Viewer Discretion

Language-Based Protection Mechanisms for Dynamically Extensible Systems

Philip W. L. Fong

pwlfong@cs.uregina.ca

Department of Computer Science

University of Regina

Regina, Saskatchewan, Canada

Dynamically Extensible Systems

Dynamically Extensible Systems

- **Dynamically Extensible Systems**

  Executable extensions are dynamically linked into the address space of a software system, either to deliver a short-lived service, or to augment the capability of the host system in a permanent manner.

Dynamically Extensible Systems

- **Dynamically Extensible Systems**

  Executable extensions are dynamically linked into the address space of a software system, either to deliver a short-lived service, or to augment the capability of the host system in a permanent manner.

- Examples:
  - Mobile code systems
  - Scriptable applications
  - Software systems with plug-in architectures

Dynamically Extensible Systems

- **Dynamically Extensible Systems**

  Executable extensions are dynamically linked into the address space of a software system, either to deliver a short-lived service, or to augment the capability of the host system in a permanent manner.

- Examples:
  - Mobile code systems
  - Scriptable applications
  - Software systems with plug-in architectures

- The most challenging form of dynamically extensible systems are those that dynamically download and execute foreign code.

Mobile Code Systems

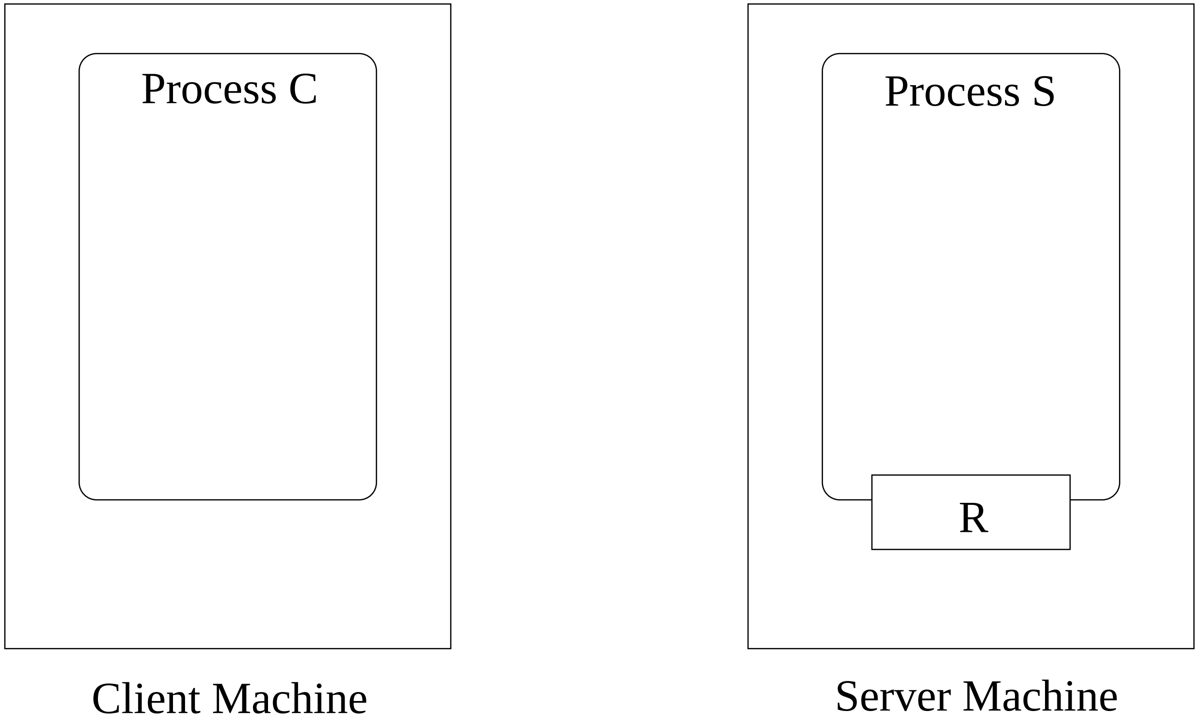
- Two paradigms for structuring distributed systems:
  1. **Client-Server Systems**

$$\text{Client} \xrightarrow{\text{Request}} \text{Server}$$

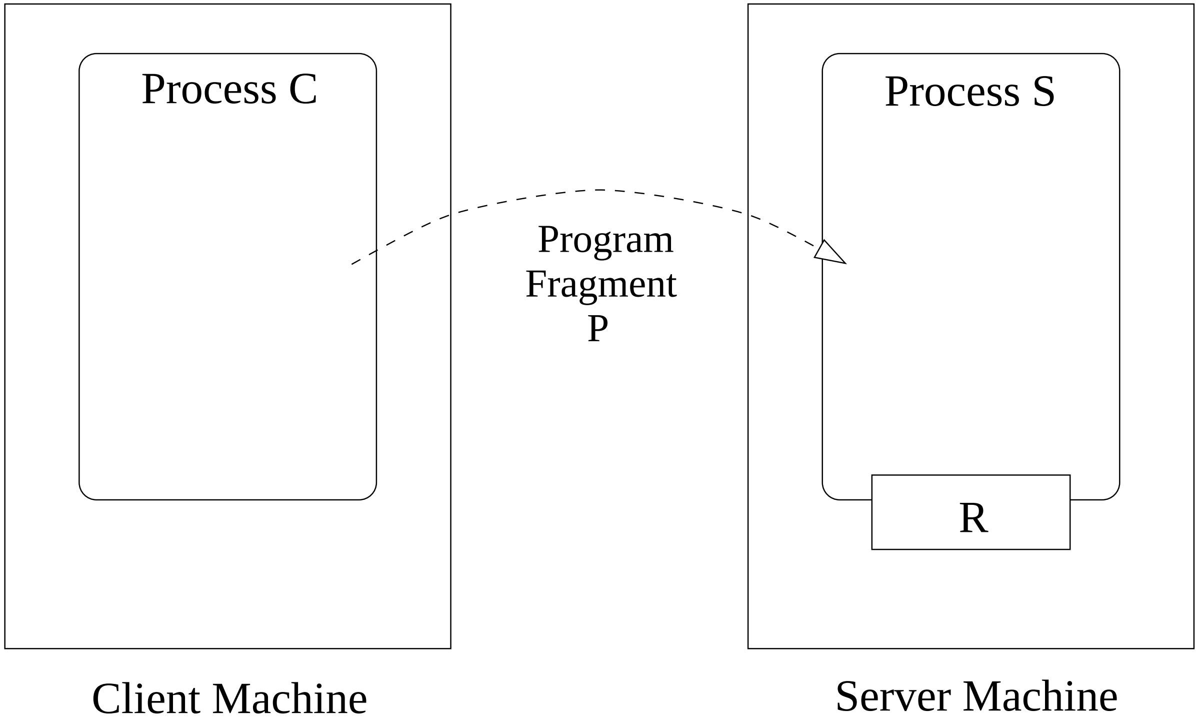
  2. **Mobile Code Systems**

$$\text{Code Producer} \xrightarrow{\text{Code Unit}} \text{Code Consumer}$$

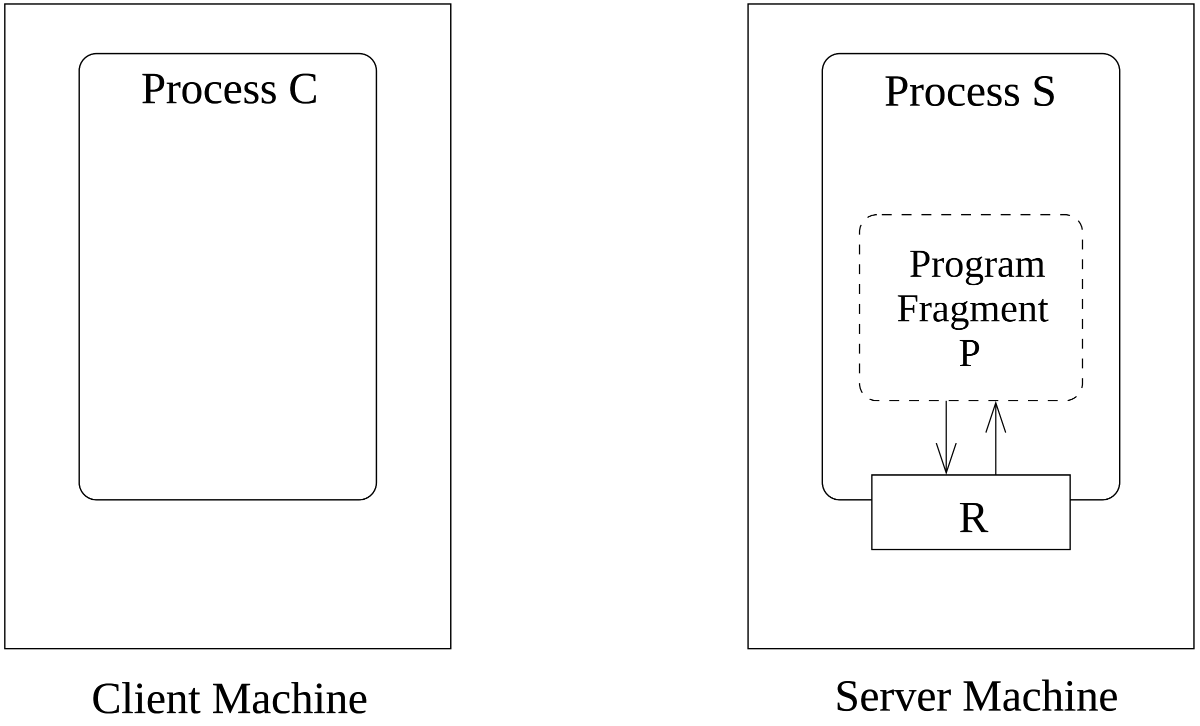
Code Mobility



Code Mobility



Code Mobility



Motivation of Code Mobility

1. **Extension of system capabilities**
   Example: Active network
2. **Real-time interaction with remote resources**
   Example: Java applets
3. **Reduction of communication traffic**
   Example: Active disks
4. **Avoiding distribution of state**

Security Challenges of Dynamically Extensible Systems

The Grand Challenge

Subject only to *time-bounded, automated checking*, code units originating from *any arbitrary source* collaborate with one another in the *same address space*.

1. **Anonymous trust**

   . . . any arbitrary source . . .

2. **Mutual suspicion**

   . . . same address space.

3. **Implicit acquisition**

   . . . time-bounded, automated checking . . .

Anonymous Trust

- Traditional discretionary access control is based on trusted identities.
- **Fallacy of the "*Identity Assumption*" [Chess 1998]**

  The most important assumption that mobile code systems violate is:

  > *Whenever a program attempts some action, we can easily identify a person to whom that action can be attributed, and it is safe to assume that that person intends the action to be taken.*

  For all intents and purposes, that is, every program that you run may be treated as though it were an extension of yourself.

Anonymous Trust

Anonymous Trust How can a host system establish trust for a code unit originating from an unknown origin and developed by an unknown party?

Mutual Suspicion

- **Assumption of “*Benign Peers*”:**

  However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread’s stacks. This [multi-threading] structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.

From a standard OS textbook [Silberschatz and Galvin 1994].

Mutual Suspicion

- Because peer code units may originate from arbitrary sources, resource-sharing peers may not trust one another.
- **Mutual Suspicion** How can protection be established among mutually suspicious code units residing in the same address space?
- Also called the ***secure cooperation*** problem [Rees 1996].

Implicit Acquisition

- Software is traditionally acquired through a gradual, manual, and explicit process.
- **Fallacy of the "Explicit Acquisition Assumption" [De Paoli *et al* 1998]**:

  Conventional computing paradigms assume that programs are installed and configured once on any and every machine and that these programs only exchange data. This means that a user can make all possible checks over a new program before running it. This assumption, however, is no longer valid for open and mobile environments, such as Java and the web.

Implicit Acquisition

Implicit Acquisition In the absence of an explicit acquisition process, how can trust be established automatically within a limited time frame?

The Language-Based Approach to Protection

Language-Based Security

- **Language-Based Security**
  - Employing programming language technologies to address the security challenges of dynamically extensible systems.

Language-Based Security

- **Language-Based Security**
  - Employing programming language technologies to address the security challenges of dynamically extensible systems.
- **Protection Mechanisms**
  - Static analysis & program verification
  - Execution monitoring
  - Program transformation

Example: The Java Platform

- Java is an archetypical language-based system.
  - Low-level memory protection
  - High-level access control

Java: Low-Level Memory Protection

- Unforgeable, strongly-typed object references.

Java: Low-Level Memory Protection

- Unforgeable, strongly-typed object references.
  - Java programs are compiled into strongly-typed bytecode.
    - Bytecode programs annotated with source-level type information.

Java: Low-Level Memory Protection

- Unforgeable, strongly-typed object references.
  - Java programs are compiled into strongly-typed bytecode.
    - Bytecode programs annotated with source-level type information.
  - Bytecode programs executed in Java Virtual Machine (JVM).
    1. Link-time bytecode verification
       - Type checking through data flow analysis
    2. Type-safe dynamic linking
    3. Run-time checks:
       - Array bounds checks.
       - Null reference checks.
       - Checked type casting.

Java: High-Level Access Control

- Stack inspection [Wallach *et al* 1998, Gordon & Fournet 2002]

Java: High-Level Access Control

- Stack inspection [Wallach *et al* 1998, Gordon & Fournet 2002]
  - Every class is assigned a set of access rights
    - Assignment is based on code source or digital signatures.
    - User may define custom access control policies.
    - When a stack frame is created in the run-time stack, it inherits the access rights of the class in which the invoked method is declared.

Java: High-Level Access Control

- Stack inspection [Wallach *et al* 1998, Gordon & Fournet 2002]
  - Every class is assigned a set of access rights
    - Assignment is based on code source or digital signatures.
    - User may define custom access control policies.
    - When a stack frame is created in the run-time stack, it inherits the access rights of the class in which the invoked method is declared.
  - To decide if an access is to be granted:
    - Every stack frame in the current run-time stack must be granted the required access right.
    - *Why?* To guard against the *Confused Deputy Problem*.

Java: High-Level Access Control

- Stack inspection [Wallach *et al* 1998, Gordon & Fournet 2002]
  - Every class is assigned a set of access rights
    - Assignment is based on code source or digital signatures.
    - User may define custom access control policies.
    - When a stack frame is created in the run-time stack, it inherits the access rights of the class in which the invoked method is declared.
  - To decide if an access is to be granted:
    - Every stack frame in the current run-time stack must be granted the required access right.
    - *Why?* To guard against the *Confused Deputy Problem*.
  - Provision for access rights amplification:
    - A method may annotate its stack frame to grant an access right to all the preceding frames.

Proof-Carrying Code

Proof-Carrying Code

Proof-Carrying Code (PCC) [Necula & Lee 1996, Necula 1997]

- Application of Floyd-style program verification to memory protection in native code.
  - Verified code runs in full speed
- Safety policies encoded in first-order logic
  - Edinburgh Logical Framework (LF)

Proof-Carrying Code

Proof-Carrying Code (PCC) [Necula & Lee 1996, Necula 1997]

- Application of Floyd-style program verification to memory protection in native code.
  - Verified code runs in full speed
- Safety policies encoded in first-order logic
  - Edinburgh Logical Framework (LF)
- Applications:
  - OS kernel extensions
  - Type-safe assembly language
  - Java bytecode
  - Mobile agents

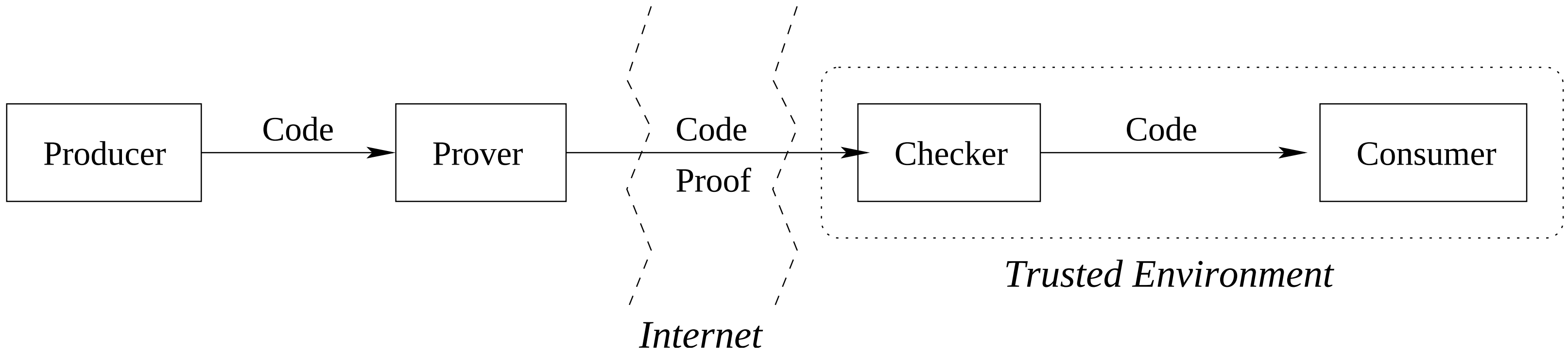
Proof-Carrying Code

Proof-Carrying Code (PCC) [Necula & Lee 1996, Necula 1997]

- Application of Floyd-style program verification to memory protection in native code.
  - Verified code runs in full speed
- Safety policies encoded in first-order logic
  - Edinburgh Logical Framework (LF)
- Applications:
  - OS kernel extensions
  - Type-safe assembly language
  - Java bytecode
  - Mobile agents
- *Question:* Who is to perform verification?

Proof-Carrying Code

- Interactive proof system:



  - Resilient to malicious code generator or tampering.
  - No cryptography or trusted certification authorities are necessary.

Proof-Carrying Code

- Efficient proof checking:

- Efficient proof generation:

Proof-Carrying Code

- Efficient proof checking:
  - *Code Producer*:
    1. Generate verification condition for code
    2. Generate proof of verification condition
    3. Ship both code and proof
- Efficient proof generation:

Proof-Carrying Code

- Efficient proof checking:
  - *Code Producer*:
    1. Generate verification condition for code
    2. Generate proof of verification condition
    3. Ship both code and proof
  - *Code Consumer*:
    1. Acquire proof-carrying code
    2. Generate verification condition for code
    3. Check if proof establishes verification condition
- Efficient proof generation:

Proof-Carrying Code

- Efficient proof checking:
  - *Code Producer*:
    1. Generate verification condition for code
    2. Generate proof of verification condition
    3. Ship both code and proof
  - *Code Consumer*:
    1. Acquire proof-carrying code
    2. Generate verification condition for code
    3. Check if proof establishes verification condition
  - *Intuition*:

    Proof checking faster than proof generation
- Efficient proof generation:

Proof-Carrying Code

- Efficient proof checking:
  - *Code Producer*:
    1. Generate verification condition for code
    2. Generate proof of verification condition
    3. Ship both code and proof
  - *Code Consumer*:
    1. Acquire proof-carrying code
    2. Generate verification condition for code
    3. Check if proof establishes verification condition
  - *Intuition*:

    Proof checking faster than proof generation
- Efficient proof generation:
  - *Certifying compiler* [Necula *et al* 1998/2000].

Type Systems for Information Flow Control

Information Flow Control

- Multilevel security [Bell & LaPadula 1973]
  - *e.g.*, unclassified, restricted, confidential, secret, top secret
- Information flow control
  - Simple Security Property (No read up)
  - *-Property (No write down)

Information Flow Control

- Multilevel security [Bell & LaPadula 1973]
  - *e.g.,* unclassified, restricted, confidential, secret, top secret
- Information flow control
  - Simple Security Property (No read up)
  - *-Property (No write down)
- Application to program certification [Denning & Denning 1977]
  - Every expression is statically assigned a security label.
  - Static analysis to ensure no information flow from high security values to low security variables.

Information Flow Control

- Explicit flow:
- Implicit flow:

Information Flow Control

- Explicit flow:

$$X := H + 1;$$
$$L := X - 1;$$

- Implicit flow:

Information Flow Control

- Explicit flow:

  $X := H + 1$;
  $L := X - 1$;

- Implicit flow:

  if $H > 0$ **then**
  $\quad L := 1$;
  else
  $\quad L := 0$;
  end if

Information Flow Control

- Explicit flow:

  $X := H + 1$;
  $L := X - 1$;

- Implicit flow:

if $H > 0$ **then**
    $L := 1$;
else
    $L := 0$;
end if

$L := 0$;
while $H > 0$ **do**
    $H := H - 1$;
    $L := L + 1$;
end while

Information Flow Control

- Explicit flow:

  $X := H + 1;$
  $L := X - 1;$

- Implicit flow:

if $H > 0$ **then**
    $L := 1;$
else
    $L := 0;$
end if

$L := 0;$
while $H > 0$ **do**
    $H := H - 1;$
    $L := L + 1;$
end while

- Reasoning about information flow is equivalent to performing dependency analysis [Abadi *et al* 1999].
  - "Does the value of L depends on H?"

Information Flow Control

- Volpano & Smith (1996) were the first to establish the *soundness* of an information flow type system for a core procedural language.

Information Flow Control

- Volpano & Smith (1996) were the first to establish the *soundness* of an information flow type system for a core procedural language.
  1. Define a security policy under a standard operational semantics:

     Non-interference *A variation of confidential (high) input does not cause a variation of public (low) output.*
  2. Define an information flow type system.
  3. Prove that all well-typed programs observe the security policy.

Information Flow Control

- Recent trends in type-based information flow control [Sabelfeld & Myers 2003]:
  1. Enriching language *expressiveness*
     e.g., procedures, functions, exceptions, objects
  2. Exploring the impact of *concurrency*
     e.g., non-determinism, multi-threading, distribution
  3. Analyzing *covert channels*
     e.g., timing channels, probabilistic channels
  4. Refining *security policies*
     e.g., downgrading

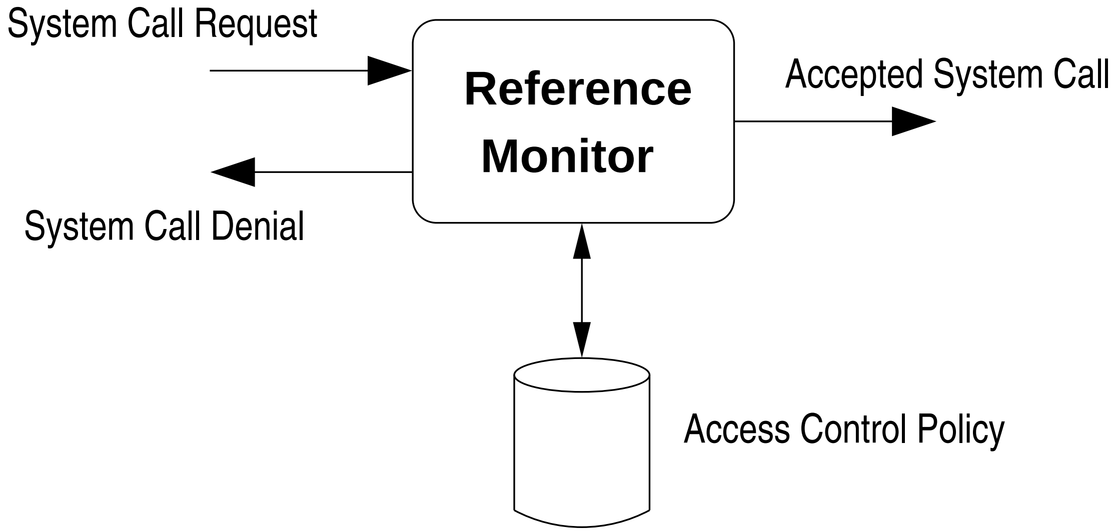
Information Flow Control

- Recent trends in type-based information flow control [Sabelfeld & Myers 2003]:
  1. Enriching language *expressiveness*
     e.g., procedures, functions, exceptions, objects
  2. Exploring the impact of *concurrency*
     e.g., non-determinism, multi-threading, distribution
  3. Analyzing *covert channels*
     e.g., timing channels, probabilistic channels
  4. Refining *security policies*
     e.g., downgrading
- A Java implementation of information flow control is Jif [Myers 1999].
  - `http:www.cs.cornell.edu/jif`

Inlined Reference Monitors

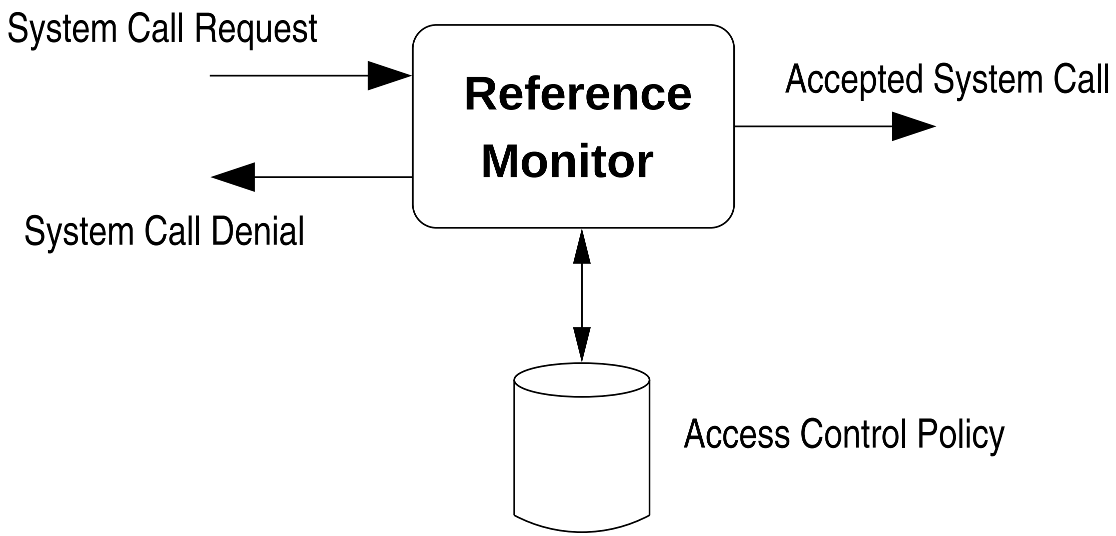
Reference Monitors

- Execution monitoring via *interposition*:



Reference Monitors

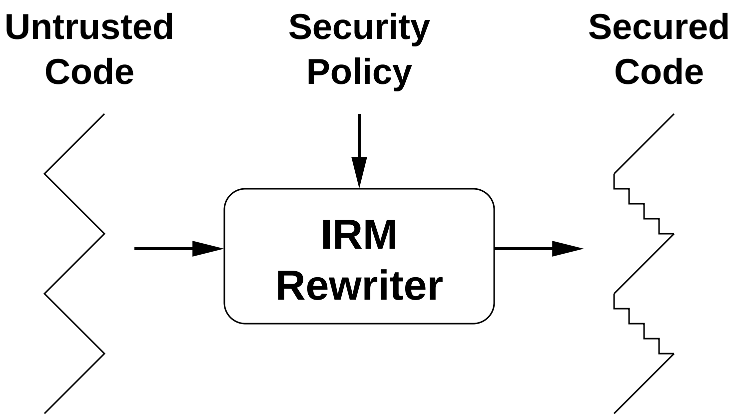
- Execution monitoring via *interposition*:



- **Problem:** Hard-coded into the host system.
  - Fail to account for the evolution of . . .
    - software configuration
    - security model

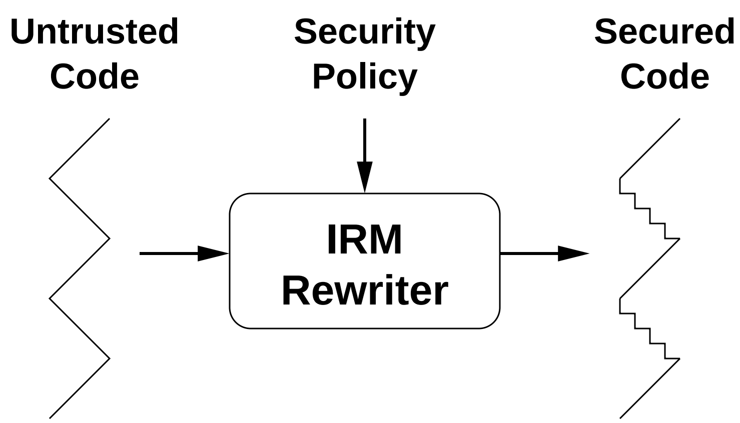
Inlined Reference Monitors

- Inlined Reference Monitors (IRM) [Erlingsson & Schneider 99/00]
  - Execution monitoring logic is *weaved* into untrusted code units by a trusted binary rewriter.



Inlined Reference Monitors

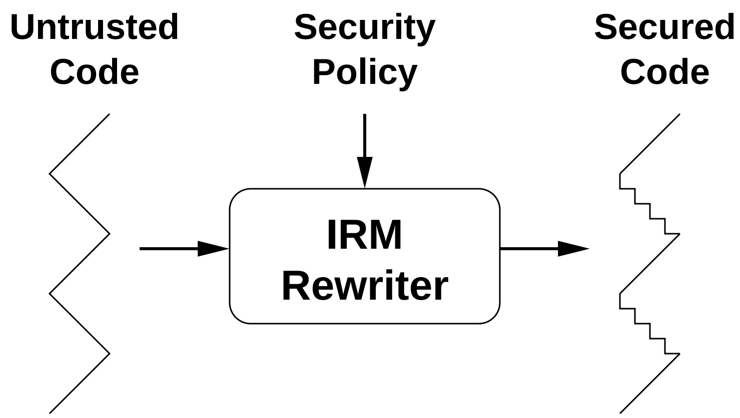
- Inlined Reference Monitors (IRM) [Erlingsson & Schneider 99/00]
  - Execution monitoring logic is *weaved* into untrusted code units by a trusted binary rewriter.



  - Administrated by code consumer $\Rightarrow$ non-bypassable.

Inlined Reference Monitors

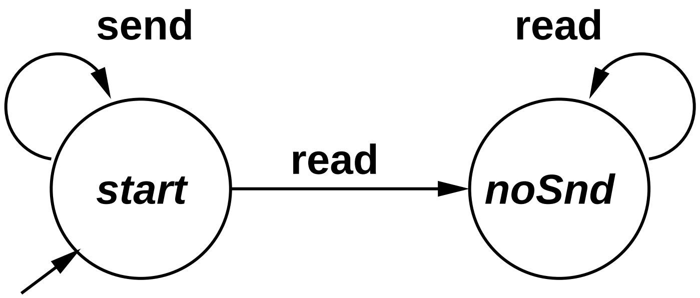
- Inlined Reference Monitors (IRM) [Erlingsson & Schneider 99/00]
  - Execution monitoring logic is *weaved* into untrusted code units by a trusted binary rewriter.



  - Administrated by code consumer $\Rightarrow$ non-bypassable.
  - 2 implementation options:
    - off-line rewriting
    - dynamic rewriting at load time

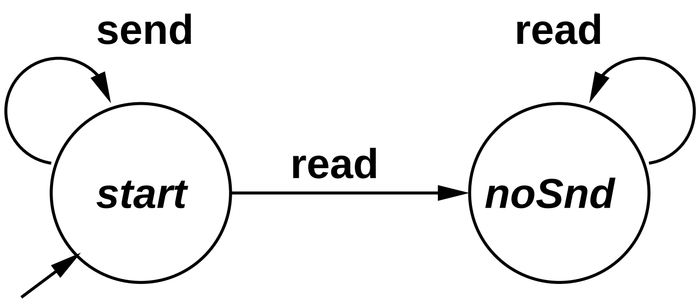
IRM: Security Automata

- Security policies are specified as *Security Automata* (SA) [Schneider 2000]:



IRM: Security Automata

- Security policies are specified as *Security Automata* (SA) [Schneider 2000]:



- Policy language PSLang is an improvement over a first-generation SA-based policy language.
  - Security events
  - Security states
  - Security updates

IRM: Java Stack Inspection

- Rewriting is applied to Java bytecode.
  - Source code not necessarily available
  - Compiler not part of the Trusted Computing Base
  - Interoperable with standard JVM.

IRM: Java Stack Inspection

- Rewriting is applied to Java bytecode.
  - Source code not necessarily available
  - Compiler not part of the Trusted Computing Base
  - Interoperable with standard JVM.
- Rewriter performs peephole optimization on generated code.
- General enough to enforce Java stack inspection.
  - an under-optimized implementation:
    - 3.0 – 72.5% slow down
  - a highly optimized implementation:
    - 0.4 – 6.4% slow down

Future Directions

Active Areas of Research

- More type systems for information flow control
- Secure program partitioning
- Characterization of enforceable policies
- Beyond stack inspection
- Access control type systems
- Detection and avoidance of software vulnerabilities