$, then for every l assigned to in c, $v(l) \leq \tau$.

Typing Compositions

$$\frac{\lambda\,;\gamma \vdash c : \tau \textit{ cmd} \qquad \lambda\,;\gamma \vdash c' : \tau \textit{ cmd}}{\lambda\,;\gamma \vdash c; c' : \tau \textit{ cmd}}$$

- The subsumption rule masks the combination of two command types:

$$\frac{\lambda\,;\gamma \vdash c : \tau \textit{ cmd} \quad \lambda\,;\gamma \vdash c' : \tau' \textit{ cmd}}{\lambda\,;\gamma \vdash c; c' : \tau \sqcap \tau' \textit{ cmd}}$$

Typing IF and WHILE

$$\frac{\lambda\,;\gamma \vdash e : \tau \quad \lambda\,;\gamma \vdash c : \tau\ \textit{cmd} \quad \lambda\,;\gamma \vdash c' : \tau}{\lambda\,;\gamma \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c' : \tau\ \textit{cmd}}$$

$$\frac{\lambda\,;\gamma \vdash e : \tau \quad \lambda\,;\gamma \vdash c : \tau\ \textit{cmd}}{\lambda\,;\gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \tau\ \textit{cmd}}$$

- To prevent implicit flows: c and c' can any update location l that satisfies $type(e) \leq \lambda(l)$.

Typing LETVAR

$$\frac{\lambda\,;\gamma \vdash e : \tau \quad \lambda\,;\gamma[x\!:\!\tau\ \textit{var}] \vdash c : \tau'\ \textit{cmd}}{\lambda\,;\gamma \vdash \texttt{letvar}\, x := e\, \texttt{in}\, c : \tau'\ \textit{cmd}}$$

- The local variable x is not observable outside the command.
- Similar to the function application: $(\lambda x.c)e$.

Proving the Noninterference Theorem

- By induction on one of the two evaluations $\mu \vdash c \Rightarrow \mu'$.
- The core language is pleasantly simple.
  - No first-class functions: the two executions run the same code.
- Syntax-directed typing rules

After 1996

SLam	Heintze&Riecke (98)	Induction on typing derivation, denotational semantics
The secure CPS calculus	Zdancewic&Myers (01)	Induction on evaluation, small-step semantics
MLIF	Pottier&Simonet (02)	Induction on evalution, small-step semantics for pairing two executions
Java-light	Banerjee&Naumann (02)	Induction on typing derivation, dentational semantics

Discussion

- "How should secrets be introduced?"
  - *Safety Versus Secrecy*, Dennis Volpano, 99
    "Instead, we associate secrecy with the origin of a value which in our case will be the free variables of a program. ... This origin-view of secrecy differs from the view held by others working with assorted lambda calculi and type system for secrecy [1,3]. There secrecy is associated with values like boolean constants. It does not seem sensible to attribute any level of security to such constants. After all, what exacly is high-security boolean?"

- Is information-flow policy EM-enforceable?
  - Suppose the operational semantics manipulates security labels and does run-time label checking.