Contributions

- Identify a central notion of dependency
- Connection between secure information flow and 3 types of program analyses
  - Program slicing
  - Binding-time analysis
  - Call-tracking
- Develop dependency core calculus (DCC) and translate calculi into DCC
- Define a semantic model for DCC that simplifies noninterference proofs

Outline

- Why information flow (SLam), slicing, binding-time, call-tracking are all dependency analyses
- SLam proof of noninterference
  - uses a logical-relations argument and denotational semantics
  - Heintze and Riecke, POPL ’98
- Dependency Core Calculus

Information Flow – SLam

- Heintze and Riecke, POPL ’98
- Lambda calculus with security annotations on types
- Well-typed programs have noninterference property:
  - No information flows from high-security values to low-security ones
  - Low-security data does not depend on high-security data.

Information Flow – SLam

Types

\[ s ::= (t, \kappa) \]
\[ t ::= \text{bool} \mid s \rightarrow s \mid s 	imes s \]
\[ \kappa \in \text{Security Lattice} \]

Exprs

\[ \text{bv ::= true} \mid \text{false} \mid \lambda x.e \]
\[ v ::= \text{bv}_\kappa \]
\[ e ::= x \mid v \mid (e \ e') \mid \text{protect}_\kappa e \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \]

SLam – Typing Rules

- [True] \[ \Gamma \vdash \text{true}_\kappa : (\text{bool}, \kappa) \]
- [False] \[ \Gamma \vdash \text{false}_\kappa : (\text{bool}, \kappa) \]
- [Lam] \[ \Gamma, x : s_1 \vdash e : s_2 \]
  \[ \Gamma \vdash (\lambda x : s_1 . e)_\kappa : (s_1 \rightarrow s_2, \kappa) \]
- [If] \[ \Gamma \vdash e : (\text{bool}, \kappa) \quad \Gamma \vdash e_1 : s \quad \Gamma \vdash e_2 : s \]
  \[ \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : s \]
SLam – Typing Rules

Example
if trueH then trueL else falseL : (bool,L)
Wrong!

Increase security level of result type to security level of “trueH”. Let (t,κ1)•κ2 = (t,κ1⊕κ2)

[If]  Γ|− e:(bool,κ)  Γ|− e1:s  Γ|− e2:s
Γ|− if e then e1 else e2 : s•κ

if trueH then trueL else falseL : (bool,L)
(booleL)•H = (bool,L⊕H) = (bool,H)

SLam – Subtyping

[Protect]  Γ|− e:s
Γ|− (protectκ,e) : s•κ

[Sub]  Γ|− e : s  s ≤ s'
Γ|− e : s'

SLam – Typing Rules

Principle: At every elimination rule, properties (security level) of the destructed constructor are transferred to the result type of the expression.

[App]  Γ|− e:(s1Æs2,κ)  Γ|− e':s1
Γ|− (ee') : s2•κ

Slicing

Determine which parts of the program (subterms) may contribute to the output
Parts that do not contribute may be replaced by any expression of the same type
Idea: label each part of the program and track dependency using type system

Slicing Calculus

Types  s ::= (t,κ)
t ::= bool | s+s | ...
κ ∈ Security Lattice

Example: (ix.true)false

(ix:(bool, {n3}),true{h1},n3)(false{h3})

Func: ((bool, {n3})Æ(bool, {n2}), {n1})

Prog: (bool, {n2})Æ{n1} = (bool, {n1,n2})
### Binding-Time Calculus

- Separate static from dynamic computation
- Dynamic values may be replaced by any expr of same type without affecting static results
- Types: 
  \[ t ::= \text{bool} | s \rightarrow s | \ldots \]
- Example: \((\lambda x:(\text{bool},\text{dyn}).\text{true}_{\text{sta}}) e_{\text{dyn}}\)
- Func: \(((\text{bool},\text{dyn}) \rightarrow (\text{bool},\text{sta}),\text{sta})\)
- Prog: \((\text{bool},\text{sta}) - \text{i.e.},\text{ result cannot depend on } e\)

### Call-tracking Calculus

- Determine which functions are called during evaluation; others may be replaced
- Types: 
  \[ s ::= \text{bool} | s \rightarrow s | \ldots \]
- Example: \(((\lambda x:(\text{bool},\text{dyn}).\text{true}_{\text{sta}}) e_{\text{dyn}})\)
- Func: \(((\text{bool},\text{dyn}) \rightarrow (\text{bool},\text{sta}),\text{sta})\)
- Prog: \((\text{bool},\text{sta}) - \text{i.e., result cannot depend on } e\)

### SLam

- **Operational Semantics**
  
  \[
  (\lambda x:s.e) v \rightarrow (\text{protect}_\kappa e[v/x])
  \]

  \[
  (\text{if }\text{true}_\kappa \text{ then } e_1 \text{ else } e_2) \rightarrow (\text{protect}_\kappa e_1)
  \]

  \[
  (\text{protect}_\kappa v) \rightarrow v \cdot \kappa
  \]

### SLam – Specifying Views

- Views can be specified using binary relations
- If \((x,y) \in R\) then \(x\) and \(y\) “look the same”

<table>
<thead>
<tr>
<th>Concrete View</th>
<th>Abstract View</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)</td>
<td>(A)</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concrete View</th>
<th>Abstract View</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)</td>
<td>(A)</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

### SLam – Semantics of Types

- \([\text{bool},\kappa]\) = \{\text{true},\text{false}\}
- \([s_1 \rightarrow s_2,\kappa]\) = \([s_1] \rightarrow [s_2]\)
- all partial continuous functions from \([s_1]\) to \([s_2]\)

- \(R[s,\kappa] = \text{“view of } s\text{ at level } \kappa\”\)

- \(R[s,\kappa] \subseteq [s] \times [s]\)
SLam – Views of Types

- If \( s = (t, \kappa) \), then for all lower \( \kappa' \) (\( \kappa \not\subseteq \kappa' \))
  \[
  R[\langle t, \kappa \rangle] = |s| \times |s| = A
  \]
- If \( s = (\text{bool}, \kappa) \) and \( \kappa \not\subseteq \kappa' \)
  \[
  R[\langle \text{bool}, \kappa \rangle] = C
  \]
- If \( s = (s_1 \rightarrow s_2, \kappa) \) and \( \kappa \not\subseteq \kappa' \)
  \[
  R[\langle s_1 \rightarrow s_2, \kappa \rangle] = \{(f,g) | \forall (x,y) \in R[\langle s_1, \kappa' \rangle]. (f(x),g(y)) \in R[\langle s_2 \cdot \kappa, \kappa' \rangle]\}
  \]

Equivalence, Related Environments

- Type context \( \Gamma = \lambda x.e : t \) and \( \kappa \not\subseteq \kappa' \)
  \[
  R[\langle t, \kappa \rangle] = \text{abstract}
  \]
- Theorem (Equivalence):
  \[
  \text{If } \emptyset \vdash e : t \text{ then } [[[\emptyset \vdash e : t]]]_{\eta} \text{ is defined iff } e \rightarrow^* v
  \]
- Theorem (Related Environments):
  \[
  \text{Suppose } \emptyset \vdash e : t \text{ and } \eta, \eta' \in [[\Gamma]] \text{ are related environments at } \kappa, \text{ then }([[\emptyset \vdash e : t]]_{\eta}, [[\emptyset \vdash e : t]]_{\eta'}) \in R[\langle t, \kappa \rangle]
  \]

Proof

- Consider open term: \( y(t,\kappa) \rightarrow C[y] : (\text{bool}, \kappa') \)
  \[
  d_{\Gamma} = [[[\emptyset \vdash y(t,\kappa)]](\kappa)]
  \]
- We must show \( (d_1, d_2) \in R[\langle t, \kappa \rangle] \)
  \[
  \text{Proof: } \kappa \not\subseteq \kappa' \text{ is abstract.}
  \]
- \( f_{\kappa} = [[[y(t,\kappa) \rightarrow C[y] : (\text{bool}, \kappa')]]](\kappa) \)
- By Related Environments theorem, we have:
  \[
  (f_1, f_2) \in R[\langle \text{bool}, \kappa' \rangle] \]
- Thus, \( f_1 \approx f_2 \). Easy to show that
  \[
  f_{\kappa} = [[[\emptyset \vdash \lambda x.e : (\text{bool}, \kappa')]]](\kappa). \text{ Since } v_1 \approx v_2, \text{ done.}
  \]

Recursion

- Need to deal with termination issues
- Call-by-name vs. Call-by-value
  - Strong vs. Weak noninterference
- Strong Noninterference: if a program terminates with one input and produces result \( v \), then it also terminates with any other “related” input and the result is related to \( v \)
- Weak Noninterference: if 2 related inputs cause a program to terminate the outputs are related

Dependency Core Calculus

- Types \( s ::= \text{unit} | s + s | \lambda x.e \) \( \kappa \in \text{Security Lattice} \)
- Expressions \( e ::= x | b_v \cdot e | \lambda x.e | \text{lift } e | \text{eta } e | ... \)
- Pointed types – to deal with termination
- Protected types
  - if \( \kappa \subseteq \kappa_1 \), then \( T_{\kappa_1}(s) \) is protected at level \( \kappa \)
DCC – Protected Types

- Protected types
  - if $\kappa \subseteq \kappa_1$, then $T_{\kappa_1}(s)$ is protected at level $\kappa$
  - $T_{\kappa_1}$ adjusts the views: makes views of lower security levels abstract
- Semantics of protected types
  - $|T_{\kappa}(s)| = |s|$
  - $R[T_{\kappa}(s), \kappa'] = R[s, \kappa']$ if $\kappa \subseteq \kappa'$
    otherwise $|s| \times |s|$

DCC

- DCC: CBN operational semantics
  - easy to translate CBN calculi to DCC and prove strong interference
  - hard to translate CBV calculi to DCC
- vDCC: CBN operational semantics, but definition of protected types is slightly different
  - if $t$ is protected at level $\kappa$ then $t_1$ is protected at level $\kappa$
  - can translate CBV calculi to vDCC and prove weak noninterference

Discussion

- Limitations?
  - Cannot translate Davies and Pfenning’s binding-time analysis into DCC – cannot model coercion of run-time objects to compile-time objects
  - Can DCC help with other analyses?
    - semantic dependencies in optimizing compilers
    - region-based memory management
  - How about a call-by-value DCC?
    - Uniform Type Structure for Secure Information Flow – Honda, Yoshida, POPL 02
    - Translate DCCv into linear/affice Pi-calcul for info flow
  - Extensions: imperative features, concurrency, …