Shahriar Bijani

Information Leakage Analysis in Open Multi-agent Systems:

A Case Study in Cloud Computing

30 Novermber 2012

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Introduction

- Open Multi-agent Systems (MAS): open systems in which autonomous agents can join and leave freely.

- In this talk:
  Open (peer to peer) MAS that agents can invent protocols for different applications and share them.

- Open MAS have growing popularity (Poslad 07).

Introduction

- Security is a major practical limitation to open MAS.
- Security means: confidentiality, integrity and availability.
- Our Assumption: there exists confidential information in open MAS.

Introduction

- My thesis :
  - Study and categorise various attacks on open MAS
  - Review and classify different security solutions
  - Focus on information leakage in LCC-based systems
  - Propose an information flow security analysis based on a language-based approach
  - Case study of cloud configuration management

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Language-based Security

- a convenient complement to traditional security mechanisms
- Why?
  - **Access control** prevents information release…
  - **Encryption** could guarantee the origin, confidentiality and integrity of information, **but not its behaviour**.
  - A fundamental limitation: **can not prevent information from being propagated**

Language-based Security

- Sound type systems are a promising language-based technique to specify and enforce an information flow policy. (Sabelfeld & Myers, 2003)

- In type checking approach:
  - Every program term has a ***security type***
  - Security is enforced by **type checking**

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Lightweight Coordination Calculus (LCC)

- LCC is a declarative language to execute agents' organisational models in a peer to peer style.
- a choreography language based on pi-calculus and logic programming.

LCC Syntax

Interaction Model := {Clause,...}

Clause := Role::Def

Role := a(Type, Id)

Def := Role | Message | Def then Def | Def or Def | null ← Constraint

Message:= M Ĝ Role | M Ĝ Role ← Constraint | M Ě Role | Constraint ← M Ě Role

Constraint:= Constant | Term | Constraint Ę Constraint | Constraint ė Constraint

Type := Term

Id := Constant | Variable

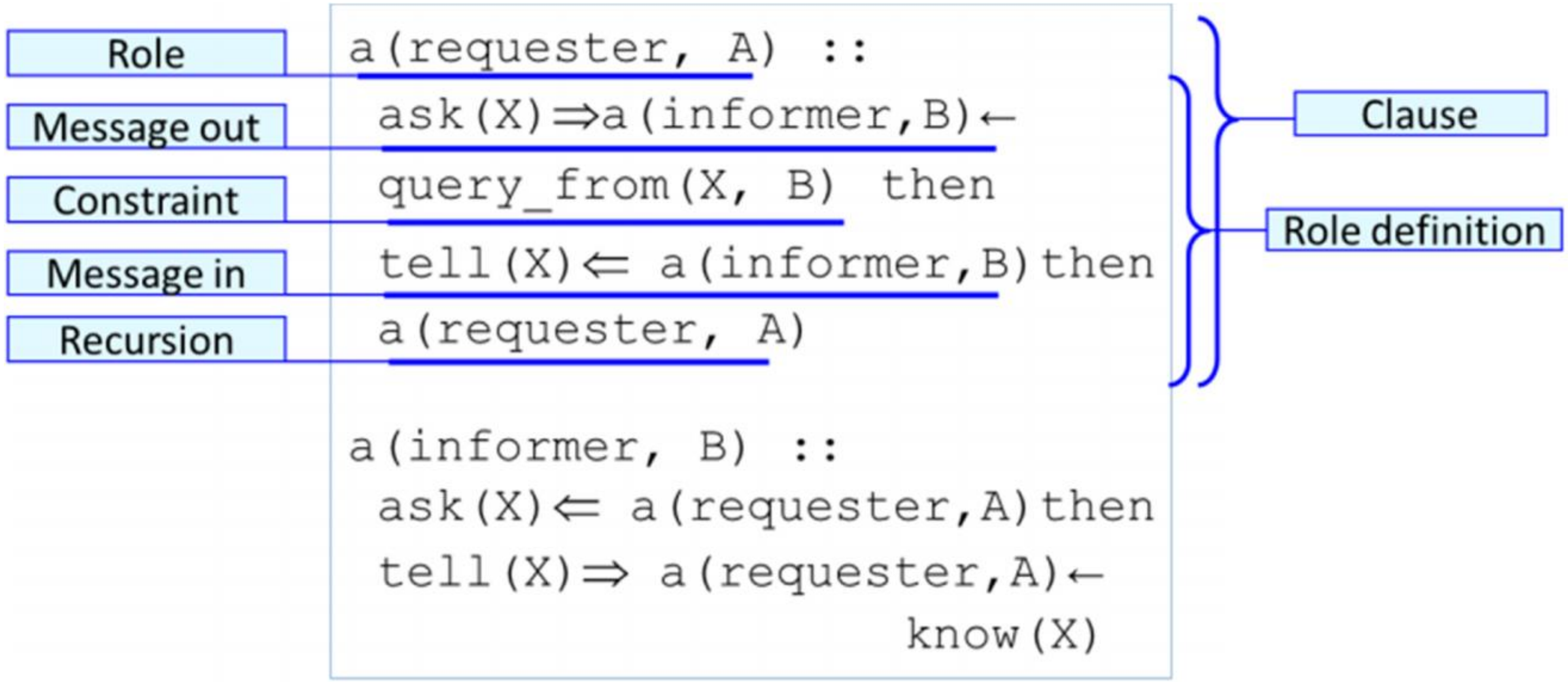
M := Term

Term := Constant | Variable | a structured term in Prolog syntax

Constant := lower case character sequence or number

Variable := upper case character sequence or number

A LCC Example



LCC language syntax:

Outgoing message: Ĝ

Incoming message: Ě

Conditional: ←

Sequence: **then**

Committed choice: **or**

Security Type System for LCC

- The rules are judgments of the form: $\Gamma \vdash T : \varphi$
- Γ : a type environment that maps a LCC term T to the type φ and its secrecy level.
- **Security types**: $\varphi = id\ \tau \ \mid \tau \ \mid op\ \tau \ \mid con\ \tau_1 / \tau_2$
- Variables have only type *id* ^{2} .
- Other terms have only type ^{2} .
- ***Def*** commands have only type $op\ \tau$.

```
Def := Role | Message | Def then Def| Def or Def | null <- Constraint
```

- **Constraint** expressions have only types $con\ \tau_1 / \tau_2$.

```
Constraint:= Constant | Term | Constraint ė Constraint | Constraint Ę Constraint
```

Security Levels

- Security levels:
  For simplicity it could be assumed that there are two levels of secrecy L (low) and H (high).

- Security levels are directly assigned to LCC terms by annotations in the code.

- the terms which are not annotated may be assigned to the highest security level.

Security Typing Rules for LCC

$$\frac{T:\tau \in \Gamma}{\Gamma \vdash T:\tau}\,Id \qquad \frac{\Gamma \vdash S:\tau_0,\ \Gamma \vdash t_1:\tau_1, \ldots, \Gamma \vdash t_n:\tau_n}{\Gamma \vdash S(t_1, \ldots, t_n): \tau_0 \vee \tau_1 \vee \ldots \vee \tau_n}\,Struct$$

$$\frac{\Gamma \vdash R:\tau,\ \Gamma \vdash ID: id\,\tau}{\Gamma \vdash a(R,ID): agent\,\tau}\,Agnt \qquad \frac{\Gamma \vdash a(R,ID): agent\,\tau,\ \Gamma \vdash E\ op\,\tau}{\Gamma \vdash a(R,ID) :: E :\ op\,\tau}\,Role$$

$$\frac{a(R,ID) :: Def,\ \Gamma \vdash a(R,ID)\ agent\,\tau}{\Gamma \vdash my_L: agent\,\tau}\,Self \qquad \frac{\Gamma \vdash C: con\,\tau'/\tau''}{\Gamma \vdash \neg C: op\,\tau'/\tau''}\,Not$$

$$\frac{\Gamma \vdash A : agent\,\tau,\ \Gamma \vdash M:\tau}{\Gamma \vdash M \Rightarrow A : op\,\tau}\,Snd \qquad \frac{\Gamma \vdash my_L: agent\,\tau,\ \Gamma \vdash M:\tau}{\Gamma \vdash M \Leftarrow A: op\,\tau}\,Rsv$$

$$\frac{\Gamma \vdash F:\tau,\ \Gamma \vdash t_1:\tau, \ldots, \Gamma \vdash t_k: id\,\tau, \ldots, \Gamma \vdash t_n:\tau}{\Gamma \vdash F(t_1, \ldots, t_k, \ldots, t_n): op\,\tau}\,Call$$

$$\frac{\Gamma \vdash C_1: con\,\tau'_1/\tau''_1, \Gamma \vdash C_2: con\,\tau'_2/\tau''_2}{\Gamma \vdash C_1 \wedge C_2 : con\,\tau'_1 \wedge \tau'_2/\tau''_1 \vee \tau''_2}\,And \qquad \frac{\Gamma \vdash C_1: con\,\tau'_1/\tau''_1, \Gamma \vdash C_2: con\,\tau'_2/\tau''_2}{\Gamma \vdash C_1 \vee C_2 : con\,\tau'_1 \wedge \tau'_2/\tau''_1 \vee \tau''_2}\,Or$$

$$\frac{\Gamma \vdash C: con\,\tau/\tau',\ \Gamma \vdash M \Leftarrow A: op\,\tau}{\Gamma \vdash C \leftarrow M \Leftarrow A: op\,\tau}\,If1 \qquad \frac{\Gamma \vdash C: con\,\tau'/\tau,\ \Gamma \vdash M \Rightarrow A: op\,\tau}{\Gamma \vdash M \Rightarrow A \leftarrow C: op\,\tau}\,If2$$

$$\frac{\Gamma \vdash null:\tau,\ \Gamma \vdash C: op\,\tau'/\tau}{\Gamma \vdash null \leftarrow C: op\,\tau}\,If3 \qquad \frac{\Gamma \vdash a(R,I): agent\,\tau,\ \Gamma \vdash C: con\,\tau'/\tau}{\Gamma \vdash a(R,I) \leftarrow C: op\,\tau}\,If4$$

$$\frac{\Gamma \vdash A_1: op\,\tau,\ \Gamma \vdash A_2: op\,\tau}{\Gamma \vdash A_1\ then\ A_2: op\,\tau}\,Seq \qquad \frac{\Gamma \vdash A_1: op\,\tau,\ \Gamma \vdash A_2: op\,\tau}{\Gamma \vdash A_1\ or\ A_2: op\,\tau}\,Choice$$

LCC Subtyping Rules

$$\varphi \leq \varphi \quad Reflex$$

$$\frac{\varphi_1 \leq \varphi_2 \, , \varphi_2 \leq \varphi_3}{\varphi_1 \leq \varphi_3} \; Trans$$

$$\frac{\Gamma \vdash T\colon \varphi, \;\; \varphi \leq \varphi'}{\Gamma \vdash T\colon \varphi'} Subsum$$

≤ means information flow is permitted from left to right

Information Flows in LCC

Source of illegal information flows in LCC:

- **Explicit flows** (operations are independent of the value of their terms)
  I. Message passing
  II. Role assignment
  III. Constraints

- **Implicit Flows** disclose some information through the program **control flow**.

Explicit Information Flows

- Permissible information flows in sending a message based on the security levels of the sender, the receiver and the message
- `Message => a(receiver, R)`

Sender	Receiver	Message	Permissible Flow
L	L	L	**Yes**
L	L	H	**No**
L	H	L	**Yes**
L	H	H	**No**
H	L	L	**Yes**
H	L	H	**No**
H	H	L	**Yes**
H	H	H	**Yes**

Explicit Information Flows

- Permissible information flows regarding the security levels of the role and the agent identifier
- `a(role, agentID)::`

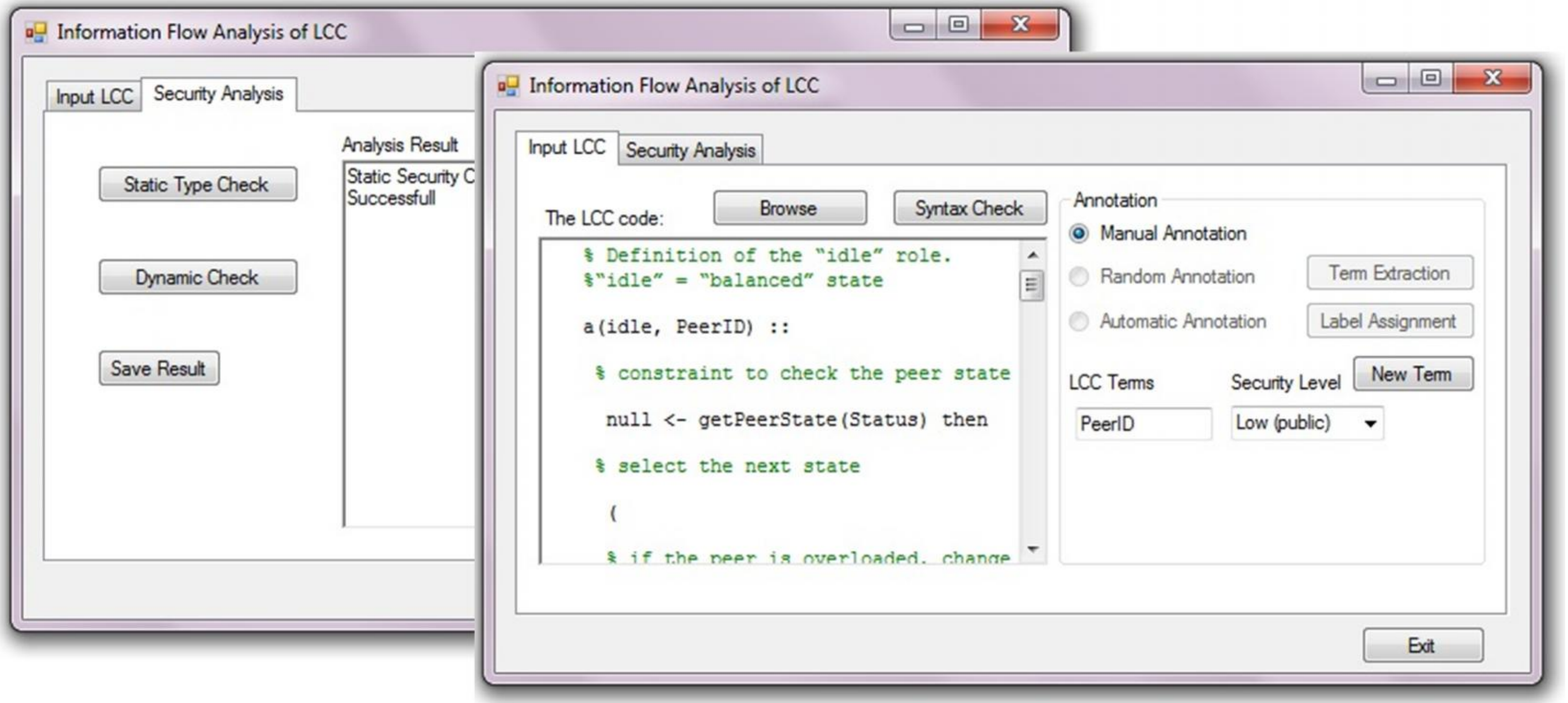
Agent Identifier	Role	Permissible Flow
L	L	**Yes**
L	H	**No**
H	L	**Yes**
H	H	**Yes**

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Implementation

- The type system has been implemented in Prolog
- A GUI prototype in C#.NET.

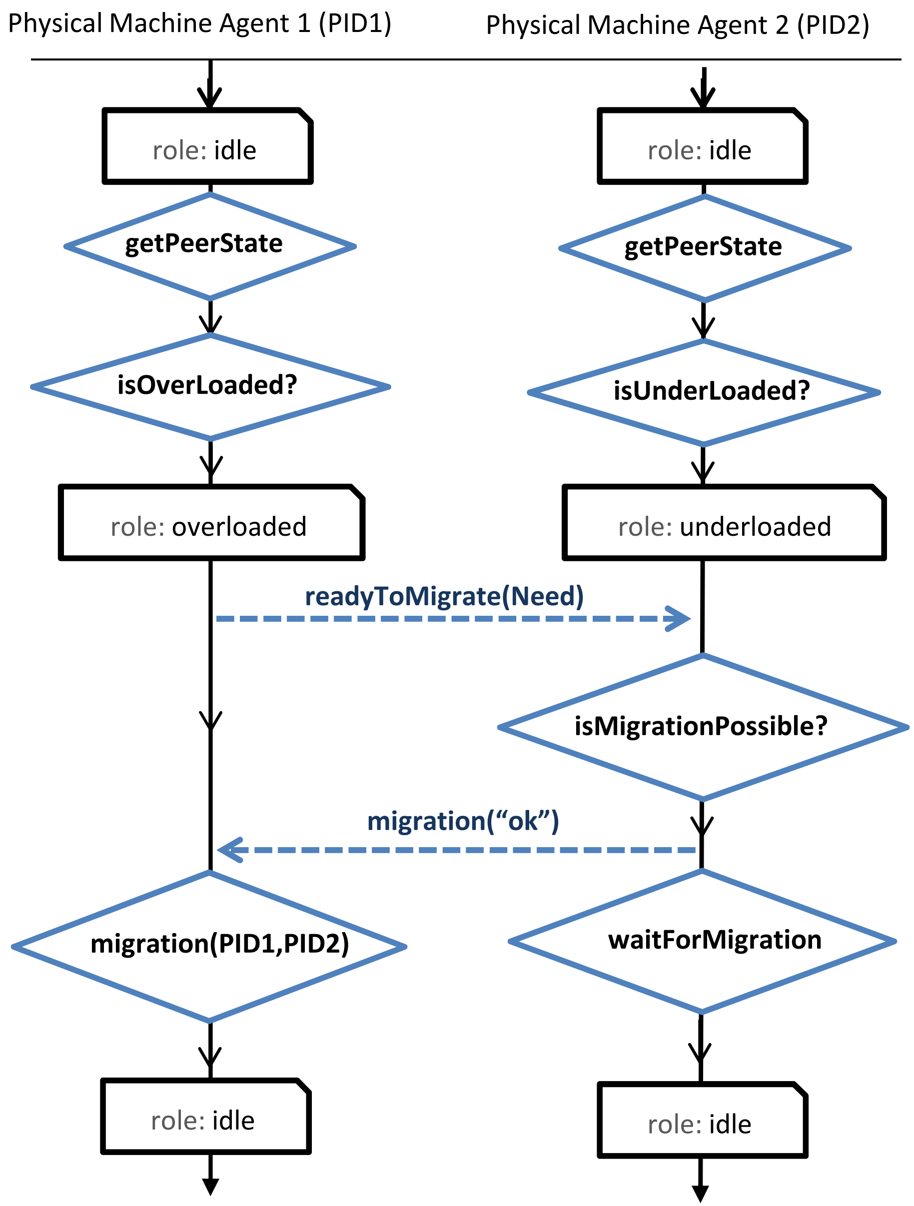


Case Study: Cloud Configuration

- Existing commercial tools for VMs management are usually based on a centralised control.
- Centralised solution may not be interesting for large scale and complex clouds.
- We proposed a less centralised multi-agent VM management framework:
  - P. Anderson, S. Bijani, A. Vichos, **Multi-agent Negotiation of Virtual Machine Migration Using LCC**, 6th KES Int. Conf., KES-AMSTA 2012.
  - P. Anderson, S. Bijani, H. Herry, **Multi-Agent Virtual Machine Management Using the Lightweight Coordination Calculus**, IJICIC Journal (selected to be published).

A Simple VM Management Policy (Interaction Diagram)

Migration policy: unbalanced peers interact to balance their loads.



A Simple VM Management Policy (LCC code)

```
% Here, "idle" means the "balanced" state

a(idle, PeerID) ::

  % the constraint to check the state of the peer

  null <- getPeerState(Status) then  (

  % if the peer is overloaded, change role to overloaded

    a(overloaded(Status), PeerID)<- isOverLoaded() then

  ) or

  ( % if the peer is underloaded, change to underloaded

    a(underloaded(Status), PeerID)<- isUnderLoaded() then

  ) or

  a(idle, PeerID) %otherwise, remain as idle (recursion)
```

```
% "Capacity" is the amount of free resources}

a(underloaded(Capacity), PID2) ::

 % receive a request from  an overloaded peer

 readyToMigrate(Need)<= a(overloaded,PID1) then

 %if free "Capacity" of the underloaded peer >
 %                  "Need" of the overloaded peer

 ( migration(ok) => a(overloaded, PID1) <-
       isMigrationPossible(Capacity, Need)

         then    null <- waitForMigration() )

  or

 ( migration(notOk) => a(overloaded, PID1)) then

a(idle, PID1) % change the peer's role to "idle"
```

```
% "Need" = amount of resources required

a(overloaded(Need), PID1) ::

   readyToMigrate(Need) => a(underloaded, PID2) then

   migration(ok)<= a(underloaded, PID2) then

  % live migration: send VMs to the underloaded peer

  null <- migration(PID1, PID2) then

  a(idle, PID1) % change the peer's role to "idle"
```

A Simple VM Management Policy: Information Leakage

```
% Here, "idle" means the "balanced" state
a(idle, PeerID) ::
  % the constraint to check the state of the peer
  null <- getPeerState(Status) then  (
  % if the peer is overloaded, change role to overloaded
    a(overloaded(Status), PeerID)<- isOverLoaded() then
  ) or
  ( % if the peer is underloaded, change to underloaded
    a(underloaded(Status), PeerID)<- isUnderLoaded() then
  ) or
  a(idle, PeerID) %otherwise, remain as idle (recursion)
```

```
% "Capacity" is the amount of free resources}
a(underloaded(Capacity), PID2) ::
 % receive a request from  an overloaded peer
 readyToMigrate(Need)<= a(overloaded,PID1) then
 %if free "Capacity" of the underloaded peer >
 %                  "Need" of the overloaded peer
 ( migration(ok) => a(overloaded, PID1) <-
       isMigrationPossible(Capacity, Need)
         then    null <- waitForMigration() )
  or
 ( migration(notOk) => a(overloaded, PID1)) then
a(idle, PID1) % change the peer's role to "idle"
```

```
% "Need" = amount of resources required
a(overloaded(Need), PID1) ::
   readyToMigrate(Need) => a(underloaded, PID2) then
   migration(ok)<= a(underloaded, PID2) then
  % live migration: send VMs to the underloaded peer
  null <- migration(PID1, PID2) then
  a(idle, PID1) % change the peer's role to "idle"
```

Consider the following annotations:

```
label(readyToMigrate, l).
label(Need, h).
label(underloaded, l).
label(PID2, l).
```

Explicit Flow from PID1 to PID2: Forbidden

A Simple VM Management Policy: Information Leakage

```
% Here, "idle" means the "balanced" state

a(idle, PeerID) ::

  % the constraint to check the state of the peer

  null <- getPeerState(Status) then  (

  % if the peer is overloaded, change role to overloaded

    a(overloaded(Status), PeerID)<- isOverLoaded() then

  ) or

  ( % if the peer is underloaded, change to underloaded

    a(underloaded(Status), PeerID)<- isUnderLoaded() then

  ) or

  a(idle, PeerID) %otherwise, remain as idle (recursion)
```

```
% "Capacity" is the amount of free resources}

a(underloaded(Capacity), PID2) ::

 % receive a request from  an overloaded peer

 readyToMigrate(Need)<= a(overloaded,PID1) then

 %if free "Capacity" of the underloaded peer >
 %                  "Need" of the overloaded peer

 ( migration(ok) => a(overloaded, PID1) <-
       isMigrationPossible(Capacity, Need)

         then    null <- waitForMigration() )

  or

 ( migration(notOk) => a(overloaded, PID1)) then

a(idle, PID1) % change the peer's role to "idle"
```

```
% "Need" = amount of resources required

a(overloaded(Need), PID1) ::

   readyToMigrate(Need) => a(underloaded, PID2) then

   migration(ok)<= a(underloaded, PID2) then

  % live migration: send VMs to the underloaded peer

  null <- migration(PID1, PID2) then

  a(idle, PID1) % change the peer's role to "idle"
```

Consider the following annotations:

```
label(migration, l).
label(ok, l).
label(isMigrationPossible, l).
label(Need, l).
label(Capacity, h).
label(notOk, l).
label(overloaded, l).
label(PID1, l).
```

Explicit Flow from *underloaded* to PID2:
Forbidden

A Simple VM Management Policy: Information Leakage

```
% “Capacity” is the amount of free resources}

a(underloaded(Capacity), PID2) ::

 % receive a request from  an overloaded peer

 readyToMigrate(Need)<= a(overloaded,PID1) then

 %if free “Capacity” of the underloaded peer >
 %                  “Need” of the overloaded peer

 ( migration(ok) => a(overloaded, PID1) <-
        isMigrationPossible(Capacity, Need)

          then    null <- waitForMigration() )

  or

 ( migration(notOk) => a(overloaded, PID1)) then

a(idle, PID1) % change the peer’s role to “idle”
```

Consider the following annotations:

```
label(migration, l).
label(ok, l).
label(isMigrationPossible, l).
label(Need, l).
label(Capacity, h).
label(notOk, l).
label(overloaded, l).
label(PID1, l).
```

Implicit Flow from PID2 to PID1:

Forbidden

A Simple VM Management Policy: Information Leakage

```
% “Capacity” is the amount of free resources}

a(underloaded(Capacity), PID2) ::

 % receive a request from  an overloaded peer

 readyToMigrate(Need)<= a(overloaded,PID1) then

 %if free “Capacity” of the underloaded peer >
 %                  “Need” of the overloaded peer

 ( migration(ok) => a(overloaded, PID1) <-
        isMigrationPossible(Capacity, Need)

          then    null <- waitForMigration() )

  or

 ( migration(notOk) => a(overloaded, PID1)) then

a(idle, PID1) % change the peer’s role to “idle”
```

Consider the following annotations:

```
label(migration, l).
label(ok, l).
label(isMigrationPossible, l).
label(Need, l).
label(Capacity, h).
label(notOk, l).
label(overloaded, l).
label(PID1, l).
```

Implicit Flow from PID2 to PID1:
Forbidden

Dynamic vs. Static Security Check

Dynamic Check (false negative result)

Γ={ ready: **L**, overload: **L**, pid1: **L**, need: **L**, capacity: **H**, migratePossible: **L ,** migrate: **L** }

LCC code	**LCC Interpreter's Action**	**Security Type Rule**	**Result**
ready <= a(overload,pid1) then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity, need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	**Closed(c(**ready <= a(overload,pid1)**))** **sTypeCheck(**ready<= a(overload,pid1)**)**	$\dfrac{\Gamma \vdash my{:}\,agent\ L,\ \Gamma \vdash ready{:}\,L}{\Gamma \vdash ready(need) <= a(overload,pid1){:}\ op\ L}\ Rsv$	**OK**
ready <= a(overload,pid1) then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity, need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	**satisfy(**migratPossible(capacity,need)**)** **returns FALSE or** **could not find the constraint**	--	
ready <= a(overload,pid1) then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity, need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	**Close(c(**migrate(notOk)=> a(overload,pid1)**)))**	$\dfrac{\Gamma \vdash a(overload,pid1){:}\ agent\ \ L, \Gamma \vdash migrate(notOk){:}\,L}{\Gamma \vdash migrate(notOk) => a(overload,pid1){:}\ op\ L}\ Snd$ $\dfrac{\Gamma \vdash overload{:}\,L,\ \ \Gamma \vdash pid1\ L}{\Gamma \vdash a(overload,pid1){:}\ agent\ L}\ Agnt$ $\dfrac{\Gamma \vdash migrate{:}\,L,\ \Gamma \vdash notOk{:}\,L}{\Gamma \vdash migrate(notOk){:}\ L \vee L}\ Struct$	**OK**

Static Type Check

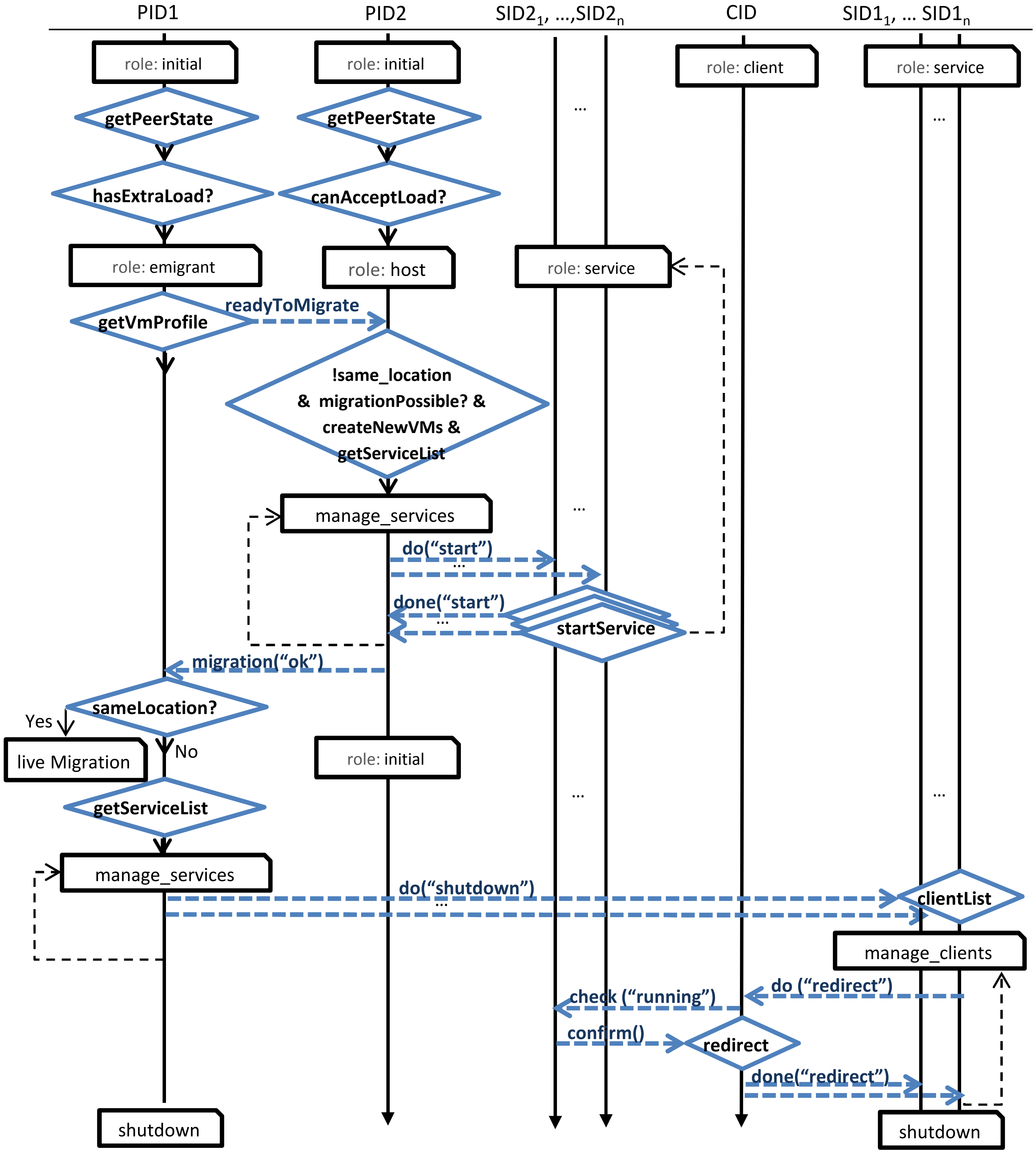
Γ={ ready: **L**, overload: **L**, **Pid1**: L, **Need**: L, **Capacity**: **H**, migratePossible: **L ,** migrate: **L, ...** }

LCC code	Security Type Rule	Result
ready <= a(overload,Pid1) then (migrate(ok) =>a(overload, Pid1) <- migratPossible(Capacity, Need) then null <- wait()) or (migrate(notOk)=> a(overload,Pid1))	$\dfrac{\Gamma \vdash my: agent\ L,\ \Gamma \vdash ready: L}{\Gamma \vdash ready(need) <= a(overload, Pid1): op\ L}\ Rsv$	OK
ready <= a(overload,pid1) then (migrate(ok) =>a(overload, Pid1) <- migratPossible(Capacity, Need) then null <- wait()) or (migrate(notOk)=> a(overload,Pid1))	$\dfrac{\Gamma \vdash migratPsbl(Capacity, Need): H\ \tau,\ \Gamma \vdash migrate(ok) => a(overload, Pid1): op\ L}{\Gamma \vdash migrate(ok) => a(overload, Pid1) <- migratPsbl(Capacity, Need): op\ \tau}\ If2$ $\dfrac{\Gamma \vdash a(overload, Pid1): agent\ L,\ \Gamma \vdash migrate(ok): L}{\Gamma \vdash migrate(ok) => a(overload, pid1): op\ L}\ Snd$ $\dfrac{\Gamma \vdash overload: L,\ \Gamma \vdash Pid1\ L}{\Gamma \vdash a(overload, pid1): agent\ L}\ Agnt$ $\dfrac{\Gamma \vdash migrate: L,\ \Gamma \vdash ok: L}{\Gamma \vdash migrate(ok): L \vee L}\ Struct$ $\dfrac{\Gamma \vdash migratePossible: L,\ \Gamma \vdash Capacity: H,\ \Gamma \vdash Need: L}{\Gamma \vdash migratPossible(Capacity, Need): L \vee H \vee L}\ Struct$	Alarm

Offline VM Migration

- Between different datacentres
- E.g. from a private cloud to a public cloud
- Not transparent to the user
- Needs more negotiation and configurations

An Interaction diagram for an offline VM migration



Offline VM Migration: Information Leakage

```
a(initial, PeerID) ::
  null <- getPeerState(Status) then
  ( % if the peer is underloaded (e.g.≤ %50), change the role to "emigrant"
    ( a(emigrant(Status), PeerID) <- isUnderLoaded(Status) )     or
     % if the peer's load > threshold (e.g. %50 ), but it still has free resources
     % change the peer's role to "host" and send the peer's status
    ( a(host(Status), PeerID) <- canAcceptMoreLoad(Status) )
    or
     % if the peer has no load, change the role to "shutdown"
    ( a(shutdown, PeerID) <- hasNoLoad(Status) )
    or
    ( a(initial, PeerID) )  % otherwise, the peer is fully-loaded (recursion)
  )
```

Annotations:

```
label(initial, l).
label(PeerID, l).
label(getPeerState, l).
label(Status, h).
```

Explicit flow from *Status* to *PeerID*:
Forbidden

Outline

- Introduction
- Language-based Security
- A Security Type System for LCC
- Information Leakage in Clouds
- Conclusion

Static Type Check

- Pros
  - proof of program correctness with reasonable computation cost
  - conservatively detects implicit and explicit information flows and provides stronger security assurance.
- Cons
  - high false positive because possibility of run-time information manipulation

Dynamic Check

- Cons
  - It can not detect **implicit information flows**.
  - **It is not sound,** because does not check all execution paths of the program.

Summary

- A security type system is proposed for LCC to analyse information flow.
- LCC interpreter is augmented with security type check (dynamic check).
- A static security type check is implemented for LCC.
- A Case study of cloud computing has been analysed