



Types for Resource Control

Ian Stark

Laboratory for Foundations of Computer Science
The University of Edinburgh

Formal Methods for Components and Objects
CWI, Amsterdam, 24–26 October 2007

Mobius: Mobility, Ubiquity and Security

Proof-carrying code for Java on mobile devices

FP6 Integrated Project developing novel technologies for trustworthy global computing, using **proof-carrying code** to give users independent guarantees of the safety and security of Java applications for mobile phones and PDAs.

- Innovative trust management, with digital evidence of program behaviour.
- Static enforcement, checking code before it starts.
- Modularity, building trusted applications from trusted components.

Types for Resource Control

This talk is about one of the underlying technologies: type systems that capture quantitative information about resource usage.

Work by Mobius partners in Madrid, INRIA, Munich, and Edinburgh.

Includes slides from David Aspinall, Patrick Maier and Martin Hofmann.

- 1 Resource control
- 2 Type systems
- 3 Heap space analysis
- 4 Permission analysis
- 5 Summary

- 1 Resource control
- 2 Type systems
- 3 Heap space analysis
- 4 Permission analysis
- 5 Summary

- 1 Machine resources — implicit properties of execution
 - heap space
 - execution time
 - stack height, call counting, ...
- 2 Program resources — explicitly manipulated in code
 - use-once permissions
 - collection sizes, thread pools, ...
- 3 External resources — exist outwith the JVM
 - billable events like text messages, phone calls
 - persistent database records, power consumption, ...

- Aim: *quantitative analysis* of resource usage.
 - Elsewhere: *patterns of access*, e.g. create, open, close.
- Objectives:
 - Simple, type-based treatment of useful cases
 - Static analysis to predict behaviour
 - Certification for PCC
- Benefits of resource control:
 - Obvious security relevance. Many security breaches amount to violating resource control: exceeding allowed bounds or gaining unauthorised access to resources.
 - Also useful beyond security: feasibility, scheduling, pricing.

Mobius work also includes approaches based on control flow graphs:

- Permissions, externally
 - native methods assigned a permissions profile
 - static analysis of control flow graph, check no errors
 - INRIA Rennes: Besson, Jensen, Pichardie.
 - see: WITS/ETAPS '07 invited talk by Thomas Jensen
- Execution cost
 - assign costs to bytecode statements (e.g., time)
 - generate a set of cost equations on control flow graph
 - Madrid: Albert, Arenas, Genaim, Puebla, Zanardini.
 - see: ESOP '07 paper by Madrid team

- 1 Resource control
- 2 Type systems**
- 3 Heap space analysis
- 4 Permission analysis
- 5 Summary

What is a type system?

A *type system* is a syntactically defined subset T of programs such that:

$$P \in T \Rightarrow \text{Compile}(P) \models \phi$$

where $\text{Compile}(P)$ is the object code corresponding to P and ϕ is some desired property of its execution.

For example,

$T =$ “well-typed Java programs”

$\phi =$ “methods always correctly invoked”

Slogan (Robin Milner): *Well-typed programs do not go wrong.*

Modern type systems guarantee more sophisticated and interesting properties. For example:

- Secure information flow.
- Bounds on resource usage.
- Absence of unwanted aliasing.
- Legal use of dynamic deallocation.

Declarative presentations of type systems

An inductively defined *typing judgement* relates program phrases e to types τ , given an assignment Γ of types to methods and variables.

Typing rules are mostly *syntax-directed*:
$$\frac{\Gamma \vdash x_1 : \text{int} \quad \Gamma \vdash x_2 : \text{int}}{\Gamma \vdash x_1 + x_2 : \text{int}},$$
 except for ...

- side conditions involving constraints (numerical, set-based);
- method types declared up front;
- existential metavariables, e.g. in subsumption rule:
$$\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation and declarative presentation.

Inference: generic algorithms available to suggest appropriate types.

Declarative presentations of type systems

An inductively defined *typing judgement* relates program phrases e to types τ , given an assignment Γ of types to methods and variables.

Typing rules are mostly *syntax-directed*:
$$\frac{\Gamma \vdash x_1 : \text{int} \quad \Gamma \vdash x_2 : \text{int}}{\Gamma \vdash x_1 + x_2 : \text{int}},$$
 except for ...

- side conditions involving constraints (numerical, set-based);
- method types declared up front;
- existential metavariables, e.g. in subsumption rule:
$$\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation and declarative presentation.

Inference: generic algorithms available to suggest appropriate types.

Declarative presentations of type systems

An inductively defined *typing judgement* relates program phrases e to types τ , given an assignment Γ of types to methods and variables.

Typing rules are mostly *syntax-directed*:
$$\frac{\Gamma \vdash x_1 : \text{int} \quad \Gamma \vdash x_2 : \text{int}}{\Gamma \vdash x_1 + x_2 : \text{int}},$$
 except for ...

- side conditions involving constraints (numerical, set-based);
- method types declared up front;
- existential metavariables, e.g. in subsumption rule:
$$\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation and declarative presentation.

Inference: generic algorithms available to suggest appropriate types.

Comparing types systems with program analysis

Advantages of type systems:

- Soundness separated from inference algorithms. No need to understand inference algorithm to grasp meaning of type system.
- Inherently interprocedural and modular.
- Interaction with user, e.g. via type annotations.
- Potential to connect to program logics.

Disadvantages of type systems:

- Less experience than with program analysis.
- More often ad-hoc due to lack of suitable approximation theory.
- Sometimes typing rules become very complicated.

- 1 Resource control
- 2 Type systems
- 3 Heap space analysis**
- 4 Permission analysis
- 5 Summary

Simple type system for heap space

Types are natural numbers.

Assignment Σ of types to methods M . Assign types to expressions with rules like:

$$\frac{\vdash e : n \quad n \leq m}{e : m} \quad (\text{WEAK})$$

$$\frac{\vdash e_1 : n_1 \quad \vdash e_2 : n_2}{\vdash \text{let } x=e_1 \text{ in } e_2 : n_1 + n_2} \quad (\text{LET})$$

$$\frac{}{\vdash \text{new } C(x_1, \dots, x_n) : 1} \quad (\text{NEW})$$

$$\frac{\Sigma(M) = n}{\vdash M(x_1, \dots, x_n) : n} \quad (\text{INVOKE})$$

Checking correctness of a derivation is easy; finding one can be harder.

- Assign a variable to each method.
- Derive “skeleton” type derivation using `WEAK` only next to `INVOKE`.
- Try to solve resulting constraints.

Output:

- Typing is static evidence that program satisfies some resource bound
- Security model: client may refuse to execute code that has no bound, or whose bound is beyond device limits

Context:

- Provably equivalent to graph-based analysis.
- Can be extended to deallocation and input-dependent bounds.

Extending to deallocation

Typing judgement: $\Gamma \vdash e : m \rightarrow n$.

Meaning: If freelist has size $s \geq m$ then evaluation of e will succeed and leave freelist of size $\geq (n + s - m)$.

$$\frac{\vdash e : m \rightarrow n \quad m' \geq m \quad n' \leq n + m' - m}{e : m' \rightarrow n'} \quad (\text{WEAK}')$$

$$\frac{\vdash e_1 : m \rightarrow k \quad \vdash e_2 : k \rightarrow n}{\vdash \text{let } x=e_1 \text{ in } e_2 : m \rightarrow n} \quad (\text{LET}')$$

$$\frac{}{\vdash \text{new } C(x_1, \dots, x_n) : 1 \rightarrow 0} \quad (\text{NEW}')$$

$$\frac{\Sigma(M) = m \rightarrow n}{\vdash M(x_1, \dots, x_n) : m \rightarrow n} \quad (\text{INVOKE}')$$

Applying this to PCC for trustworthy computing:

- Typing derivations can be used directly as certificates. But: need to believe or understand type soundness.
- Likewise, successful runs of a program analysis, perhaps in condensed form can be used as certificates. But: need to believe or understand correctness of analysis.

Two better options:

- Formally prove correctness of analysis / type system
- Translate typing judgements into judgements of a formalised program logic, translate typing derivations into proofs of those translations.

Translating heap space type system into program logic

For example, $e : n$ becomes $\llbracket e \rrbracket : \phi_n$ where ϕ_n is the specification

$$\phi_n \equiv \begin{array}{l} \text{pre} \\ \text{post} \quad |h| \leq |h_{\text{old}}| + n \\ \text{inv} \quad \quad |h| \leq |h_{\text{old}}| + n \end{array}$$

Here $\llbracket e \rrbracket$ is the bytecode corresponding to e and $|h_{\text{old}}|$, $|h|$ are the initial and current heap respectively.

Typing derivations can be translated into program logic rule by rule. Each typing rule can be translated (as an admissible rule) once and for all.

The Mobius Base Logic expresses assertions like this and provides proof rules that allow modular translation of typing rules. Resources can be tracked with **ghost fields** in the heap. [See Hofmann talk later]

Translating heap space type system into program logic

For example, $e : n$ becomes $\llbracket e \rrbracket : \phi_n$ where ϕ_n is the specification

$$\phi_n \equiv \begin{array}{l} \text{pre} \\ \text{post} \quad |h| \leq |h_{\text{old}}| + n \\ \text{inv} \quad \quad |h| \leq |h_{\text{old}}| + n \end{array}$$

Here $\llbracket e \rrbracket$ is the bytecode corresponding to e and $|h_{\text{old}}|$, $|h|$ are the initial and current heap respectively.

Typing derivations can be translated into program logic rule by rule. Each typing rule can be translated (as an admissible rule) once and for all.

The Mobius Base Logic expresses assertions like this and provides proof rules that allow modular translation of typing rules. Resources can be tracked with **ghost fields** in the heap. [See Hofmann talk later]

- 1 Resource control
- 2 Type systems
- 3 Heap space analysis
- 4 Permission analysis**
- 5 Summary

Starting assumption: at any point some set of permissions is available.

Certain operations may:

- **grant** a permission; increases the permission set
- **request** a permission; this
 - either checks presence — classical access control,
 - or checks presence and **consumes an instance** — e.g. billable events

A program is safe if for every execution trace the requested permissions are present (i.e. no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes the classical case.

Starting assumption: at any point some set of permissions is available.

Certain operations may:

- **grant** a permission; increases the permission set
- **request** a permission; this
 - either checks presence — classical access control,
 - or checks presence and **consumes an instance** — e.g. billable events

A program is safe if for every execution trace the requested permissions are present (i.e. no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes the classical case.

Starting assumption: at any point some set of permissions is available.

Certain operations may:

- **grant** a permission; increases the permission set
- **request** a permission; this
 - either checks presence — classical access control,
 - or checks presence and **consumes an instance** — e.g. billable events

A program is safe if for every execution trace the requested permissions are present (i.e. no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes the classical case.

Example: SMS in MIDP

Text messaging (SMS) by Java phone applications (MIDP, MIDlets). Widely used for basic connectivity. But: Cost concerns and draconian security policies.

Bulk messaging MIDlet code

```
msg.edit();
grp = addr_book.sel_grp();    // user selects recipients
for (addr in grp) {
    num = addr.get_mobile();
    msg.send(num);           // user re-confirms each recipient
}
```

- Hard to detect statically that re-confirmation is unnecessary:
 - Selection and sending may happen in different threads
 - User should confirm phone number rather than address book entry
- Intrusive security even introduces new social engineering attacks
- Also: multi-stage protocols; unattended actions; ...

API for Resource Managers

Idea: reify permission sets into code to allow explicit accounting.

- New API for resource managers, with three classes:
 - `Resource` — abstract class for resources; subclasses immutable
 - `Multiset` — for manipulating permission sets
 - `ResourceManager` — encapsulation of granted permission set
- Standard operations instrumented with resource consumption.
- Separates granting resources from using them
- Supports **block booking**: requesting several resources at the same time, in advance.
- API needs careful design: resource manager should be tamper-proof and new information flows limited.
- Compared with leaving permissions implicit:
 - Pros: visible to programmer, flexible static/dynamic policies.
 - Cons: additional checks, complexity, platform library extension

API for Resource Managers

Idea: reify permission sets into code to allow explicit accounting.

- New API for **resource managers**, with three classes:
 - **Resource** — abstract class for resources; subclasses immutable
 - **Multiset** — for manipulating permission sets
 - **ResourceManager** — encapsulation of granted permission set
- Standard operations instrumented with resource consumption.
- Separates granting resources from using them
- Supports **block booking**: requesting several resources at the same time, in advance.
- API needs careful design: resource manager should be tamper-proof and new information flows limited.
- Compared with leaving permissions implicit:
 - Pros: visible to programmer, flexible static/dynamic policies.
 - Cons: additional checks, complexity, platform library extension

API for Resource Managers

Idea: reify permission sets into code to allow explicit accounting.

- New API for **resource managers**, with three classes:
 - **Resource** — abstract class for resources; subclasses immutable
 - **Multiset** — for manipulating permission sets
 - **ResourceManager** — encapsulation of granted permission set
- Standard operations instrumented with resource consumption.
- Separates granting resources from using them
- Supports **block booking**: requesting several resources at the same time, in advance.
- API needs careful design: resource manager should be tamper-proof and new information flows limited.
- Compared with leaving permissions implicit:
 - Pros: visible to programmer, flexible static/dynamic policies.
 - Cons: additional checks, complexity, platform library extension

API for Resource Managers

Idea: reify permission sets into code to allow explicit accounting.

- New API for **resource managers**, with three classes:
 - **Resource** — abstract class for resources; subclasses immutable
 - **Multiset** — for manipulating permission sets
 - **ResourceManager** — encapsulation of granted permission set
- Standard operations instrumented with resource consumption.
- Separates granting resources from using them
- Supports **block booking**: requesting several resources at the same time, in advance.
- API needs careful design: resource manager should be tamper-proof and new information flows limited.
- Compared with leaving permissions implicit:
 - Pros: visible to programmer, flexible static/dynamic policies.
 - Cons: additional checks, complexity, platform library extension

Using resource managers

MIDlet with resource manager

```
msg.edit(); // user edits message
nums = addr_book.sel_nums(); // user selects recipients
Multiset r = new Multiset();
for (num in nums) { // gather resources
    r.add(Resource.from_string(num));
}
ResMgr m = new ResMgr();
r = m.enable(r); // user grants resources
if (r.is_empty()) {
    for (num in nums) { // send loop, pass mgr
        msg.send_rm(m, num);
    }
}
```

Use a type system to analyze statically the use of external resources.

- Work with simple procedural language (bytecode-level), featuring:
 - resource type `res` with constructors, equality
 - multiset type `mset` with union, intersection, ...
 - abstract type `mgr` with copying prohibited
- Statements in SSA form:

$$s ::= \text{skip} \mid \text{assert } x_1 \text{ in } x_2 \mid y := e \mid s_1; s_2 \mid \\ \text{if } x \text{ then } s_1 \text{ else } s_2 \text{ fi} \mid p(x_1, \dots, x_m \mid y_1, \dots, y_n)$$

- Procedure parameters partitioned by input/output mode.
- All `mgr`-variables must be used exactly once (linear).
- `assert` checks $x_1 \subseteq x_2$, raises error \perp otherwise
- Built-in procedures for manipulating managers, etc.

Built-in procedures

- `empty(| m : mgr)`
 - Set m to the empty multiset.
- `enable(r_1 : mset | m : mgr, r_2 : mset)`
 - Ask user/policy to grant resources r_1 .
 - Place granted resources in m , denied resources in r_2 .
- `split(m : mgr, r : mset | m_1 : mgr, m_2 : mgr)`
 - Split off from m all resources contained in r .
 - Place split off resources in m_2 , remaining resources in m_1 .
- `join(m_1 : mgr, m_2 : mgr | m : mgr)`
 - Set m to the sum of resources in m_1 and m_2 .
- `consume(m : mgr |)`
 - No-op (required due to strict linearity).

Bulk messaging (1)

Main procedure

```
bulk_send(msg:string, nums:string[] |) {
  r,r':mset;
  m,m':mgr;
  b:bool;

  res_from_nums(nums | r);
  enable(r | m, r');
  b := r' = ∅;
  if b
  then msg_send(msg, nums, m | m');
      assert m' in r';
      consume(m' |)
  else consume(m |)
  fi
}
```

Bulk Messaging (2)

Send loop — body

```
msg_send'(msg:string, nums:string[], m:mgr, i:int | m':mgr) {  
  b:bool;  
  num:string;  
  i':int;  
  m'':mgr;  
  
  b := i < #nums;  
  if b  
  then num := nums[i];  
       send_rm(msg, num, m | m'');  
       i' := i+1;  
       msg_send'(msg, nums, m'', i' | m')  
  else move(m | m')  
  fi  
}
```

Instrumented send procedure

```
send_rm(msg:string, num:string, m:mgr | m':mgr) {  
  r:mset;  
  m'':mgr;  
  
  r := {from_string(num):1};  
  split(m, r | m', m'');  
  assert r in m'';  
  send(msg, num |);  
  consume(m'' |)  
}
```

Effect types

Could still just use runtime checks, and get block booking (Aspinall et al. REM '07).

But static types prevent runtime errors, give programmer feedback, can be independently checked, etc.

Use *effect* type $\Phi \rightarrow \Psi$ where pre-post constraints Φ and Ψ are quantified FO formulae over expressions.

Type judgement

$\Gamma \vdash s : \Phi \rightarrow \Psi$ says that for all runs of s from state α with $\alpha \models \Phi$:

- the error \perp will not be raised, and
- if the execution terminates in a state β then $\beta \models \Phi \wedge \Psi$.

Judgement is relative to a program $Prog$ and an effect environment Eff .

Could still just use runtime checks, and get block booking (Aspinall et al. REM '07).

But static types prevent runtime errors, give programmer feedback, can be independently checked, etc.

Use *effect* type $\Phi \rightarrow \Psi$ where pre-post constraints Φ and Ψ are quantified FO formulae over expressions.

Type judgement

$\Gamma \vdash s : \Phi \rightarrow \Psi$ says that for all runs of s from state α with $\alpha \models \Phi$:

- the error \perp will not be raised, and
- if the execution terminates in a state β then $\beta \models \Phi \wedge \Psi$.

Judgement is relative to a program $Prog$ and an effect environment Eff .

skip

$$\overline{\Gamma \vdash \text{skip} : \top \rightarrow \top}$$

assertion

$$\overline{\Gamma \vdash \text{assert } x_1 \text{ in } x_2 : (x_1 \subseteq x_2) \rightarrow \top}$$

assignment

$$\overline{\Gamma \vdash y := e : \top \rightarrow (y = e)}$$

Effect types for built-in procedures

`empty(| m : mgr)`

- $Eff(\text{empty}) = \top \rightarrow (m = \emptyset)$

`enable(r_1 : mset | m : mgr, r_2 : mset)`

- $Eff(\text{enable}) = \top \rightarrow (r_1 = m \uplus r_2)$

`split(m : mgr, r : mset | m_1 : mgr, m_2 : mgr)`

- $Eff(\text{split}) = \top \rightarrow (m_2 = m \cap r) \wedge (m = m_1 \uplus m_2)$

`join(m_1 : mgr, m_2 : mgr | m : mgr)`

- $Eff(\text{join}) = \top \rightarrow (m = m_1 \uplus m_2)$

`consume(m : mgr |)`

- $Eff(\text{consume}) = \top \rightarrow \top$

procedure call

$$\frac{\begin{array}{l} \text{Prog}(p) = p(x_1 : \sigma_1, \dots, x_m : \sigma_m | y_1 : \tau_1, \dots, y_n : \tau_n) [\{\dots\}] \\ \text{Eff}(p) = \Phi \rightarrow \Psi \quad \Phi' = \Phi(x'_i/x_i) \quad \Psi' = \Psi(x'_i, y'_i/x_i, y_i) \end{array}}{\Gamma \vdash p(x'_1, \dots, x'_m | y'_1, \dots, y'_n) : \Phi' \rightarrow \Psi'}$$

sequential composition

$$\frac{\Gamma \vdash s_1 : \Phi \rightarrow \Psi_1 \quad \Gamma \vdash s_2 : \Phi \wedge \Psi_1 \rightarrow \Psi_2}{\Gamma \vdash s_1; s_2 : \Phi \rightarrow \Psi_1 \wedge \Psi_2}$$

conditional branching

$$\frac{\Gamma \vdash s_1 : z \wedge \Phi \rightarrow \Psi \quad \Gamma \vdash s_2 : \neg z \wedge \Phi \rightarrow \Psi}{\Gamma \vdash \text{if } z \text{ then } s_1 \text{ else } s_2 \text{ fi} : \Phi \rightarrow \Psi}$$

weakening

$$\frac{\Gamma \vdash s : \Phi \rightarrow \Psi \quad \begin{array}{l} \Phi' \models \Phi \\ (\Phi \Rightarrow \Psi) \models (\Phi' \Rightarrow \Psi') \end{array}}{\Gamma \vdash s : \Phi' \rightarrow \Psi'}$$

procedure body

$$\begin{array}{l} \text{Prog}(p) = p(x_1 : \sigma_1, \dots, x_m : \sigma_m | y_1 : \tau_1, \dots, y_n : \tau_n) \{z_1 : \rho_1, \dots, z_l : \rho_l; s\} \\ \Gamma = x_1 : \sigma_1, \dots, x_m : \sigma_m, y_1 : \tau_1, \dots, y_n : \tau_n, z_1 : \rho_1, \dots, z_l : \rho_l \\ \text{Eff}(p) = \Phi \rightarrow \Psi \\ \Gamma \vdash s : \Phi \rightarrow \Psi \\ \hline \text{Eff} \vdash p \end{array}$$

Big-step semantics $\alpha \vdash s \triangleright \beta$

- there is an execution of statement s starting in state α and terminating in state β .

Soundness Theorem

If

- $Eff \vdash p$ for all procedures in the program
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \vdash s \triangleright \beta$

then $\alpha \neq \perp \wedge \alpha \models \Phi$ implies $\beta \neq \perp \wedge \beta \models \Phi \wedge \Psi$.

Big-step semantics $\alpha \vdash s \triangleright \beta$

- there is an execution of statement s starting in state α and terminating in state β .

Soundness Theorem

If

- $Eff \vdash p$ for all procedures in the program
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \vdash s \triangleright \beta$

then $\alpha \neq \perp \wedge \alpha \models \Phi$ implies $\beta \neq \perp \wedge \beta \models \Phi \wedge \Psi$.

Erasing resource managers

- Erasure operation: $\text{mgr}^\circ = \text{unit}$, $(\text{assert } x_1 \text{ in } x_2)^\circ = \text{skip}$.
- After erasure, enable retains user/policy interaction:
 - $\text{Prog}^\circ(\text{enable}) = \text{enable}(r_1 : \text{mset} \mid m : \text{unit}, r_2 : \text{mset})$
 - $\text{Eff}^\circ(\text{enable}) = \top \rightarrow (r_2 \subseteq r_1)$.

Erasure Theorem

Resource managers can be safely erased from well-typed programs: if

- $\text{Eff} \vdash p$ for all procedures in the program Prog ,
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \neq \downarrow \wedge \alpha \Vdash \Phi$

then for all states β , $\alpha \vdash s \triangleright \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \triangleright \beta^\circ$.

Corollary

Resource managers cannot be used as covert channels.

Erasing resource managers

- Erasure operation: $\text{mgr}^\circ = \text{unit}$, $(\text{assert } x_1 \text{ in } x_2)^\circ = \text{skip}$.
- After erasure, `enable` retains user/policy interaction:
 - $\text{Prog}^\circ(\text{enable}) = \text{enable}(r_1 : \text{mset} \mid m : \text{unit}, r_2 : \text{mset})$
 - $\text{Eff}^\circ(\text{enable}) = \top \rightarrow (r_2 \subseteq r_1)$.

Erasure Theorem

Resource managers can be safely erased from well-typed programs: if

- $\text{Eff} \vdash p$ for all procedures in the program Prog ,
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \neq \downarrow \wedge \alpha \models \Phi$

then for all states β , $\alpha \vdash s \triangleright \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \triangleright \beta^\circ$.

Corollary

Resource managers cannot be used as covert channels.

Erasing resource managers

- Erasure operation: $\text{mgr}^\circ = \text{unit}$, $(\text{assert } x_1 \text{ in } x_2)^\circ = \text{skip}$.
- After erasure, `enable` retains user/policy interaction:
 - $\text{Prog}^\circ(\text{enable}) = \text{enable}(r_1 : \text{mset} \mid m : \text{unit}, r_2 : \text{mset})$
 - $\text{Eff}^\circ(\text{enable}) = \top \rightarrow (r_2 \subseteq r_1)$.

Erasure Theorem

Resource managers can be safely erased from well-typed programs: if

- $\text{Eff} \vdash p$ for all procedures in the program Prog ,
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \neq \downarrow \wedge \alpha \models \Phi$

then for all states β , $\alpha \vdash s \triangleright \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \triangleright \beta^\circ$.

Corollary





Resource managers cannot be used as covert channels.

- 1 Resource control
- 2 Type systems
- 3 Heap space analysis
- 4 Permission analysis
- 5 Summary

Specialized type systems can provide quantitative analysis of resource usage; with static checks suitable for proof-carrying code.

- Several kinds of resource: machine intrinsic; explicit in code; external.
- Type-based analysis: modular, declarative, certifying.
- Certification approaches: base logic proofs; admissible embedding of type systems; formalize correctness of analysis.
- Meta properties like *erasure* for resource managers: certified code may safely skip runtime checks.
- Particular resources: heap space, permissions, text message block booking.

Continuing work in Mobius includes: improving analyses and automation; handling more Java and Java bytecode; implementing certification; combining the approaches of logic, types, abstract interpretation and program analysis as sources of digital evidence for program behaviour.

-  E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini.
Cost analysis of Java bytecode.
Proc. ESOP 2007, LNCS 4421. Springer-Verlag, 2007.
-  D. Aspinall, P. Maier, and I. Stark.
Monitoring external resources in Java MIDP.
Proc. REM 2007: Run Time Enforcement for Mobile and Distributed Systems, ENTCS. Elsevier, 2007.
-  L. Beringer and Martin Hofmann.
A bytecode logic for JML and types.
Proc. APLAS 2006, LNCS 4279. Springer-Verlag, 2006.
-  F. Besson, G. Dufay, and T. Jensen.
A formal model of access control for mobile interactive devices.
Proc. ESORICS 2006, LNCS 4189. Springer-Verlag, 2006.

Mobius: Mobility, Ubiquity and Security

Mobius is funded from 2005–2009 as project IST-015905 under the Global Computing II proactive initiative of the Future and Emerging Technologies objective in the Information Society Technologies priority of the European Commission's 6th Framework programme.

Disclaimer: This document reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein.