

A Systematic Approach to Static Access Control

François Pottier (INRIA),
Christian Skalka and Scott Smith (JHU)

Motivations

- 1 To develop types as tools for *statically* analyzing general purpose PL's with *security features*:
 - Types provide declarative security mechanism
 - Types can improve efficiency, eagerly detect security errors
- 2 To present an instance of a general *transformational* approach to type system development:
 - New type systems systematically developed from established ones
 - Instance improves upon a previous system developed by Skalka and Smith

Types for PL Security

“Static Enforcement of Security with Types”, Skalka & Smith ICFP00

Java JDK1.2 provides a language-based 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 ACL's
- Privileges may be explicitly enabled and checked dynamically by *Stack Inspection*

Stack Inspection

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

- $\text{enablepriv}(r)$ places privilege r on current stack frame if stack frame owner is authorized for r
- $\text{checkpriv}(r)$ initiates stack inspection; 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:

(* owned by System *)

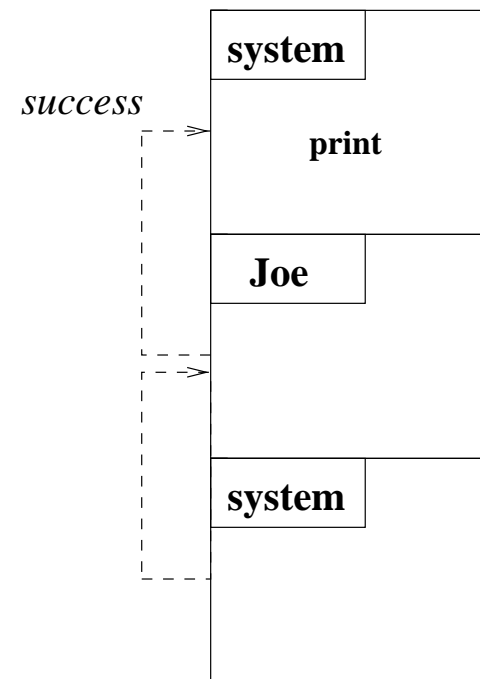
$enablePrint(g, x) = \{enablepriv(\mathbf{print}); g(x)\}$

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

(* owned by Joe *)

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

$enablePrint(joeProg, file)$



Stack inspection example

Assuming Joe is *not* authorized for print privilege:

(* owned by System *)

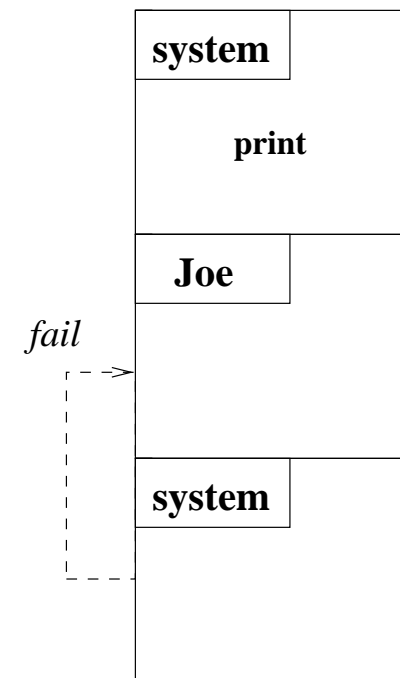
$enablePrint(g, x) = \{enablepriv(\mathbf{print}); g(x)\}$

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

(* owned by Joe *)

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

$enablePrint(joeProg, file)$



Static Stack Inspection

Our contribution: In a simplified language, types can be used to statically enforce the stack inspection model:

- Types provide declarative security mechanism
- Static approach improves runtime efficiency by eliminating need for dynamic checks

However: Language must be extended, monomorphic types, constraint-based types hard to read.

Transformational Approach

A general approach developed in:

“Information Flow Inference for Free”, Pottier & Conchon ICFP00

We describe an instance of the approach: expressions e in a “stack inspection” language λ_{sec} are typed via transformation $\llbracket e \rrbracket$:

- $\llbracket e \rrbracket$ is a term in a *target language* with a sufficiently rich type system
- Transformation is *correct*— program semantics are preserved
- Types for source expressions e obtained from target type system; if $\llbracket e \rrbracket : \tau$ then $e : \tau$

Source language: λ_{sec}

The language is a lambda calculus extended with constructs for modeling stack inspection.

Access control lists: $\mathcal{A} = \{p_0 \mapsto \emptyset, p \mapsto \{r\}\}$

Signed functions: $\lambda x.p.e$

Privilege management: $\text{checkpriv } r \text{ for } e$
 $\text{enablepriv } r \text{ in } e$

A formalization of stack inspection and an operational semantics faithfully model the Java system:

$\dots \text{enablepriv } r \text{ in } ((\lambda x.p.\text{checkpriv } r \text{ for } x)(1)) \rightarrow^* 1$

$(\lambda x.p.\text{checkpriv } r \text{ for } x)(1)$ *goes wrong!*

Target language: λ_{set}

We use Wallach's *security-passing-style (SPS)* conversion:

- Security information is *passed down the stack* as an explicit function parameter, not maintained in stack annotations
- “Stack inspection” becomes a trivial set membership test

The target language λ_{set} contains sets and set operations: $R = \{r_1, \dots, r_n\}$, expressions $R.r$ which reduce only if $r \in R$, union \vee and intersection \wedge .

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

Our transformation is defined (in part) as follows:

$$\begin{aligned} \llbracket \lambda x. f \rrbracket_p &= \lambda x. \lambda s. \llbracket f \rrbracket \\ \llbracket e_1 e_2 \rrbracket_p &= \llbracket e_1 \rrbracket_p \llbracket e_2 \rrbracket_p s \\ \llbracket \text{enablepriv } r \text{ in } e \rrbracket_p &= \text{let } s = s \vee (\{r\} \cap \mathcal{A}(p)) \text{ in } \llbracket e \rrbracket_p \\ \llbracket \text{checkpriv } r \text{ for } e \rrbracket_p &= s.r; \llbracket e \rrbracket_p \\ \llbracket p.e \rrbracket &= \text{let } s = s \wedge \mathcal{A}(p) \text{ in } \llbracket e \rrbracket_p \end{aligned}$$

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

Correctness of transformation

Transformational correctness is stated as follows:

Theorem: If $e \rightarrow^* v$, then $\langle e \rangle \rightarrow^* \langle v \rangle$. If e goes wrong, then $\langle e \rangle$ goes wrong. If e diverges, then $\langle e \rangle$ diverges.

This straightforward *syntactic* result allows us to apply type safety and inference in λ_{set} to λ_{sec} .

Typing the target language

We want a sound type system that will ultimately be applied to λ_{SEC} :

- Types should track contents of sets
- Primitive operations should be typed *accurately*
- Types should be readable; unification-based inference

A problem: no existing satisfactory type system! But we don't have to start from scratch...

The HM(X) system

“Type Inference with Constrained Types”. Odersky, Sulzmann and Wehr 1999

HM(X) is a generalized *type framework*:

- Let-polymorphism for core λ -calculus is provided
- System is specialized by instantiating X with a *constraint system*:
 - Type grammars may be extended
 - Initial typing environment defined for new constants
 - Interpretation of a subsumption (\leq) rule defined
- Type inference provided, modulo constraint solution algorithm

The HM(X) system

The framework contains expressions, functions types, type schemes and basic constraints:

$$\begin{array}{ll} e ::= x \mid \lambda x.e \mid e e \mid \text{let } x = e \text{ in } e & \textit{expressions} \\ \tau ::= \alpha, \beta, \dots \mid \tau \rightarrow \tau & \textit{types} \\ \sigma ::= \forall \bar{\alpha}[C].\tau & \textit{type schemes} \\ C ::= C \wedge C \mid \tau = \tau \mid \tau \leq \tau & \textit{constraints} \end{array}$$

Type judgements are of the form $C, \Gamma \vdash e : \sigma$.

- For simplicity in this presentation, we interpret \leq as *equality* and constraint solution as *unification*.

Set types: SETS

We use specialized *row types*, called *set types*, to accurately type sets in λ_{set} :

- The fields in set types are always assigned the constructor **Pre** or **Abs**, denoting which elements of a set are present or absent

$$\{r_1, \dots, r_n\} \quad : \quad \{r_1 : \mathbf{Pre}; \dots ; r_n : \mathbf{Pre}; \partial \mathbf{Abs}\}$$

- Sets and set types can be viewed as shorthand for an encoding of sets as records:

$$[r_1 = \mathbf{true}, \dots, r_n = \mathbf{true}] \quad : \quad \{r_1 : \mathbf{Pre}(bool); \dots ; r_n : \mathbf{Pre}(bool); \partial \mathbf{Abs}\}$$

SETS is the constraint system that uses equality set type constraints.

Typing the set operations

To type the set operations in λ_{set} , we define an initial typing environment Γ_1 with bindings for the language constants:

$$\begin{aligned} .r & : \forall \beta. \{r : \mathbf{Pre}; \beta\} \rightarrow \text{unit} \\ \bigvee_{\{r_1, \dots, r_n\}} & : \forall \beta \bar{\gamma}. \{r_1 : \gamma_1; \dots; r_n : \gamma_n; \beta\} \rightarrow \{r_1 : \mathbf{Pre}; \dots; r_n : \mathbf{Pre}; \beta\} \\ & \vdots \end{aligned}$$

Note that $.r$'s type (for example) is sound, since the expression $R.r$ is defined and reduces to $()$ only if $r \in R$.

λ_{set} to λ_{sec} type safety

\mathcal{S} is the type system HM(SETS) with Γ_1 as an initial environment.

Theorem: If e is a λ_{set} expression and $C, \Gamma \vdash e : \sigma$ is a valid judgment in \mathcal{S} , then e does not go wrong.

Definition: Let e be in λ_{sec} ; then $C, \Gamma \vdash e : \sigma$ iff $C, \Gamma \vdash (e) : \sigma$.

Theorem: If e is a λ_{sec} expression and $C, \Gamma \vdash e : \sigma$ is a valid judgment in \mathcal{S} , then e does not go wrong.

Direct types for λ_{sec}

For a realistic *implementation* of types for λ_{sec} , we want to type the language *directly*:

- Transformation may introduce inefficiencies at compile and/or runtime
- Type error reporting difficult using indirect approach

Direct type judgments are of the form $p, \zeta, \Gamma \vdash e : \sigma$, and are derived from the types given by the transformation.

Direct type judgements (highlights)

The CHECKPRIV rule is easily derived from the binding for $.r$:

$$\text{CHECKPRIV} \frac{p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash e : \tau}{p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash \text{checkpriv } r \text{ for } e : \tau}$$

Function types in the transformation are all of the form $\tau \rightarrow \zeta \rightarrow \tau'$, which we abbreviate $\tau \xrightarrow{\zeta} \tau'$ allowing this ABS rule:

$$\text{ABS} \frac{\star, \zeta_2, (\Gamma; x : \tau_1) \vdash f : \tau_2}{p, \zeta_1, \Gamma \vdash \lambda x. f : \tau_1 \xrightarrow{\zeta_2} \tau_2}$$

Direct λ_{sec} type examples

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

$$check_r = \lambda x.p.\text{checkpriv } r \text{ for } x$$

$$wrapper_r = \lambda f.p.\lambda x.p.\text{enablepriv } r \text{ in } f x$$

$$id : \forall \alpha \beta \gamma. \alpha \xrightarrow{\{r:\gamma; \beta\}} \alpha$$

$$check_r : \forall \alpha \beta. \alpha \xrightarrow{\{r:\mathbf{Pre}; \beta\}} \alpha$$

$$wrapper_r : \forall \dots . (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}; s:\gamma_1; \partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\gamma_2; s:\gamma_1; \beta_2\}} \alpha_2)$$

λ_{sec} type inference

The same approach may be used to provide *unification-based type inference* for λ_{set} and λ_{sec} types:

- General row unification algorithm by Didier Rémy *
- HM(X) provides type inference framework (constraint inference with constraint solver unspecified)
- To infer type of e :
 - compose transformation $(\lfloor e \rfloor)$ with HM(X) inference instantiated with row type unification

Direct type inference for λ_{sec} can easily be derived from this algorithm

* *in Theoretical Aspects Of Object-Oriented Programming. MIT Press, 1993*

Related work

- *Type & effects systems*
 - A security annotation ζ on a function type $f : \tau \xrightarrow{\zeta} \tau'$ can be viewed as the “security effect” of f
- *Monads*
 - Effects systems can be transformed into monadic representation
 - Our approach similar, but provides a sufficiently rich type system for application to Stack Inspection

Conclusion

- Types can be used for PL-based security:
 - Efficient, static Stack Inspection for access control
 - Declarative mechanism for expressing policies
- Transformational approach allows re-use of existing row type system for application to static Stack Inspection:
 - Approach conserves proof effort
 - Row type unification yields inference of readable, expressive types
 - Provides insight into structure and static analysis of λ_{sec}

For more information: <http://www.cs.jhu.edu/~ces/work.html>