IsoMod: A Module System for Isolating Untrusted Software Extensions

Philip W. L. Fong

pwlfong@cs.uregina.ca

Department of Computer Science

University of Regina

Regina, Saskatchewan, Canada S4S 0A2

Overview

1. Motivation: Name Visibility Management
2. The ISOMOD Architecture and Policy Language
3. Sample Applications
4. On-going Work

Name Visibility Management

Secure Cooperation of Mutually Suspicious Code

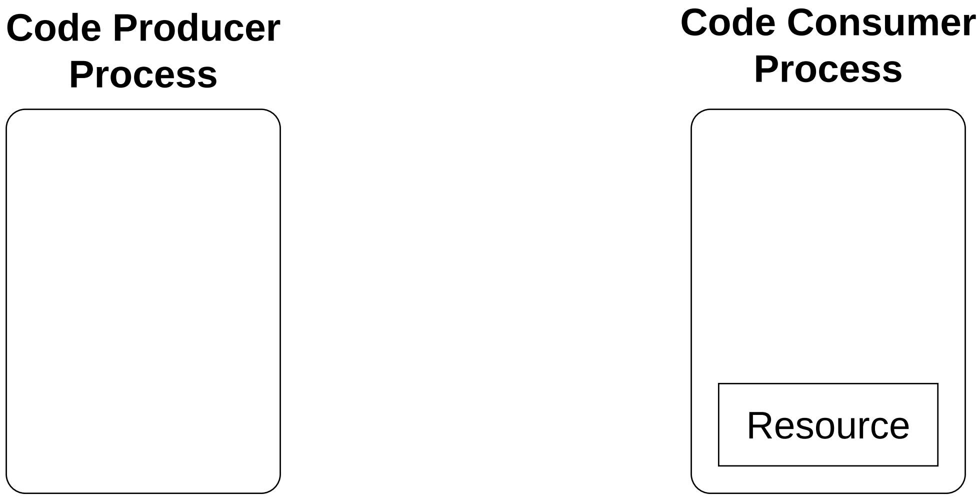
The Challenge of Secure Cooperation

Protecting mutually suspicious code units from one another while they are executing in the same run-time environment.

[Schroeder 1972, Rees 1996]

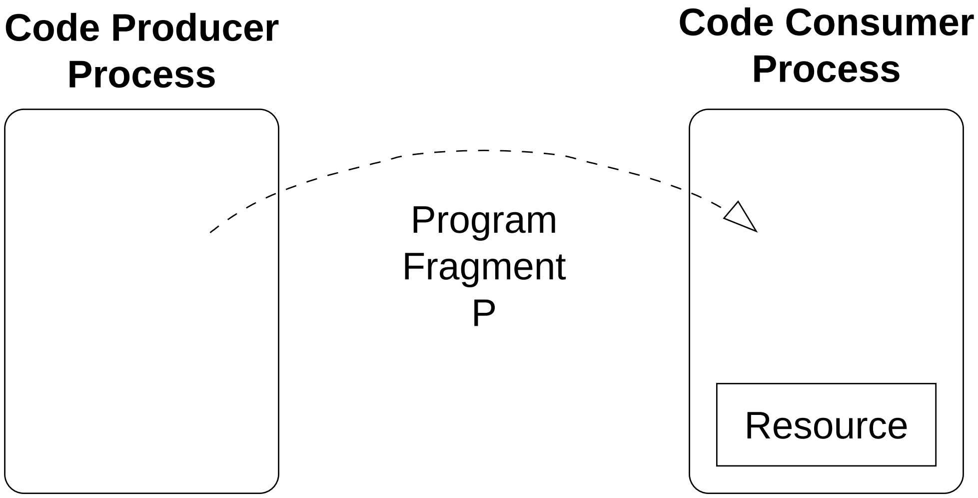
Dynamically Extensible Systems

- Dynamically-loaded software extensions



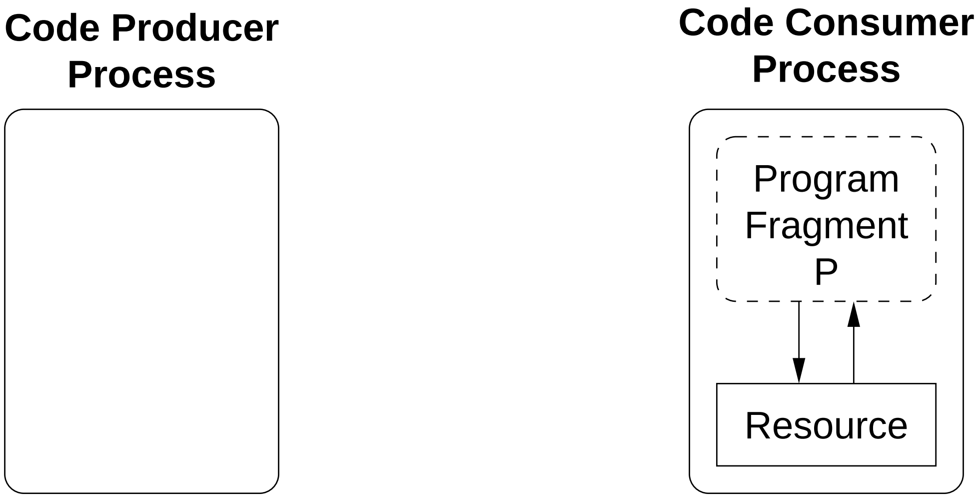
Dynamically Extensible Systems

- Dynamically-loaded software extensions



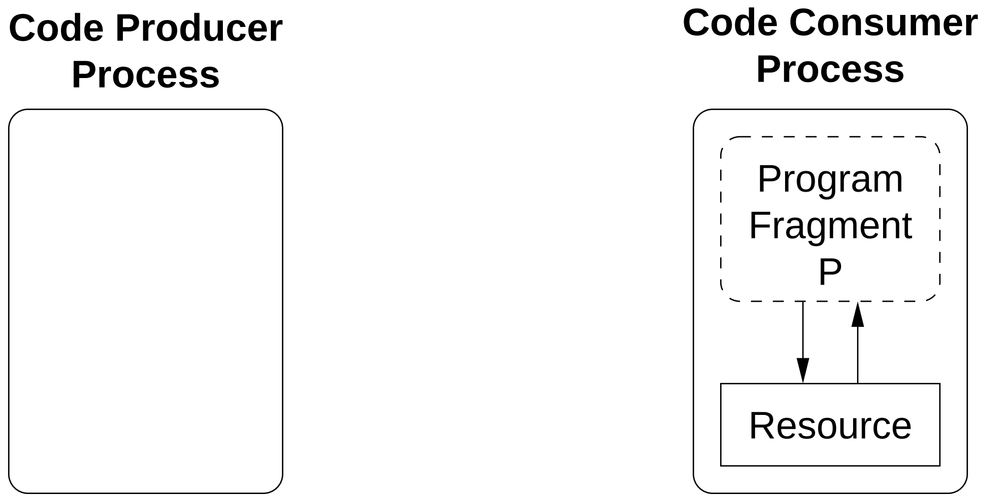
Dynamically Extensible Systems

- Dynamically-loaded software extensions



Dynamically Extensible Systems

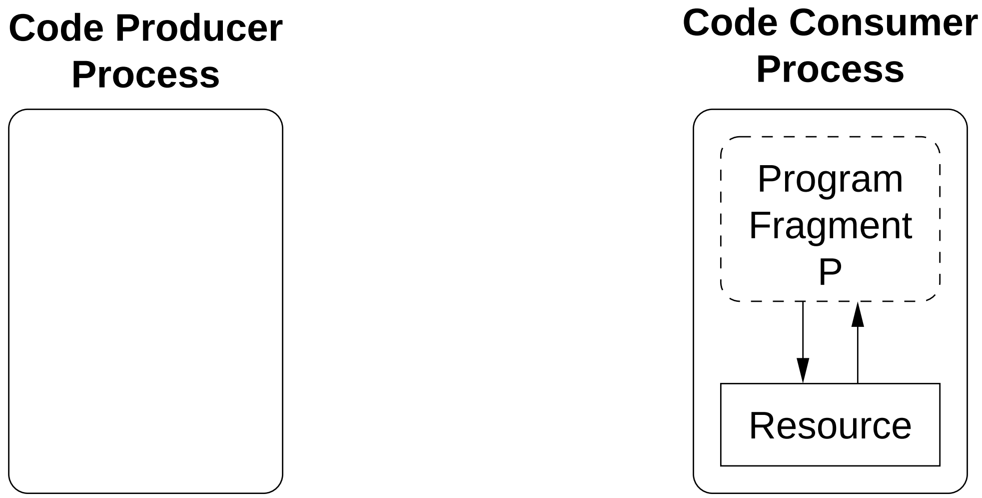
- Dynamically-loaded software extensions



- Examples
  - Mobile code platforms
  - Scriptable applications
  - Systems with plug-in architecture

Dynamically Extensible Systems

- Dynamically-loaded software extensions



- Examples
  - Mobile code platforms
  - Scriptable applications
  - Systems with plug-in architecture
- **Challenge:** *Secure Cooperation!*

Language-Based Security

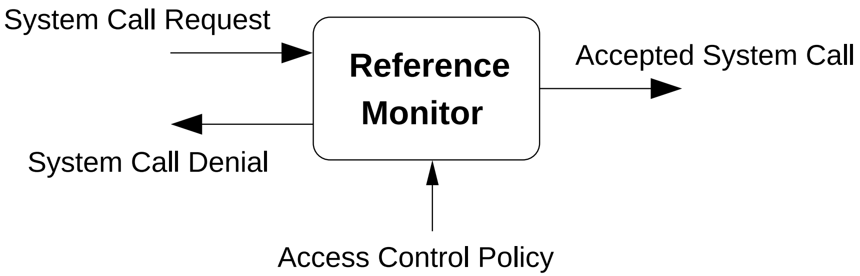
- Encode untrusted extensions in safe language
- Run untrusted code in secure run-time environment
- Protection mechanisms based on programming language technologies:
  - type systems
  - program rewriting
  - execution monitoring
- Examples
  - Java Virtual Machine (JVM)
  - Common Language Runtime (CLR)

Language-based Access Control

1. **Low-level:** Encapsulation via *visibility control*
   - e.g., public, protected, private
2. **High-level:** Execution monitoring via *interposition*
   - e.g., stack inspection, inlined reference monitors

Direct Interposition

- Execution monitoring via *interposition*:

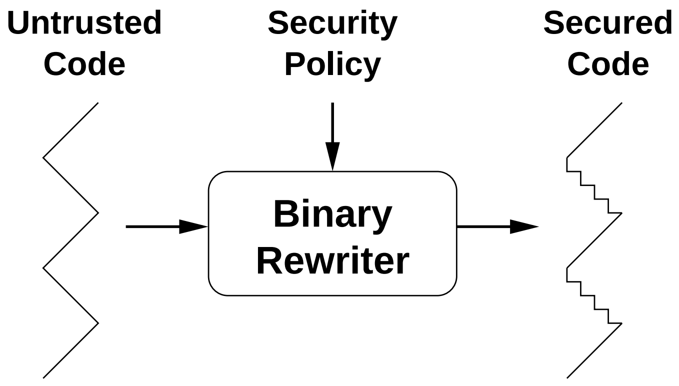


- Stack inspection [Wallach *et al* 2000]
  - Guard code examines call chain leading to the request
    - to avoid Confused Deputy [Hardy 1988]
  - **Problems:**
    - lack of declarative semantics
    - brittle in the face of evolving system configurations
      - guard code hard-coded into system

Inlined Reference Monitors

Inlined Reference Monitors [Erlingsson & Schneider 2000]

- Guard code is *weaved* into untrusted code by a trusted binary rewriter.



- **Pros:**
  - Policy maintained separately from system code
  - Good for evolving system configurations
- **Cons:**
  - Non-trivial run-time overhead
    [Wallach *et al* 2000, Erlingsson & Schneider 2000]

Question

- Don't always need full-fledged execution monitoring
  - tracking of execution history is not always needed
  - Confused Deputy is not always the major concern
- Can execution monitoring be complemented by a protection mechanism with the following properties?
  - lightweight
  - declarative characterization
  - copes with evolving system configuration gracefully

Name Visibility Management

Intuition If the name of a service isn't visible then it can't be accessed.

⇒ Run untrusted code in a name space that enforces name visibility policy

Name Visibility Policy

- what names are visible
- to whom they are visible
- to what extent they are visible

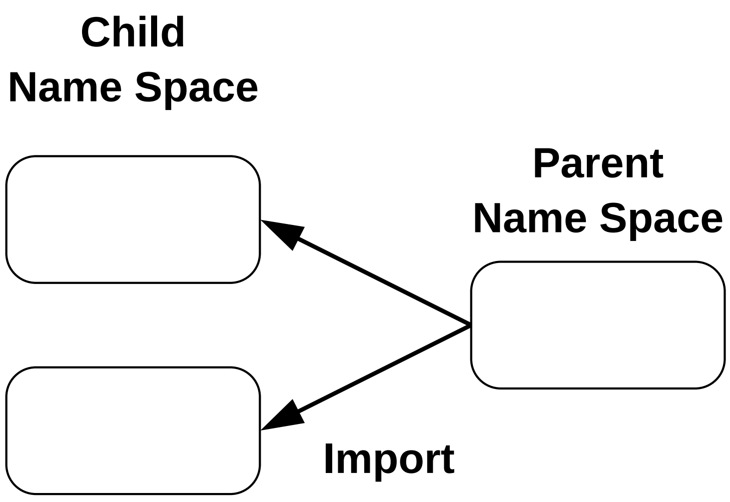
Goal To investigate the degree to which name visibility management can serve the purpose of access control when full-fledged execution monitoring is not necessary.

ISOMOD

- A module system for Java that manages the visibility of names in run-time name spaces
- ISOMOD name visibility policies are:
  1. enforced at class loading time
     $\Rightarrow$ no run-time overhead
  2. declarative and separately maintained
     $\Rightarrow$ disentangled from core system code
  3. expressive
     $\Rightarrow$ captures a rich family of access control policies

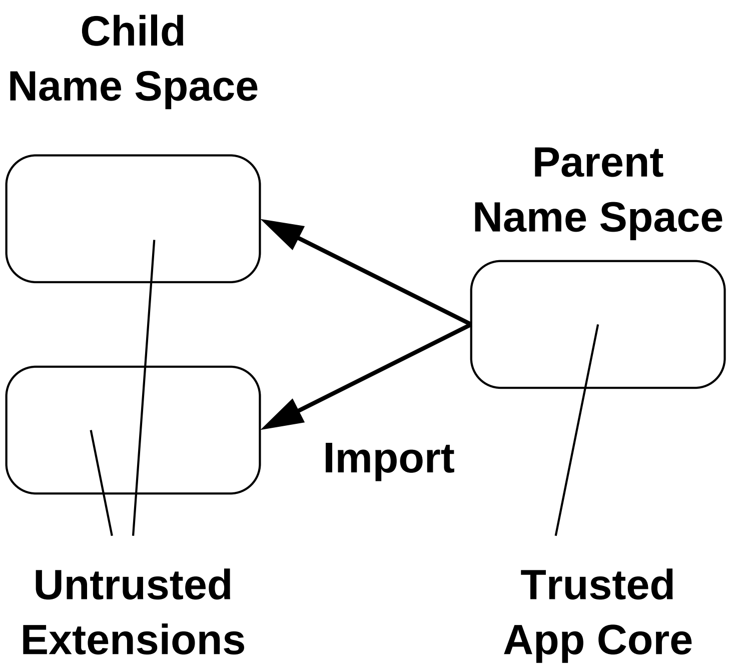
The IsoMod Architecture and Policy Language

Delegation-Style Class Loading in Java



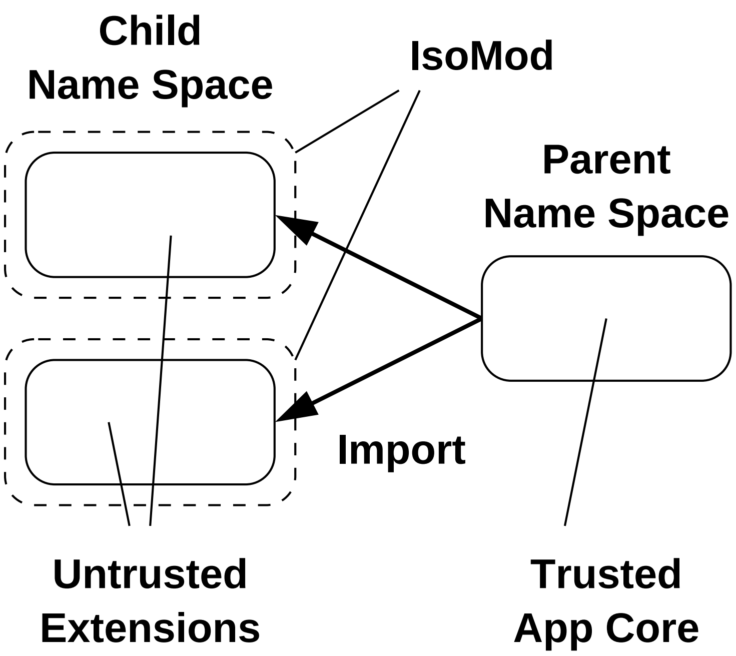
- class loader = run-time name space
- name space partitioning
- names from a parent name space are implicitly imported into its child name spaces

Dynamically Extensible Systems



- core application services are exposed to untrusted extensions via implicit import of names

Enter ISOMOD



- ISOMOD is a custom class loader . . .
  - configured with user-defined name visibility policy
  - enforces visibility restrictions on:
    1. imported names
    2. locally defined names

Now You See It... Now You Don't

- Visibility control can be exercised to:
  1. control *which* locally defined class may *"see"* a name, and
  2. present an *alternative, restricted view* of the entity to which a name is bound.

IsoMod Policy

- Scan classfile at load time to identify ***accesses***
  - *access* $= \langle$*subject*, *right*, *object*$\rangle$
  - e.g., $\langle$*method* $A.m$, *invoke*, *method* $B.n\rangle$

  A class is loaded into a name space only if its accesses are granted by the policy of the name space.
- An IsoMod policy is a list of policy clauses:

$$O \;\; (\textbf{grant}|\textbf{deny}) \;\; \{r_1, \ldots, r_k\} \;\; [\; \textbf{to} \;\; S \;] \;\; [\; (\textbf{when}|\textbf{unless}) \;\; c \;]$$

  - O and S may be universally quantified variables.
  - Condition c specifies a static relation between O and S.

Sample Applications

Sample Applications

1. Selective Hiding of System Services
2. Systematic Control of Reference Acquisition
3. Discretionary Capability Confinement

Selective Hiding of System Services (1)

- Simulating the **`getClassLoader`** permission of the Java 2 platform:

class `ClassLoader`

  method `getParent`

    deny { invoke }

  method `getSystemClassLoader`

    deny { invoke }

class `Class`

  method `getClassLoader`

    deny { invoke }

  method `forName(String,boolean,Classloader)`

    deny { invoke }

Selective Hiding of System Services (2)

- Most **`BasicPermission`**s defined in Java 2 can be simulated by IsoMod.
- Finer-grained than **`BasicPermission`**:

Example: What if we want to . . .

  - disallow the use of the Reflection API to invoke methods, access fields, and create class instances, but
  - permit the use of the Reflection API to examine class interface

Systematic Control of Reference Acquisition (1)

- Rethinking the **`getClassLoader`** permission . . .
  - What if the Java API is changed in the next release?
  - What if a platform extension library is installed?
  - What if an evolving application core exposes more ways to leak **`ClassLoader`** references?

$\Rightarrow$ exhaustive code audit to avoid leaking **`ClassLoader`** references.

Systematic Control of Reference Acquisition (1)

- Rethinking the **`getClassLoader`** permission . . .
  - What if the Java API is changed in the next release?
  - What if a platform extension library is installed?
  - What if an evolving application core exposes more ways to leak **`ClassLoader`** references?

$\Rightarrow$ exhaustive code audit to avoid leaking **`ClassLoader`** references.

Bad!

Systematic Control of Reference Acquisition (2)

class C

deny { new, cast, catch }

when *subclass*(C, **`ClassLoader`**)

field F

deny { get, put }

when *subclass*(*field-type*(F), **`ClassLoader`**)

method M

deny { invoke }

when *subclass*(*return-type*(M), **`ClassLoader`**)

method M

deny { invoke }

when exists A **in** *argument-types*(M) :

subclass(A, **`ClassLoader`**)

Discretionary Capability Confinement (1)

- **Discretionary Capability Confinement (DCC)** is a static type system for modeling capabilities in the JVM bytecode language. *[Pending submission]*
- Under mild conditions, DCC enforces classical confinement properties:
  - **No Theft**
  - **No Leakage**

  The two properties have been formally verified in the framework of **Featherweight JVM**. *[Under review]*
- DCC type rules can be completely encoded in a IsoMod policy.

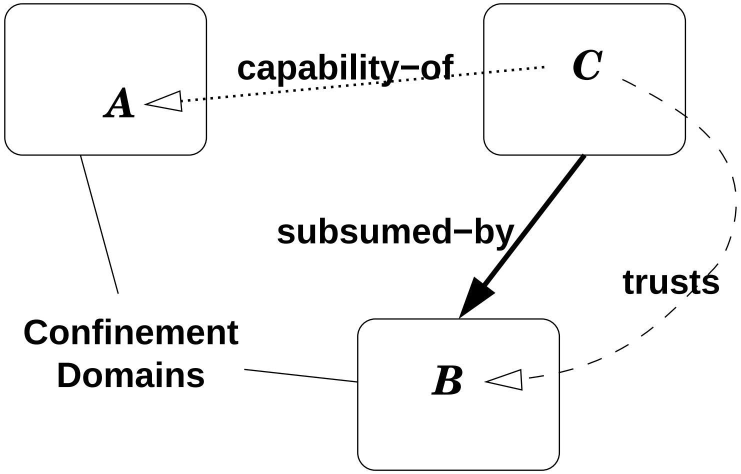
Discretionary Capability Confinement (2)

- **Intuition:** A statically typed reference specifies a pair:

$$\langle \textit{handle}, \textit{access rights} \rangle$$

$\Rightarrow$ Capability!

- **Trust** and **capabilities**:



- Write "$C \triangleright B$" to denote "*C trusts B.*"

Discretionary Capability Confinement (3)

($\mathcal{DCC}1$). Unless $B \triangleright A$, A shall not invoke a static method declared in B.

($\mathcal{DCC}2$). The sole means by which a domain acquires a capability is through argument passing.

($\mathcal{DCC}3$). If $A.m$ invokes $B.n$, and C is the type of a formal parameter of n, then $C \triangleright B \vee A \bowtie B \vee (B \triangleright m \wedge C \triangleright m)$.

($\mathcal{DCC}4$). A method m may invoke another method n only if $n \triangleright m$.

($\mathcal{DCC}5$). If $A <: B$ then $B \triangleright A$.

($\mathcal{DCC}6$). Suppose $B.n$ is overridden by $B'.n'$.

1. $n' \triangleright n$.
2. If the method return type is C, then $C \triangleright B \vee B \bowtie B'$.
3. If C is the type of a formal parameter, then $C \triangleright B' \vee B \bowtie B'$.

($\mathcal{DCC}7$). Suppose neither $A \triangleright B$ nor $B \triangleright A$. If $A' <: A$ and $B' <: B$, then neither $A' \triangleright B'$ nor $B' \triangleright A'$.

Discretionary Capability Confinement (4)

$(\mathcal{DCC}3)$. If $A.m$ invokes $B.n$, and C is the type of a formal parameter of n, then $C \triangleright B \vee A \bowtie B \vee (B \triangleright m \wedge C \triangleright m)$.

class B
  method n
    deny { invoke } **to** $A.m$
      when for C **in** *argument-types*(n) :
        trusts(C, B) **or**
        (*trusts*(A, B) **and** *trusts*(B, A)) **or**
        (*trusts*(B, m) **and** *trusts*(C, m))

On-going Work

Implementation Experience

- Master's student: Simon Orr
  - Pure Java implementation of IsoMod class loader
  - To be open-sourced
  - Extensive built-in predicates, functions and access rights
  - User-defined predicates/functions
  - XML encoding of IsoMod policies
  - Over 200 Java classes
  - Encouraging performance figures

Enforcing Communication Integrity

- Master's student: Jason Zhang
  - Ensuring untrusted software extensions conform to the architectural constraints of the application core.
  - Architectural constraints under consideration:
    1. Encapsulation policies [Schärli *et al* 2004]
    2. Module systems
    3. Software architectures: components, ports and connectors
    4. Layers, facade, etc
  - **Idea:** compiling a high-level architectural description language into ISOMOD policies.

Summary

- IsoMod
- Discretionary Capability Confinement
- Featherweight JVM
- Communication Integrity via IsoMod

Thank You