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: $\text{certified}(P) \Rightarrow S(P)$. 
Program Certification as Type Checking

\( v \ := e \text{ is certified if } e \rightarrow v. \)

\( v \ := e \text{ is welltyped if } \text{type}(e) \leq \text{type}(v). \)
Program Certification as Type Checking

\[ v := e \text{ is certified if } e \rightarrow v. \]

\[ v := e \text{ is welltyped if } \text{type}(e) \leq \text{type}(v). \]

- Security levels \(\simeq\) Types
- Lattice order on security levels \(\simeq\) Subtyping
- Program certification \(\simeq\) Type checking
Program Certification as Type Checking

\[ v := e \text{ is certified if } e \rightarrow v. \]
\[ v := e \text{ is welltyped if } \text{type}(e) \leq \text{type}(v). \]

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

\[ \text{welltyped}(P) \Rightarrow \text{noninterference}(P) \]
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 Banâtre&Métayer&Beaulieu)

\[
\forall S, P. \text{ if } \vdash_1 \{\text{Init}\} S \{P\} \text{ then } C(P, S)
\]
The Core Language

Phrases \[ p ::= e | c \]

Expressions \[ e ::= x | l | n | e + e' | e - e' | e = e' | e < e' \]

Commands \[ c ::= e := e' | ; c' | if e then c else c' | while e do c | letvar x := e in c \]

Security classes \[ s ∈ SC \] (partially ordered by \( ≤ \))

Types \[ τ ::= s \]

Phrase types \[ ρ ::= τ | τ var | τ cmd \]
Typing Assertion

\[
\lambda ; \gamma \vdash p : \rho
\]

Heap: map \( l \) to \( \rho_l \)  
Stack: map \( x \) to \( \rho_x \)

• \( \tau \text{ cmd} \): if \( \lambda ; \gamma \vdash c : \tau \text{ cmd} \), then for any \( l \) assigned to in \( c \), \( \tau \leq \lambda(l) \). (Lemma 6.4)

• \( \tau \text{ var} \): a variable that can store values with type \( \tau \).
Noninterference Theorem

Theorem 6.8 (Type Soundness) Suppose

(a) $\lambda \vdash c : \rho$

$c$ is well-typed
Noninterference Theorem

Theorem 6.8 (Type Soundness) Suppose

(a) $\lambda \vdash c : \rho$

(b) $\mu \vdash c \Rightarrow \mu'$

$c$ is well-typed

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) \( \nu \vdash c \Rightarrow \nu' \)  
execution two
Noninterference Theorem

Theorem 6.8 (Type Soundness) Suppose

(a) $\lambda \vdash c : \rho$  \hspace{2cm} $c$ is well-typed
(b) $\mu \vdash c \Rightarrow \mu'$  \hspace{2cm} execution one
(c) $\nu \vdash c \Rightarrow \nu'$  \hspace{2cm} execution two
(d) $\text{dom}(\mu) = \text{dom}(\nu) = \text{dom}(\lambda)$
(e) $\nu(l) = \mu(l)$ for all $l$ such that $\lambda(l) \leq \tau$ \hspace{2cm} the same low inputs

CS 711: Language-Based Security and Information Flow
Noninterference Theorem

Theorem 6.8 (Type Soundness) Suppose

(a) \( \lambda \vdash c : \rho \) \hspace{1cm} \text{c is well-typed}

(b) \( \mu \vdash c \Rightarrow \mu' \) \hspace{1cm} \text{execution one}

(c) \( \nu \vdash c \Rightarrow \nu' \) \hspace{1cm} \text{execution two}

(d) \( \text{dom}(\mu) = \text{dom}(\nu) = \text{dom}(\lambda) \)

(e) \( \nu(l) = \mu(l) \) for all \( l \) such that \( \lambda(l) \leq \tau \) \hspace{1cm} \text{the same low inputs}

Then \( \nu'(l) = \mu'(l) \) for all \( l \) such that \( \lambda(l) \leq \tau \). \hspace{1cm} \text{the same low outputs}
Typing Arithmetic Operations

\[ \frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e + e' : \tau} \]

- Example:

\[
\frac{
  x : L, y : H \vdash x : H \quad x : L, y : H \vdash y : H
}{
  x : L, y : H \vdash x + y : H
} \]

- Subsumption rule:

\[
\frac{
  \lambda; \gamma \vdash e : \tau \quad \vdash \tau \subseteq \tau'
}{
  \lambda; \gamma \vdash e : \tau'
} \]

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

\[
\begin{align*}
\tau &\leq \tau' \\
\Gamma \vdash \tau &\subseteq \tau' \\
\Gamma \vdash \tau &\subseteq \tau' \\
\Gamma \vdash \rho &\subseteq \rho' \\
\Gamma \vdash \rho' &\subseteq \rho'' \\
\tau \text{ var} &\text{ is invariant with respect to } \tau.
\end{align*}
\]

Corollary: \( \tau \text{ var} \) is invariant with respect to \( \tau \).

\[
\begin{align*}
\tau \equiv \tau' \\
\Gamma \vdash \tau \text{ var} &\subseteq \tau' \text{ var}
\end{align*}
\]
Typing Assignments

\[\begin{align*}
\lambda; \gamma \vdash e : \tau \quad \text{var} & \quad \lambda; \gamma \vdash e' : \tau \\
\lambda; \gamma \vdash e := e' : \tau \quad \text{cmd}
\end{align*}\]

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

\[\lambda; \gamma \vdash c : \tau \text{ cmd} \quad \lambda; \gamma \vdash c' : \tau \text{ cmd}\]
\[\lambda; \gamma \vdash c; c' : \tau \text{ cmd}\]

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

\[\lambda; \gamma \vdash c : \tau \text{ cmd} \quad \lambda; \gamma \vdash c' : \tau' \text{ cmd}\]
\[\lambda; \gamma \vdash c; c' : \tau \cap \tau' \text{ cmd}\]
Typing IF and WHILE

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau & \qquad \lambda; \gamma \vdash c : \tau \text{ cmd} & \qquad \lambda; \gamma \vdash c' : \tau \\
\lambda; \gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}
\end{align*}
\]

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau & \qquad \lambda; \gamma \vdash c : \tau \text{ cmd} \\
\lambda; \gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}
\end{align*}
\]

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

\[
\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma[x:\tau\; var] \vdash c : \tau' \; cmd}{\lambda; \gamma \vdash \text{letvar}\; x := e \; \text{in} \; c : \tau' \; cmd}
\]

- 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Authors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLam</td>
<td>Heintze&amp;Riecke (98)</td>
<td>Induction on typing derivation, denotational semantics</td>
</tr>
<tr>
<td>The secure CPS</td>
<td>Zdancewic&amp;Myers (01)</td>
<td>Induction on evaluation, small-step semantics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIF</td>
<td>Pottier&amp;Simonet (02)</td>
<td>Induction on evaluation, small-step semantics for pairing two executions</td>
</tr>
<tr>
<td>Java-light</td>
<td>Banerjee&amp;Naumann (02)</td>
<td>Induction on typing derivation, denotational semantics</td>
</tr>
</tbody>
</table>
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 exactly is high-security boolean?”
• Is information-flow policy EM-enforceable?
  – Suppose the operational semantics manipulates security labels and does run-time label checking.