Securing the Future — An Information Flow Analysis of a Distributed OO Language

Martin Pettai
joint work with Peeter Laud
University of Tartu / Cybernetica AS

January 24, 2012

Introduction

- We analyze a concurrent language with objects, asynchronous method calls and futures
- We want to guarantee that there are no insecure flows in the programs written in this language
  - The set of variables is partitioned into low (public) and high (private)
  - Insecure flow occurs when the initial value of a high variable may influence the final value of a low variable
  - We consider both direct and indirect flows and also flows through non-termination
  - Flows can also occur through synchronization
- We use an information-flow type system to prevent insecure flows in the programs written in this language
- Some leaks are prevented using operational semantics and some using the type system

The language

- A simplified version of the concurrent object level of Core ABS (Abstract Behavioural Specification)

Syntax (1)

- In the following, $\overline{X}$ denotes a sequence of X-s

$$Pr ::= \overline{Cl}\ B \qquad \text{program}$$

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

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 concurrent object groups (b), objects (o), and tasks (n).

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

- The operational semantics of some statements ($\mathrm{suspend}_l$, await_l, $e_p.\mathrm{get}_l$, $\mathrm{while}_l\ (e_p)\ s$, $e_p!_l m(\overline{e_p})$) depends on the security context (high or low)
- We distinguish high and low tasks
  - High tasks execute entirely in high context
  - Low tasks begin and end in low context but can temporarily switch into high context
- We also distinguish high and low steps
  - Low steps are the reduction of the statements with $_L$ annotation, assignment to low variables, and new and $\mathrm{new\ cog}$ statements
    - Low side effects can occur only here
  - High steps reduce any other statement

Operational semantics (2): scheduling

- The next step is chosen nondeterministically: any task (in any cog) that can make step
- Different cogs are running in parallel
  - Similar to the scheduling of parallel threads where the scheduler can switch to a different thread after each step
- Tasks inside one cog use cooperative scheduling
  - Tasks use explicit suspend statements to create scheduling points
  - Two locks (low lock and high lock) are used in each cog
  - The high lock is needed to make any step
  - The number of suspends executed may depend on high variables
  - We distinguish high and low suspends and forbid low steps during high suspends
  - The low lock is needed to make a low step

Operational semantics (3): scheduling

- Suspending in low context releases both locks
- Suspending in high context releases the high lock

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

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

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

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

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

Operational semantics (4): creating

- 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 (5): loops

- An example of an insecure flow
  - 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

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

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

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

Operational semantics (6): high-low tasks

- High-low tasks are high tasks that are guaranteed to terminate (other high tasks are called high-high tasks)
- An example of an insecure flow (an indirect flow between different tasks)
  - Low task n in cog b is in high context and awaits for a high-low task n_2 in cog b'
  - The high lock of b' is held by a low task n_3 in cog b'
  - Here it may depend on the high variables in n whether low steps must be made in n_3 before the next low step in n or not
- The following rule removes this dependency

$$\frac{\text{the next step of } s_1 \text{ is low and the task } n_2 \text{ is high-low}}{\begin{array}{c} n\,\langle b, o, \sigma', \mathrm{await}_H(n_2?); s\rangle \parallel n_2\,\langle b', o', \sigma, \mathrm{grab}_H; s'; x\rangle \parallel \\ \parallel n_3\,\langle b', o_1, \sigma_1, s_1\rangle \parallel b'[n_3, n_3] \rightsquigarrow n\,\langle b, o, \sigma', \mathrm{suspend}_H; \mathrm{await}_H(n_2?); s\rangle \parallel \\ \parallel n_2\,\langle b', o', \sigma, s'; x\rangle \parallel n_3\,\langle b', o_1, \sigma_1, \mathrm{grab}_H; s_1\rangle \parallel b'[n_3, n_2] \end{array}} \quad (\mathrm{await}_3$$

Security types

- The types in the type system are the following:

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

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

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

- 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

Insecure information flows (1)

- There can be direct flows, indirect flows, and flows through non-termination
  - Security of direct flows and indirect flows inside a single task is easily enforced by the type system

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

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

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

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

Insecure information flows (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

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

- The type of a high-low method has the form $(l, \overline{T}) \stackrel{H,i}{\rightarrow} \mathrm{Cmd}^L(T_1)$
- The corresponding future has type $\mathrm{Fut}_H^i(T_1)$

Low-equivalence

- We can define a low-equivalence relation $\sim_\gamma$ on configurations
- For $P_1 \sim_\gamma P_2$ to hold, P_1 and P_2 must have
  - sets of low tasks in one-to-one correspondence
  - sets of objects in one-to-one correspondence
  - sets of cogs in one-to-one correspondence
  - the same object identifier for corresponding tasks
  - the same cog identifier for corresponding objects
  - equal values of low local variables in corresponding low tasks
  - equal values of low fields in corresponding objects
  - low lock of each cog belonging to the corresponding tasks
  - equivalent statements in corresponding low tasks (modulo addition or removal of high statements)
- A high step cannot change the low-equivalence class of a configuration, a low step may change it

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 saw how these flows can be eliminated by the operational semantics and the type system
- We have a non-interference proof

The End