Secure Information Flow Analysis for a Distributed OO Language

Martin Pettai
University of Tartu / Cybernetica AS

October 8, 2011

Introduction

- We analyze a language with objects, asynchronous method calls and futures
- We use an information-flow type system to prevent insecure flows in the programs written in this language
- Synchronization creates additional flows
- We consider both direct and indirect flows and also flows through non-termination

The language

- A simplified version of the concurrent object level of Core ABS
- No synchronous method calls
- No boolean guards
- No interfaces

Syntax (1)

$Pr ::= \overline{Cl}\ B$		program
$Cl ::= \text{class}\ C\{\overline{T\,f}\ \overline{M}\}$		class definition

Syntax (2)

$x \mid n \mid o \mid b \mid f$	local variable | task | object | cog | field name
$M ::= (m : (l, \overline{T}) \xrightarrow{l[,i]} \mathrm{Cmd}^l(T))(\overline{x})\ B$	method definition
$B ::= \{\overline{T\,x}\ s; x\}$	method body
$v ::= x \mid \mathrm{this} \mid \mathrm{this}.f$	variable
$i ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	integer
$e ::= e_p \mid e_s$	expression
$e_p ::= v \mid \mathrm{null} \mid i \mid e_p = e_p$	pure expression
$e_s ::= e_p!_l m(\overline{e_p}) \mid e_p.\mathrm{get}_l \mid \mathrm{new}\ C \mid \mathrm{new\ cog}\ C$	expression with side effects
$s ::= v := e \mid e \mid \mathrm{skip} \mid \mathrm{suspend}_l \mid \mathrm{await}_l\ g$	statement
$\quad \mid \mathrm{if}\ (e_p)\ s\ \mathrm{else}\ s \mid \mathrm{while}_l\ (e_p)\ s \mid s; s$	
$g ::= v?$	guard
$l ::= L \mid H$	security level
$\ell ::= l \mid i$	security level or integer
$T ::= \mathrm{Int}_l \mid C_l \mid \mathrm{Fut}_l^{\ell}(T) \mid \mathrm{Guard}_l^{\ell}$	security type

Operational semantics (1)

- The run-time configurations consist of cogs (b), objects (o), and tasks (n).

$$P ::= b[n_1, n_2] \mid o[b, C, \sigma] \mid n\,\langle b, o, \sigma, s\rangle \mid P \parallel P$$

- Creating new tasks, objects, cogs:

$$\frac{n' \text{ fresh} \qquad \text{body}(m) = s(\bar{x}); x' \qquad s_{task} = \text{grab}_l; s[\bar{a}/\bar{x}]; \text{release}_l; x'}{o'[b', C, \sigma'] \parallel n\,\langle b, o, \sigma, R_1[o'!_l m(\bar{a})]; s\rangle \rightsquigarrow\ \rightsquigarrow o'[b', C, \sigma'] \parallel n\,\langle b, o, \sigma, R_1[n']; s\rangle \parallel n'\,\langle b', o', \sigma_{init}, s_{task}\rangle} \text{ (acall)}$$

$$\frac{o' \text{ fresh}}{n\,\langle b, o, \sigma, R_1[\text{new } C]; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, R_1[o']; s\rangle \parallel o'[b, C, \sigma_{init}]} \text{ (new)}$$

$$\frac{b' \text{ fresh} \qquad o' \text{ fresh}}{n\,\langle b, o, \sigma, R_1[\text{new cog } C]; s\rangle \rightsquigarrow\ \rightsquigarrow n\,\langle b, o, \sigma, R_1[o']; s\rangle \parallel b'[\bot, \bot] \parallel o'[b', C, \sigma_{init}]} \text{ (newcog)}$$

Operational semantics (2)

- Synchronization:

$$\frac{}{n\,\langle b, o, \sigma, \mathrm{suspend}_l; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, \mathrm{release}_l; \mathrm{grab}_l; s\rangle}\ (\mathsf{suspend})$$

$$\frac{}{b[\bot, \bot] \parallel n\,\langle b, o, \sigma, \mathrm{grab}_L; s\rangle \rightsquigarrow b[n, n] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{grab_L})$$

$$\frac{}{b[n', \bot] \parallel n\,\langle b, o, \sigma, \mathrm{grab}_H; s\rangle \rightsquigarrow b[n', n] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{grab_H})$$

$$\frac{}{b[n, n] \parallel n\,\langle b, o, \sigma, \mathrm{release}_L; s\rangle \rightsquigarrow b[\bot, \bot] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{release_L})$$

$$\frac{}{b[n', n] \parallel n\,\langle b, o, \sigma, \mathrm{release}_H; s\rangle \rightsquigarrow b[n', \bot] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{release_H})$$

$$\frac{}{n\,\langle b, o, \sigma', \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \rightsquigarrow n\,\langle b, o, \sigma', s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle}$$

$$\frac{}{\begin{array}{c} n\,\langle b, o, \sigma', \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma', \mathrm{suspend}_l; \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \end{array}}\ (\mathsf{await_2})$$

Locks

- Every cog has a high and a low lock
- A task can execute only when it has the high lock
- A task can change the low (publicly visible) part of the state only when it also has the low lock (this is checked statically by the type system)

Security types

- The types in the type system are the following:

$$T ::= \mathrm{Int}_l \mid C_l \mid \mathrm{Fut}_l^{\ell}(T) \mid \mathrm{Guard}_l^{\ell} \mid \mathrm{Exp}^l(T) \mid \mathrm{Cmd}^l \mid$$

$$\mid \mathrm{Cmd}^l(T) \mid (l, \overline{T}) \overset{l[,i]}{\to} \mathrm{Cmd}^l(T)$$

$$l ::= L \mid H$$

- The possible types of futures are $\mathrm{Fut}_L^L(T)$ (corresponding to a low task), $\mathrm{Fut}_H^L(T)$ (high-low task), and $\mathrm{Fut}_H^H(T)$ (high-high task)
- Both low and high tasks can await for high-low tasks
- Only low tasks can await for low tasks
- Only high-high tasks can await for high-high tasks

Subtyping rules

$$l \leq l \qquad L \leq H \qquad \mathrm{Guard}^{i}_{H} \leq \mathrm{Guard}^{L}_{H}$$

$$\frac{l_2 \leq l_1 \qquad \ell_3 \leq \ell_4}{\mathrm{Guard}^{\ell_3}_{l_1} \leq \mathrm{Guard}^{\ell_4}_{l_2}} \qquad \frac{l_2 \leq l_1 \qquad \ell_3 \leq \ell_4 \qquad T_5 \leq T_6}{\mathrm{Fut}^{\ell_3}_{l_1}(T_5) \leq \mathrm{Fut}^{\ell_4}_{l_2}(T_6)}$$

$$\frac{l_1 \leq l_2}{C_{l_1} \leq C_{l_2}} \qquad \frac{l_1 \leq l_2}{\mathrm{Int}_{l_1} \leq \mathrm{Int}_{l_2}} \qquad \frac{\gamma, l \vdash e : T}{\gamma, l \vdash e : \mathrm{Exp}^{L}(T)}$$

$$\frac{\gamma, l \vdash e : T_1 \qquad T_1 \leq T_2}{\gamma, l \vdash e : T_2} \qquad \frac{\gamma, l \vdash s : \mathrm{Cmd}^{l_1} \qquad l_1 \leq l_2}{\gamma, l \vdash s : \mathrm{Cmd}^{l_2}}$$

$$\frac{\gamma, l_1 \vdash s : \mathrm{Cmd}^{l} \qquad l_1 \geq l_2}{\gamma, l_2 \vdash s : \mathrm{Cmd}^{l}}$$

Some type rules

$$\frac{\gamma, l \vdash e : C_{l_0} \qquad \gamma, l \vdash \overline{e} : \overline{T} \qquad \gamma(C.m) = l_0, \overline{T} \xrightarrow{l} \mathrm{Cmd}^{l_1}(T_2) \qquad l_0 \geq l \qquad \overline{T} \geq l \qquad l_1 = l}{\gamma, l \vdash e!_l m(\overline{e}) : \mathrm{Fut}_l^{l_1}(l \vee l_1 \vee T_2)} \;(\mathsf{ACall}_1)$$

$$\frac{\gamma, l \vdash e : \mathrm{Guard}_l^{l_1}}{\gamma, l \vdash \mathrm{await}_l(e) : \mathrm{Cmd}^{l_1}} \;(\mathsf{Await}_1)$$

$$\frac{\gamma, l \vdash e : \mathrm{Int}_l \qquad \gamma, l \vdash s : \mathrm{Cmd}^l}{\gamma, l \vdash \mathrm{while}_l\ (e)\ s : \mathrm{Cmd}^l} \;(\mathsf{While})$$

Low-equivalence

$$\frac{\gamma, l \vdash s : \mathrm{Cmd}^H}{s \sim_\gamma s} \qquad \frac{\gamma, H \vdash s : \mathrm{Cmd}^H \qquad \gamma, H \vdash s' : \mathrm{Cmd}^H}{s \sim_\gamma s'}$$

$$\frac{\gamma, l \vdash s : \mathrm{Cmd}^H(T)}{s \sim_\gamma s} \qquad \frac{\gamma, H \vdash s : \mathrm{Cmd}^H(T) \qquad \gamma, H \vdash s' : \mathrm{Cmd}^H(T)}{s \sim_\gamma s'}$$

$$\frac{\gamma, H \vdash s_1 : \mathrm{Cmd}^H \qquad s_2 \sim_\gamma s_2'}{s_1; s_2 \sim_\gamma s_2'} \qquad \frac{\gamma, H \vdash s_1 : \mathrm{Cmd}^H \qquad s_2 \sim_\gamma s_2'}{s_2 \sim_\gamma s_1; s_2'} \qquad \frac{s_2 \sim_\gamma s_2'}{s_1; s_2 \sim_\gamma s_1; s_2'}$$

$$\sigma \sim_\gamma \sigma' \equiv dom(\sigma) = dom(\sigma') \wedge \forall v \in dom(\sigma).\ level(\gamma(v)) = L \Rightarrow \sigma(v) = \sigma'(v)$$

$$b[n_1, n_2] \sim_\gamma b[n_1, n_2']$$

$$\frac{\sigma \sim_\gamma \sigma'}{o[b, C, \sigma] \sim_\gamma o[b, C, \sigma']} \qquad \frac{\sigma \sim_\gamma \sigma' \qquad s \sim_\gamma s'}{n\,\langle b, o, \sigma, s\rangle \sim_\gamma n\,\langle b, o, \sigma', s'\rangle} \qquad \frac{P_1 \sim_\gamma P_1' \qquad P_2 \sim_\gamma P_2'}{P_1 \parallel P_2 \sim_\gamma P_1' \parallel P_2'}$$

$$\frac{\gamma, H \vdash s : \mathrm{Cmd}^{l_1}(T_2) \qquad P \sim_\gamma P'}{n\,\langle b, o, \sigma, s\rangle \parallel P \sim_\gamma P'} \qquad \frac{\gamma, H \vdash s : \mathrm{Cmd}^{l_1}(T_2) \qquad P \sim_\gamma P'}{P \sim_\gamma n\,\langle b, o, \sigma, s\rangle \parallel P'}$$

High and low steps and locks

- A high step cannot change the low-equivalence class of a configuration, a low step may change it
- Each cog has two locks for synchronization of its tasks
  - The high lock is needed to make a high step
  - Both locks are needed to make a low step
  - Suspending in high context releases only the high lock

Insecure information flows

- Within one task, there can be direct flows, indirect flows, and flows through non-termination
  - Security of these flows is easily enforced by the type system

$$\frac{\gamma, l \vdash s_1 : \mathrm{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathrm{Cmd}^{l_2}}{\gamma, l \vdash s_1; s_2 : \mathrm{Cmd}^{l_1 \vee l_2}} \ (\mathsf{Seq}_1)$$

$$\frac{\gamma, l \vdash s_1 : \mathrm{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathrm{Cmd}^{l_2}(T)}{\gamma, l \vdash s_1; s_2 : \mathrm{Cmd}^{l_1 \vee l_2}(T)} \ (\mathsf{Seq}_2)$$

- Synchronization between tasks introduces additional flows

Flows through synchronization (1)

- An example
  - A high task n_1 in cog b_1 makes a high while loop (e.g. $\mathrm{while}\ h\ \mathrm{do\ skip}$) whose termination depends on secret data
  - A low task n_2 in cog b_1 is about to make a low side effect (e.g. call a method in cog b_2 that does $\mathrm{l} := 0$)
  - The low side effect can be blocked by a non-terminating high loop
- To prevent this, while and await loops suspend after each iteration

$$\frac{}{\begin{array}{c} n\,\langle b, o, \sigma, \mathrm{while}_l\ (e)\ s_1; s_2\rangle \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma, \mathrm{if}\ (e)\ (s_1; \mathrm{suspend}_l; \mathrm{while}_l\ (e)\ s_1)\ \mathrm{else\ skip}; s_2\rangle \end{array}}\ (\text{while})$$

$$\frac{}{\begin{array}{c} n\,\langle b, o, \sigma', \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma', s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \end{array}}\ (\text{await}_1)$$

$$\frac{}{\begin{array}{c} n\,\langle b, o, \sigma', \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma', \mathrm{suspend}_l; \mathrm{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \end{array}}\ (\text{await}_2)$$

Flows through synchronization (2)

- For a high-low task n_4, non-termination must not be allowed, as it can leak secret information to any low task awaiting for n_4
- It is not enough to disallow loops, infinite recursion must also be prevented

$$\frac{\gamma, l, i \vdash e : \mathrm{Guard}_l^{i_1} \qquad i_1 < i}{\gamma, l, i \vdash \mathrm{await}_l(e) : \mathrm{Cmd}^L} \ (\mathsf{Await}_2)$$

Flows through synchronization (3)

- An example
  - Low task n_1 in cog b_1 is in high context and awaits for a high-low task n_2 in cog b_2
  - The high lock of b_2 is held by a low task n_3 in cog b_2
  - Here it may depend on the high variables in n_1 whether low steps must be made in n_3 before the next low step in n_1 or not
- The following rule removes this dependency

$$\frac{\text{the next step of } s_1 \text{ is low and the task } n' \text{ is high-low}}{\begin{array}{c} n\,\langle b, o, \sigma', \mathrm{await}_H(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, \mathrm{grab}_H; s'; x\rangle \parallel \\ \parallel n_1\,\langle b', o_1, \sigma_1, s_1\rangle \parallel b'[n_1, n_1] \rightsquigarrow n\,\langle b, o, \sigma', \mathrm{suspend}_H; \mathrm{await}_H(n'?); s\rangle \parallel \\ \parallel n'\,\langle b', o', \sigma, s'; x\rangle \parallel n_1\,\langle b', o_1, \sigma_1, \mathrm{grab}_H; s_1\rangle \parallel b'[n_1, n'] \end{array}} \;(\text{await}_3$$

Non-interference

- We have proved concurrent non-interference

Definition (Non-interference)

A program $\overline{Cl}\ \{\overline{T\ x}\ s; x_0\}$ is *non-interferent* if for any three states σ_0, $\sigma_0^\bullet$ and σ_1 satisfying $\sigma_0 \sim_{\overline{x:T}} \sigma_1$,

$$b_0[n_0, n_0] \,\|\, n_0 \,\langle b_0, null, \sigma_0, s; release_L; x_0\rangle \overset{*}{\rightsquigarrow} n_0 \,\langle b_0, null, \sigma_0^\bullet, x_0\rangle \,\|\, \ldots$$

implies that there exists a state $\sigma_1^\bullet$ with $\sigma_1^\bullet(x_0) = \sigma_0^\bullet(x_0)$ and

$$b_0[n_0, n_0] \,\|\, n_0 \,\langle b_0, null, \sigma_1, s; release_L; x_0\rangle \overset{*}{\rightsquigarrow} n_0 \,\langle b_0, null, \sigma_1^\bullet, x_0\rangle \,\|\, \ldots \quad .$$

Theorem (Subject reduction)

If P_1 and P_2 are well typed under γ and $P_1 \sim_\gamma P_2$ then if $P_1 \rightsquigarrow P_1'$ then there exists P_2' such that $P_2 \rightsquigarrow^* P_2'$ and $P_1' \sim_\gamma P_2'$.

Conclusion

- We have demonstrated a type-based information flow analysis for a language with several features
- We saw that synchronization between tasks can create some interesting flows
- We have a non-interference proof

The End