A Core Calculus of Dependency

Abadi, Banerjee,
Heintze, Riecke

POPL '99

CS711

Amal Ahmed

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

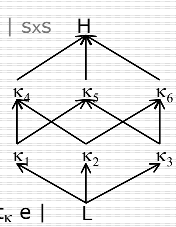
- Heintze and Riecke, POPL '98
- Lambda calculus with security annotations on types
- Well-typed programs have noninterference property:
  - No information flows from high-security values to low-security ones
  - Low-security data does not *depend* on high-security data.

Information Flow – SLam

- Types $s ::= (t,\kappa)$

  $t ::= bool \mid s \rightarrow s \mid s+s \mid s \times s$

  $\kappa \in$ Security Lattice



- Exprs

  $bv ::= true \mid false \mid \lambda x.e$

  $v ::= bv_\kappa$

  $e ::= x \mid v \mid (e\ e') \mid protect_\kappa\ e \mid$ if e then e1 else e2

SLam – Typing Rules

[True] $\Gamma \vdash true_\kappa : (bool,\kappa)$

[False] $\Gamma \vdash false_\kappa : (bool,\kappa)$

[Lam] $$\frac{\Gamma, x{:}s1 \vdash e : s2}{\Gamma \vdash (\lambda x{:}s1.e)_\kappa : (s1 \rightarrow s2, \kappa)}$$

[If] $$\frac{\Gamma \vdash e:(bool,\kappa) \qquad \Gamma \vdash e1:s \qquad \Gamma \vdash e2:s}{\Gamma \vdash \text{if } e \text{ then } e1 \text{ else } e2 : s}$$

SLam – Typing Rules

- Example
  if true_H then true_L else false_L : (bool,L) *Wrong!*
- Increase security level of result type to security level of "true_H". Let $(t,\kappa 1) \bullet \kappa 2 = (t, \kappa 1 \oplus \kappa 2)$

$$[\text{If}] \quad \frac{\Gamma \mid\!- e:(\text{bool},\kappa) \qquad \Gamma \mid\!- e1:s \qquad \Gamma \mid\!- e2:s}{\Gamma \mid\!- \text{if } e \text{ then } e1 \text{ else } e2 : s \bullet \kappa}$$

- if true_H then true_L else false_L : (bool,L)$\bullet$H
- (bool,L)$\bullet$H = (bool,L $\oplus$ H) = (bool,H)

SLam – Typing Rules

- *Principle:* At every elimination rule, properties (security level) of the destructed constructor are transferred to the result type of the expression.
- $[\text{App}] \quad \dfrac{\Gamma \mid\!- e:(s1 \rightarrow s2,\kappa) \qquad \Gamma \mid\!- e':s1}{\Gamma \mid\!- (e\,e') : s2 \bullet \kappa}$

SLam – Typing Rules

$$[\text{Protect}] \quad \frac{\Gamma \mid\!- e:s}{\Gamma \mid\!- (\text{protect}_\kappa\, e) : s \bullet \kappa}$$

$$[\text{Sub}] \quad \frac{\Gamma \mid\!- e : s \qquad s \le s'}{\Gamma \mid\!- e : s'}$$

SLam – Subtyping

$$[\text{SubBool}] \quad \frac{\kappa \sqsubseteq \kappa'}{(\text{bool},\kappa) \le (\text{bool},\kappa')}$$

$$[\text{SubFun}] \quad \frac{\kappa \sqsubseteq \kappa' \qquad s1' \le s1 \qquad s2 \le s2'}{(s1 \rightarrow s2,\kappa) \le (s1' \rightarrow s2',\kappa')}$$

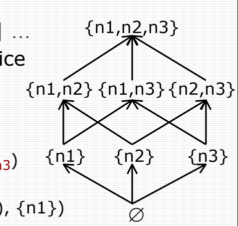
$$[\text{SubTrans}] \quad \frac{s1 \le s2 \qquad s2 \le s3}{s1 \le s3}$$

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 $s ::= (t,\kappa)$
  $t ::= \text{bool} \mid s \rightarrow s \mid \ldots$
  $\kappa \in$ Security Lattice



- Example: (λx.true)false
- $(\lambda x:(\text{bool},\{n3\}).\text{true}_{n2})_{n1}(\text{false}_{n3})$
- Func: ((bool,{n3})$\rightarrow$(bool,{n2}), {n1})
- Prog: (bool,{n2})$\bullet${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 $s ::= (t,\kappa)$
  $t ::= bool \mid s \to s \mid \ldots$
  $\kappa ::= sta \mid dyn$ *where* $sta \le dyn$
- Example: $(\lambda x{:}(bool,dyn).true_{sta})_{sta}\ e_{dyn}$
- Func: ((bool,dyn)→(bool,sta),sta)
- Prog: (bool,sta) – i.e., result cannot depend on e

Call-tracking Calculus

- Determine which functions are called during evaluation; others may be replaced
- Types $s ::= bool \mid s \to^{\kappa} s \mid \ldots$
  $\kappa ::=$ <sets of labels of lambda exprs>

[Lam] $$\frac{\Gamma, x{:}s1 \vdash e{:}s2,\kappa}{\Gamma \vdash (\lambda x{:}s1.\ e)_n : (s1 \to^{\{n\}\oplus\kappa} s2),\varnothing}$$

[App] $$\frac{\Gamma \vdash e{:}(s1 \to^{\kappa} s2),\kappa 1 \qquad \Gamma \vdash e'{:}s1,\kappa 2}{\Gamma \vdash (e\,e') : s2,\ \kappa \oplus \kappa 1 \oplus \kappa 2}$$

SLam

- Operational Semantics

$((\lambda x{:}s.e)_\kappa\, v) \to (protect_\kappa\ e[v/x])$

$(if\ true_\kappa\ then\ e1\ else\ e2) \to (protect_\kappa\ e1)$

$(protect_\kappa\ v) \to v \bullet \kappa$

SLam – Proving Noninterference

- Give a denotational semantics for SLam
- A high-security computation can depend on a high-security input, but a low-security computation cannot; the 2 computations have different "views" of the same high-security input
  - ((bool,H)→(bool,L),L) looks like $\forall\alpha.\alpha \to bool$
  - ((bool,H)→(bool,L),H) looks like bool→bool
- For each type (t,κ), specify CPO as well as a view for each level $\kappa \in$ Lattice
- Functions must preserve the view

SLam – Specifying Views

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

Concrete View

C	true	false
true	1	0
false	0	1

Abstract View

A	true	false
true	1	1
false	1	1

SLam – Semantics of Types

- $|(bool,\kappa)| = \{true,false\}$
- $|(s1 \to s2,\kappa)| = |s1| \to_p |s2|$
  - all partial continuous functions from |s1| to |s2|
- $R[s,\kappa]$ = "view of s at level κ"
- $R[s,\kappa] \subseteq |s| \times |s|$

SLam – Views of Types

- If s = (t,κ), then for all lower κ' (κ ⋢ κ')
  R[s,κ'] = |s| x |s| = **A**
- If s = (bool,κ) and κ ⊑ κ' then
  R[s,κ'] = **C**
- If s = (s1→s2,κ) and κ ⊑ κ' then
  R[s,κ'] = {(f,g) | ∀(x,y)∈R[s1,κ'].
  (f(x),g(y))∈R[s2•κ,κ']}

Adequacy, Related Environments

- Typing context Γ = x1:s1, x2:s2, … , xn:sn
  |Γ| = |s1| x |s2| x … x |sn|
  Environment η ∈|Γ|
- Theorem (Adequacy):
  If ∅ |– e:s then [[∅ |– e:s]]η is defined iff e →* v
- Theorem (Related Environments):
  Suppose Γ |– e:s and η,η'∈|Γ| are related environments at κ, then
  ([[Γ |– e:s]]η, [[Γ |– e:s]]η') ∈ R[s,κ]

Equivalence, Noninterference

- C[] is a context with a hole
- e ~ e' = whenever e→* v and e' →* v', v=v'
- Theorem(Noninterference):
  Suppose ∅ |– ei:(t,κ) and ∅ |– C[e1]:(bool,κ') where κ ⋢ κ' then C[e1] ~ C[e2].

Proof

- Consider open term: y:(t,κ) |– C[y] : (bool,κ')
- di = [[∅ |– ei:(t,κ)]]()
- We must show (d1,d2)∈ R[(t,κ),κ']
  - Proof: Since κ ⋢ κ' R[(t,κ), κ'] is abstract.
- fi = [[y:(t,κ) |– C[y] : (bool,κ')]]di
- By Related Environments theorem, we have:
  (f1, f2) ∈ R[(bool,κ'), κ'] = **C**
- Thus, f1=f2. Easy to show that
  fi = [[∅ |– v:(bool,κ')]](). Since v1~v2, 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 ::= unit | s→s | $s_\perp$ | $T_\kappa(s)$ | s+s | sxs
  κ ∈ Security Lattice
- Exprs bv ::= () | λx.e
  e ::= x | bv_κ | (e e') | lift e | η_κ e |…
- Pointed types – to deal with termination
- Protected types
  - if κ ⊑ κ1, then $T_{\kappa 1}(s)$ is protected at level κ

DCC – Protected Types

- Protected types
  - if $\kappa \sqsubseteq \kappa l$, then $T_{\kappa l}(s)$ is protected at level κ
  - $T_{\kappa l}$ 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 $\kappa \sqsubseteq \kappa'$

  $= |s| \times |s|$ otherwise

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 κ then $t_\perp$ is protected at level κ
  - 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-calc for info flow
- Extensions: imperative features, concurrency, …