Automated Verification of Security Policies of Mobile Programs

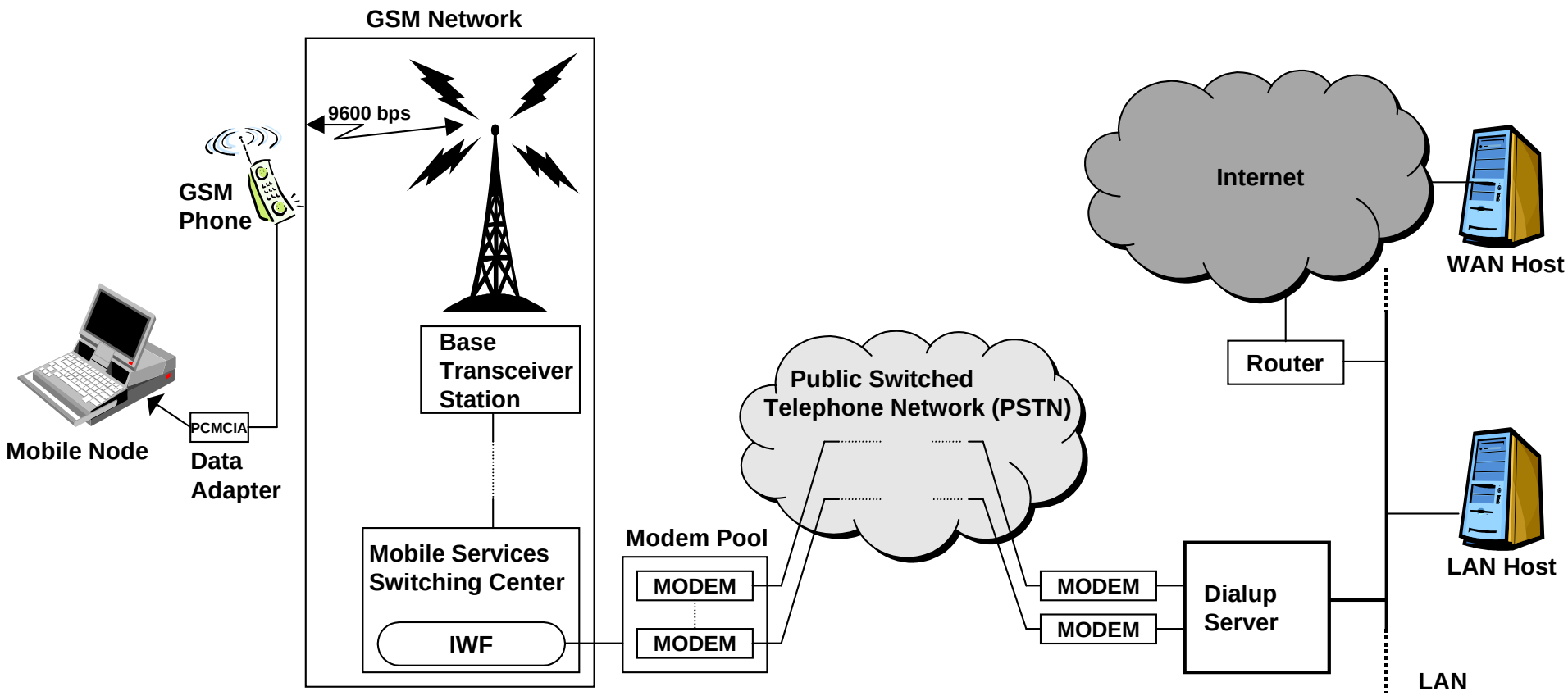
Natasha Sharygina

University of Lugano, Switzerland
and Carnegie Mellon University, USA

joint work with Chiara Braghin and Katerina Barone-Adesi
University of Lugano

EPFL, January 12, 2009

Motivation



- Exhaustive formal techniques are essential for verification of *mobile programs*, focusing on *security issues*.

Sample Security Policies

- *Mandatory Access Control*
  - If ℓ is a location/host that contains confidential information, we need to check that the information is not leaked (i.e., no write to remote locations).
- *Information Flow*
  - If a user is shopping using a credit card, we need to validate to whom/under what conditions the threads reveal the confidential information (by exploiting the trail of the actions of the threads and checking parameters of exchange actions).
- *"Bad" Traces*
  - If T_i is a downloaded applet, need to check for Trojan horses, i.e., sequences of malicious instructions (e.g., installing a backdoor or a keylogger).

State of the Art

- Programming languages for mobile code:
  - Java, Telescript, Obliq, Klaim, C, $\cdots$
- Current validation techniques:
  - Testing and simulation
  - Type system or Abstract Interpretation:
    - on process algebrae (π-calculus, Mobile Ambients, Dpi, etc.)
    - on very small fragments of programming languages
  - Dynamic policy verification (e.g., Java sandbox)
  - Model checking (only for process algebra models, not for the actual mobile code)

Objectives

- Expressive formalism for specifying mobile systems:
  - explicit specification of the *thread location* and *location net*
  - formalization of different synchronization and communication mechanisms
- Security policies specification language
  - not restricted to a single security policy
  - supports both access control and information flow specification
- Formal analysis of mobile systems by model checking of security and safety properties:
  - abstraction-based approach for efficiency
  - sound modeling of unbounded thread creation
  - verification of actual programs not abstract models

Our domain

- Mobile systems are a special case of multi-threaded programs.
- Differences:
  - Threads may run at different locations (e.g., different administrative domains, hosts, physical locations);
  - Threads may migrate from a location to another;
  - Thread communication takes into account geographical distribution:
    - distinction among local and global (remote) communication;
    - remote communication may be restricted by security policies.

Multi-threads - Classical Representation

T	::=		threads
		$T_1 \mid T_2$	parallel comp.
	$\mid$	$Instr$	sequential exec.
$Instr$	::=		instructions
		$Instr_1$ `;` $Instr_2$	sequential exec.
	$\mid$	`x := e`	assignment
	$\mid$	`if` ($Expr$ `!= 0`) $Instr$	condition
	$\mid$	`while` ($Expr$ `!= 0`) $Instr$	loop
	$\mid$	`skip`	skip
	$\mid$	`m`	sync. call
	$\mid$	`fork`	thread creation

Location-aware Threads (1)

LT	::=		location-aware threads
	$\mid$	$\ell[\,T\,]$	single thread
	$\mid$	$LT_1 \parallel LT_2$	parallel composition
T	::=		threads
		$T_1 \mid T_2$	parallel comp.
	$\mid$	$Instr$	sequential exec.
$Instr$	::=		instructions
		$Instr_1$ `;` $Instr_2$	sequential exec.
	$\mid$	`x := e`	assignment
	$\mid$	`if` ($Expr$ `!= 0`) $Instr$	condition
	$\mid$	`while` ($Expr$ `!= 0`) $Instr$	loop
	$\mid$	`skip`	skip
	$\mid$	`m`	sync. call
	$\mid$	`fork`	thread creation
	$\mid$	M_Instr	moving instr, `go_in(`ℓ`)`, `go_out(`ℓ`)`

Location-aware Threads (2)

- Explicit notion of location ($\ell[\,T\,]$).
- Ability to capture the geographical distribution of the Web:
  - *location net* is used to encapsulates the hierarchical nesting
    - location net: a tree, with nodes labeled by unique location names;
    - mobility: updates of the location net;
  - the structure of the location net can be constrained to formalize various security policies.
- Ability to locate multi-threaded programs.

Location Net - Example

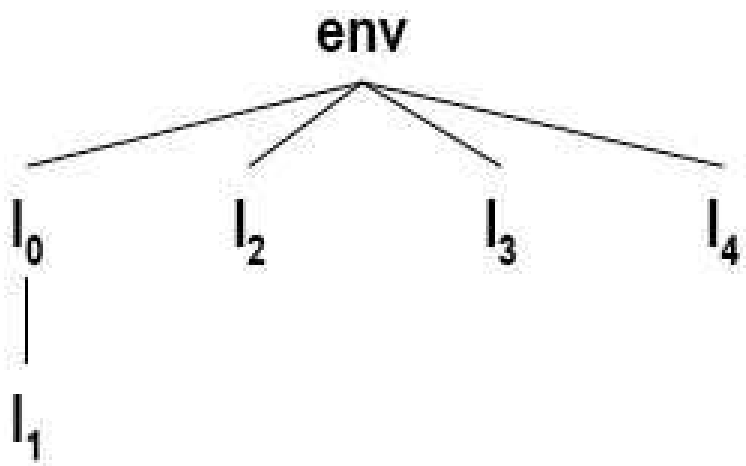
Consider a system with the following set of locations:

$$Loc = \{env, \ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}.$$

The location net is:

$$\{env.\ell_0.\ell_1, env.\ell_2, env.\ell_3, env.\ell_4\},$$

which corresponds to the following tree:



The Computational Model (1)

- A location-aware thread is a Labeled Kripke Structure $T = (S, Init, AP, \mathcal{L}, \Sigma, \mathcal{R})$, with:
  - The set of states $S = (V_l, V_g, pc, \varphi, \eta)$:
    - V_l evaluation of local variables,
    - V_g evaluation of global variables,
    - pc the program counter,
    - $\varphi : Loc \rightarrow Loc$ is a partial order function denoting the *location net* (where Loc is the set of location names),
    - $\eta : \mathbb{N} \rightarrow Loc$ is a partial function denoting the *thread location*.
- $\mathcal{R} \subseteq S \times \Sigma \times S \cup \{ S \times S \}$ is a total *labeled* transition relation
  - notice that $s \xrightarrow{\texttt{fork}} (s', \overline{s})$, with s' the next state of s, and $\overline{s} = Init$ the initial state of the newly created thread.

The Computational Model (2)

Inference rules for the labeled transition relation $\mathcal{R}_i$ for thread T_i.

(FORK-ACTION)

$$\frac{Instr(s.pc) = \texttt{fork}}{s \xrightarrow{\texttt{fork}}_i (s', \overline{s}) \;\; [s'.pc = s.pc + 1; \overline{s} = Init_i]}$$

(**in**-ACTION)

$$\frac{Instr(s.pc) = \texttt{go\_in}(\ell) \wedge (\exists \ell_1 \text{ s.t.: } \ell_1 := s.\eta(i) \wedge s.\varphi(\ell_1) = s.\varphi(\ell))}{s \xrightarrow{\texttt{go\_in(l)}}_i s' \;\; [s'.pc = s.pc + 1; s.\varphi \cup \{\ell_1 \mapsto \ell\}]}$$

(SYNC-ACTION)

$$\frac{Instr(s.pc) = \texttt{m}}{s \xrightarrow{\texttt{m}}_i s' \;\; [s'.pc = s.pc + 1]}$$

The Computational Model (3)

- The set of Σ is split into mutually disjoint sets Σ^M, Σ^S, Σ^T, Σ^τ, representing *moving*, *synchronization*, *thread creation*, and τ actions.

The labeled transition relation for the composition of two threads $T_1 \parallel T_2$.

(SYNC-ACTION)

$$\frac{a \in \Sigma_1^S \wedge s^1 \xrightarrow{a}_1 s'^1 \wedge a \in \Sigma_2^S \wedge s^2 \xrightarrow{a}_2 s'^2 \wedge s^1.\eta(1) = s^2.\eta(2)}{(s^1, s^2) \xrightarrow{a} (s'^1, s'^2)}$$

(L-PAR)

$$\frac{a \in \Sigma_1^M \wedge s \xrightarrow{a}_1 s'^1}{(s^1, s^2) \xrightarrow{a}_1 (s'^1, s^2)}$$

(R-PAR)

$$\frac{a \in \Sigma_2^M \wedge s^2 \xrightarrow{a}_2 s'^2}{(s^1, s^2) \xrightarrow{a}_2 (s^1, s'^2)}$$

Policy Specification Language (1)

- A security policy consists of a set of rules or conditions stating which actions are prohibited in the system:

$$\begin{aligned}
\langle \text{policy} \rangle &-> \{\langle \text{sec_levels} \rangle \mid \langle \text{operation_def} \rangle \mid \langle \text{deny statement} \rangle\} \\
\langle \text{deny statement} \rangle &-> \texttt{deny to}\ \langle \text{deny_target} \rangle\ [\langle \text{code base} \rangle][\langle \text{code origin} \rangle] \\
&\quad \texttt{\{}\ \langle \text{permission entry} \rangle\ \{\texttt{,}\ \langle \text{permission entry} \rangle\}\ \texttt{\}} \\
\langle \text{deny_target} \rangle &-> \texttt{public} \mid \langle \text{entity list} \rangle
\end{aligned}$$

- The location may play an important role in determining access rights:

$$\begin{aligned}
\langle \text{code base} \rangle &-> \texttt{codeBase}\ \langle \text{IPv4 addr} \rangle \\
\langle \text{code origin} \rangle &-> \texttt{codeOrigin}(\langle \text{location} \rangle \mid \texttt{remote}) \\
\langle \text{location} \rangle &-> \langle \text{location_id} \rangle\ \{\texttt{:}\ \langle \text{location_id} \rangle\}
\end{aligned}$$

Policy Specification Language (2)

- A permission entry specifies which actions are denied:

$$\begin{aligned} \langle \text{permission entry} \rangle &-> \texttt{permission}\ \langle \text{action} \rangle \\ \langle \text{action} \rangle &-> \langle \text{function} \rangle \mid \langle \text{operation} \rangle \end{aligned}$$

- In case of a function, it is possible to specify (i) formal parameters (variable names), (ii) actual parameters (the value of the arguments passed), (iii) an empty string, denying access to the function regardless of the arguments to it, or (iv) the keyword `high` (no high variables can be passed as arguments to this function).

$$\begin{aligned} \langle \text{function} \rangle &-> \texttt{function}\ \langle \text{function\_id} \rangle\ \langle \text{parameters} \rangle \\ \langle \text{parameters} \rangle &-> \langle \text{actual par} \rangle \mid \langle \text{formal par} \rangle \mid \texttt{high} \mid \varepsilon \end{aligned}$$

Examples of Security Policies Specification

- Java sandbox policy: it is responsible for protecting a number of resources by preventing applets from accessing the local hard disk and the network.

```
operation read_file_system { fread, read, scanf, gets, fgets}
deny to codeOrigin remote
{ permission function connect_to_location,
  permission operation read_file_system }
```

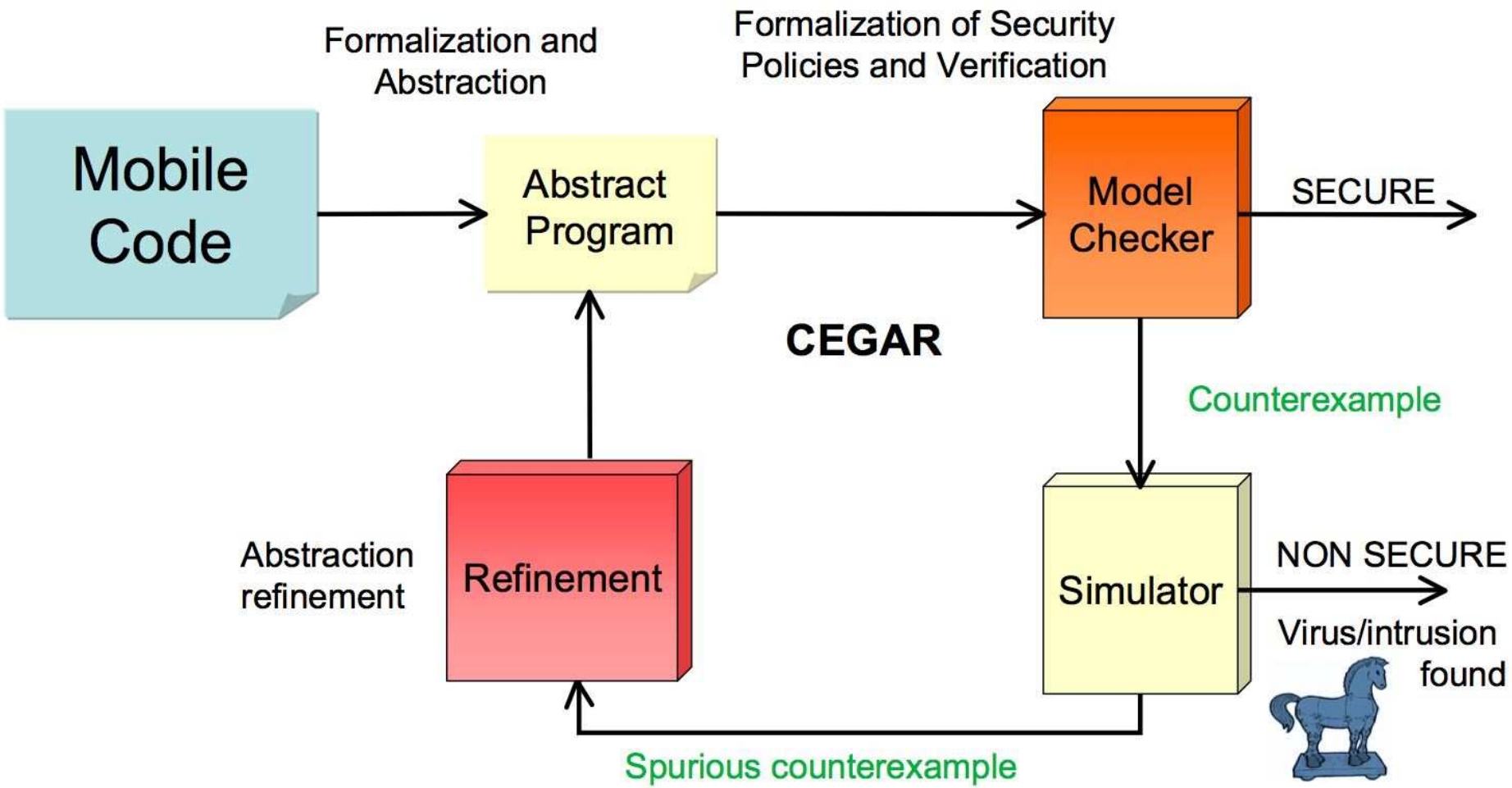
- A naive multi-level security policy:

```
High={confidential_var, x}
deny to public
{ permission function fopen high}
```

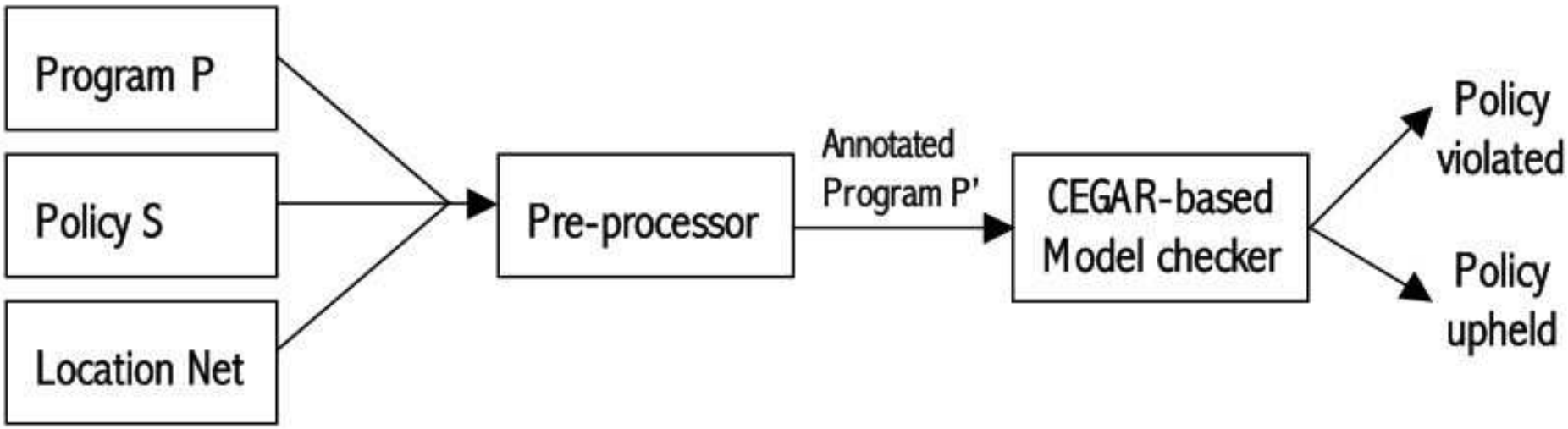
Implementation and Validation

- Implemented as part of SATABS, CEGAR-based model checker for ANSI-C;
  – SAT-based abstraction of programs (TACAS 2005);
  – unbounded thread creation is treated soundly by over-approximating thread creation (FMCAD 2006 paper);
  – Direct encoding of data (valuation of program variables) and control (control actions);
  – Abstract transition relation encodes location net (threads distribution);
  – Various location net-specific absraction mechanisms are available to manage size of state space.
- Experimented with mobile programs implemented in C (web-services, shopping systems, airline booking)

The CEGAR-based Mobile Code Analysis



Automated Security Verification



- Program P' is annotated with the security invariants expressed in the security policy;
- A security monitor function is created for each permission statement in the policy;
- P' consists of the product of P with all monitor functions corresponding to the security policy S.

An Efficient Security-driven abstraction

- Idea: Remove from the system behaviors not related to the location-specific policy:
  - Location Projection, $\pi \downarrow \ell$: thread executions for a particular location.
  - Moving Projection, $\pi \downarrow_M i$: all *moving* actions executed by thread T_i.
  - Thread Projection, $\pi \downarrow i$: all actions executed by thread T_i.

Verifying a Shopping Agent System

policy	time (s)	# iterations	# predicates	pv?
1 (sa)	151.644	7	17	yes
2 local (sa)	100.234	5	15	no
2 remote (sa)	524.866	12	36	yes
3 codeBase (sa)	340.011	12	22	yes

Sample policy for the shopping agent system (2(sa)):
deny to public
{ permission function fopen "/etc/passwd"}

Summary

- Expressive formalism for specifying mobile systems:
  – it explicitly models thread location, location distribution and thread moving operations;
  – it preserves both data and communication structures of mobile systems.
- Specification language for specifying security policies of mobile code:
  – generic policies;
  – information flow;
  – access control policies;
- Integrated model checking framework to support exhaustive analysis of security policies.
  – preprocessing and automated program anotation with security monitor functions;
  – CEGAR-based approach;
  – location-aware abstractions.

References

- Automated Verification of Security Policies in Mobile Code, Proceedings of 6th International Conference on Integrated Formal Methods, UK, 2007, LNCS, Vol. 4591, p. 37 - 53.
- Mobile Code Verification Project at Formal Verification and Security Group at University of Lugano, Switzerland, www.verify.inf.unisi.ch

Questions?