SPOPS

Secure Operating Systems for POPSs

Everest Team, INRIA Sophia-Antipolis
R2DP Team, LIFL, Lille University
SSIR Team, SUPELEC Rennes

http://www-sop.inria.fr/everest/projects/spops

December 12th, 2003

Context and Objective

POPSs require

- security: POPSs are widely used
  - as secure authentication tokens (SIM Cards)
  - to store and manipulate sensible data
- flexibility: POPSs must
  - interact with numerous heterogenous environments
  - provide execution support for a large panel of applications
  - execute several applications simultaneously
  - load applications and OS components post-issuance

The objective of SPOPS is to propose a compromise for addressing both needs simultaneously.

Topics

- Secure application loading/executing
  - Real-time operating systems for availability
  - Enhanced bytecode verification for stronger confidentiality (and integrity)
  - Logic-based methods for application verification
- Modular and secure operating systems
  - Modular and reconfigurable operating systems
  - Secure component loading
    - Modular verification of OS components
    - PCC

Availability

- Java security architecture does not address availability
- Ressources:
  - Memory
  - Communication
  - CPU
- Solutions:
  - Ticketing mechanisms for memory and communication.
  - WCET and real-time mechanisms for CPU

Remark: no trust between applications, hence OS must ensure equity

Availability in Camille NG

Results:

- Validation of dynamically loaded schedulers
- Split on-card/off-card computations for WCET

Further work

- Implementation of split computations
- Extension to JVM/OS

Confidentiality/Integrity

- Java security architecture only addresses a limited form of confidentiality/integrity
- A basic recipe for enforcing stronger confidentiality/integrity
  - Maintain the principle (dataflow analysis of an abstract virtual machine)
  - Enrich the type structure with security levels
- Information flow types guarantee that executing a program does not reveal otherwise unaccessible data to applets

Non-interference

Results

- Indistinguishability on JVM states
- Define a transition relation that rejects harmful programs

$$\Delta, C, m, i \vdash st, se \Rightarrow st', se'$$

- (Termination-insensitive) non-interference
- Compilation
- Non-interference for Java with exceptions (joint work with D. Naumann)

Further work

- Multi-threading
- Trusted downgrading and logic-based analyses

Types vs. logic

Type-based analyses

- are efficient and compositional
- are imprecise and do not capture certain properties

Logic-based analyses are

- are precise (and sometimes even complete) and capture many forms of security, and functionality
- complex to conduct

Our proposal

Proof finding is complex in general, but proof checking is simple

- Use proof finding for simple problems
- Use proof checking for complex problems

Weakest precondition calculi lie at the core of our approach

- Operate on annotated programs: pre-conditions, post-conditions, invariants
- Generate proof obligations from annotated programs

Security auditing

- Security auditing for high-level security properties, e.g.

no run-time exception at top-level
no nested transaction
no call to X between calling Y and returning from Z

- Generates core annotations from high-level properties
- Propagate annotations globally throughout the code
- Generate proof obligations with the WP calculus
- Discharge proof obligations with efficient provers

JITS

- Modular JVM used as an OS for POPS
- Ideal platform for experimenting with secure dynamic update
- System components (existing or under development):
  - (OO) Memory components: garbage collector, transactional memory model, etc.
  - CPU: scheduler, etc.
  - Communication: IP stack, etc.

Proof carrying code

- Principles:
  - Code comes with proof of correctness
  - Proof is checked, not inferred
  - No trust infrastructure is required
- Problems:
  - What to prove?
  - How to prove it?
  - How to package proofs?
- Applications: secure component loading

Work programme

- Complete work on availability and non-interference
- Develop modular system components: access controllers, protocol stacks, schedulers, etc.
- Develop generic specifications for components and verify components against specifications
- Implement a WP for Java bytecode
- Develop a PCC infrastructure and experiment with it