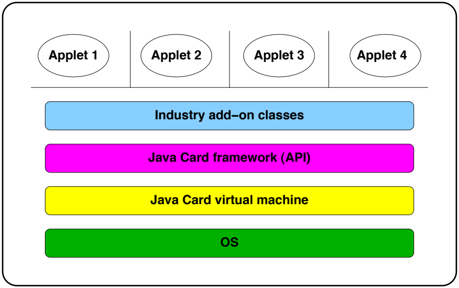
Language-based access control

February 1, 2007

Access control for the Java platform

- Codes from different trust levels execute within the same runtime.
  Java Cards cardlets, J2ME midlets, J2SE applets
- Security architectures use dynamic monitoring checks
  - Java Card firewall
  - J2ME interactive permissions
  - J2SE stack inspection



Local checks *vs* global security property

A study of the major Java security architectures:

- Analysis of Java Card firewall [TSI'04]
- Inference of security interfaces for stack-inspection [JFP'05]
- New security model for interactive devices [Esorics'06]

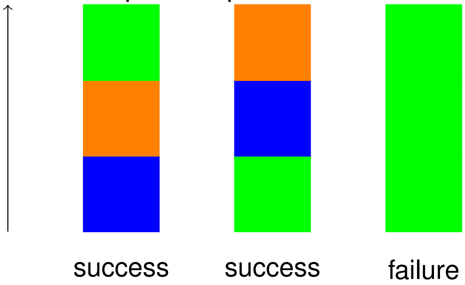
Are local checks sufficient to ensure a global security property?

Stack inspection

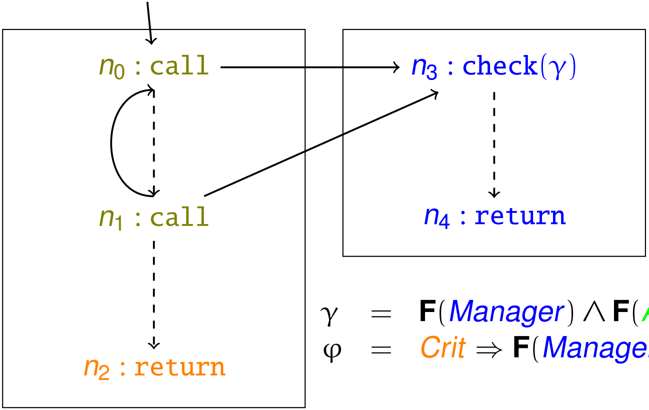
The stack inspection mechanism

Dynamic access control mechanism (Java, .NET)

- Security policy $\mathit{Code} \to \mathcal{P}(\mathit{attr})$
  origin, signature
- Stack inspection primitive ■ $\wedge$ ■



Control graph model for libraries



$$\gamma = \mathbf{F}(\mathit{Manager}) \wedge \mathbf{F}(\mathit{Accountant})$$

$$\varphi = \mathit{Crit} \Rightarrow \mathbf{F}(\mathit{Manager}) \wedge \mathbf{F}(\mathit{Accountant})$$

$$n_0, n_1 \mapsto \{\mathit{System}\}$$

$$n_2 \mapsto \{\mathit{System}, \mathit{Crit}\}$$

$$n_3, n_4 \mapsto \{\mathit{Manager}\}$$

Specification of secure contexts

- Secure call contexts

$$secure(s, n_0) = \forall(s' : Stack).s{:}n_0 \xrightarrow{[s]}{}^{*} s' \Rightarrow s' \vDash \varphi$$

- Call contexts that permit *node traversal*

$$transits(s, n) = \exists n', s{:}n \xrightarrow{[s]}{}^{+} s{:}n'$$

- Call contexts that permit *method returns*

$$returns(s, n) = \exists r, is(r) = return \wedge s{:}n \xrightarrow{[s]}{}^{*} s{:}r$$

Symbolic computation of secure contexts

- Constraint solving over a lattice of LTL formulae
- A weakest condition operator $\delta : LTL \to LTL$

$$s \vDash \delta_n(\phi) \iff s{:}n \vDash \phi$$

- Flavor of the constraints to solve
  - Traversal of check nodes

$$\frac{is(n) = check(\gamma)}{\tau_n \Leftarrow \delta_n(\gamma)}$$

  - Traversal of method calls

$$\frac{n \stackrel{inter}{\to} m}{\tau_n \Leftarrow \delta_n(\rho_m)}$$

  - Secure contexts for method calls

$$\frac{n \stackrel{inter}{\to} m}{\sigma_n \Rightarrow \delta_n(\sigma_m)}$$

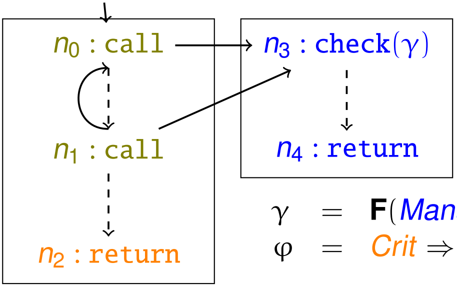
Inference of secure call contexts

Weakest precondition over the call stack

$$secure(s, n_0) = \forall(s' : Stack).s{:}n_0 \xrightarrow{[s]}{}^{*} s' \Rightarrow s' \vDash \varphi$$

is given by a LTL formulae

$$\sigma_{n_0} = \neg F(Accountant) \vee F(Manager)$$



$$\begin{aligned}\gamma &= \mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)\\ \varphi &= Crit \Rightarrow \mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)\end{aligned}$$

Stack inspection: a long-standing effort

- T. Jensen, D. Le Métayer and T. Thorn,
  Verification of control flow based security properties.
  In Proc. of the 20th IEEE Symp. on Security and Privacy, pages 89–103, IEEE Computer Society, 1999.
- F. Besson, T. Jensen, D. Le Métayer and T. Thorn.
  Model checking security properties of control flow graphs.
  Journal of Computer Security, 9:217–250, 2001.
- F. Besson, T. de Grenier de Latour and T. Jensen,
  Secure calling contexts for stack inspection.
  In Proc. of 4th Int Conf. on Principles and Practice of Declarative Programming, pages 76–87, ACM Press, 2002.
- F. Besson, T. de Grenier de Latour, and T. Jensen.
  Interfaces for stack inspection.
  Journal of Functional Programming, 15(2):179–217, 2005.

Access control for interactive devices

Current security model for interactive devices

Resources accesses is protected by permissions

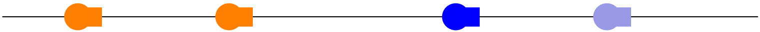
- Signed applications
  permissions granted forever
- unsigned applications
  permissions granted&consumed at resource access time

Drawback: a coarse-grained control of permissions

- Unsigned applications may flood the user with security screens
- Operators are reluctant to sign

Resource usage scenario (current model)

Inflexible usage of permissions



- permission is granted
- permission is consumed (resource access)

Resource usage scenario (enhanced model)

Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access

- Permissions are assigned quotas

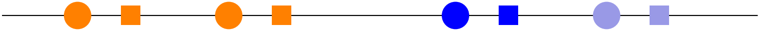
- Permissions denote sets of resources

- Permissions of different kinds are independent

Resource usage scenario (enhanced model)

Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access



- Permissions are assigned quotas

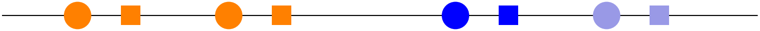
- Permissions denote sets of resources

- Permissions of different kinds are independent

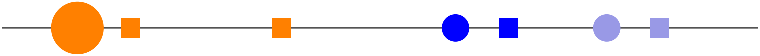
Resource usage scenario (enhanced model)

Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access



- Permissions are assigned quotas



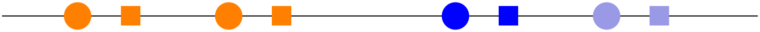
- Permissions denote sets of resources

- Permissions of different kinds are independent

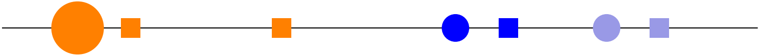
Resource usage scenario (enhanced model)

Towards a fined-grained control of permissions

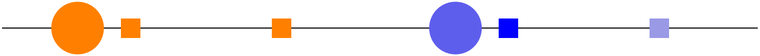
- Permissions are granted in advance before resource access



- Permissions are assigned quotas



- Permissions denote sets of resources

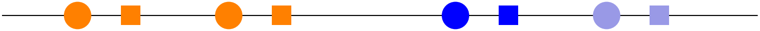


- Permissions of different kinds are independent

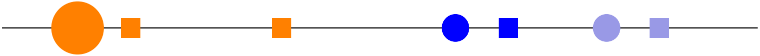
Resource usage scenario (enhanced model)

Towards a fined-grained control of permissions

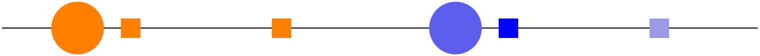
- Permissions are granted in advance before resource access



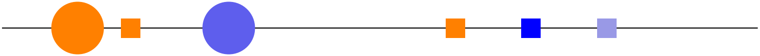
- Permissions are assigned quotas



- Permissions denote sets of resources

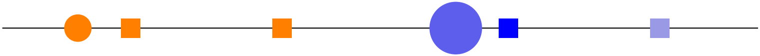


- Permissions of different kinds are independent



Enforcement of the enhanced model

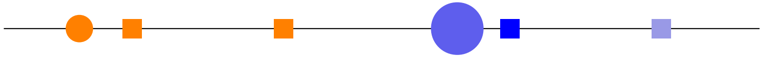
Programs will not use more permissions than they have been granted.



- Dynamic monitoring
  - runtime overhead
  - security exception
- Static enforcement
  - no runtime overhead
  - no security exception

Enforcement of the enhanced model

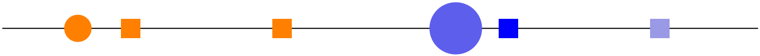
Programs will not use more permissions than they have been granted.



- Dynamic monitoring
  - runtime overhead
  - security exception
- Static enforcement
  - no runtime overhead
  - no security exception

Enforcement of the enhanced model

Programs will not use more permissions than they have been granted.



- Dynamic monitoring
  - runtime overhead
  - security exception
- Static enforcement
  - no runtime overhead
  - no security exception

Program as control-flow graphs

- A permission centric control-flow graph
  - Permission nodes:

$$\begin{array}{l}\texttt{grant} : \mathit{Kind} \times \mathcal{P}(\mathit{Permission}) \times \mathbb{N} \cup \infty \\ \texttt{consume} : \mathit{Permission}\end{array}$$

  - Control-flow nodes: `call`, `return`, `throw`
- A model of execution

$$\mathit{State} = \mathit{Stack}(\mathit{Node}), \mathit{Exception}?, \mathit{BagOf}(\mathit{Permission})$$

$$\frac{\mathit{Kind}(n) = \texttt{grant}(p, m) \quad n \stackrel{\mathit{intra}}{\rightarrow} n'}{n{:}s, \epsilon, \pi \longrightarrow n'{:}s, \epsilon, \mathit{grant}(\pi)(p, m)}$$

$$\frac{\mathit{Kind}(n) = \texttt{call} \quad n \stackrel{\mathit{inter}}{\rightarrow} m}{n{:}s, \epsilon, \pi \longrightarrow m{:}n{:}s, \epsilon, \pi} \qquad \frac{\mathit{Kind}(n) = \texttt{throw}(\mathit{ex}) \quad \forall h, n \stackrel{\mathit{ex}}{\nrightarrow} h}{n{:}s, \epsilon, \pi \longrightarrow n{:}s, \mathit{ex}, \pi}$$

Safe traces and permissions

1. Formalise the notion of *safe traces*

   Safe traces do not use more permissions than they have been granted

2. Prove the soundness theorem (Coq proof)

Theorem

$\forall n \in Node, P_n \neq Err \Rightarrow \forall tr \in Trace, Safe(tr)$

Static analysis of permission usage

- Compute an under-approximation of the permissions

$$P : \mathit{Node} \to \mathit{BagOf}(\mathit{Permission})$$

- Greatest solution of a set of recursive constraints

$$\frac{\mathit{Kind}(n) = \mathtt{grant}(p) \quad n \overset{\mathit{intra}}{\to} n'}{P_{n'} \sqsubseteq_p \mathit{grant}(P_n)(p)}$$

$$\frac{\mathit{Kind}(n) = \mathtt{call} \quad n \overset{\mathit{inter}}{\to} m \quad n \overset{\mathit{intra}}{\to} n'}{P_{n'} \sqsubseteq_p R_m(P_n)}$$

$\Rightarrow$ Iterative constraint solving

Inter-procedural analysis

Constraints summarise the effect of method calls

$$\frac{Kind(n) = \texttt{grant}(p, m) \quad n \stackrel{intra}{\longrightarrow} n'}{R_n^e \sqsubseteq grant(p, m); R_{n'}^e}$$

$$\frac{Kind(n) = \texttt{return}}{R_n \sqsubseteq \lambda\rho.\rho} \quad \frac{Kind(n) = \texttt{call} \quad n \stackrel{inter}{\longrightarrow} m \quad n \stackrel{intra}{\longrightarrow} n'}{R_n^e \sqsubseteq R_m; R_{n'}^e}$$

$\Rightarrow$ Amenable to symbolic resolution

Further enhancements

- Language features
  permission objects, multi-threading
- Precise program models
  dataflow analyses, integer analyses
  (nasty interaction with multi-threading)
- Strengthened security policy
  Beyond *enforceable security properties* (eventually, all the permissions are consumed)
- Bytecode verifier for the security model
  trade-off verification power/efficiency