

Types for Access Control: Foundations and Methodology

Christian Skalka, The University of Vermont

Joint work with François Pottier, Inria Rocquencourt, and Scott Smith, The Johns
Hopkins University

Topics

1. Programming language (PL)-based access control:
 - Benefits
 - Examples, formal models
2. Using types to statically enforce PL-based security:
 - Benefits
 - Particular systems, results
3. Developing type systems: methodology
 - Re-use existing results and implementations
 - Inspire design

PL-based access control

- The problem: protecting *local resources* against foreign, *hostile* code
 - Resources: disk space, memory, printing, clock cycles
- PL-based security adds security mechanisms to language base as primitives or add-on libs:
 - Program annotations allow definition of *security policy*
 - Embedded *checks* enforce security policy

Benefits of PL-based access control

- Systems level security (EROS, SSL) largely transparent to the applications programmer
- PL-based security gives applications programmer control of security policy:
 - Recent proliferation of *mobile code* demands flexible approach to security
 - Greater adaptability and expressiveness leads to more robust security policies
- PL-based security nicely incorporates *context* and *history* into access control decisions

Example: Java JDK1.2

Java JDK1.2 provides a PL-based security mechanism for defining and enforcing access control policies:

- All programs are signed by the code *owner* (principal)
- Local policy maps owners to sets of authorized local privileges in *access control lists* (ACLs)
- Privileges may be explicitly enabled in code regions with `doPrivileged` method
- Activated privileges are checked with `checkPermission` method

Any `checkPermission` encountered during execution sets off a *dynamic* check...

Stack inspection

Each program call-stack frame is annotated with identity of frame owner and any privileges activated within frame:

- $\text{doPrivileged}(r)$ places privilege r on current stack frame if stack frame owner is authorized for r
- $\text{checkPermission}(r)$ initiates *stack inspection algorithm*. Stack frames are searched from most to least recent:
 - **fail** if a stack frame owned by a principal unauthorized for r is encountered
 - Otherwise **succeed** if r is found

Stack inspection example

Assuming Joe is authorized for print privilege:

(* code owned by System *)

$enablePrint(g, x) = \{doPrivileged(\mathbf{print}); g(x)\}$

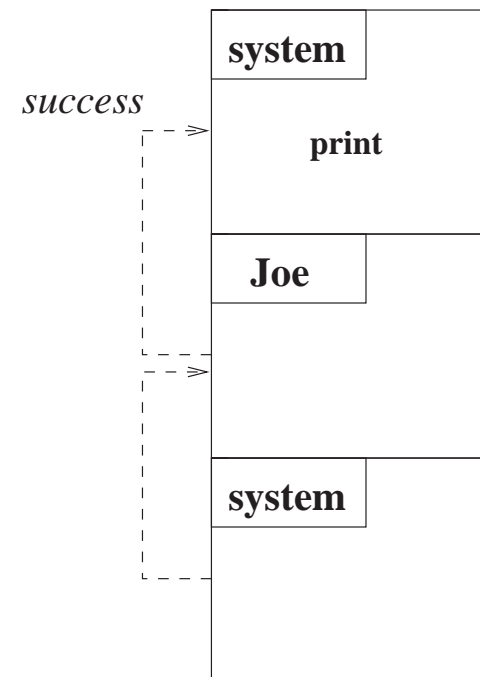
$safePrint(f) = \{checkPermission(\mathbf{print}); \mathbf{print}(f)\}$

(* code owned by Joe *)

$joeProg(f) = \{safePrint(f)\}$

(* code executing locally *)

$enablePrint(joeProg, file)$



Stack inspection example

Assuming Joe is *not* authorized for print privilege:

(* code owned by System *)

$enablePrint(g, x) = \{doPrivileged(\mathbf{print}); g(x)\}$

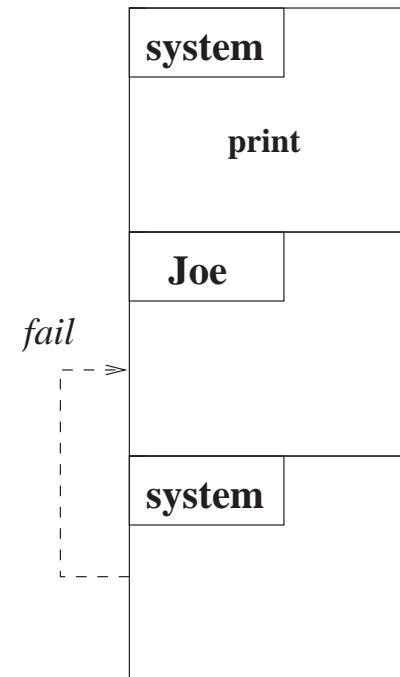
$safePrint(f) = \{checkPermission(\mathbf{print}); \mathbf{print}(f)\}$

(* code owned by Joe *)

$joeProg(f) = \{safePrint(f)\}$

(* code executing locally *)

$enablePrint(joeProg, file)$



Reconsidering stack inspection

Dynamic stack inspection proven useful in practice, e.g. *sandboxing*; but mechanism has several drawbacks:

- Checks incur a runtime penalty
- Preservation of call stack prevents many compiler optimizations (e.g. tail-call optimizations)
- Dynamic check delays detection of security flaws
- Security policy buried in code annotations

What's the policy?

```
if (!sclSet) {
    // Workaround for 4154308 (JDK 1.2FCS build breaker)
    // Launcher l = Launcher.getLauncher();
    sun.misc.Launcher l = sun.misc.Launcher.getLauncher();
    if (l != null) {
        scl = l.getClassLoader();
    }
    sclSet = true;
}
if (scl == null) {
    return null;
}
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    ClassLoader ccl = getCallerClassLoader();
    if (ccl != null && ccl != scl && !scl.isAncestor(ccl)) {
        sm.checkPermission(GetClassLoaderPerm);
    }
}
return scl;
}
```

Types for PL-based access control

We argue that static type analysis of stack inspection enhances the system:

- Static enforcement of security means runtime checks, and associated performance penalties, can be eliminated
- Elimination of runtime checks allows safe compiler optimizations
- Static detection of errors means quicker detection of errors
- Declarative nature of types makes security policy more explicit, understandable

These benefits generalize to other access control models.

Formalizing stack inspection

To develop the stack inspection type analysis in a manner that accommodates rigorous proof, we begin with as simplified language model, λ_{sec} :

- Purely *functional* core
 - Function definitions: $\lambda x.e$
 - Function application: $(\lambda x.e)e'$
- Additions for modelling stack inspection
 - Access control list: \mathcal{A}
 - Signed expressions: $p.e$
 - Privilege enabling: $\text{check } r \text{ then } e$
 - Privilege checking: $\text{enable } r \text{ in } e$

Formalizing stack inspection

For example, the following are λ_{sec} expressions:

$$\begin{aligned} id &\triangleq \lambda x.p.x \\ check_r &\triangleq \lambda x.p.\text{check } r \text{ then } x \\ enable_r &\triangleq p.\text{enable } r \text{ in } check_r(id) \end{aligned}$$

λ_{sec} is sufficient to study essential aspects of security model:

- Function call/return basic issue
- Other features, e.g. state and objects, engineering issue

λ_{sec} semantics

The λ_{sec} definition also comprises an *operational semantics*:

- A mathematical, inductive definition of stacks S :

$$S ::= \emptyset \mid \langle p, R \rangle :: S$$

- A formalization of stack inspection: $\text{inspect}(S, r)$
- A well-defined set of evaluation rules on configurations S, e

λ_{sec} operational semantics (highlights)

$$\begin{aligned} S, (\lambda x.p.e)v &\rightarrow \langle p, \emptyset \rangle :: S, \cdot e[v/x] \cdot \\ \langle p, R \rangle :: S, \text{enable } r \text{ in } e &\rightarrow \langle p, R \cup \{r\} \rangle :: S, e \\ S, \text{check } r \text{ then } e &\rightarrow S, e \quad \text{if } \text{inspect}(S, r) = \mathbf{true} \\ \langle p, R \rangle :: S, \cdot v \cdot &\rightarrow S, v \\ &\vdots \end{aligned}$$

Note: programs that fail security checks become stuck (go wrong)

Types for λ_{sec}

Intuition: type *terms* express security requirements of functions.

- Function types of the form: $\tau \xrightarrow{\{\rho\}} \tau$
- Security annotations $\zeta = \{\rho\}$ are *set types*
- Contents ρ inductively defined:
 - Either empty \emptyset , completely full ω or
 - of the form $(r+, \rho)$ or $(r-, \rho)$

Annotations ζ represent the set of privileges checked upon execution of the function.

Types for λ_{sec}

Types judgements are valid with respect to security environment, represented as ζ , and code owner p in current scope:

- Type judgements $p, \zeta, \Gamma \vdash e : \tau$
- Environment ζ represents privileges on stack, p owner in active frame
- Type validity defined as derivability in inference system:

$$\text{CHECK} \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau}{p, \{r+, \rho\}, \Gamma \vdash \text{check } r \text{ then } e : \tau}$$

Types for λ_{sec}

Other rules, where φ ranges over $\{+, -\}$:

$$\text{APP} \quad \frac{p, \zeta, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\zeta} \tau \quad p, \zeta, \Gamma \vdash e_2 : \tau_2}{p, \zeta, \Gamma \vdash e_1 e_2 : \tau}$$

$$\text{ENABLE SUCCESS} \quad \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau \quad r \in \mathcal{A}(p)}{p, \{r\varphi, \rho\}, \Gamma \vdash \text{enable } r \text{ in } e : \tau}$$

An expression e is *well-typed* iff

$$\text{nobody}, \{\emptyset\}, \Gamma \vdash e : \tau$$

is derivable, in which case we write $e : \tau$

λ_{sec} type examples

$$\mathcal{A} = \{\dots, p \mapsto \{r\}, \dots\}$$

$$id \triangleq \lambda x.p.x$$

$$check_r \triangleq \lambda x.p.\text{check } r \text{ then } x$$

$$enable_r \triangleq \text{enable } r \text{ in } check_r(id)$$

$$id : \tau \xrightarrow{\{\emptyset\}} \tau$$

$$check_r : \tau \xrightarrow{\{r+, \emptyset\}} \tau$$

$$enable_r : \tau \xrightarrow{\{\emptyset\}} \tau$$

$check_r(id)$ is *not* well-typed!

Type safety implies security

Given our formal model:

- Operational semantics (where failing security checks result in stuck expressions)
- Type derivation system

We may prove a *syntactic* type safety result:

Theorem (λ_{sec} **Type Safety**): If e is well-typed, then e does not go wrong.

Corollary : If e is well-typed, then e has no runtime security errors.

How to prove type safety?

An *ab initio* approach to proving syntactic type safety for λ_{sec} :

- Feasible (via subject reduction), but:
- Tedious
- Error-prone

Alternative *methodology* allows development on basis of existing results...

Building on foundations

We use two techniques for building type systems on foundations of previous results:

1. *Transformational* approach— novel languages embedded into existing languages
2. The system *HM(X)*, a polymorphic type-constraint framework

These foundations provide several benefits:

- Less proof effort, greater confidence
- Existing implementations (type inference)
- Design inspiration

Transformational Approach

Type system for expressions e in a novel *source language* (e.g. λ_{sec}) is obtained by transformation $\langle e \rangle$:

- $\langle e \rangle$ is a term in a familiar *target language* pre-equipped with sound type system, including inference algorithm
- Transformation preserves semantics (is a *simulation*):

Theorem: If e safely evaluates to v , then $\langle e \rangle$ safely evaluates to $\langle v \rangle$. If e has runtime errors, then so does $\langle e \rangle$. If e diverges, then $\langle e \rangle$ diverges.

Transformational Approach

Correctness of term transformation $\langle e \rangle$ yields a source language type system “for free”— without further proof effort:

- Sound *indirect* type system for expressions e obtained from target type system: if $\langle e \rangle : \tau$ then $e : \tau$
- Since $\langle e \rangle : \tau$ can be inferred, compose transformation and type inference to infer $e : \tau$
- Method yields insight into semantics and/or desired structure of *direct* types for source language, eases proof development

The system $\text{HM}(X)$

To develop the target language for our transformations, we use the system $\text{HM}(X)^*$, a constraint based type *framework*.

$\text{HM}(X)$ provides a type system for functional core that may be *instantiated* with specialized constraint systems for particular applications:

- any instantiation enjoys syntactic type safety in the framework, with minimal proof requirements
- any instantiation yields type inference by supplying constraint solution algorithm

* *Odersky, Sulzmann and Wehr, TAPOS 1999, vol. 5 no. 1*

The HM(X) language

The HM(X) framework provides a *core language*:

- Functional abstractions $\text{fix } z.\lambda x.e$, where z binds to $\text{fix } z.\lambda x.e$ in e
- Standard reference operations ref , $!$ and $:=$
- Let expressions $\text{let } x = v \text{ in } e$; note *values restriction*
- Core language *extendable*:
 - Specify constants $c \in \text{Const}$ in instantiations
 - Specify semantics of functional constants with function δ

Operational semantics of HM(X)(highlights)

Evaluation defined on *configurations* e, σ , where:

- Stores σ are partial mappings from *locations* l to values v

$$(\mathbf{fix} \ z.\lambda x.e)v, \sigma \rightarrow e[v/x][\mathbf{fix} \ z.\lambda x.e/z], \sigma$$

$$\mathbf{let} \ x = v \ \mathbf{in} \ e, \sigma \rightarrow e[v/x], \sigma$$

$$\mathbf{c} \ v, \sigma \rightarrow \delta(\mathbf{c}, v), \sigma$$

$$!l, \sigma \rightarrow \sigma(l), \sigma$$

⋮

The HM(X) type and constraint language

The HM(X) framework provides a basic language of types:

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \tau \text{ ref}$$

constraints:

$$C ::= \mathbf{true} \mid \tau = \tau \mid \tau \leq \tau \mid C \wedge C \mid \exists \alpha. C$$

and constrained polymorphic type schemes:

$$\sigma ::= \forall \bar{\alpha}[C]. \tau$$

Core type and constraint language *extendable*:

- Add new specialized terms to language of types and constraints
- Defines initial type bindings Δ for constants in $Const$

HM(X) type inference rules (highlights)

$$\text{SUB} \quad \frac{C, \Gamma \vdash e : \tau \quad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}$$

$$\text{CONST} \quad C, \Gamma \vdash \mathbf{c} : \Delta(\mathbf{c})$$

$$\text{LET} \quad \frac{C, \Gamma \vdash e_1 : \sigma \quad C, (\Gamma; x : \sigma) \vdash e_2 : \tau}{C, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

$$\text{APP} \quad \frac{C, \Gamma \vdash e_1 : \tau' \rightarrow \tau \quad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1 e_2 : \tau}$$

$$\forall \text{INTRO} \quad \frac{C \wedge D, \Gamma \vdash e : \tau \quad \bar{\alpha} \cap \text{fv}(C, \Gamma) = \emptyset}{C \wedge \exists \bar{\alpha}. D, \Gamma \vdash e : \forall \bar{\alpha}[D]. \tau}$$

$$\forall \text{ELIM} \quad \frac{C, \Gamma \vdash e : \forall \bar{\alpha}[D]. \tau' \quad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

HM(X) type safety

HM(X) framework includes results applicable to *any* instance:

- *Subject reduction* for core language
- *Normalization*, treating subtyping and polymorphism issues

Theorem (λ_{sec} Type Safety): If $e : \tau$ is valid, then e does not go wrong.

- Type safety extends to full language by proving Δ sound wrt δ (the δ -typability property)

Target language: pml_B

The *target language* of our transformations, called pml_B , is defined as an instance of $\text{HM}(X)$.

The language extends the $\text{HM}(X)$ core with various features:

- *records* in the style of Projective ML*
- *sets* of atomic elements, set operations

The language includes an accurate type system that captures dynamic properties of records and sets:

- *row types* accurately describe contents of records and sets
- *conditional constraints* accurately describe behavior of set operations

* Rémy, CLFP92

Language details

In the $\text{HM}(X)$ instantiation, we populate Const with constants that define a language of records:

- elevation: $\{v\}$, records with default values v
- modification: $v\{a = v'\}$, updates v to associate v' with a
- projection: $v.a$, selects the value stored in a field from v

and a language of sets:

- sets of atomic elements: $B \subseteq \mathcal{L}_b$, with $\mathcal{L}_b = \{b_1, b_2, \dots\}$
- cosets: \bar{B} , where $\bar{B} = \mathcal{L}_b - B$
- set operations: membership check \ni_b , union \vee , intersection \wedge , difference \ominus

pml_B operational semantics (highlights)

We define the behavior of pml_B functional constants in $\text{HM}(X)$ by extending δ with:

<i>intersection:</i>	$B_1 \wedge B_2 \rightarrow B_1 \cap B_2$
<i>union:</i>	$B_1 \vee B_2 \rightarrow B_1 \cup B_2$
<i>membership:</i>	$B \ni b \rightarrow B \quad \text{if } b \in B$
<i>default project:</i>	$\{v\}.a \rightarrow v$
<i>skip project:</i>	$v\{a' = v'\}.a \rightarrow v.a \quad a' \neq a$
<i>project:</i>	$v\{a = v'\}.a \rightarrow v'$
	\vdots

Types for pml_B

The pml_B type system is defined by extending $\text{HM}(X)$ with the constraint system RS (**R**ows and **S**ets), as specified by the framework:

- A *type and constraint language* comprising row types and conditional constraints
- A standard *interpretation in a model*, specifying e.g. behavior of conditional constraints
- *Initial type bindings* Δ for records, sets and associated operations

The system RS

We extend the basic $\text{HM}(X)$ system with row types and *presence constructors*:

$$\begin{array}{ll}
 \zeta ::= \alpha, \beta, \dots \mid (b : \tau; \zeta) \mid \partial\tau & \text{rows} \\
 c ::= + \mid - & \text{constructors} \\
 \tau ::= \alpha, \beta, \dots \mid \tau \rightarrow \tau \mid \{\zeta\} \mid c & \text{types}
 \end{array}$$

These types describe sets by asserting which elements are present (+) and which are absent (-); letting $B = \{b_1, b_2\}$:

$$\begin{array}{l}
 B : \{b_1 : +; b_2 : +; \partial-\} \\
 B : \{b_1 : +; b_2 : +; b_3 : -; \partial-\}
 \end{array}$$

We define the following syntactic sugar for types:

$$(b : \tau; \zeta) \triangleq (b\tau, \zeta) \qquad \partial- \triangleq \emptyset \qquad \partial+ \triangleq \omega$$

Hence:

$$B : \{b_1+, b_2+, \emptyset\}$$

Conditional constraints

The RS language of constraints extends the $\text{HM}(X)$ constraint language with *conditional constraints**:

$$C ::= \dots \mid \text{if } c \leq \tau \text{ then } \tau' \leq \tau'' \quad \text{constraints}$$

The behavior of conditional constraints is specified by the RS interpretation, and includes an “intuitive” interpretation:

$$\frac{c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

* *Pottier, Nord. J. Comp., Nov. 2000*

Conditional constraints

Conditional constraints are also equipped with a more complex interpretation:

$$\frac{\forall b \in \mathcal{L}_b . (c \leq \rho(\zeta)(b) \Rightarrow \rho(\zeta')(b) \leq \rho(\zeta'')(b))}{\rho \vdash \text{if } c \leq \zeta \text{ then } \zeta' \leq \zeta''}$$

This interpretation is used for an accurate description of the behavior of set operations:

$$\begin{aligned} \wedge : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\} \\ \text{where } C = \quad & \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ & \wedge \text{ if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

Conditional constraint example

$$\begin{aligned} \wedge & : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\} \\ & \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ & \quad \wedge \text{ if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

Define \leq as $=$; then to type the expression $\{b_1, b_2\} \wedge \{b_2, b_3\}$:

$$\beta_1 = \{b_1+, b_2+, b_3-, \emptyset\}$$

$$\beta_2 = \{b_1-, b_2+, b_3+, \emptyset\}$$

$$\beta_3 = \{b_1\gamma_1, b_2\gamma_2, b_3\gamma_3, \beta\}$$

Conditional constraint example

$$\begin{aligned} \wedge : \forall \beta_1 \beta_2 \beta_3 [C]. \{ \beta_1 \} \rightarrow \{ \beta_2 \} \rightarrow \{ \beta_3 \} \\ \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ \quad \wedge \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

The interpretation of constraints will force the following “splitting”:

$$\begin{aligned} C = \quad & \text{if } - \leq + \text{ then } - \leq \gamma_1 \wedge \text{if } + \leq + \text{ then } - \leq \gamma_1 \\ & \wedge \text{if } - \leq + \text{ then } - \leq \gamma_2 \wedge \text{if } + \leq + \text{ then } + \leq \gamma_2 \\ & \wedge \text{if } - \leq - \text{ then } - \leq \gamma_3 \wedge \text{if } - \leq + \text{ then } - \leq \gamma_3 \\ & \wedge \text{if } - \leq \emptyset \text{ then } \emptyset \leq \beta \wedge \text{if } + \leq \emptyset \text{ then } \emptyset \leq \beta \end{aligned}$$

Conditional constraint example

$$\begin{aligned} \wedge & : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\} \\ & \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ & \quad \wedge \text{ if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

This splitting will force the following unification:

$$\beta_3 = (b_1-, b_2+, b_3-, \emptyset)$$

or

$$\beta_3 = (b_2+, \emptyset)$$

This describes the contents of $\{b_1\}$, and $\{b_1, b_2\} \wedge \{b_2, b_3\} \rightarrow^* \{b_2\}$.

Other pml_B initial bindings

We also populate Δ with:

$$\ni b : \forall \beta. \{b+, \beta\} \rightarrow \{b+, \beta\}$$

$$\vee : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}$$

$$\text{where } C = \quad \text{if } + \leq \beta_1 \text{ then } \omega \leq \beta_3$$

$$\quad \wedge \text{if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3$$

$$\ominus : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}$$

$$\text{where } C = \quad \text{if } + \leq \beta_2 \text{ then } \emptyset \leq \beta_3$$

$$\quad \wedge \text{if } - \leq \beta_2 \text{ then } \beta_1 \leq \beta_3$$

Type safety for pml_B

Note that the type binding

$$\ni b : \forall \beta. \{b+, \beta\} \rightarrow \{b+, \beta\}$$

requires that b be an element of B for any expression $B \ni b$, so that any operationally unsafe membership check is not well-typed.

- Type safety implies that *optimizations* may be effected; run-time membership checks may be eliminated
- Type safety obtained by proving δ -typability for pml_B (simple)

Back to stack inspection

A sound type system for λ_{sec} is obtained by simulation in pml_B .

We use Wallach's *security-passing-style (SPS)* transformation*, where security information is represented as sets of *active privileges*:

- Security information is *passed down the stack* as an explicit function parameter, rather than maintained in stack annotations
- Enabling a privilege adds it to the active privileges, checking a privilege checks its membership
- If signed code $p.e$ is encountered, any active privileges $r \notin \mathcal{A}(p)$ are deactivated (removed from active set)

* *Wallach, PhD thesis, Yale 1999*

The λ_{sec} -to- pml_B transformation (highlights)

Our security passing style transformation is defined (in part) as follows:

$$\begin{aligned} \llbracket \lambda x. f \rrbracket_p &= \lambda x. \lambda s. \llbracket f \rrbracket \\ \llbracket e_1 e_2 \rrbracket_p &= \llbracket e_1 \rrbracket_p \llbracket e_2 \rrbracket_p s \\ \llbracket \text{enable } r \text{ in } e \rrbracket_p &= \text{let } s = s \vee (\{r\} \cap \mathcal{A}(p)) \text{ in } \llbracket e \rrbracket_p \\ \llbracket \text{check } r \text{ then } e \rrbracket_p &= \text{let } _ = s \ni r \text{ in } \llbracket e \rrbracket_p \\ \llbracket p.e \rrbracket &= \text{let } s = s \wedge \mathcal{A}(p) \text{ in } \llbracket e \rrbracket_p \end{aligned}$$

Free occurrences of s are assigned \emptyset by the top-level transformation function $\langle e \rangle$, where $\langle e \rangle = \llbracket e \rrbracket_{p_0}[\emptyset/s]$.

Transformation example

$$check_r = \lambda x.p.check\ r\ then\ x$$

$$id = \lambda x.p.x$$

$$\langle\langle check_r \rangle\rangle = \lambda x.\lambda s.let\ s = s \wedge \mathcal{A}(p)\ in\ (s \ni r; x)$$

$$\langle\langle id \rangle\rangle = \lambda x.\lambda s.let\ s = s \wedge \mathcal{A}(p)\ in\ x$$

$$\langle\langle check_r\ id \rangle\rangle = (\langle\langle check_r \rangle\rangle)(\langle\langle id \rangle\rangle)(\emptyset)$$

$$\langle\langle check_r\ id \rangle\rangle \rightarrow^* let\ s = \emptyset \wedge \mathcal{A}(p)\ in\ (s \ni r; \langle\langle id \rangle\rangle)$$

Indirect types for λ_{sec}

An *indirect* static analysis for λ_{sec} , ensuring runtime security safety, may be defined via the pml_B type system:

- Compose λ_{sec} -to- pml_B transformation and pml_B type judgements— that is, $e : \tau$ if $(e) : \tau$
- Type safety immediate by correctness of transformation and pml_B type safety

Methodology allows a minimum of proof effort.

Direct types for λ_{sec}

Direct λ_{sec} type system (defined at beginning of presentation) better than indirect:

- Transformation may be inefficient, problems with type error reporting
- To establish direct type safety for λ_{sec} , we prove a correspondance lemma for the indirect and direct type systems:

Lemma: Let $e : \tau$ be a typing in the direct λ_{sec} type system; then $e : \tau$ is valid iff $(e) : \tau$ is valid in the pml_B type system.

The proof is easy, since the systems are closely related.

Direct types for λ_{sec}

λ_{sec} type safety is a corollary of the preceding lemma and pml_B type safety:

Theorem: (Direct λ_{sec} Type Safety) If e is well-typed in the direct λ_{sec} type system, then e does not go wrong.

A *flexible*, *efficient* and *readable* type system for static enforcement of the stack inspection model, proven sound with minimum of effort.

- Can also develop *implementation* on basis of existing methods

Other security models: Object confinement

The type-based, transformational approach has general scope in the context of PL-based security

- Type analysis is not limited to stack inspection security
- Transformational approach can exploit previous results in application to other security paradigms
 - re-use pml_B and type system, including soundness results

Object confinement security for OO languages benefits from this approach

Object confinement

Object confinement security is founded on *capability-based* security:

- Capabilities are *unforgeable references* to regions of code
- OS-based capability security: KeyKOS, EROS
- PL-based capability security: E, Secure Network Objects, *Java*

Java is a prime example of the capability model:

- Object references must be explicitly obtained
 - No backdoors such as pointer arithmetic, buffer overflow
 - Enforced by type system

Object confinement: hardening

In the *pure* model, security is based on high-level program design and sound distribution of capabilities.

Some systems also include *hardening* mechanisms to strengthen security:

- Runtime checks on safety of capability communication, access
 - Ensures that capabilities don't fall into the wrong hands, or provide stronger access than is intended
- `private`, `protected` visibility modifiers

Our analysis focuses on capability hardening mechanisms

The language `pop`

In `pop`, we model capabilities as objects with method definitions, *domain* identifiers, and fine-grained *interfaces*, e.g.:

$$[\text{read} = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, d' \mapsto \{\text{read}\}\}$$

- Conception of *domains* open to interpretation: code owner names, static scope identifiers, etc.
- Semantics defined with respect to a *current domain*; method select legal only if current domain is authorized for selected method
- Security mediates inter-object access across domains, e.g. objects can be made *truly private*

pop examples

Assume the following definition:

$$c \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, d' \mapsto \{\text{read}\}\}$$

Let d' be the current domain:

- $c.\text{write}(e)$ will *fail*, $c.\text{read}()$ will succeed

Let d be the current domain:

- $c.\text{write}(e)$ will succeed, $c.\text{read}()$ will succeed

Let $d'' \neq d, d'$ be the current domain:

- c is unusable

Enforced by *dynamic* checks!

Types for pop

As is the case for λ_{sec} , pop benefits from a type analysis:

- Obtained by transformation into pml_B

NB: same target language for distinct security models.

The pop-to-pml_B transformation (highlights)

The transformation of interfaces φ , denoted $\hat{\varphi}$, is straightforward using records:

$$\{d_1 \mapsto \widehat{\iota_1}, \dots, d_n \mapsto \iota_n\} = \{d_1 = \iota_1, \dots, d_n = \iota_n\}$$

Objects are also transformed as records:

$$\begin{aligned} \llbracket [m_1(x) = e_1, \dots, m_n(x) = e_n] \cdot d \cdot \varphi \rrbracket_{d'} \\ = \\ \{\text{obj} = \{m_1 = \lambda x. \llbracket e_1 \rrbracket_d, \dots, m_n = \lambda x. \llbracket e_n \rrbracket_d\}, \text{ifc} = \hat{\varphi}\} \end{aligned}$$

Method selects are encoded so that access rights are always verified:

$$\begin{aligned} \llbracket e_1.m(e_2) \rrbracket_d = & \text{let } c_1 = \llbracket e_1 \rrbracket_d \text{ in} \\ & c_1.\text{ifc}.d \ni m; \\ & (c_1.\text{obj}.m)(\llbracket e_2 \rrbracket_d) \end{aligned}$$

Types for pop

Type systems for pop developed on the basis of transformation into pml_B , in the same manner as the λ_{sec} type system:

- Sound indirect type system immediately obtained as composition of pop-to- pml_B transformation and pml_B type system
- Sound direct system developed on foundation of pml_B type system
- Type safety for pop easily obtained as in λ_{sec} case:
 - Type safety implies static enforcement of access control in pop

Direct pop types

We define direct type terms specifically adapted for pop, with object types of the form $[\tau_1] \cdot \{\tau_2\}$:

- τ_1 the types of methods
- τ_2 the type of the interface
- direct pop types have an *interpretation* as (are syntactic sugar for) pml_B types

$o \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, d' \mapsto \{\text{read}\}\}$

$o : [\text{read} : \text{unit} \rightarrow \tau, \text{write} : \tau \rightarrow \text{unit}] \cdot \{d : \{\text{read}, \text{write}\}, d' : \{\text{read}\}\}$

$o.\text{write}(v) : \text{unit}$ if d is current (static) domain

$o.\text{write}(v)$ *not well-typed* otherwise

Direct pop types

Direct pop types offer same benefits as direct λ_{sec} types:

- Statically verified program *safety*
- Significant run-time *optimizations*
- Enhanced *readability* of security policies

Conclusion

- Programming language-based security a useful approach for securing systems and applications:
 - Give programmers greater control and flexibility for specifying and enforcing security
 - Allow dynamic *context* to be considered in access-control decisions
- Type systems enhance programming language-based security:
 - Improve efficiency of implementation
 - Improve understanding of security policies

Conclusion

- Methodologies enhance the cycle of type system development:
 - Transformations allow novel type systems to be developed on the basis of existing ones
 - $\text{HM}(X)$ is a polymorphic type constraint framework for functional core; *extensible*
- Future work (with Scott Smith): a generalized understanding of context-based security in type-safe programming languages

<http://www.cs.uvm.edu/~skalka>

λ_{sec} evaluation examples

$$\mathcal{A} = \{\dots, p \mapsto \{r\}, \dots\}$$

$$id \triangleq \lambda x.p.x$$

$$check_r \triangleq \lambda x.p.\text{check } r \text{ then } x$$

$$enable_r \triangleq p.\text{enable } r \text{ in } check_r(id)$$

$$-, check_r(id) \rightarrow \boxed{p, \emptyset}, \text{check } r \text{ then } id \rightarrow \text{fail}$$

$$-, enable_r \rightarrow \boxed{p, \{r\}}, check_r(id) \rightarrow$$

$$\frac{\boxed{p, \{r\}}}{\boxed{p, \emptyset}}, \text{check } r \text{ then } id \rightarrow \frac{\boxed{p, \{r\}}}{\boxed{p, \emptyset}}, id \text{ succeeds}$$

Capability hardening: weak capabilities

Capability *weakening* is a kind of deep casting mechanism— a security abstraction that preserves important properties (e.g. confinement):

- in EROS, if a capability is *weak* read-only, then it is read-only, *and* any capabilities it returns are weak read-only

We generalize weakening in `pop`— where τ is a set of method names and c is a capability definition, $\text{weak}_{\tau}(c)$ is a weakening of c :

- No methods in τ may be selected from $\text{weak}_{\tau}(c)$
- Suppose $m \notin \tau$; if c' is returned by $c.m(e)$ then $\text{weak}_{\tau}(c).m(e)$ returns $\text{weak}_{\tau}(c')$

pop examples

Assume the following definition:

$$c \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, d' \mapsto \{\text{read}\}\}$$

Suppose $c.\text{read}()$ returns $c' = c$, and suppose d is the current domain:

- $(\text{weak}_{\{\text{write}\}}(c)).\text{read}(e)$ returns $\text{weak}_{\{\text{write}\}}(c')$
- $(\text{weak}_{\{\text{write}\}}(c')).\text{write}()$ *fails*

Enforced by *dynamic* checks!

\mathcal{S}_1^- -derived type deduction rules (highlights)

Where φ ranges over $\{+, -\}$:

$$\text{APP} \quad \frac{p, \zeta, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\zeta} \tau \quad p, \zeta, \Gamma \vdash e_2 : \tau_2}{p, \zeta, \Gamma \vdash e_1 e_2 : \tau}$$

$$\text{ENABLE SUCCESS} \quad \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau \quad r \in \mathcal{A}(p)}{p, \{r\varphi, \rho\}, \Gamma \vdash \text{enable } r \text{ in } e : \tau}$$

$$\text{CHECK} \quad \frac{p, \{r+, \rho\}, \Gamma \vdash e : \tau}{p, \{r+, \rho\}, \Gamma \vdash \text{check } r \text{ then } e : \tau}$$

$$\text{SIGN} \quad \frac{p, \{r_1\varphi_1, \dots, r_n\varphi_n, \emptyset\}, \Gamma \vdash e : \tau \quad \mathcal{A}(p) = \{r_1, \dots, r_n\}}{\star, \{r_1\varphi_1, \dots, r_n\varphi_n, \rho\}, \Gamma \vdash p.e : \tau}$$