

---

# *Model checking security properties of Java components*

Thomas Jensen  
IRISA  
Rennes, France

DISPO meeting  
Ecole des Mines de Nantes  
23 September 2004

# Semantics-based security analysis

---

Verification of software security using **static program analysis** and **model checking**.

**Fundamental question:** Is a global security property ensured by a collection of **dynamic security checks**, inserted into the code?

Approach inspired by the Schmidt-Steffen slogan:

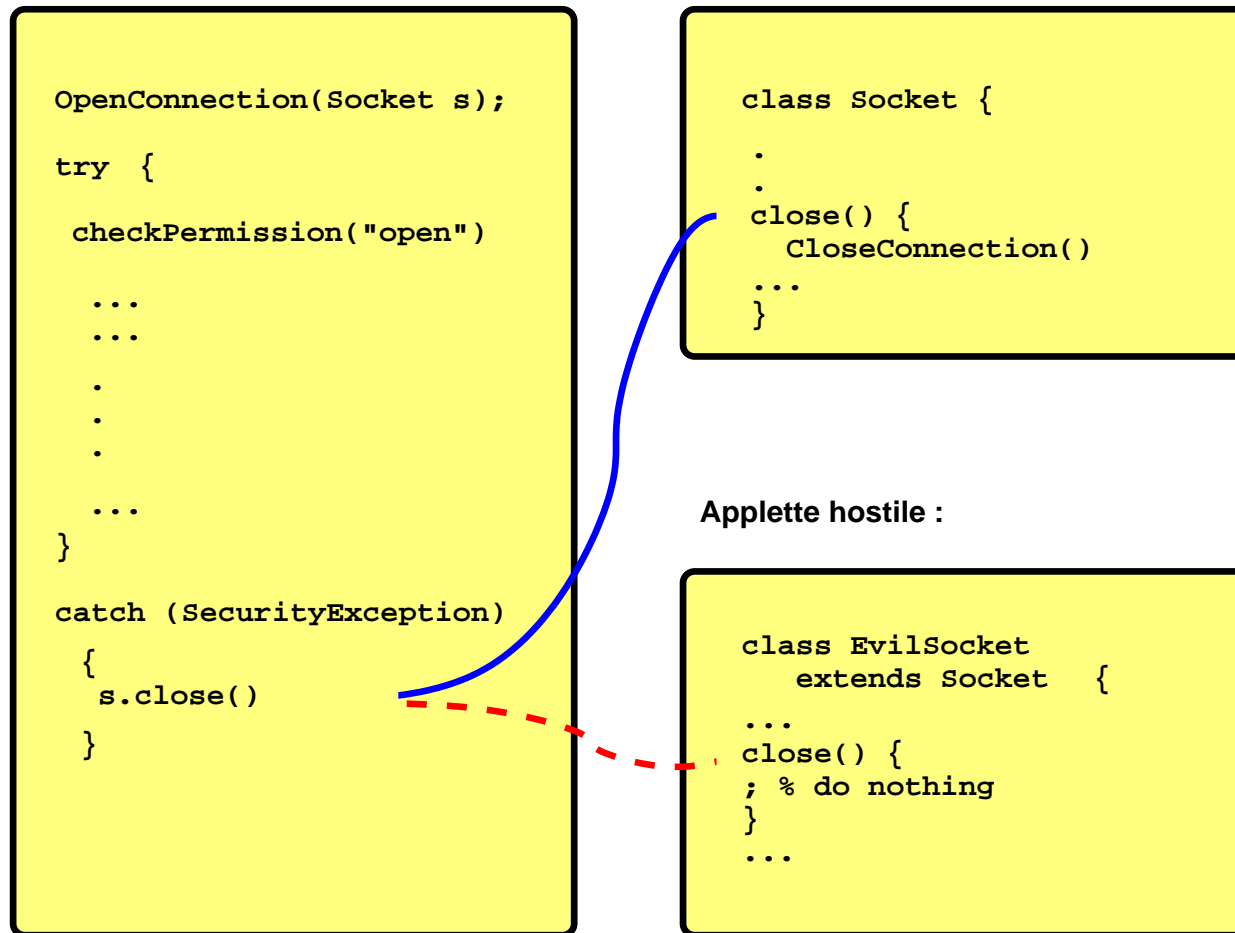
*“Data flow analysis as model checking of abstract interpretations”*

Concretely, this means:

- ▶ formalise security properties as temporal properties,
- ▶ use abstract interpretation to obtain an abstract, finitary model of the program to be analysed,
- ▶ use model checking to verify properties of this abstraction.

# Software security holes

A security hole (now fixed!) in Netscape allowed hostile applet to create illicit network connections and divulge confidential data.



# Overview

---

1. Access control in Java.
2. Modeling Java programs and security policies.
3. Extracting control flow graphs.
4. Verificaton and enforcement of security properties.
5. Towards verification of Java modules.

---

## *Access control in Java*

# Java security architecture

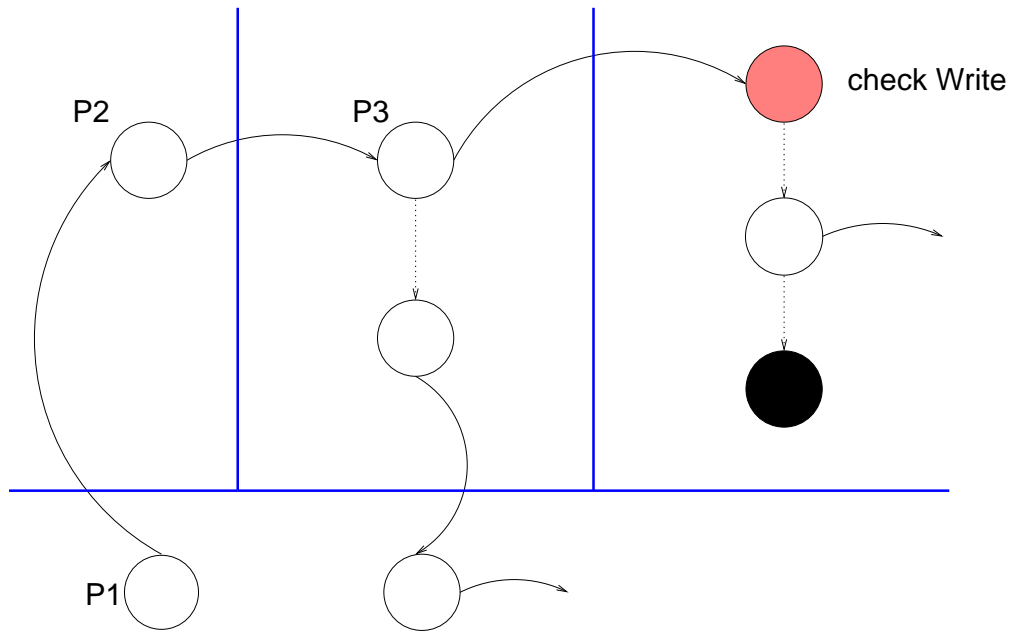
---

- ▶ Visibility modifiers (public, private, ...).
- ▶ Class loaders for name space creation.
- ▶ Code is given **permissions** depending on origin, signer, ...
- ▶ A Java application can check the permissions of its callers with :  
`AccessController.CheckPermission(P)`
- ▶ Privileged code—granting permissions to callers.

**Stack inspection**: an operation is performed only if all code participating in the call (“on the control stack”) has permission to do so.

```
public class ControlledVar {  
    private float var;  
    void write(float new) {  
        AccessController.checkPermission(Write);  
        var = new;  
    }  
}
```

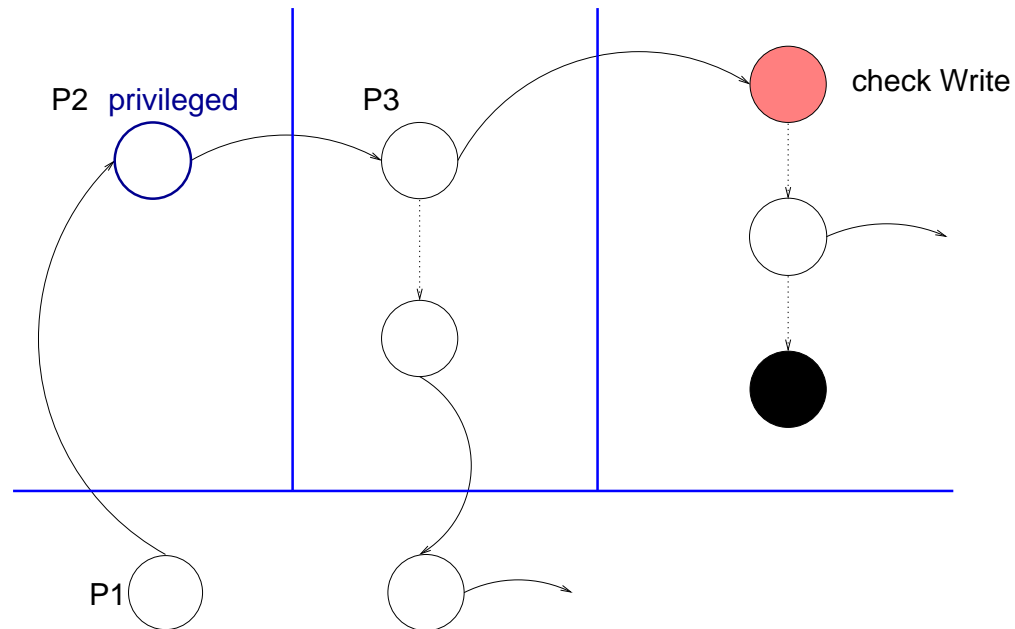
# Stack inspection



$$\text{write permission} \in \bigcap_{i=1}^3 \text{Perm}(P_i)$$

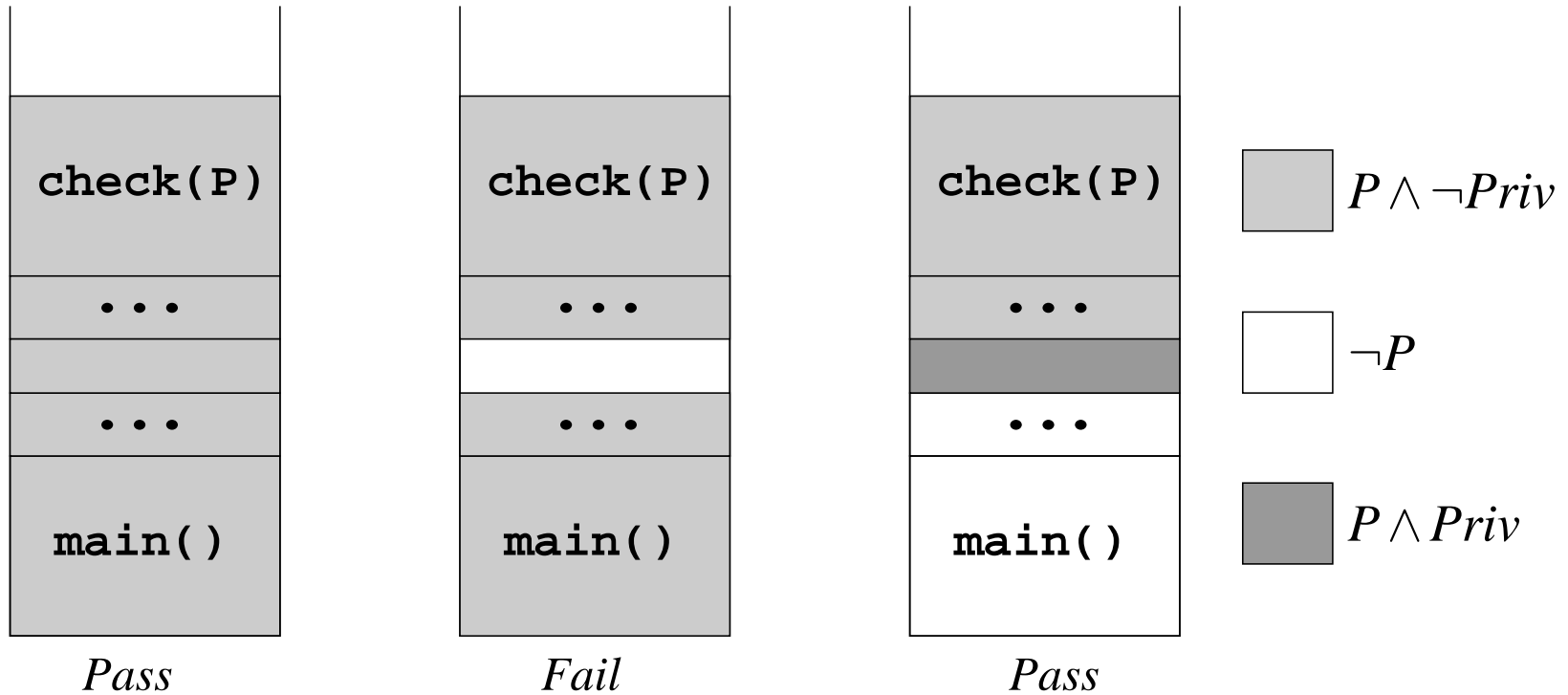
# Privileged calls

Privileged calls:  $P_2$  privileged  $\Rightarrow$  can ignore  $P_1$ .

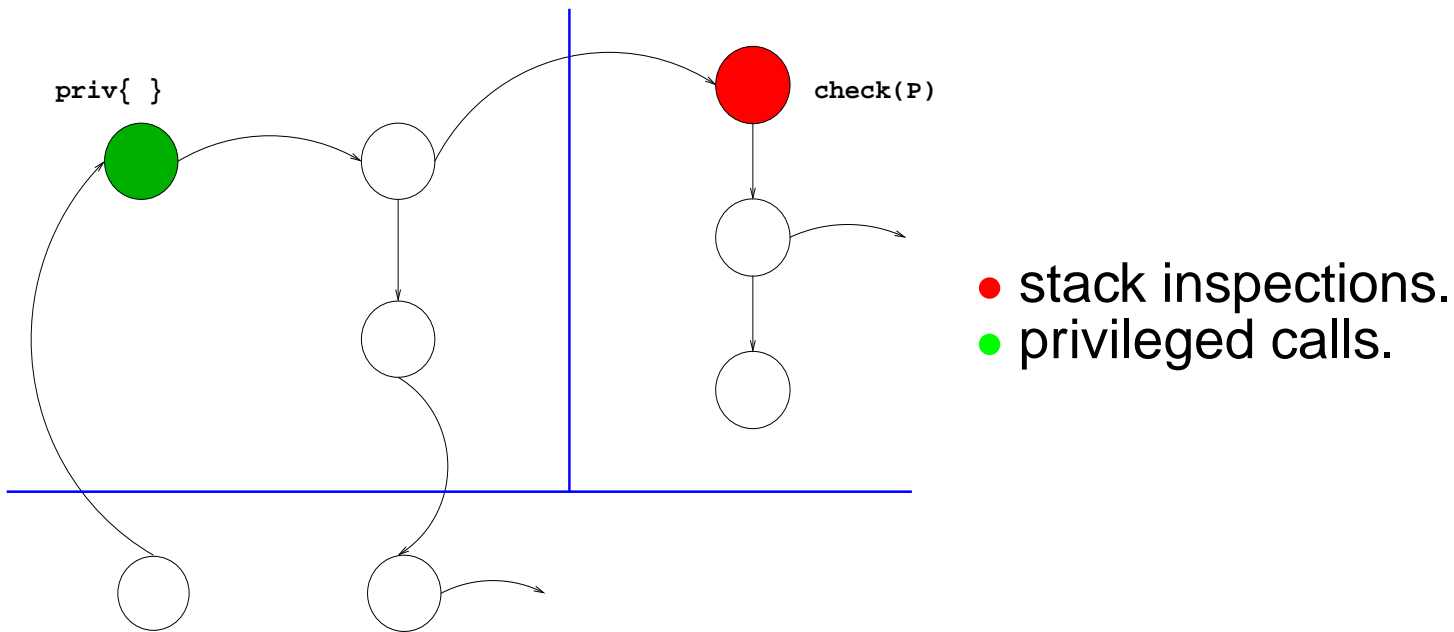


write permission  $\in \text{Perm}(P_2) \cap \text{Perm}(P_3)$

# Stack inspection - operationally



# Analysing stack-inspecting code



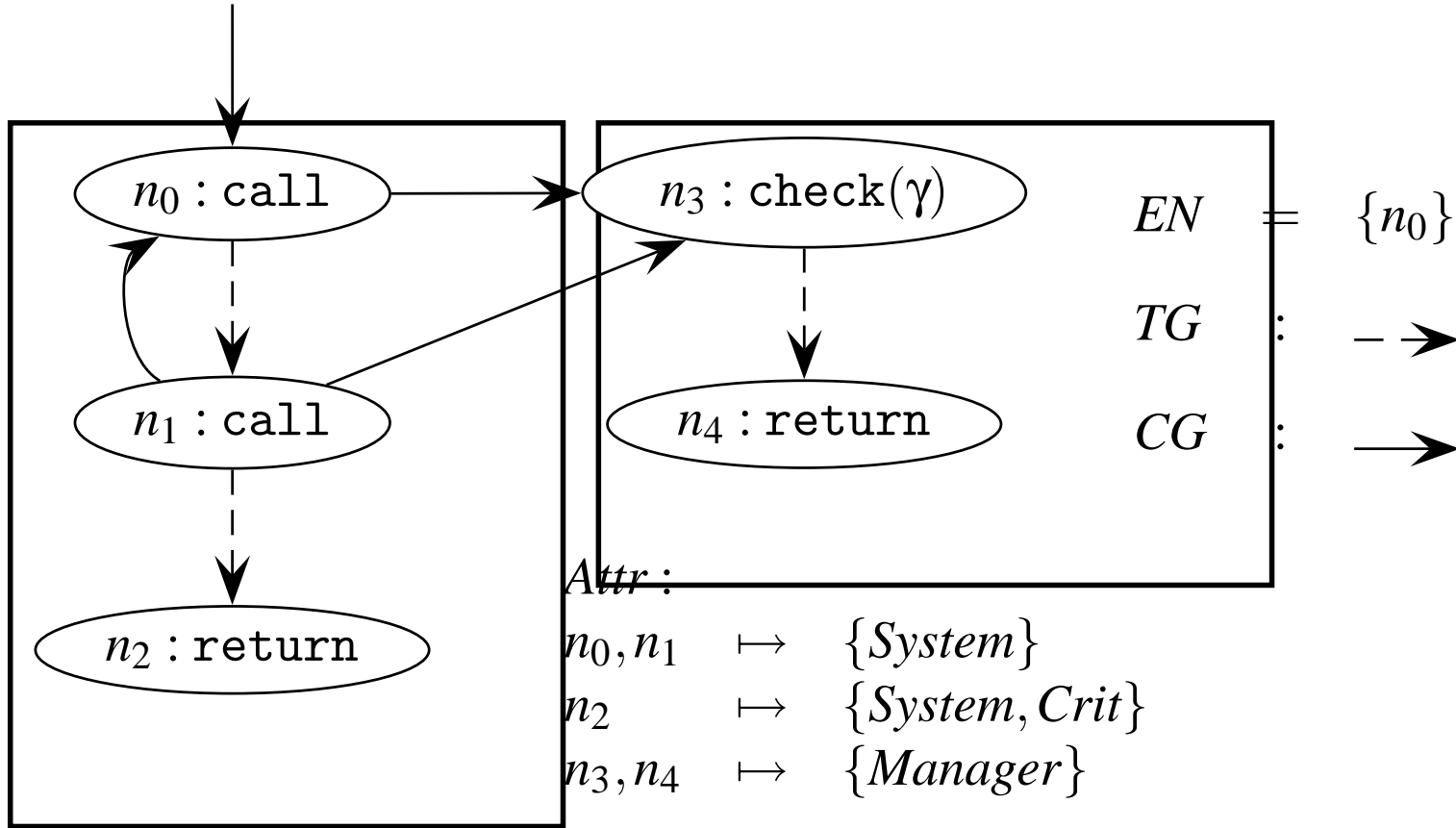
Extraction of a control flow graph from Java program using [control flow analysis](#)

Model checking reflexive push-down automata against security properties specified eg. using LTL or [security automata](#) (Schneider).

---

## *Model checking Java security*

# Program representation



# Operational semantics

$Stacks = Nodes^*$  is the set of states

$.\triangleright.$   $\subseteq Stacks \times Stacks$

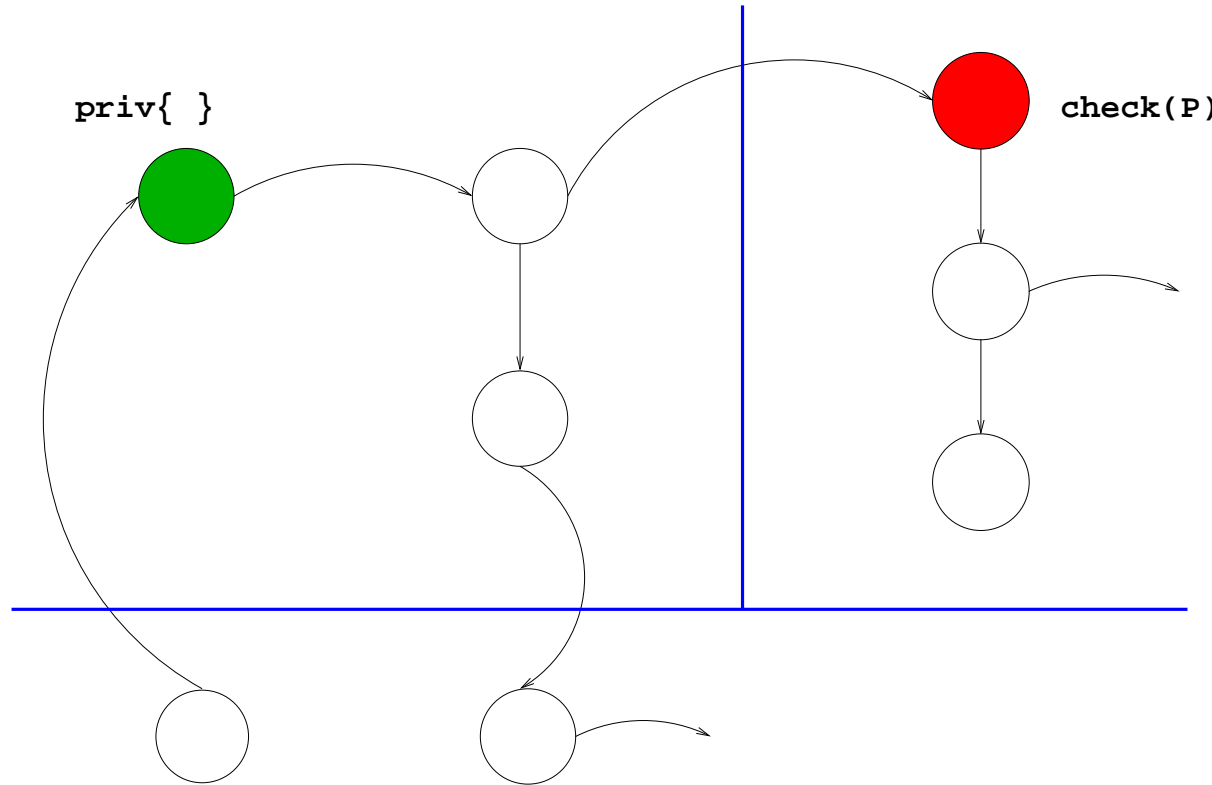
$s \triangleright s'$  iff one of the following rules applies:

$$\begin{array}{c} IS(n) = \text{check}(\gamma) \\ n \xrightarrow{TG} n' \\ s:n \models \gamma \\ \triangleright_{check} \frac{}{s:n \triangleright_{check} s:n'} \end{array} \qquad \begin{array}{c} IS(n) = \text{call} \\ n \xrightarrow{CG} m \\ \triangleright_{call} \frac{}{s:n \triangleright_{call} s:n:m} \end{array}$$

$$\begin{array}{c} IS(m) = \text{return} \\ n \xrightarrow{TG} n' \\ \triangleright_{return} \frac{}{s:n:m \triangleright_{return} s:n'} \end{array}$$

# Sandboxing

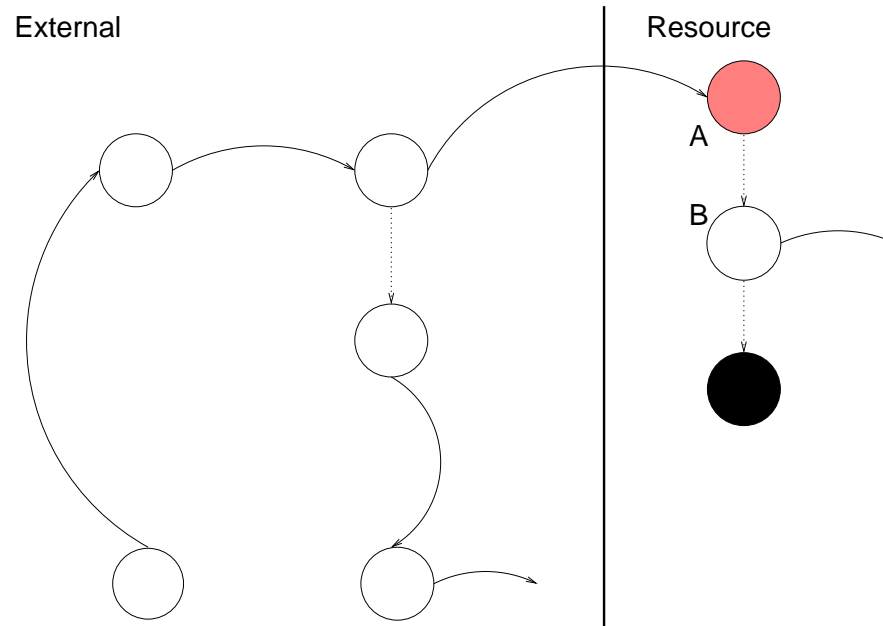
Code can call methods from own site/sandbox, or (certain) local methods.



$$\mathbf{G} (Site_S \Rightarrow \mathbf{X} (Site_S \vee Local))$$

# Resource protection

Resource protection: External code can only access a resource via a check node.



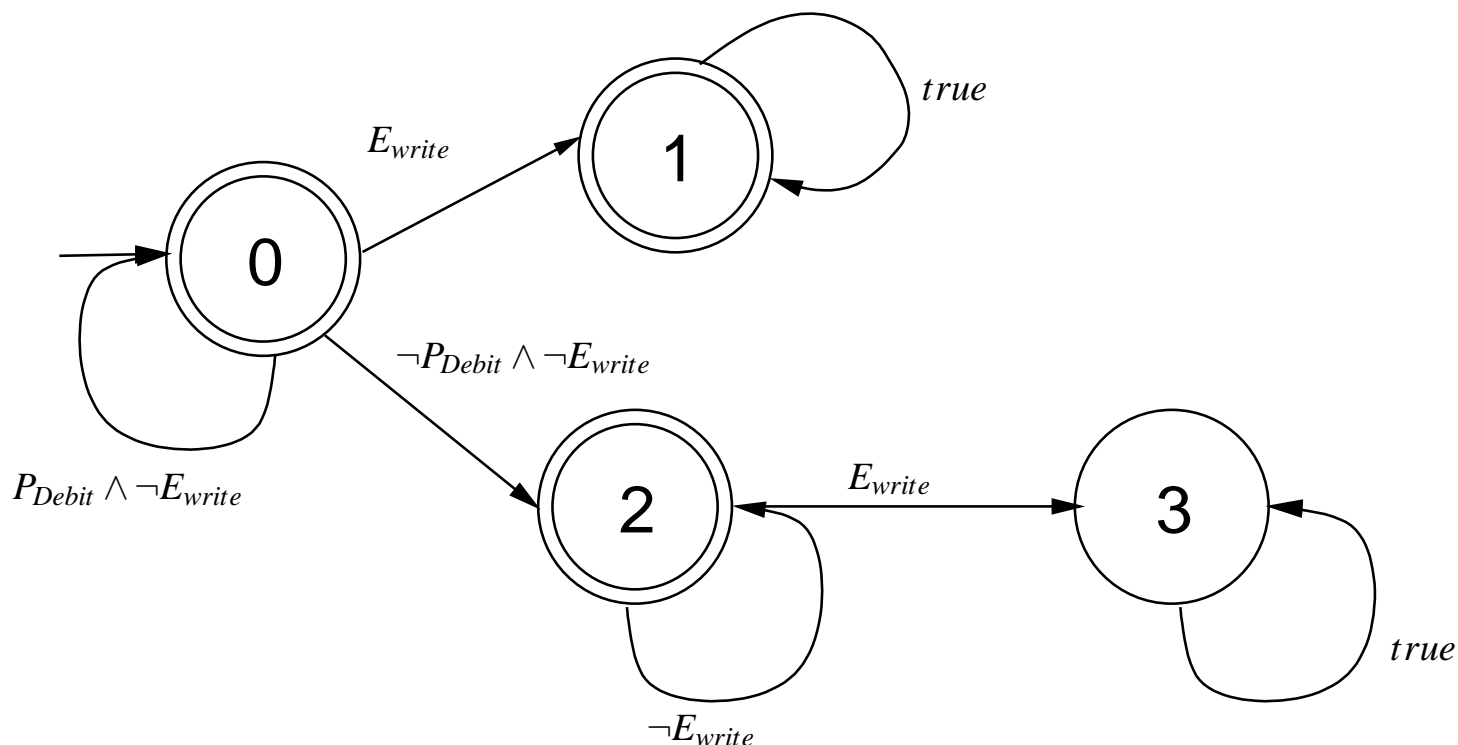
$$\mathbf{G}(\neg External \vee (\neg BU A)).$$

# Security automata

Notion proposed by Fred Schneider to represent “enforceable security policies”:

- ▶ Finite-state automata with particular sink states identifying security violations.
- ▶ Translation from LTL to automata.

The property “all calls leading to a write must have debit permission” expressed as an automaton:



# Security verification

---

Given a global security property  $\varphi$  (repr. by automaton  $A_\varphi$ ) and a control flow graph  $G$  with security checks,

- ▶ verify that all states satisfy  $\varphi$
- ▶ identify stack traces (“attacks”) leading to violation of  $\varphi$
- ▶ analyse the consequence of inserting/deleting security checks.

Standard model-checking solution:

Calculate reachable states of product trans. system

$$G \times A_\varphi.$$

# Automatic verification technique

---

The set of reachable call stacks can be **infinite**.

Identify those call stacks that satisfy the same security property and pass the same security checks.

Approach:

- ▶ **Abstract** stacks into states of the automata,
- ▶ Define a transition relation on the abstract states and verify that transition system

# Abstraction of control states

---

Keep enough information to

- ▶ decide where to call and return.
- ▶ predict outcome of local security check (stack inspection),
- ▶ check a stack against global security property

Principle:

Abstract a stack into its effect on the automata.

Concrete state = CtrlStack  $\times$  PgPoint.

Abstract state =

AutomataStates  $\times$  TopOfCtrlStack  $\times$  PgPoint.

For example, with one check  $\psi$  and global policy  $\varphi$ :

$$cs :: m :: n \mapsto (A_\psi(cs :: m), A_\varphi(cs :: m), m, n).$$

# Abstract transition system

---

**Call:**

$$\frac{IS(n) = \text{call} \quad n \longrightarrow m \quad (P, p, n) \in \llbracket G \rrbracket^\#}{(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#}$$

**Return:**

$$\frac{n \dots\dots > n' \quad (P, p, n) \in \llbracket G \rrbracket^\# \quad \delta(P, n) = N \quad (N, n, m) \in \llbracket G \rrbracket^\#}{(P, p, n') \in \llbracket G \rrbracket^\#}$$

**Check:**

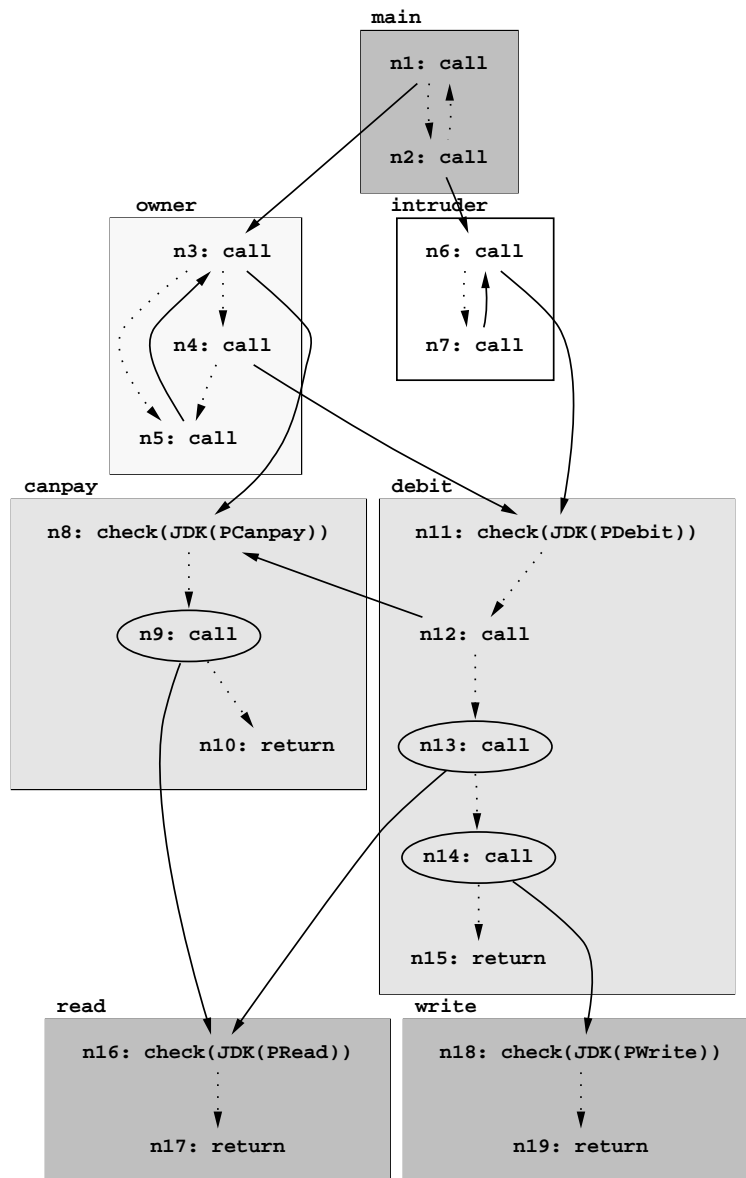
$$\frac{IS(n) = \text{check}(\gamma_i) \quad n \dots\dots > n' \quad (P, p, n) \in \llbracket G \rrbracket^\# \quad (P, p, n) \vdash \gamma_i}{(P, p, n') \in \llbracket G \rrbracket^\#}$$

**Completeness Theorem :**

$$\llbracket G \rrbracket \vdash \phi \quad \Leftrightarrow \quad \forall E \in \llbracket G \rrbracket^\# . E \vdash \phi$$

# Example: accessing an account.

- ▶ owner has *debit* permission,
- ▶ intruder hasn't.



# Analysis of example

Property to show: All calls leading to write must have debit permission and all calls leading to a read must have the *Canpay* permission:

$$\mathbf{G}(\neg E_{write}) \vee (P_{Debit} \mathbf{U} E_{write}) \quad \wedge \quad \mathbf{G}(\neg E_{read}) \vee (P_{Canpay} \mathbf{U} E_{read})$$

4 checks + 2 conjuncts in global property = 6 automata.

6 automata + TopOfCtrlStack + PgPoint  $\Rightarrow$  8-tuple as abstract state:

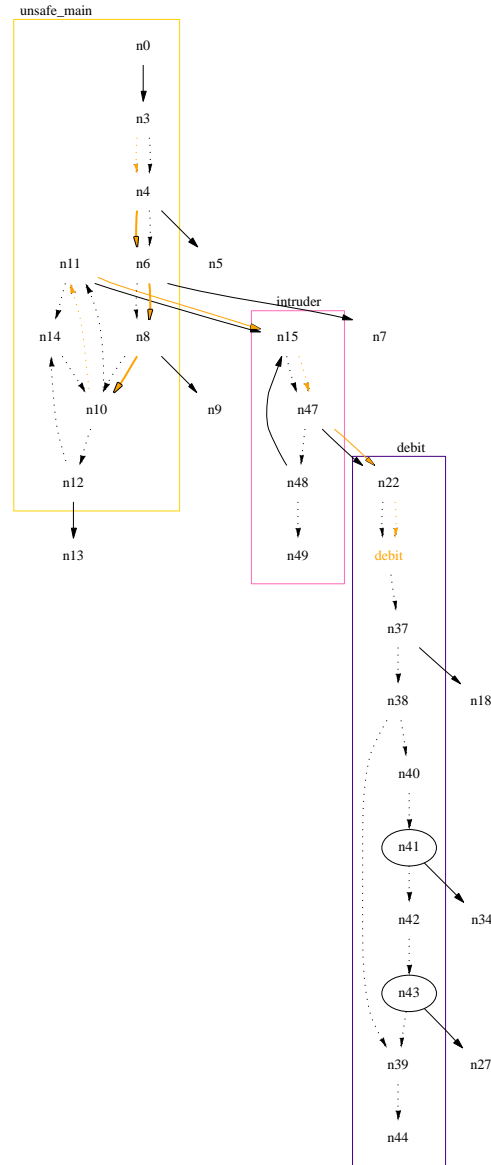
**Reachable states :**

$$\begin{array}{lll} ((0, 0, 0, 0, 0, 0), n_0, n_1) & ((0, 0, 0, 0, 0, 0), n_1, n_6), & \dots \\ ((1, 1, 1, 1, 2, 2), n_6, n_{11}), & ((0, 0, 1, 1, 0, 0), n_3, n_8), & \dots \\ ((0, 0, 0, 0, 0, 0), n_9, n_{16}), & ((0, 0, 0, 0, 0, 0), n_9, n_{17}), & \dots \\ \vdots & \vdots & 26 \text{ states} \end{array}$$

## Execution being cut

The check of the *debit* permission in the `debit()` results in a security exception being raised.

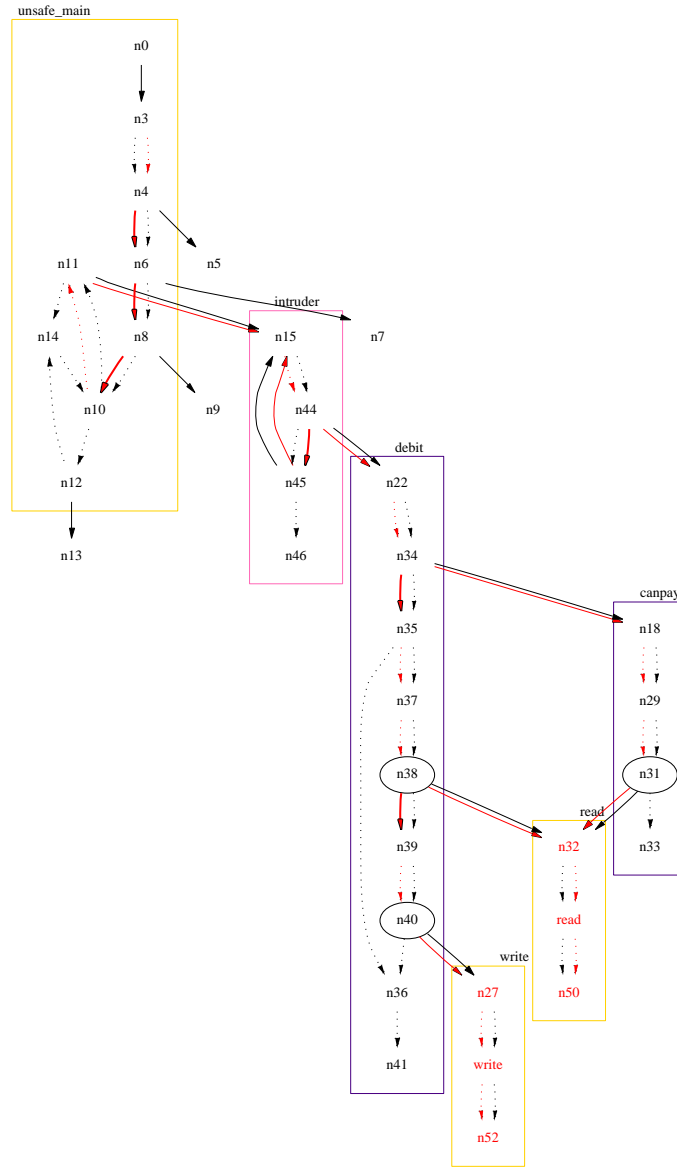
Yellow traces in the control flow graph show the path to security exception being raised.



## Security violation

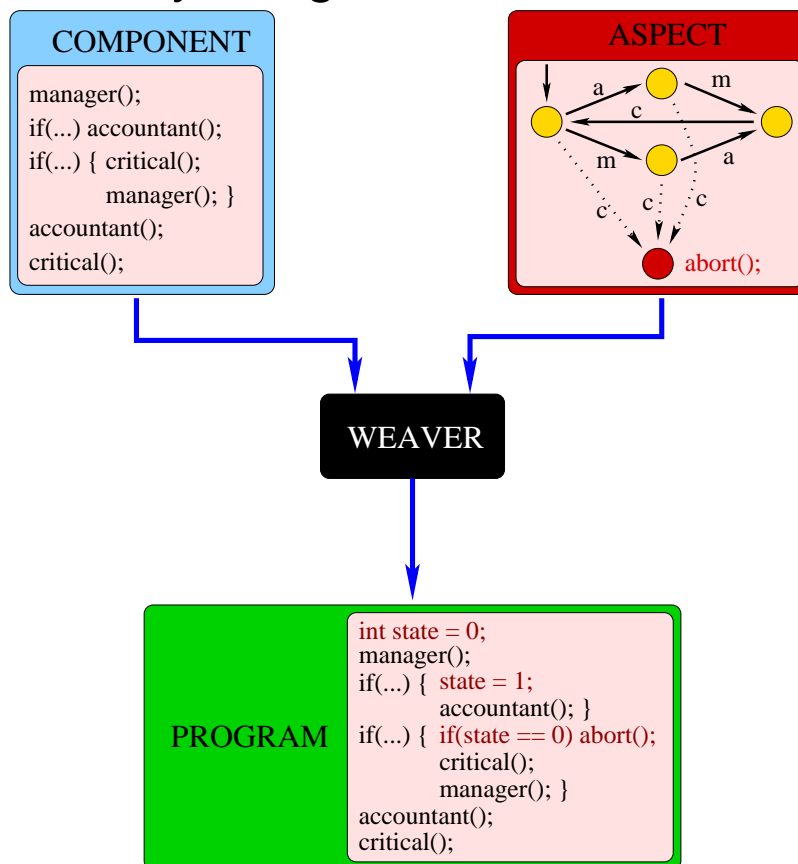
Consequence of removing the check of the *debit* permission in the `debit()`.

A red trace shows the path to the security violation.



# Enforcing security by transformation

Instead of rejecting potentially dangerous code, secure it!



---

# *Towards verification of Java components*

# Security, programs and libraries

---

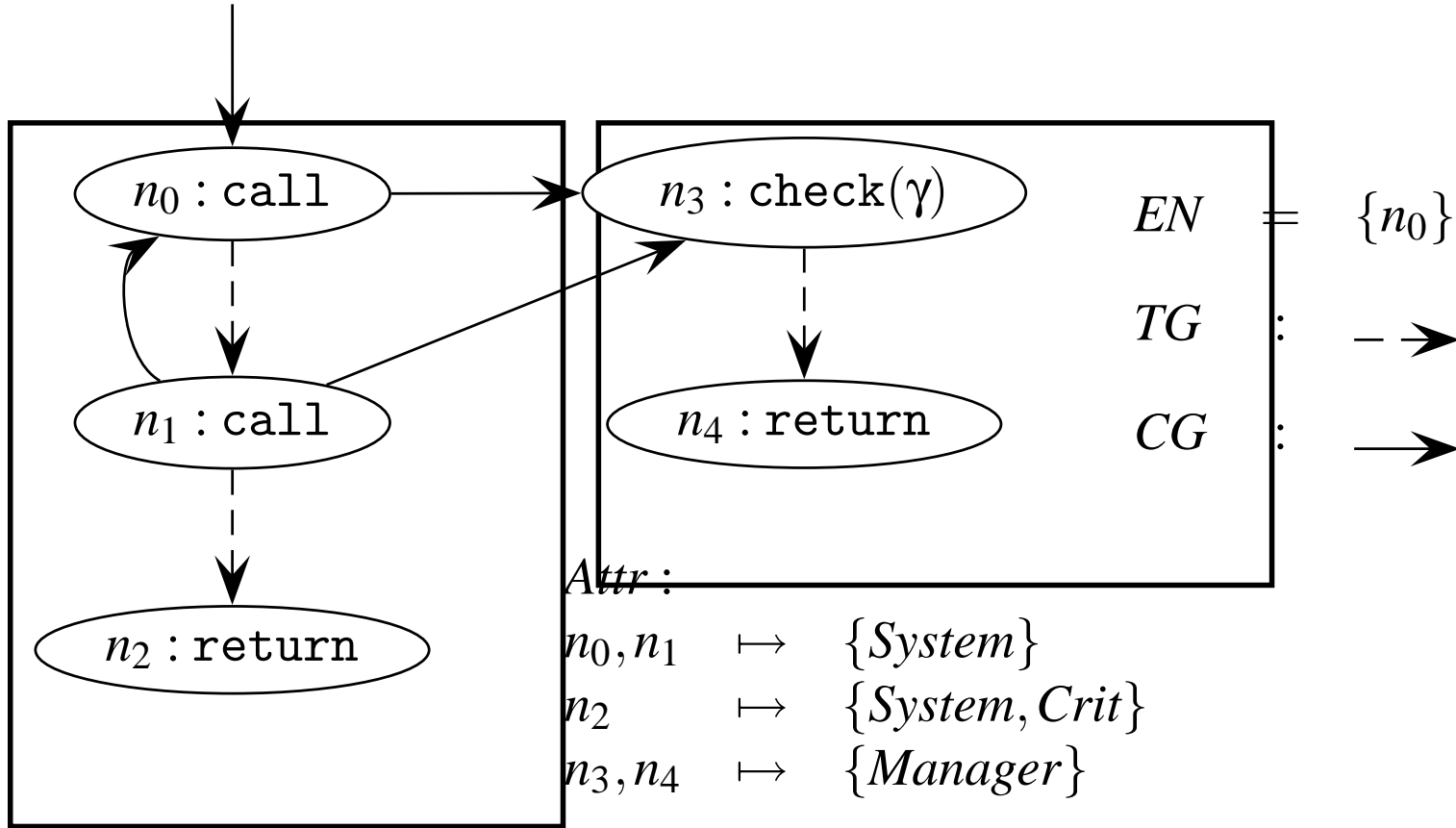
- ▶ A whole program is secure if all reachable execution states verify the global security property.
- ▶ Whole-program security analysis gives a yes/no answer:

$$CFG \rightarrow Bool$$

- ▶ In the case of a library, issue a **pre-condition**

$$CFG \times EN \rightarrow \mathcal{P}(Stacks)$$

# Graph example



# Secure calling contexts

---

- ▶ For each node  $n \in NO$ , we calculate a set  $\sigma_n$  of safe call stacks, *i.e.*,  $\forall s \in Stacks$ ,

$$s \in \sigma_n \quad \Rightarrow \quad \langle s:n \rangle^* \subseteq \mathcal{M}(\varphi)$$

where  $\mathcal{M}(\varphi) \in \mathcal{P}(Stacks)$  is the set of models of  $\varphi$ .

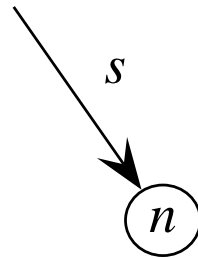
- ▶ A whole program is secure if  $\varepsilon \in \sigma_{n_0}$

Calculated as follows:

- ▶ Define the sets  $\sigma_n$  via a set of constraints.
- ▶ Solve these constraints iteratively in a symbolic domain of LTL formulae,
- ▶ proving explicitly that the iteration terminates in this infinite domain

# The $\delta$ operator on stacks

- ▶ What is the set  $S$  of stacks that doesn't immediately violate the security invariant if they call a given node  $n$ ?



$$\begin{aligned} S &= \{s \mid s:n \models \varphi\} \\ &= \{s \mid s:n \in \mathcal{M}(\varphi)\} \\ &= \delta_n(\mathcal{M}(\varphi)) \end{aligned}$$

- ▶ We introduce an operator defined as follows:  
 $\forall(S \in \mathcal{P}(\text{Stacks})), \forall(n \in \text{Nodes}),$

$$\delta_n(S) = \{s \mid s:n \in S\}$$

- ▶  $\delta$  satisfies

$$s:n \models \varphi \Leftrightarrow s \in \delta_n(\mathcal{M}(\varphi))$$

# Constraints for the $\sigma$ set

---

Execution is secure from  $s:n$  to end-of-method only if. . .

- ▶  $s:n$  is secure

$$\overline{\sigma_n \subseteq \delta_n(\mathcal{M}(\varphi))}$$

- ▶ stacks produced in methods  $m$  called from  $n$  are secure ( $s:n \in \sigma_m$ )

$$\frac{n \xrightarrow{CG} m}{\sigma_n \subseteq \delta_n(\sigma_m)}$$

- ▶ if  $s:n$  can sequentially lead to  $s:n'$ , then execution is secure from  $s:n'$

$$\frac{n \xrightarrow{TG} n'}{\sigma_n \subseteq (Stacks \setminus \tau_n) \cup \sigma_{n'}}$$

# Towards a symbolic calculation

---

- ▶ Abstraction from  $\mathcal{P}(\text{Stacks})$  to  $\mathcal{LTL}$ :
  - ▶  $(\text{Stacks}, \emptyset, \subseteq, \cup, \cap)$  becomes  $(\mathbf{True}, \mathbf{False}, \Rightarrow, \vee, \wedge)$
  - ▶  $\text{Stacks} \setminus .$  becomes  $\neg$
  - ▶  $\mathcal{M}(\phi)$  becomes  $\phi$
  - ▶  $\delta_n$  becomes  $\delta_n^\#$ , with  $s \models \delta_n^\#(\phi) \Leftrightarrow s:n \models \phi$
- ▶ Recast of our set constraints to  $\mathcal{LTL}$  constraints
  - ▶ For a given node  $n$ , if  $(\sigma_n^\#, \rho_n^\#, \tau_n^\#)$  is a solution to the abstract constraints, then  $(\mathcal{M}(\sigma_n^\#), \mathcal{M}(\rho_n^\#), \mathcal{M}(\tau_n^\#))$  is a solution to the concrete ones.
- ▶ A whole program is secure if  $\varepsilon \models \sigma_{n_0}^\#$

# Recasting constraints to $\mathcal{LTL}$ (examples)

- ▶  $\sigma$  for a  $CG$  edge:

$$\frac{n \xrightarrow{CG} m}{\sigma_n \subseteq \delta_n(\sigma_m)} \quad \text{becomes} \quad \frac{n \xrightarrow{CG} m}{\sigma_n^\# \Rightarrow \delta_n^\#(\sigma_m^\#)}$$

- ▶  $\sigma$  for a  $TG$  edge:

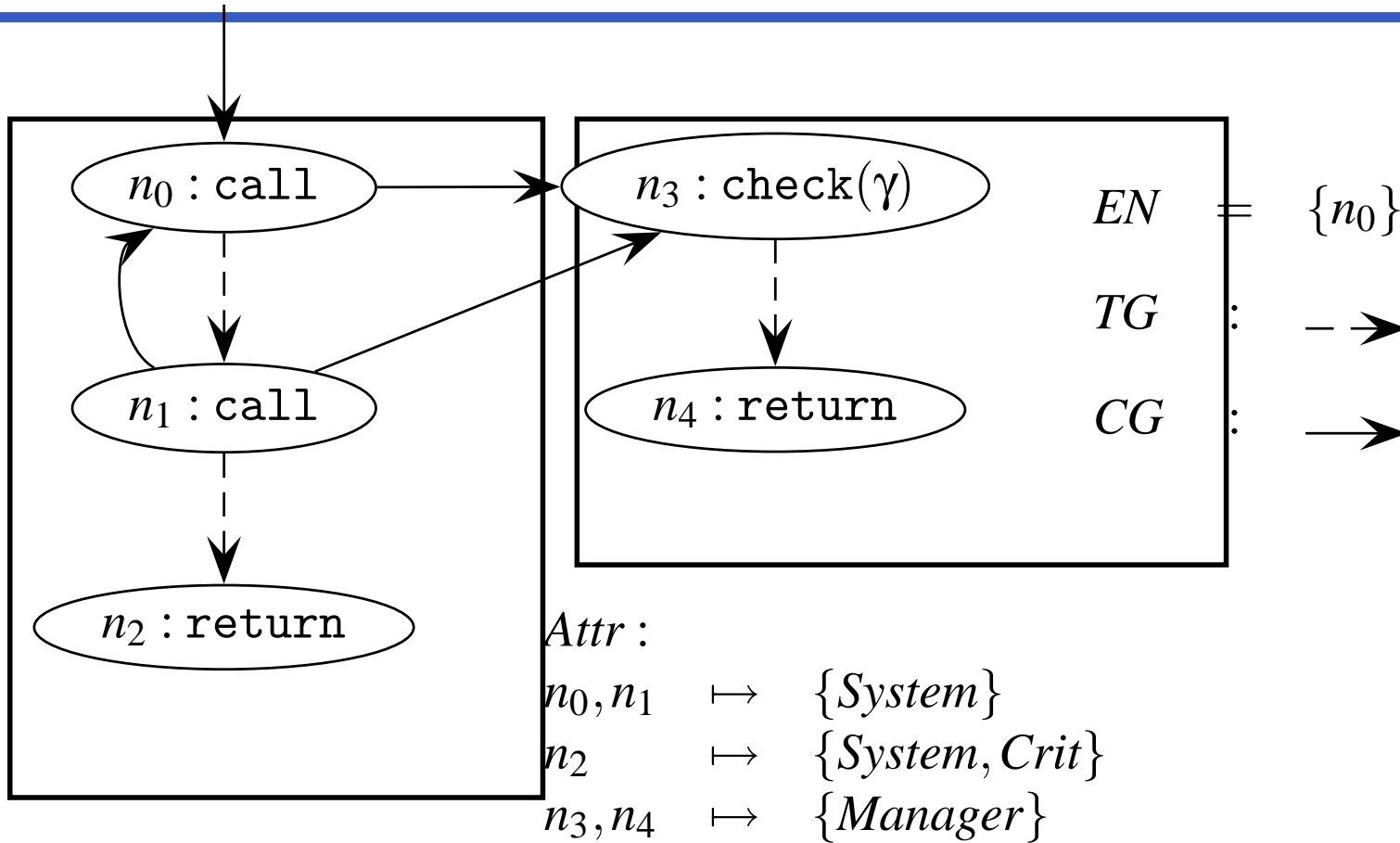
$$\frac{n \xrightarrow{TG} n'}{\sigma_n \subseteq (Stacks \setminus \tau_n) \cup \sigma_{n'}} \quad \text{becomes} \quad \frac{n \xrightarrow{TG} n'}{\sigma_n^\# \Rightarrow (\neg \tau_n^\# \vee \sigma_{n'}^\#)}$$

# Resolution

---

- ▶ Abstract constraints leads to monotone equations, solvable by fixed point iteration.
- ▶ Domain is  $(\mathcal{LTL}, \Leftarrow, \wedge, \vee)$
- ▶ But resolution may not terminates due to infinite ascending chains
- ▶ Our equations are only composed of  $\wedge$ ,  $\vee$ ,  $\neg$  and  $\delta^\#$  operators
- ▶ We can identify a finite sub-domain which:
  - ▶ contains every check formulae plus the global security invariant
  - ▶ is closed by boolean operators and  $\delta^\#$
- ▶ The resolution terminates in this domain

# Back to the example (1)



Local check:  $\gamma = \mathbf{F}(\text{Manager}) \wedge \mathbf{F}(\text{Accountant})$

Global property:  $\varphi = \text{Crit} \Rightarrow (\mathbf{F}(\text{Accountant}) \wedge \mathbf{F}(\text{Manager}))$

## Back to the example (2)

---

Result of the analysis for  $n_0$ :

$$\sigma_{n_0}^{\#} = (\mathbf{F}(\textit{Accountant}) \Rightarrow \mathbf{F}(\textit{Manager}))$$

The code is safe when called from a stack  $s$  such that  $s \models \sigma_{n_0}$ .

Informal explanation

- ▶ if  $s \models \neg\mathbf{F}(\textit{Accountant})$ , then execution is cut in  $n_3$
- ▶ if  $s \models \mathbf{F}(\textit{Manager})$ , then:
  - ▶ stack inspection succeeds in  $n_3$
  - ▶ critical operation is executed in a safe way *i.e.*,  
 $s:n_2 \models (\textit{Crit} \Rightarrow \mathbf{F}(\textit{Accountant}) \wedge \mathbf{F}(\textit{Manager}))$

# Summary

---

- ▶ Java components modelled formally using push-down systems with check points.
- ▶ Static program analysis for extracting such models.
- ▶ Temporal logic or security automata for specifying security policies.
- ▶ Model checking security of whole applications.
- ▶ Inferring safe contexts for program fragments.