Secure information sharing in social agent interactions using information flow analysis

Secure Information Sharing in Social Agent Interactions Using Information Flow Analysis

Shahriar Bijani[1], David Robertson[2] and David Aspinall[2]

[1] Computer Science Dept., Shahed University, Persian Gulf Highway, Tehran, Iran.
[2] Informatics School, University of Edinburgh, 10 Crichton St. Edinburgh, UK.
bijani@shahed.ac.ir, dr@inf.ed.ac.uk, david.aspinall@ed.ac.uk

Abstract.
When we wish to coordinate complex, cooperative tasks in open multi-agent systems, where each agent has autonomy and the agents have not been designed to work together, we need a way for the agents themselves to determine the social norms that govern collective behaviour. An effective way to define social norms for agent communication is through the use of interaction models such as those expressed in the Lightweight Coordination Calculus (LCC), a compact executable specification language based on logic programming and pi-calculus. Open multi-agent systems have experienced growing popularity in the multi-agent community and gain importance as large scale distributed systems become more widespread. A major practical limitation to such systems is security, because the very openness of such systems opens the doors to adversaries to exploit vulnerabilities introduced through acceptance of social norms.
This paper addresses a key vulnerability of security of open multi-agent systems governed by formal models of social norms (as exemplified by LCC). A fundamental limitation of conventional security mechanisms (e.g. access control and encryption) is the inability to prevent information from being propagated. Focusing on information leakage in choreography systems using LCC, we suggest a framework to detect insecure information flows. A novel security-typed LCC language is proposed to prevent information leakage.
Both static (design-time) and dynamic (run-time) security type checking are employed to guarantee no information leakage can occur in annotated agent interaction models. The proposed security type system is discussed and then formally evaluated by proving its properties.
Two disadvantages of the pure dynamic analysis are its late detection and its inability to detect implicit information flows. We overcome these issues by performing static analysis. The proposed security type system supports non-interference, i.e. high-security input to the program never affect low-security output. However, it disregards information leaks due to the termination of the program.

Keywords: Multi-Agent Systems (MASs), Open Systems, Language-based Security, Information Leakage, Information Flow Analysis, Lightweight Coordination Calculus (LCC).

1. Introduction

Security is a major practical limitation to the advancement of open systems and open multi-agent systems (MASs) is no exception. Although openness in open MASs makes them attractive for different new applications, new problems emerge, among which security is a key issue. Unfortunately, there remain many potential gaps in the security of open MASs and little research has been done in this area.

A MAS could be defined as a subcategory of a software system, a high level application on top of the OSI[1] networking model; therefore the security of MASs is not a completely different and new concept; it is a sub-category of computing security. However, some traditional security mechanisms resist use in MAS directly, because of the social nature of MASs and the consequent special security needs (Robles 2008). Open MASs are

[1] Open Systems Interconnection

particularly difficult to protect, because we can provide only minimum guarantees about the identity and behaviour of agents.

Confidentiality is one of the main features of a secure system that is challenging to be assured in open MAS. Open MASs are convenient platforms to share knowledge and information, however usually there exists some sensitive information that we want to protect. The openness of these systems increases the potential for unintentional leaking of sensitive information. Thus, it is crucial to have mechanisms that guarantee confidentiality and to assure that the publicly accessible information during the interactions is what we deliberately want to share.

Information leakage denotes disclosure of secret information to unauthorised parties via insecure information flows. Information leaks in agent interactions occur when secret data are revealed through message transfers, constraints or assigning roles to agents.

An *electronic institution* (Esteva, et al. 2001) or an *interaction model* is an organisation model for MASs that provides a framework to describe, specify and deploy agent interaction environments (Joseph, et al. 2006). It is a formalism which defines agents' interaction rules and their permitted and prohibited actions. While interaction models can be used to implement security requirements of a multi-agent system, they also can be turned against agents to breach their security in a variety of ways, as we will show in this paper.

To employ a language-based approach to secure interaction models, we need to select an agent language. We chose the *Lightweight Coordination Calculus* (LCC) as the agents' communication language (see Section 2 for a summary of LCC).

Common security techniques such as conventional access control, encryption, digital signatures, virus signature detection and information filtering are necessary but they do not address the fundamental problem of tracking information flow in information systems, therefore, they cannot prevent all information leaks. Access control mechanisms only prevent illegal access to information resources and cannot be a substitute for information flow control (Sabelfeld and Myers 2003). Encryption-based techniques guarantee the origin and integrity of information, but not its behaviour.

This paper is laid out as follows. First, different types of insecure information flows in open MAS governed by interaction models are introduced. Second, a security type system is proposed by defining *security types* and the *type inference rules*. Then, the security type system is evaluated by proving some of its properties. Next, the dynamic and the static approaches in the interaction type checking are reviewed and *non-interference* and *declassification* are discussed.

2. Lightweight Coordination Calculus (LCC)

In our security analysis *Lightweight Coordination Calculus* (LCC) is used to implement agents' interaction models and formulate attacks. LCC (Robertson 2005), is a declarative language used to specify and execute social norms in a peer to peer style. LCC is a compact executable specification based on logic programming.

An interaction model in LCC is defined as a set of clauses, each of which specifies a role and its process of execution and message passing. The LCC syntax is shown in Fig. 2-1.

```
Interaction Model := {Clause,...}

Clause := Role::Def

Role := a(Type, Id)

Def := Role | Message | Def then Def | Def or Def | null<- C | Role <- C

Message := M => Role  |  M => Role <- C  |  M <= Role  | C <- M <= Role

C:= Constant | P(Term,...) | ¬ C | C ∧ C | C ∨ C

Type := Term

Id := Constant | Variable

M:= Term

Term:= Constant | Variable | P(Term,...)

Constant:= lower case character sequence or number

Variable := upper case character sequence or number
```

Fig. 2-1: LCC language syntax; principal operators are: messages (`and`), conditional (`<-`), sequence (`then`) and committed choice (`or`)

Each role definition specifies all of the information needed to perform that role. The definition of a role starts with: `a(roleName,PeerID)`. The principal operators are outgoing message (=>), incoming message (<=), conditional (<-), sequence (then) and committed choice (or). Constants start with lower case characters and variables (which are local to a clause) start with upper case characters. LCC terms are similar to Prolog terms, including support for list expressions. Matching of input/output messages is achieved by structure matching, as in Prolog.

The right-hand side of a conditional statement is a constraint. Constraints provide the interface between the interaction model and the internal state of the agent. These would typically be implemented as a Java component which may be private to the peer, or a shared component registered with a discovery service.

Role definitions in LCC can be recursive and the language supports structured terms in addition to variables and constants so that, although its syntax is simple, it can represent sophisticated interactions. Notice also that role definitions are "stand alone" in the sense that each role definition specifies all the information needed to complete that role. This means that definitions for roles can be distributed across a network of computers and (assuming the LCC definition is well engineered) will synchronise through message passing while otherwise operating independently.

Robertson (2005) defined the following clause expansion mechanism for agents to unpack any LCC interaction model they receive and suggested applying rewrite rules to expand the interaction state:

$$C_i \xrightarrow{M_i, M_{i+1}, P, O_i} C_{i+1}s, \ldots, C_{n-1} \xrightarrow{M_{n-1}, M_n, P, O_n} C_n$$

where C_n is an expansion of the original LCC clause C_i in terms of the interaction model P and in response to the set of received messages M_i, O_n is an output message set, M_n is a remaining unprocessed set of messages.

The rewrite rules allow an agent to conform to the interaction model by unpacking clauses, finding the next step and updating the interaction state. The rewrite rules are defined in the LCC interpreter, which should be

installed on each agent running LCC codes. For more information about the LCC expansion algorithm see (Robertson 2005) and (Robertson, Barker, et al. 2009).

3. Related Work

Security of MAS has been explored extensively in the literature but only a few studies have focused on open MAS social interactions and most research has dealt with the security of mobile agent environments, so most of the security solutions address threats from agents to hosts or from hosts to agents. A review of attacks and countermeasures for open MASs is presented by (Bijani and Robertson, A Review of Attacks and Security Approaches in Open Multi-agent Systems 2014). Trust measures have also important role in implementing security strategies in open MAS. E.g. The trust service of the MAS is responsible for preventing the fake identity attacks. This issue has been considered in several recent works e.g. (Rosaci 2012) and (Buccafurri, et al. 2016).

There have been many attempts to protect mobile agents from the host platform in the literature (Jansen and Karygiannis 2000), (Oey, Warnier and Brazier 2010) and (Ngereki 2015); some are based on cryptography while others are not; e.g. code obfuscation (Majumdar and Thomborson 2005), function encryption (Lee, Alves-Foss and Harrison 2004) and (Zhu and Xiang 2011), environmental key generation (Riordan and Schneier 1998), execution tracing (Tan and Moreau 2002), and agent monitoring (Page, Zaslavsky and Indrawan 2005). Another important security issue in mobile agent systems is protecting the agent platform from mobile agents. Some example techniques are: Proof Carrying Code (Necula and Lee 1998), sandboxing (Wahbe, Lucco and Anderson 1993) and code signing (Jansen and Karygiannis 2000). However, the importance of these security issues originating from the mobility of agents should not diminish the importance of many other security threats in open MASs and a subset of these will be our concern henceforth in this paper.

Security approaches in the multi-agent security domain can be divided into two parts; the first approach is *prevention*, in which usually encryption-based techniques and authentication methods (e.g.: certificates and PKI[2]) are used. Most research on secure MASs follows this approach. (Wong and Sycara 1999), (Idrissi, Souidi and Revel 2015) and (Ohno, et al. 2016) are some examples of using encryption to prevent MASs from malicious attacks. For instance, (Poslad and Calisti 2000), (Wang, Varadharajan and Zhang 1999) and (Odubiyi and Choudhary 2007) suggest security architectures for the IEEE FIPA agent standard by means of authentication, PKI and VPN[3]. (Sarhan and Alnaser 2014) propose a public-key based solution for multi-agent virtual learning environment. Other prevention methods for secure MASs are: policy driven and secure development methodologies such as (Mouratidis, Giorgini and Weiss 2003) and (Hedina and Moradian 2015) that guarantee security requirements and design are integrated with system functionalities. Policy driven methodologies are based on applying security policies, which may be used for access control, e.g. (Quillinan, et al. 2008), definition of acceptable behaviour, e.g. (Vazquez-Salceda, et al. 2003) or policy randomisation to prevent adversaries guess the next agent action, e.g. (Tan, Poslad and Xi 2004).

[2] Public Key Infrastructure

[3] Virtual Private Network

The second approach is *detection*, which tries to detect attacks on MASs and then *respond* to them. Little research has been done in this area and the focus of the work has been on attacks and countermeasures in mobile agents, e.g., (Jansen and Karygiannis 2000), (Endsuleit and Wagner 2004), and (Ogunnusi and Ogunlola 2015). The main problems in mobile agent systems, which are not in the scope of our review, are threats from agents to hosts and vice versa.

We employed a language-based information flow analysis approach in the context of open MASs. In static information flow analysis, agent interaction models are validated before being run. Static analysis of programmes using security type systems conservatively detects implicit and explicit information flows and provides stronger security assurance (Sabelfeld and Myers 2003). Dynamic security checks may be accomplished via two similar approaches: monitors (Russo and Sabelfeld 2010) or dynamic security typing (Hennigan, et al. 2011).

4. Insecure Information Flows

The first step in language-based information flow analysis for agent interaction models is defining security levels for terms and components in the code. A set of security levels is a finite lattice i.e. a partially ordered set with a top element H and a bottom element L, ordered by ≤. Lower in the lattice denotes "less secure" and higher in the lattice indicates "more secure". Without loss of generality, a two-element security lattice is assumed with levels *l*, for low security (public information), and *h*, for high security (secret information).

The following definition characterises the concept of security levels in this paper.

Definition 4-1 (Security Levels):

We consider a simple lattice $\mathbb{L}$ with two security levels, low *l* and high *h*, security level $l \in (\mathbb{L}, \leq)$, where $l \leq h$ and ≤ is a partial order relation.

We need to ensure that information flows only upwards in the lattice (D. E. Denning 1976) e.g. when l ≤ h, permissible information flows are from l to l, from l to h and from h to h, but flow from h to l is not allowed.

A MAS keeps secrets confidential during agents' interaction if it only allows secure information flow. There are two types of information flows: *explicit flow* and *implicit flow*. Distinctions between *explicit* and *implicit* flows in LCC interaction models are shown with the following examples. It is assumed that all the LCC terms in the given examples are public information (which have security level *l*), except for the following secret variables (which have security level *h*):

```
SecretMessage, SecretID, S, PrivateAgent, secretAgent.
```

In the following examples SecretMessage is a secret message, are sec is for the following secret variables (which have security level *h*):

4.1 Explicit flows

Insecure explicit information flow denotes direct sending or assigning of secrets. Explicit flows in LCC interaction models may occur in three situations: (a) message passing, (b) invoking a constraint and (c)

assigning a role to an agent. In explicit information flows, the operations are performed independently of the value of their terms (Denning and Denning 1977), e.g. the content of an LCC message does not affect the sending operation. Insecure explicit flow may cause secret information to be leaked to a publicly observable term. Consider the following LCC codes as examples of explicit information flows:

a) Message passing

The following explicit flow, in which the instance of a variable `SecretMessage` is sent to a low level agent `P` with the risk of secret information leakage:

```
SecretMessage => a(publicAgent, P)
```

The secret message can also be received by another agent:

```
SecretMessage <= a(publicAgent, P)
```

This breach of security can occur in an LCC clause, when a public agent `P` sends the `SecretMessage` to any (public or secret) receiver agent `R`:

```
a(publicAgent, P)::
        ...
        SecretMessage => a(receiver, R)
        ...
```

On the other hand, a message passing pattern can occur without a security breach. The following explicit flows that sends (receives) a `PublicMessage` variable to (from) a `secretAgent S` is permissible.

```
PublicMessage => a(secretAgent, S)
```

and

```
PublicMessage <= a(secretAgent, S)
```

b) Invoking a constraint

An example of an explicit flow that discloses the value of a secret variable to a publicly observed variable is assigning `SecretID` to a `PublicVariable` in an LCC constraint:

```
null <- assign(PublicVariable, SecretID)
```

Any constraint that updates the value of a public term using a secret term causes an unacceptable information flow. The constraints in LCC play an important role, although the implementation details of constraint solvers are invisible to LCC clauses and the constraint solver might even be a remote web service. However, it is the responsibility of the LCC programmer to prevent any illegal information flow caused by invoking a constraint.

c) Assigning a role to an agent

When a role is assigned to an agent in the definition of an LCC clause, the security level of the role and the agent identifier need to be compatible. The following role definition is not a permissible flow, because it assigns a secret role `secretAgent` to a low security agent `PublicAgent`.

```
a(secretAgent, PublicAgent)::
  ...
```

On the other hand, a `publicAgent` role (or a `secretAgent` role) can be assigned to a `PrivateAgent`:

```
a(publicAgent, Private Agent)::   ...
```

4.2 Implicit Flows

Insecure implicit flows disclose some information through the program control flow. In other words, based on a definition from Denning and Denning (1977), we can define an implicit information flow from term T1 to term T2, when a performed operation causes a flow from some arbitrary T3 to T2, based on the value of T1. Thus, conditional LCC expressions are the sources of insecure flows.

The following example is a conditional statement, in which a public message is sent to a public agent `P`, if the constraint is satisfied (`SecretID` ≤ `10`). The explicit flow in sending the message is permitted, but the implicit flow from the constraint to the public agent `P` that leaks information about the range of `SecretID` variable is illegal. If a public message is sent to agent `P`, it reveals that the `SecretID` is less than or equal to `10` and if it is not sent, the `SecretID` is greater than `10`.

```
PublicMessage => a(publicAgent,P) <- lessOrEqual(SecretID, 10)
```

In another example below, the public agent `P` can guess the range of `SecretID`, by receiving a public message containing a public variable `X`, although the message passing part does not explicitly disclose any information. Either the public agent receives `publicMsg(X)` or `publicMsg(1)`, knowing the value of `X`, some information about `SecretID` is leaked.

```
publicMsg(X) => a(publicAgent,P) <- lessOrEqual(SecretID,X)

or

publicMsg(1) => a(publicAgent,P)
```

The above example might leak information about `SecretID`, but not the exact value of it. The following example discloses the value of `SecretID`; assuming `SecretID` is not negative, the initial value of `X` is set to `0` and the constraint `increase(X1,X,1)` means `X1=X+1`. In the recursive clause below, if `SecretID` is not equal to 0, the value of `X1` is `X+1` and the clause is called again with the updated `X1`; i.e. `a(myAgent(X1),Q)`. Finally, when `X` equals to `SecretID+1`, `publicMsg(X)` reveals the value of `SecretID` to the public agent `P`.

```
a(myAgent1(X), Q):: ...

(

   a(myAgent1(X1), Q) <- lessOrEqual(SecretID,X) ∧ increase(X1,X,1)
```

```
   or
   publicMsg(X)=>a(publicAgent,P)        % when X equals SecretID +1
 )
```

In a similar example, the following LCC clause binds `R` to the precise value of `SecretID` if the role completes successfully. So, it discloses the value of `SecretID` to the public agent `P` by sending `publicMsg(R)` message. In this example, even if `R` is not sent as a message parameter (i.e. `publicMsg` instead of `publicMsg(R)`), the public agent `P` can discover the value of `SecretID` by counting the number of received messages.

```
a(myAgent2(X,R), Q)::
 ...
 (
   publicMsg(R)=>a(publicAgent,P) <- lessOrEqual(SecretID,X) ∧ increase(X1,X,1)
   then a(myAgent2(X1,R), Q)
 )  or
   a(myAgent2(X,X), Q) <- equals(SecretID,X)
```

Information may leak because of the termination behaviour of the interaction model[4]. Recursion is the key to this type of leaks. In the following sample LCC clause, the adversary learns that the value of the `SecretID` is `0` if the interaction model terminates.

```
a(myAgent3, Q)::
   a(myAgent3, Q) <- ¬equals(SecretID,0)
```

Adversaries can exploit *explicit* or *implicit* information flows to perform attacks. We need to prevent both *explicit* and *implicit* insecure information flows in order to ensure no information leaks to unauthorised parties.

4.3 Countermeasures

Two approaches to address information flow problems in MASs governed by interaction models are *conceptual modelling* by analysing the abstract models of the code and *language-based information flow analysis*. In the first approach, an LCC interaction model is translated into an abstract model, in which information leakage is investigated using an existing reasoning tool (Bijani, Robertson and Aspinall 2011). In language-based analysis of the agents' code, we employ security types for the LCC terms and enforce a security policy by type checking.

[4] This is also called information leaks via the *termination channel*.

5. Information Flow Analysis in LCC

We propose a language-based information flow analysis technique for the LCC language to prevent information leaks problem by introducing a novel security type system. The proposed framework is inspired by the security type system of Volpano and Smith (1997).

A security type system is defined by a set of type definitions and typing rules to determine if an interaction model is well-typed.

5.1 Security Types

The type rules are judgments of the form: $\Gamma \vdash T : \varphi$, where Γ is a type environment that maps term T to type φ. Here are some definitions:

Definition 5-1 (Security Type Environment):

A security type environment (context) Γ is a finite map from LCC terms to security types and is defined by

$$\Gamma ::= empty \mid \Gamma, T: \varphi, \qquad (5\text{-}1)$$

in which Γ is empty (with no binding) or an updated environment that contains a mapping of the term T to the type φ. If there exists a φ that Γ ⊢ T : φ, then T is called a well-typed LCC expression under the security context of Γ.

Definition 5-2 (Security Types):

The security types of our system are defined as following:

$$\varphi = \tau \mid \text{uTrm } \tau \mid \text{agent } \tau \mid \text{con } \tau \mid \text{op } \tau, \qquad (5\text{-}2)$$

where τ ranges over elements of security levels, agent identifiers have only type "uTrm τ", agents have only type "agent τ", constraint expressions have only type "con τ", operational commands[5] have only type "op τ" and messages, Constraint arguments have type "uTrm τ" or τ. Role names and other terms (variables, constants and structures) have only type τ.

To have a better understanding of the meanings of the security types, the following description explains the intuition behind them:

a. $\Gamma \vdash X: uTrm\ \tau$ means that an updated agent identifier in a role assignment or message passing operation or an updated argument in a constraint has a security level higher than or equal to τ in context Γ.
b. $\Gamma \vdash T : \tau$ means that an identifier, a role name, or a message T (with every identifier inside it) has a security level lower than or equal to τ in context Γ.

[5] Operational commands are the *Def* keyword in the LCC syntax: *Def := Role | Message | Def then Def | Def or Def | null<- C | Role <- C*

c. Γ⊢ `a(Role,Id)`: *agent* τ means in the agent definition, agent identifier *Id*, to which a role is assigned has a security level τ or higher in context Γ.

d. Γ⊢ *C : con* τ means that the constraint name and every argument within *C* has a security level τ or lower in context Γ.

e. Γ⊢*D: op τ* means that every receiver of a message or any updated identifier in an operational command (i.e. *Def*) has a security level τ or higher in context Γ.

Γ (T) denotes the security level of the term T, e.g. if we have *t1 : h* and *f1 : con l*, then Γ (*t1*) = *h* and Γ (*f1*) = *l*.

Security levels are directly assigned to LCC terms by annotations of the LCC code n

```
label(Term, Level).
```

in which `label` is a keyword, `Term` is any LCC term and `Level` is the security levels high (`h`) or low (`l`). The security types are then assigned based on the term definitions. All security types can be inferred from the term structure automatically, except constraints' arguments, which need to be defined explicitly (by the user). By default, a constraint's arguments are assumed to be non-updatable and to have a security type, τ, assigned to them.

5.2 Type Inference for LCC

The proposed security type system for LCC programs is described by two sets of typing rules (Fig. 5-1) and subtyping rules (Fig. 5-2). Each rule is read from bottom left and is applied recursively, e.g. rule *Agnt* states that in order to assign a role to an agent in form of `a(R, ID)` that has security type of *agent τ*, we must first check whether the security type of the role `R` is τ and then whether the security type of the agent identifier is uTrm τ. It guarantees that a high level role will not be assigned to a low agent. The environment Γ is a confidentiality policy, which is an input of our secure interpreter (Fig. 7-1-a). Security labels are assigned to LCC terms as annotation of LCC interaction models (Fig. 5-3).

The security typing rules *Id* and *uId* explain if an LCC identifier (a constant or a variable) is defined in the environment Γ, security types τ or *uTrm* τ may be assigned to it. Selection of τ or uTrm τ is based on the structure of the LCC expression. The security label of the current clause (*this*) is important while message passing and calling a constraint. This is created and added to the security environment Γ by the *Init* rule.

$$\frac{T:\tau \in \Gamma}{\Gamma \vdash T:\tau}\,Id \qquad \frac{T:\tau \in \Gamma}{\Gamma \vdash T: uTrm\ \tau}\,uId$$

$$\frac{}{\Gamma \vdash\ null: op\ h}\,Null \qquad \frac{}{\Gamma \vdash false: op\ h}\,False$$

$$\frac{\Gamma \vdash T:\varphi}{\Gamma \vdash c(T):\varphi}\,Close \qquad \frac{\Gamma \vdash R:\tau,\ \Gamma \vdash ID: uTrm\ \tau}{\Gamma \vdash a(R,ID): agent\ \tau}\,Agnt$$

$$\frac{\Gamma \vdash a(R,ID): agent\ \tau}{\Gamma, this: agent\ \tau \vdash a(R,ID) :: agent\ \tau} Init^*$$

$$\frac{\Gamma \vdash this: agent\ \tau,\ \Gamma \vdash M: \tau,\ \Gamma \vdash A : agent\ \tau}{\Gamma \vdash M \Rightarrow A : op\ \tau} Snd$$

$$\frac{\Gamma \vdash this: agent\ \tau,\ \Gamma \vdash M: \tau, \Gamma \vdash A : agent\ \tau}{\Gamma \vdash M \Leftarrow A: op\ \tau} Rsv$$

$$\frac{\Gamma \vdash this: agent\ \tau,\ \Gamma \vdash f: \tau,\ \Gamma \vdash T_i: \rho}{\Gamma \vdash f(T_i): con\ \tau} Call \quad \rho = \tau | uTrm\ \tau, \qquad i = 1, \dots, n$$

$$\frac{\Gamma \vdash f: \tau,\ \Gamma \vdash T_i: \tau}{\Gamma \vdash f(T_i): \tau} Struct \quad i = 1, \dots, n \qquad \frac{\Gamma \vdash C: con\ \tau}{\Gamma \neg C: con\ \tau} Not$$

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

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

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

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

$$\frac{\Gamma \vdash A_1: op\ \tau\ ,\ \Gamma, constraint[A_1] \vdash A_2: op\ \tau}{\Gamma \vdash A_1\ or\ A_2: op\ \tau} Choice$$

$$\frac{\Gamma \vdash a(R,ID): agent\ \tau,\ \Gamma \vdash Def\ op\ \tau}{\Gamma, \vdash a(R,ID) :: Def: op\ \tau} Role$$

Fig. 5-1: The security typing rules for LCC

$$\varphi \leq \varphi \quad Reflex \qquad \frac{\Gamma \vdash T: \varphi,\ \varphi \leq \varphi'}{\Gamma \vdash T: \varphi'} Sub \qquad \frac{\varphi_1 \leq \varphi_2\ , \varphi_2 \leq \varphi_3}{\varphi_1 \leq \varphi_3}\ Trans$$

$$\frac{\tau' \leq \tau}{agent\ \tau \leq agent\ \tau'}\ AgentRule \qquad \frac{\tau \leq \tau'}{con\ \tau \leq con\ \tau'}\ ConRule$$

$$\frac{\tau' \leq \tau}{uTrm\ \tau \leq uTrm\ \tau'}\ uTrmRule \qquad \frac{\tau' \leq \tau}{op\ \tau \leq op\ \tau'}\ OpRule$$

Fig. 5-2: Subtyping rules

The rule *Snd* expresses that if the sender (*this*), the receiver *A* and the message *M* have security level τ, then the sending operation ($M \Rightarrow A$) can have the security type op τ. The rule *Rsv* is the same as *Snd*. We need to assure that no high security message is accessed and sent by a low security agent; checking the security level of the sender along with the security level of the massage in Snd and Rsv rules guarantees this. Sending and receiving operations in LCC are dual, so if there exists a leakage in message sending in one clause, the same leakage will be detected in receiving the message in the counterpart clause. The rules *Agnt*, *Snd* and *Rsv* in

conjunction with subtyping rules prevent explicit flows; they imply that assigning or sending public information to secret agents is possible, but not vice versa. This is similar to the concepts of "write up is possible" and "write down is forbidden" in the security type system for imperative programming languages, e.g. (Volpano and Smith 1997).

The *Call* rule states that when we call a constraint, the security level of its functor[6], the security level of the current clause (*this*) and the security level of either read-only arguments (T_i: τ) or write-only arguments (T_i: *uTrm* τ) have to be the same. This ensures us that a public agent can not access secret constraints and a public constraint may not reveal secret information to a public agent. The *Struct* rule denotes that in structured non-updatable terms (such as messages, role names and read-only arguments) the security types and levels of the functor *f* and the arguments T_i must be the same. The rules *And*, *Or* and *Not* regulate the composition of constraints in LCC. The rule *If1* states that the security type of constraint C and the message sending operation ($M \Rightarrow A$) needs to be matched so that the conditional expression is allowed. Security typing of other conditional expressions (*If2 to If4*) is performed in a similar way to *If1*.

The rule *Seq* say that if two LCC expressions have the same security level, their composition has also that security level. The *Choice* rule functions in the same way, only it also considers the security level of the constraint of the first part A_1 to prevent implicit information flow from the constraint in A_2. The rule *Role* checks whether the role definition *a(R,ID)* agrees with the body of the LCC clause. The remaining rules of the security type system are subtyping rules in Fig. 5-2. The subtyping rules *AgentRule*, *uTrmRule* and *opRule* are contravariant[7] and the *conRule* is covariant[8].

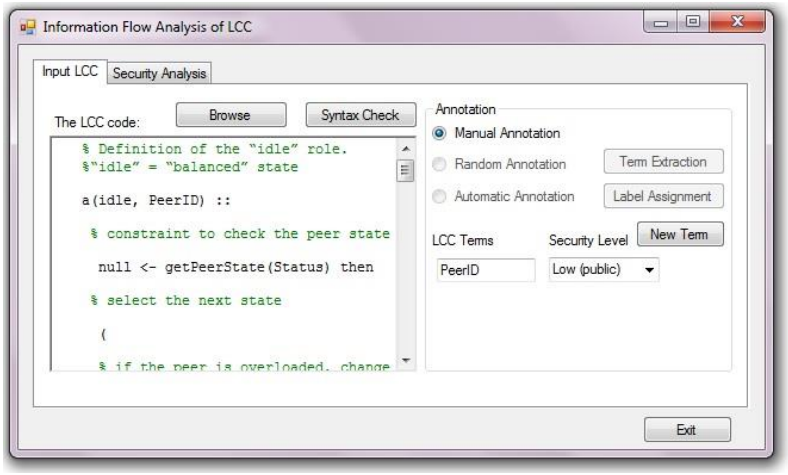
5.3 Implementation

The security type system and a prototype of dynamic security checking application have been implemented to demonstrate that the proposed framework is feasible and can be automated. The original version of LCC which is implemented in Prolog has been extended to support security type checking. The security type system is implemented in SICStus Prolog and a user interface for security analysis of LCC codes is designed in Visual C#.NET. This tool is designed for annotation of LCC interaction models with security labels and performing the security type checking.

[6] Non-numeric constant

[7] Contravariant denotes the possibility of converting from a narrower type to a wider type, e.g. from *h* to *l*.

[8] Covariant means the possibility of converting from a wider type to a narrower type, e.g. from *l* to *h*.



An example annotation of an LCC interaction model that assigns security levels to LCC terms is shown in Fig. 5-3.

```
a(buyer, B) ::
   ask(X) => a(Seller, S)    then    price(X,P) ⇐ a(Seller, S)   then
   buy(X,P) => a(Seller, S) <- afford(X, P)   then
   sold(X, P) <= a(Seller, S)

a(Seller, S) ::
   ask(X) <= a(buyer, B) then  price(X,P)=>a(buyer, B)<-in_stock(X,P) then
   buy(X,P) <= a(buyer, B) then sold(X, P) => a(buyer, B)

   label(buyer, l).    label(B, l).      label(ask, l).    label(X, l).
   label(Seller, h).   label(S, h).      label(price, l).  label(P, l).
   label(buy, l).      label(afford, l). label(sold, l).   label(P, l).
```

Fig. 5-3: Annotation of an LCC interaction model

6. Key Properties of the Type System

Having defined a type system for a class of security properties, our purpose in this section is to prove key security properties of the system. Other work (S. Bijani 2013) gives empirical examples of the consequences of these properties in specific interactions but, to save space, we focus here on generic properties across all appropriate LCC interactions.

Type soundness (or *type safety*) is the most basic feature of a type system (Pierce 2002). Two properties that show the type soundness in a type system are *progress* and *preservation*. In our security type system, *preservation* means that expansion of a well-typed term by the LCC rewrite rules is a well-typed term (clause expansion preserves well-typedness). Progress guarantees that a well-typed LCC expression does not get stuck in the execution of LCC clauses, assuming that agents can evaluate (satisfy/dissatisfy) the constraints and the necessary input/output messages are generated.

Definition 6-1 (Final Step): An LCC expression is in its final step when either it can be marked as a *closed* expression by an LCC rewrite rule or it is a constraint that is evaluated by a *satisfy* or *satisfied* rule.

Definition 6-2 (Transition $L \rightsquigarrow L'$): transition of $L \rightsquigarrow L'$ means L' is an expansion of LCC expression L, either as a result of an LCC rewrite rule $L \xrightarrow{R_i, M_i, M_O, P, O} L'$ or as a structural expansion of a compound constraint.

This is an example of a compound constraint expansion: assume L is $null \leftarrow C$ and the compound constraint C is $C_1 \wedge C_2$, when C is unfolded into $C_1 \wedge C_2$ then L' is $null \leftarrow C_1 \wedge C_2$ and we can write $L \rightsquigarrow L'$.

Theorem 6-1 (Progress):

If $\Gamma \vdash L : \varphi$, i.e. L is a well-typed LCC expression, then either L is a *final step* or else there exists some L' that L $\rightsquigarrow$L'.

By induction on the structure of $\Gamma \vdash L : \varphi$ and proceed by case analysis (Appendix) □

Theorem 6-2 (Preservation):

If $\Gamma \vdash L : \varphi$, i.e. L is a well-typed LCC expression and $L \rightsquigarrow L'$, then $\Gamma \vdash L' : \varphi'$.

By induction on the structure of $\Gamma \vdash L : \varphi$ and proceed by case analysis (Appendix) □

Two important properties of a security type system are '*No Read Up*' and '*No Write Down*' or '*simple security'* and '*confinement'* as referred to by (Smith and Volpano 1998). *No Read Up* means that identifiers within a message or a constraint can not have security level higher than the massage level or the constraint level. In other words, when a message (or a constraint) has a security level τ, it assures us that it will not reveal any information with security level more than τ.

'No Write Down' means having an operational command with the security level of $op\ \tau$ (any operational command), any *updatable identifier* within it has a security level higher than or equal to τ. By *updatable identifier*, we mean an agent when a role is assigned to it or a message is sent to it. We also mean an argument in a constraint whose value is updated. E.g. this denotes that it is not possible to assign (send) a higher role (higher message) to a lower agent.

Proposition 6-3 (No Read Up):

If T is a well-typed LCC constraint, message or identifier with security type τ*; i.e.* $\Gamma \vdash T: \tau$ *or* $\Gamma \vdash T: con\ \tau$, then T contains only identifiers with security level not higher than τ.

This can be proved by induction on derivation of $\Gamma \vdash T : \tau$ *and* $\Gamma \vdash T : con\ \tau$*; i.e.* induction on the smaller derivations that are used to derive $\Gamma \vdash T : \tau$ *and* $\Gamma \vdash T : con\ \tau$, then proceeding by case analysis on the typing rule that was applied last in the proof of $\Gamma \vdash T$.

Proposition 6-4 (Confinement):

If T is a well-typed agent definition or LCC operation; *i.e.* $\Gamma \vdash T: agent\ \tau$ *or* $\Gamma \vdash T: op\ \tau$, then any agent identifier in the agent definition, any receiver of a message, or any updated term in an operation, has a security level equal or higher than τ.

This can be proved by case analysis on the rule that was applied last in the proof of $\Gamma \vdash T : \varphi$ and by induction on the type rules that are used to derive $\Gamma \vdash T : \varphi$.

In the next section, we show how the security type system can be used in the agent interaction to verify whether an interaction model is secure.

7. Discussion

7.1 Dynamic Information Analysis

We used both dynamic and static security typing approaches to implement our type system for agent interactions. Dynamic (run-time) information flow analysis such as (Santos, et al. 2015) can appropriately be added to the LCC language interpreter because of the dynamic nature of LCC language.

Based on the reaction policy, type checking could result in termination of the execution or breach detection and continuation of the clause expansion ((a) (b)

Fig. 7-1).

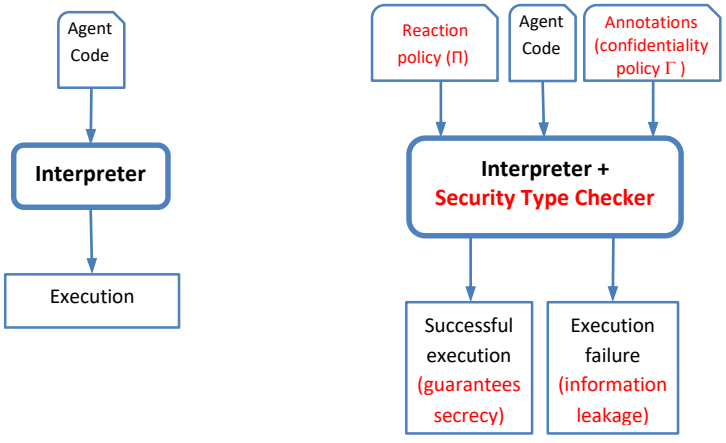


Fig. 7-1: Upgrading the agent code interpreter (a) The interpreter executes codes (b) The improved interpreter performs the security type checking and executes the annotated agent codes

LCC clauses are well-typed by ensuring that every expansion of them is performed according the corresponding security typing rule. Security type checking is performed using the proposed formal type system which ensures that the security types of LCC terms are used consistently.

In order to integrate dynamic information flow analysis into the LCC interpreter and to detect or prevent information leakage, the LCC *clause expansion mechanism* (Robertson 2005) (explained in section 2) has been upgraded by amending the LCC *rewrite rules*.

The extended LCC rewrite rules augmented with dynamic type checking are shown in Fig. 7-2. The updated rewrite rules in Fig. 7-2 are of the form $X \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} Y$, where Y is the expansion of X performing role R_i, M_i is the initial set of messages, O is the output message set, and M_o is the subset of M_i which is not yet processed and P is the interaction model. Δ is the current security environment. $\Delta = (\Gamma, K, \Pi, L, \Sigma)$, where Γ is the mapping between LCC terms and secrecy labels (the confidentiality policy), K is the agents' current state of knowledge, Π is the reaction policy defining the desired behaviour when an unacceptable information flow is found and L is the set of possible information leakages found. Σ is an optional part of Δ that keeps a record of provenance information about the agent's counterparts, who have interacted with the current agent. Elements of Δ could be denoted as Δ(member); e.g. $\Delta(\Gamma)$ *is* Γ or $\Delta(\Pi)$ *is* Π. Consequently, the LCC expansion of the initial clause C_i to the final clause C_n under the security environment Δ is as follows.

$$C_i \xrightarrow{M_i, M_{i+1}, P, O_i}_{\Delta} C_{i+1}, \dots, C_{n-1} \xrightarrow{M_{n-1}, M_n, P, O_n}_{\Delta} C_n$$

typeChk(X,Δ) is in charge of checking the possibility of information leakage from LCC expression X using the security type system introduced in section 5.2. Type checking is performed when the other conditions for rewriting an expression are met. E.g. only if $(M \Leftarrow A) \in M_i$ in (8) or *satisfied(C)* in (9) return *true*, then *typeChk(X,Δ)* is called.

As a result of rewrite rules in Fig. 7-2, the clause of the interaction model appropriate to the given role is expanded. The first rule starts unpacking a clause by expanding its body (B) and the rules (2) to (12) expand different parts of the clause body. The *closed* rules in (13) to (18), determine whether an interaction rule has been completed through earlier interpretation (in which case we say that it is closed).

The interpreter tries to find a matching rewrite rule for each LCC expression, if no match is found, it means that there is a syntax error in the LCC code. If a match is found but the conditions of the rewrite rule are not fulfilled, it returns false and continues to search for another rewrite rule that matches the expression.

$$a(R,I) :: B \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} a(R,I) :: E \quad if \; B \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \wedge typeChk(a(R,I) :: E, \Delta) \quad (1)$$

$$A_1 \; or \; A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \quad if \; \neg closed(A_2) \wedge A_1 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \quad (2)$$

$$A_1 \; or \; A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \quad if \; \neg closed(A_1) \wedge A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \quad (3)$$

$$A_1 \text{ then } A_2 \xrightarrow{R_i, M_i, M_o, P, O} E \text{ then } A_2 \quad \text{if } A_1 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \tag{4}$$

$$A_1 \text{ then } A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} A_1 \text{ then } E \quad \text{if } closed(A_1) \wedge A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \wedge typeChk(A_1 \text{ then } E, \Delta) \tag{5}$$

$$A_1 \text{ par } A_2 \xrightarrow{R_i, M_i, M_o, P, O_1 \cup O_2}_{\Delta} E_1 \text{ par } E_2$$
$$\text{if } A_1 \xrightarrow{R_i, M_i, M_n, P, O_1}_{\Delta} E_1 \wedge A_2 \xrightarrow{R_i, M_n, M_o, P, O_2}_{\Delta} E_2 \wedge typeChk(E_1 \text{ par } E_2, \Delta) \tag{6}$$

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{M \Leftarrow A\}, P, \emptyset}_{\Delta} c\big(C \leftarrow M \Leftarrow A, \Delta(L)\big)$$
$$\text{if } (M \Leftarrow A) \in M_i \wedge typeChk(C \leftarrow M \Leftarrow A, \Delta) \wedge satisfy(C) \tag{7}$$

$$M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{M \Leftarrow A\}, P, \emptyset}_{\Delta} c(M \Leftarrow A, \Delta(L)) \quad \text{if } (M \Leftarrow A) \in M_i \wedge typeChk(M \Leftarrow A, \Delta) \tag{8}$$

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}}_{\Delta} c(M \Rightarrow A \leftarrow C, \Delta(L)) \quad \text{if } satisfied(C) \wedge typeChk(M \Rightarrow A \leftarrow C, \Delta) \tag{9}$$

$$M \Rightarrow A \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}}_{\Delta} c(M \Rightarrow A, \Delta(L)) \quad \text{if } typeChk(M \Rightarrow A, \Delta) \tag{10}$$

$$null \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \emptyset,}_{\Delta} c(null \leftarrow C, \Delta(L)) \quad \text{if } satisfied(C) \wedge typeChk(null \leftarrow C, \Delta) \tag{11}$$

$$a(R, I) \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \emptyset}_{\Delta} a(R, I) :: B$$
$$\text{if } clause(P, a(R, I) :: B) \wedge satisfied(C) \wedge typeChk(a(R, I) \leftarrow C, \Delta) \tag{12}$$

$$a(R, I) \xrightarrow{R_i, M_i, M_o, P, \emptyset}_{\Delta} a(R, I) :: B \quad \text{if } clause(P, a(R, I) :: B) \tag{13}$$

$$closed\big(c(X, L)\big) \tag{14}$$
$$closed(A \text{ or } B) \leftarrow closed(A) \vee closed(B) \tag{15}$$
$$closed(A \text{ par } B) \leftarrow closed(A) \wedge closed(B) \tag{16}$$
$$closed(A \text{ then } B) \leftarrow closed(A) \wedge closed(B) \tag{17}$$
$$closed(X :: B) \leftarrow closed(B) \tag{18}$$

Fig. 7-2: The amended LCC rewrite rules, which include security type checking, for expansion of one clause in an interaction model in the LCC interpreter.

It is the agents' responsibility to satisfy the constraints in the clause and it is assumed that agents have a mechanism to fulfil the constraints. *satisfied(C)* is true if C can be satisfied from the current knowledge state K of the agent and *satisfy(C)* is true when K can be made to fulfil the constraint C. *clause (P, X)* is true if clause *X* exists in the interaction model *P*.

The algorithm for simple type checking (*typeChk*) is defined in Fig. 7-3. In Fig. 7-4, an updated *typeChk* algorithm is defined, in which based on the result of the type checking and the reaction policy Π, *true* or *false* is returned.

In this version of LCC clause expansion, three secrecy policies affect the behaviour of the LCC interpreter: prevention, detection and no-detection. The default policy is prevention (*prevMode*) that averts expansion of the current expression when a leakage is found. If the detection policy (*detectMode*) is selected in Π, the interpreter only keeps a record of the confidentiality breaches and continues to expand the expression X. Selection of the no-detection policy (no*ChkMode*) bypass the information flow analysis and the LCC interpreter do not perform the type checking procedure. The *false* result from *typeChk(X,Δ)* shows that a breach is found and the *true* result

means either the type checking option is off, no leakage is found, or a leakage is found but the detection mode is on.

Table 7-1: Different reaction policy modes in security type checking: prevention, detection and no-detection modes.

Reaction Policy modes	Priority	(Pre,	Det,	NoCk)	Type checking	typeChk result when a leakage is found
prevMode	1	1	0	0	Yes	False
detectMode	2	0	1	0	Yes	True
noChkMode	3	0	0	1	No	True

```
typeChk(X, Δ) {
        TR = findTypeRule(X);  // find a security typing rule that matches X
        Br = checkBreach(X, TR, Δ);      // type check to find a breach
        if ( Br ≠ null ) {               // if a leakage is found
            Update (Δ(L), Δ(Σ), Br, X); // save the new leakage info in L and Σ
            Continue = FALSE;            // prevent the clause expansion
        } else   Continue = TRUE;       // no information leaks
    return Continue;
}
```

Fig. 7-3: A basic security type checking algorithm of *typeCkh(X,Δ)*

```
typeChk(X, Δ) {
  if( ¬noChkMode(Δ) ){        // perform the information flow analysis
        TR = findTypeRule(X);  // find a security typing rule that matches X
        Br = checkBreach(X, TR, Δ);      // type check to find a breach
        if ( Br ≠ null ) {               // if a leakage is found
            Update (Δ(L), Δ(Σ), Br, X); // save the new breach in L and Σ
            if ( prevMode(Δ) )
               Continue = FALSE;        // prevent the clause expansion
            else                        // detectMode(Δ)
                 Continue = TRUE;       // detect and continue
            }
        } else   Continue = TRUE;       // no information leaks
   } else Continue = TRUE;             // no information flow analysis
   return Continue;
}
```

Fig. 7-4: The updated security type checking algorithm of *typeCkh(X,Δ)*

When a leakage is found, there might be cases in which the clause expansion failure itself leaks some information to the adversary and informs them that some high level information is blocked from them.

To minimise this kind of information leakage and to have more flexible secrecy policies, new options forming the type checking strategy can be defined in Π. In Table 7-1, the following three reaction policy modes and their priorities are shown: prevention (prevMode), detection (detectMode) and no-detection (noChkMode). The new secrecy policy is defined as the following: Π = (*Pre*, *Det*, *NoCk*),

in which users can choose a policy by selecting one of the Boolean values *Pre*, *Det* and *NoCk* (Table 7-1). Only one policy may be activated at a time; which means if more than one option is chosen, based on the defined priorities they may be overridden. E.g. both Π=(1,0,1) and (1,0,0) have the same effect, as *Pre* overrides *Det* and *NoChk* values and results in prevention mode.

7.2 Drawbacks of Dynamic Type Checking

The main disadvantage of purely run-time information flow analysis similar to the one discussed above, is that they can produce false negative results as they cannot detect implicit information flows. This is because in dynamic security analysis of LCC, not all execution paths of the program are checked. The following simple example show when the dynamic analysis can go wrong. All terms are low security and the only terms with high security levels are `Secret` and `this` (i.e. the current clause environment).

```
( publicMessage1 => a(publicAgent, P) <- check(Secret) )

or

 publicMessage2 => a(publicAgent, P)
```

The following rewrite rule handles the first part of the code:

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}}_{\Delta} c(M \Rightarrow A \leftarrow C, \Delta(L)) \quad if\ satisfied(C) \wedge typeChk(M \Rightarrow A \leftarrow C, \Delta)$$

Let us assume the constraint does not hold; i.e. `satisfied (check(Secret))` return *false*, so the first part of the conditional statement fails and the second part is processed by this rewrite rule:

$$M \Rightarrow A \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}}_{\Delta} c(M \Rightarrow A, \Delta(L)) \qquad if\ typeChk(M \Rightarrow A, \Delta),$$

then the type checking is as below:

$$\dfrac{\dfrac{this{:}agent\ h \in \Gamma}{\Gamma \vdash PID2{:}uTrm\ h}Id, agent\ h \leq agent\ l}{\Gamma \vdash this{:}agent\ l}Sub, \quad \dfrac{\dfrac{publicMessage2{:}l \in \Gamma}{\Gamma \vdash publicMessage2{:}l}Id, \dfrac{\dfrac{publicAgent{:}l \in \Gamma}{\Gamma \vdash publicAgent{:}l}Id, \dfrac{P{:}l \in \Gamma}{\Gamma \vdash P{:}uTrm\ l}Id}{\Gamma \vdash a(publicAgent, P){:}\ agent\ l}Agnt}{\Gamma \vdash publicMessage2 => a(publicAgent, P){:}op\ l}Snd$$

This is detected as a well-typed LCC command, which is wrong! This is because of a high security constraint as described in the Appendix.

Another possible problem is late detection of the insecure flow in run-time security checking of LCC interaction models. This may result in the rewriting of some illegal LCC expressions, thus changing the state of the agent before finding the breach - for example,detection of the breach after a high security message is sent to a low security agent is too late.

Generally, dynamic checking (in the best case), may assure that the current execution of an interaction model does not leak information, but does not tell us that the code is safe and will never reveal any confidential information in future, because it does not check all possible execution paths of the LCC program. In other words, if no breach occurs in dynamic checking, it means that there exists a secure execution path in the LCC interaction model. This is a *Liveness* property, which specifies that eventually "good things" do happen versus a *Safety* property, which states that no "bad things" occur during program execution (Halpern and Schneider 1987).

7.3 Static Information Flow Analysis

We can perform static analysis to overcome the drawbacks of dynamic methods.

The static checking explores all execution paths in LCC interaction models, hence it guarantees that detection of any insecure flow based on the defined type system. To perform a static type check, we can modify the LCC rewrite rules for the static type check, in a way that the whole expansion tree of an LCC clause is explored. In recursions, the clause is expanded if it has not already been expanded (Fig. 7-5).

$$a(R,I) :: B \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: E \quad if \ B \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(a(R,I)::E,\Delta) \tag{19}$$

$$A_1 \ or \ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \quad if \ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E \tag{20}$$

$$A_1 \ or \ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \quad if \ closed(A_1) \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(A_1 \ or \ E,\Delta) \tag{21}$$

$$A_1 \ then \ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \ then \ A_2 \quad if \ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E \tag{22}$$

$$A_1 \ then \ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} A_1 \ then \ E \quad if \ closed(A_1) \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(A_1 \ then \ E,\Delta) \tag{23}$$

$$A_1 \ par \ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E_1 \ par \ E_2 \quad if \ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E_1 \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E_2 \wedge typeChk(E_1 \ par \ E_2,\Delta) \tag{24}$$

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c\big(C \leftarrow M \Leftarrow A, \Delta(L)\big) \quad if \ typeChk(C \leftarrow M \Leftarrow A,\Delta) \tag{25}$$

$$M \Leftarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Leftarrow A, \Delta(L)) \quad if \ typeChk(M \Leftarrow A,\Delta) \tag{26}$$

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Rightarrow A \leftarrow C, \Delta(L)) \quad if \ typeChk(M \Rightarrow A \leftarrow C,\Delta) \tag{27}$$

$$M \Rightarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Rightarrow A, \Delta(L)) \quad if \ typeChk(M \Rightarrow A,\Delta) \tag{28}$$

$$null \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} c(null \leftarrow C, \Delta(L)) \quad if \ typeChk(null \leftarrow C,\Delta) \tag{29}$$

$$a(R,I) \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: B \quad if \ newClause(P, a(R,I) :: B) \wedge typeChk(a(R,I) \leftarrow C,\Delta) \tag{30}$$

$$a(R,I) \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: B \quad if \ newClause(P, a(R,I) :: B) \wedge typeChk(a(R,I) \leftarrow C,\Delta) \tag{31}$$

$$closed\big(c(X,L)\big) \tag{32}$$

$$closed(A \ or \ B) \leftarrow closed(A) \wedge closed(B) \tag{33}$$

$$closed(A \ par \ B) \leftarrow closed(A) \wedge closed(B) \tag{34}$$

$$closed(A \ then \ B) \leftarrow closed(A) \wedge closed(B) \tag{35}$$

$$closed(X :: B) \leftarrow closed(B) \tag{36}$$

Fig. 7-5: Static analysis of an LCC clause by expansion of an LCC clause.

7.4 Drawbacks of Static Type Checking

Static type checking to prevent insecure information flows conservatively detects implicit and explicit information flows, provides stronger security assurance and proves program correctness with reasonable computation cost (Sabelfeld and Myers 2003) and (Huang, et al. 2004), but it has some drawbacks. The main disadvantages of static type checking are:

1) False positive results: non-permissiveness of some secure information flows; static type checks suffer from over-approximation and may prevent genuinely useful interaction models.

2) Lack of information in static checking; we may not know the security level of all peers and components of the program, especially in an open MAS we may not know who will join the system during the interactions. In practice, security policies cannot be determined at the time of program analysis and may vary dynamically.

3) The proposed type system which is based on Denning's work ignores leaks via the termination behaviour of programs. Therefore they satisfy only termination-insensitive non-interference (Sabelfeld and Russo 2010), which is defined in the next section.

4) Exhaustive checking of every possible path in the execution tree of the LCC code is time-consuming, while dynamic checking is faster, because it concerns only one execution path of the program.

Some role names, constraints, variables and the security level of the terms may not be available to our static analysis. The LCC programmer or the expert who annotates the code by security levels may not know about the behaviour of some constraints and other variables, which will be available at run-time. E.g. in the cloud configuration case study, some general patterns are used and some constraints and roles' arguments are defined at execution time by the counterpart agent.

The following codes presents some examples that the static type checking rejects, although they do not cause any information leakage:

```
SecretMessage => a(publicAgent, P)<- smallerThan(PublicVar, PublicVar)
```

in which the constraint is never satisfied (because the public variable PublicVar cannot be smaller than itself), so under no circumstances will the secret message be sent to the public agent P. In a similar example bellow, the constraint is always satisfied, therefore the second part of the conditional statement, in which a secret message leaks, never runs and no message is sent.

```
( null <- equals(PublicVar, PublicVar)
 or
 SecretMessage => a(publicAgent, P) )
```

In general, any LCC code containing a low security expression within a high security constraint, which does not hold at run-time is rejected by static type checking, even though it is permissible. This is due to the fact that

the security checker is not guaranteed to know whether or not a constraint holds at the time the interaction model is checked, so it conservatively rejects the interaction model.

As mentioned before, information might also leak via termination behaviour of the program, e.g. in the following code:

```
a(secretAgent, S)::
          null <- notEqual(SecretID, 0) then
a(secretAgent, S)
```

The adversary learns that SecretID was 0, by observing the termination of the clause.

7.5 Non-interference

Non-interference is a popular information flow property that guarantees secrecy of information flow and tells us whether there is any information leakage in the information system. Non-interference was introduced by Goguen and Meseguer (1982), but its concept goes back to the notion of strong dependency introduced by Cohen (1977).

The intuition behind the non-interference property is that high-security input to the program must never affect low-security output. In other words, public outputs are not dependent on secret inputs. In the following secrecy analysis of the agents' interaction models, we consider received messages, role arguments, and sometimes constraint arguments as *input* and the sent messages as *output*. There are formulations of non-interference. In this section, we define the notion of non-interference for the LCC interaction models inspired by the definitions of Hedin and Sabelfeld (2011) and Becker (2010).

Before defining non-interference, we need to define *visibility*, *alikeness*, and *observational equivalence* as prerequisites:

Definition 7-1 (Visibility): The set $\mathbf{visible}_l(\Gamma)$ denotes the LCC terms in the context Γ that can be observed by other agents (or adversaries) with the security level l or higher:

$$\mathbf{visible}_l(\Gamma) = \{ T \in \Gamma \mid \Gamma(T) \leq l \}$$

Definition 7-2 (Alikeness[9])

$\mathbf{\Gamma 1 \approx_l \Gamma 2}$: Two security contexts $\Gamma 1$ and $\Gamma 2$ are alike up to the level l iff: $\mathbf{visible}_l(\Gamma 1) = \mathbf{visible}_l(\Gamma 2)$.

For example, if we have the following two contexts: $\Gamma 1 = \{ m1: l, m2: l, m3: h \}$ and $\Gamma 2 = \{ m1: l, m2: l, m3: h , m4: h \}$, then: $\boldsymbol{visible}_l(\Gamma 1) = \{m1, m2\}$ and $\boldsymbol{visible}_l(\Gamma 2) = \{m1, m2\}$, which means other agents with security level of at least l can see these information. We also have $\Gamma 1 \approx_l \Gamma 2$.

[9] *Alikeness* up to level *l* is known as *low equivalence* in the literature.

Recall the LCC clause expansion mechanism of an original LCC clause C_i into C_n in terms of the interaction model P:

$$C_i \xrightarrow{M_i, M_{i+1}, P, O_i}_{\Delta} C_{i+1}, \ldots, C_{n-1} \xrightarrow{M_{n-1}, M_n, P, O_n}_{\Delta} C_n$$

where security environment $\Delta = (\Gamma, K, \Pi, L, \Sigma)$ *and* O_n is an output message set that can expresse the observable *behaviour* of an agent by its counterpart agents. We now define the *Observational Equivalence* relation on *behaviour* as follows.

Definition 7-3 (Observational Equivalence[10])

$O_{n1} \equiv_l O_{n2}$: The observable behaviours of two clause expansions in terms of the interaction model P are observationally equivalent up to level *l*, if an adversary of level *l* cannot distinguish between O_{n1} and O_{n2}.

Observational equivalence of O_{n1} and O_{n2} can (imprecisely) be understood as two runs of an interaction model that are the same from the adversary's point of view. *Alikeness* and *observational equivalence* are then used to define the notion of *non-interference* for the LCC interaction models. In the following, for the sake of clarity, the notion of the security context Γ is used instead of the security environment Δ. This is safe to do, because in our investigation, Γ only changes within Δ.

Definition 7-4 (Non-interference)

$$\forall\, \Gamma 1, \Gamma 2.\ (\Gamma 1 \approx_l \Gamma 2) \wedge C_i \xrightarrow{M_i, M_{n1}, P, \boldsymbol{O_{n1}}}_{\boldsymbol{\Gamma 1}} C_{n1} \wedge C_i \xrightarrow{M_i, M_{n2}, P, \boldsymbol{O_{n2}}}_{\boldsymbol{\Gamma 2}} C_{n2} \Rightarrow (O_{n1} \equiv_l O_{n2})$$

This states that for any two contexts Γ1 and Γ2 which are *alike* up to level of *l*, a successful expansion of the LCC clause C_i in one of the contexts with behaviour O_{n1} and a successful expansion in the other context with behaviour O_{n1} guarantee that the behaviours are observationally equivalent.

Informally, if two clauses look the same to an adversary, they also behave the same. In other words, low output (the sent messages to an adversary) depends on low inputs (the immutable visible parts of the contexts).

The proposed security type system supports non-interference; Suppose C_i is a message sending operation $M \Rightarrow A$. If the type of the agent A is $agent\ h$, the typing rule Snd allows sending a message (with any security level) to the high security agent A, in either case, an adversary of level *l* cannot observe any output message. If the type of A is $agent\ l$, then the type system requires that $M : l$, then any the observable output of the LCC rewrite rule for an adversary of level *l* will be message M. The other cases of C_i that can have an observable behaviour are similar.

This definition of non-interference is termination-insensitive, by which we mean that it disregards information leaks due to the termination of the program (e.g. the last example in section 4.2). Thus, our type system cannot detect this type of insecure flow.

[10] *Observational Equivalence* is also called *indistinguishability*.

Although the notion of non-interference is a popular and natural way of describing confidentiality and integrity, it may be too restrictive for many applications (Hedin and Sabelfeld 2011). The next section addresses this issue.

7.6 Declassification

Declassification is intentional release of secret information by lowering security levels of information (Zdancewic and Myers 2001). Sometimes, we need a way of information declassification in our security system.

A typical example is any system that asks the user credentials for authentication. Consider the access request to a patient record by a specialist. Rejection of an incorrect password violates non-interference, because of the dependency between high input (i.e. password) and low output (i.e. rejection message). That implies the system leaks partial information about the password (i.e. incorrectness of the password) to a potential attacker. However, this leakage is unlikely, in this case, to give valuable information to the attacker.

To support declassification in our security type system, we can deliberately downgrade the security classification of information by adding the following rule:

$$declassify(h) = l$$

This violates non-interference, but it may be necessary for some applications. We should carefully declassify information. In (Sabelfeld and Sands 2005) the principles and dimensions of declassification are described by identifying *what* can be declassified, *who* controls the declassification, *where* the declassification happens and when the declassification can occur relative to other events in the program.

8. Conclusions and Future Work

In this paper, we have addressed information leakage problems in open MAS governed by interaction models and, consequently, developed secrecy analysis frameworks for an agent language called LCC. Explicit and implicit insecure information flows have been explained using a number of LCC examples.

We have proposed and implemented a language-based information flow framework to analyse information leaks in LCC interaction models. The security-typed LCC has been introduced by inventing a security type system, which formally defines security levels, security types and the type inference rules. Next, the proposed type system has been evaluated and proven to hold basic, important properties: *type soundness*, *simple security* and *confinement*.

We have discussed two approaches for applying the security type system on the agent interaction models; dynamic (run-time) and static type checking. Two disadvantages of dynamic information flow analysis are its inability to detect implicit information flows and late detection of insecure flow. All execution paths of the program are not checked in dynamic analysis and some paths are disregarded, which could lead to implicit information flows. To overcome this problem, we provide the following options:

a) Extending the dynamic approach with the control flow stack mechanism described in (S. Bijani 2013, 103)

b) Using the static approach instead of dynamic analysis:

The static approach is a promising method that prevents insecure implicit and explicit flows, but it suffers mainly from non-permissiveness, so it may also reject genuine flows. Another drawback of the static analysis problem is that due to the dynamic behaviour of open MAS, there is a lack of information about the security classification of agents, constraints, etc. before run-time.

c) The combined approach: using both static and dynamic methods

In this approach, static analysis is performed on an interaction model and if it is rejected, the system informs the user. The user then can decide to continue with the interaction model and perform dynamic checking at run-time. There is also a hybrid approach (Russo and Sabelfeld 2010), in which static and dynamic analysis are merged to take the best of both worlds. This is especially useful in flow sensitive analysis. In flow sensitivity, variables may store values of different sensitivity (low and high) over the course of the interaction. We leave flow-sensitivity analysis in LCC interaction models as a topic for future research.

To address the false alarm of static approach, static analysis is performed on an interaction model and if it is rejected, the system informs the user. The user then can decide to continue with the interaction model and perform dynamic checking at run-time. The proposed security type system supports non-interference. The intuition behind the non-interference property is that high-security input to the program must never affect low-security output. This definition of non-interference is termination-insensitive, by which we mean that it disregards information leaks due to the termination of the program. As non-interference may be too restrictive for many applications, the proposed framework supports declassification.

Adaptation of the proposed security type system for similar first-class agent protocol languages such as *MAP*[11] and *RASA* is straightforward. Similar idea can be applied on other agent languages with further edition. We have focused on one aspect of security, i.e. confidentiality. The other important aspect of security is integrity. We would suggest defining other security properties for security typing that guarantee integrity through analysis of agents' interactions in this regard. We also leave automatic security annotation of agent interaction models (with secrecy labels) as another topic for future research.

Bibliography

Becker, Moritz Y. 2010. "Information Flow in Credential Systems." *Computer Security Foundations Symposium, IEEE* (IEEE Computer Society) 0: 171-185.

Bierman, Elmarie, and Elsabe Cloete. 2002. "Classification of Malicious Host Threats in Mobile Agent Computing." *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology.* South Africa: South African Institute for Computer Scientists and Information Technologists. 141-148.

Bijani, S., D. Robertson, and D. Aspinall. 2011. "Probing Attacks on Multi-agent Systems using Electronic Institutions." *Declarative Agent Languages and Technologies Workshop (DALT), AAMAS 2011.*

[11] MAP: Multi-Agent Protocol language

Bijani, Shahriar. 2013. *Securing Open Multi-agent Systems Governed by Electronic Institutions, PhD Thesis.* Edinburgh University.

Bijani, Shahriar, and David Robertson. 2014. “A Review of Attacks and Security Approaches in Open Multi-agent Systems.” *Artificial Intelligence Review* (Springer) 42: 607-636.

Buccafurri, F, A. Comi, G Lax, and D Rosaci. 2016. “Experimenting with Certified Reputation in a Competitive Multi-Agent Scenario.” *IEEE Intelligent Systems* (IEEE) 31 (1): 48-55.

Cohen, Ellis. 1977. “Information Transmission in Computational Systems.” *ACM SIGOPS Operating Systems Review* 11 (5): 133-139.

Denning, D. E. 1976. “A Lattice Model of Secure Information Flow.” *Communications of the ACM* 19 (5): 236-243.

Denning, Dorothy E., and Peter J. Denning. 1977. “Certification of Programs for Secure Information Flow.” *Communications of the ACM* 20 (7): 504-513.

Endsuleit, Regine, and Arno Wagner. 2004. “Possible Attacks on and Countermeasures for Secure Multi-Agent Computation.” *Proceedings of the International Conference on Security and Management, SAM '04,.* Las Vegas,Nevada, USA. 221-227.

Esteva, Marc, Juan-Antonio Rodriguez-Aguilar, Carles Sierra, Pere Garcia, and Josep L Arcos. 2001. “On the Formal Specification of Electronic Institutions.” In *Agent Mediated Electronic Commerce*, 126-147.

Goguen, Joseph A, and Jose Meseguer. 1982. “Security Policies and Security Models.” *IEEE Symposium on Security and Privacy.*

Halpern, Bowen, and Fred B Schneider. 1987. “Recognizing Safety and Liveness.” *Distributed Computing* 2 (3): 117-126.

Hedin, Daniel, and Andrei Sabelfeld. 2011. *A Perspective on Information-flow Control.* Proc. of the 2011 Marktoberdorf Summer School. IOS Press.

Hedina, Yaqin, and Esmiralda Moradian. 2015. “Security in Multi-Agent Systems.” *Procedia Computer Science, Knowledge-Based and Intelligent Information & Engineering Systems 19th Annual Conference, KES-2015.* 1604-1615.

Hennigan, E, C Kerschbaumer, S Brunthaler, and M Franz. 2011. *Implementation Details of Dynamic Information Flow Security Type Systems.* Technical Report 11-03, Dept of Information and Computer Science, University of California, Irvine.

Huang, Yao-Wen, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. “Securing Web Application Code by Static Analysis and Runtime Protection.” *the 13th international conference on World Wide Web.* ACM. 40-52.

Idrissi, Hind, El Mamoun Souidi, and Arnaud Revel. 2015. “Security of Mobile Agent Platforms Using Access Control and Cryptography.” *Agent and Multi-Agent Systems: Technologies and Applications* (Springer) 27-39.

Jansen, Wayne, and Tom Karygiannis. 2000. “Mobile Agent Security.” *National Institute of Standards and Technology (NIST) Special Publication 800-19.*

Joseph, Sindhu, Adrian P. Perreau de Pinninck, David Robertson, Carles Sierra, and Chris Walton. 2006. “OpenKnowledge Deliverable 1.1: Interaction Model Language Definition.” *http://groups.inf.ed.ac.uk/OK/Deliverables/D1.1.pdf.*

Lee, Hyungjick, Jim Alves-Foss, and Scott Harrison. 2004. “The Use of Encrypted Functions for Mobile Agent Security.” *the 37th Annual Hawaii International Conference on System Sciences (HICSS'04).* IEEE Computer Society. 10.

Majumdar, Anirban, and Clark Thomborson. 2005. “On the Use of Opaque Predicates in Mobile Agent Code Obfuscation.” In *Intelligence and Security Informatics*, 255-236. Springer Berlin / Heidelberg.

Mouratidis, Haralambos, Paolo Giorgini, and Michael Weiss. 2003. “Integrating Patterns and Agent-Oriented Methodologies to Provide Better Solutions for the Development of Secure Agent Systems.” *Workshop on Expressiveness of Pattern Languages 2003, at ChiliPLoP (2003).*

Necula, George, and Peter Lee. 1998. “Safe, Untrusted Agents Using Proof-Carrying Code.” In *Mobile Agents and Security*, by Giovanni Vigna, 61-91. Springer Berlin / Heidelberg.

Ngereki, Anthony M. 2015. *Protecting mobile agents from malicious hosts in a distributed network.* University of Nairobi.

Odubiyi, J B, and Abdur R Choudhary. 2007. “Building security into an IEEE FIPA compliant multiagent system.” *Proceedings of the 2007 IEEE Workshop on Information Assurance, IAW.* West Point, NY, United states: IEEE Computer Society. 49-55.

Oey, M. A. , M. Warnier, and F. M. T. Brazier. 2010. “Security in Large-Scale Open Distributed Multi-Agent Systems.” In *Autonomous Agents*, by Vedran Kordic, 107-130. IN-TECH.

Ogunnusi, Olumide Simeon, and Olasunkanmi Okunola Ogunlola. 2015. “Solutions to Mobile Agent Security Issues in Open-Multi-agent Systems.” *International Research Journal of Engineering and Technology* 601-609.

Ohno, Ken, Takahiro Uchiya, Ichi Takumi, and Tetsuo Kinoshita. 2016. “Security mechanism for DASH agent framework.” *Consumer Electronics, 2016 IEEE 5th Global Conference on.* IEEE. 1-2.

Page, John P, Arkady B Zaslavsky, and Maria T Indrawan. 2005. “Extending the buddy model to secure variable sized multi agent communities.” *Proceedings of the Second International Workshop on Safety and Security in Multiagent Systems.* Utrecht, Netherlands. 59-75.

Pierce, B. 2002. *Types and Programming Languages.* The MIT Press.

Poslad, S, and M Calisti. 2000. “Towards improved trust and security in FIPA agent platforms.” *Workshop on Deception, Fraud and Trust in Agent Societies.* Spain.

Poslad, Stefan, Patricia Charlton, and Monique Calisti. 2002. “Specifying Standard Security Mechanisms in Multi-agent Systems.” *Trust, Reputation, and Security: Theories and Practice, AAMAS 2002 International Workshop.* Bologna, Italy: Springer Berlin - Heidelberg. 122--127.

Quillinan, Thomas B, Martijn Warnier, Michel A Oey, Reinier J Timmer, and Frances M Brazier. 2008. “Enforcing Security in the AgentScape Middleware.” *Proceedings of the 1st International Workshop on Middleware Security (MidSec).* ACM.

Riordan, James, and Bruce Schneier. 1998. “Environmental Key Generation Towards Clueless Agents.” *Mobile Agents and Security.* Springer-Verlag. 15-24.

Robertson, David. 2005. *A Lightweight Coordination Calculus for Agent Systems.* Vol. 3476/2005, in *Declarative Agent Languages and Technologies II*, 183--197. Springer Berlin / Heidelberg.

Robertson, David, Adam Barker, Paolo Besana, Alan Bundy, Yun Heh Chen-Burger, David Dupplaw, Fausto Giunchiglia, et al. 2009. “Models of Interaction as a Grounding for Peer to Peer Knowledge Sharing.” *Advances in Web Semantics I* (Springer-Verlag) 4891: 81--129.

Robles, Sergi. 2008. *Trust and Security.* Vol. Chapter 4, in *Issues in Multi-Agent Systems: the AgentCities.ES Experience*, by A. Moreno and Juan Pavn, 87- 115. BirkhÃ¤user Basel.

Rosaci, D. 2012. “Trust Measures for Competitive Agents.” *Knowledge-based Systems (KBS)* (Elsevier) 28 (46): 38-46.

Russo, A., and A. Sabelfeld. 2010. “Dynamic vs. Static Flow-sensitive Security Analysis.” *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE.* IEEE. 186-199.

Sabelfeld, A., and A. C. Myers. 2003. “Language-based Information-flow Security.” *IEEE Journal on Selected Areas in Communications* 21 (1): 5-19.

Sabelfeld, Andrei, and Alejandro Russo. 2010. “From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research.” In *Perspectives of Systems Informatics*, 352-365. Springer Berlin Heidelberg.

Sabelfeld, Andrei, and David Sands. 2005. “Dimensions and Principles of Declassification.” *18th IEEE Workshop Computer Security Foundations. CSFW-18.* 255-269.

Santos, J. F., T. Jensen, T. Rezk, and A. Schmitt. 2015. “Hybrid Typing of Secure Information Flow in a JavaScript-Like Language.” *International Symposium on Trustworthy Global Computing.* Springer . 63-78.

Sarhan, Zahi A M Abu, and As'ad Mahmoud As'ad. Alnaser. 2014. “Information Security Approach in Open Distributed Multi-Agent Virtual Learning Environment.” *International Journal of Computer Science & Information Technology* 6 (1): 15-28.

Smith, Geoffrey, and Dennis Volpano. 1998. “Secure Information Flow in a Multi-threaded Imperative Language .” *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 355-364.

Tan, H.K., and L. Moreau. 2002. “Extending Execution Tracing for Mobile Code Security.” *Second International Workshop on Security of Mobile MultiAgent Systems (SEMAS 2002).* Bologna, Italy. 51–59.

Tan, Juan J, Stefan Poslad, and Yanmin Xi. 2004. “Policy Driven Systems for Dynamic Security Reconfiguration.” *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).* IEEE Computer Society. 1274--1275.

Vazquez-Salceda, J, J A Padget, U Cortes, A Lopez-Navidad, and F Caballero. 2003. “Formalizing an electronic institution for the distribution of human tissues.” *Artificial Intelligence in Medicine* 27: 233-258.

Volpano, Dennis M., and Geoffrey Smith. 1997. “A Type-Based Approach to Program Security.” *7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development.* Springer-Verlag. 607-621.

Wahbe, R., S. Lucco, and T. Anderson. 1993. “Efficient Software-Based Fault Isolation.” *the Fourteenth ACM Symposium on Operating Systems Principles* 203-216.

Wang, Hongzue, Vijay Varadharajan, and Yan Zhang. 1999. “A Secure Communication Scheme for Multiagent Systems.” *PRIMA '98: Selected papers from the First Pacific Rim International Workshop on Multi-Agents, Multiagent Platforms.* London, UK: Springer-Verlag. 174--185.

Wong, Hao C, and Katia Sycara. 1999. “Adding Security and Trust to Multi-Agent Systems.” *Proceedings of Autonomous Agents '99 Workshop on Deception, Fraud, and Trust in Agent Societies.* 149 - 161.

Zdancewic, Steve, and Andrew C Myers. 2001. “Robust Declassification.” *IEEE Computer Security Foundations Workshop.* 15-23.

Zhu, Ping, and Guangli Xiang. 2011. “The Protection Methods for Mobile Code Based on Homomorphic Encryption and Data Confusion.” *Management of e-Commerce and e-Government (ICMeCG), 2011 Fifth International Conference on.* doi:10.1109/ICMeCG.2011.48.

Appendix A

Table 0-1 to Table 0-3 summarise the acceptable and unacceptable explicit and implicit information flows in message passing, role assignment and conditional statements in LCC codes. It is assumed that a secret LCC term and a public LCC term are shown by *H* and *L*, respectively.

In Table 0-1, permissible and impermissible information flows in sending a message, based on the security levels of the sender, the receiver and the message are shown. The three undesirable flows are: 1) sending a high security message by a low security sender to a low security receiver, 2) sending a high security message by a low security sender to a high security receiver and 3) sending a high security message by a high security sender to a low security receiver.

Table 0-1. Permissible and impermissible information flows in sending a message based on the security levels of the sender, the receiver and the message

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

Table 0-2 `shows different combinations of role allocation (without arguments) to` agent identifiers, in which the only illegal flow is from a high security role to a low security agent.

Table 0-2. Permissible information flows in the LCC role definition regarding the security levels of the role and the agent identifier

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

The sources of implicit information flows are conditional operations. Table 0-3 summarises secure and insecure information flows in LCC via conditional expressions in the form of `(Operation1 <- Constraint) or Operation2`. There is one generic insecure flow from constraints to operations, when the operation is public but the constraint is secret. In Table 0-3, `Max_Operation` is the maximum security level of `Operation1` and `Operation2`.

Table 0-3. Permissible and impermissible information flows in LCC conditional expressions regarding the security levels of the operations and the constraint. `Max_Operation = max` (`Operation1` level, `Operation2` level)**.**

Constraint	Max_Operation	Permissible Flow
L	L	Yes
L	H	Yes
H	L	No
H	H	Yes

Appendix B

Proof of Theorem 6.1 (Progress):

By induction on the structure of $\Gamma \vdash L : \varphi$; we apply the induction on the smaller derivations of typing rules assuming this property holds for all of these sub-derivations (above the line in typing rules) and proceed by case analysis.

Case *Seq*: $L = A_1\ then\ A_2$ and L: $op\ \tau$, so A_1: $op\ \tau$, A_2: $op\ \tau$

By the induction hypothesis either A_1 is a final step or else some A_1' exists that $A_1 \rightsquigarrow A_1'$. *Similarly,* either A_2 is a final step or else some A_2' exists that $A_2 \rightsquigarrow A_2'$. If both A_1 and A_2 are final steps (*closed*), based on the following LCC rewriting rule in Fig 7-2:

$$closed(A\ then\ B) \leftarrow closed(A)\ \wedge\ closed(B)$$

$A_1\ then\ A_2$ is a final step. If A_1 is a final step and $A_2 \rightsquigarrow A_2'$, according to the following rewrite rule:

$$A_1\ then\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} A_1\ then\ E \qquad if\ closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} E$$

$A_1\ then\ A_2 \rightsquigarrow A_2'$. If both A_1 and A_2 are not final steps, which means $A_1 \rightsquigarrow A_1'$ and $A_2 \rightsquigarrow A_2'$, based on the following LCC rewriting rule:

$$A_1\ then\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} E\ then\ A_2 \qquad if\ A_1 \xrightarrow{R_i, M_i, M_o, P, O} E$$

$A_1\ then\ A_2 \rightsquigarrow A_1'$. If $A_1 \rightsquigarrow A_1'$ and A_2 is a final step, it is not an acceptable state in LCC, so the result is *false*: $A_1\ then\ A_2 \rightsquigarrow false$.

Case *If1*: $L = M \Rightarrow A \leftarrow C$ and L: $op\ \tau$, so $M \Rightarrow A$: $op\ \tau$ and C: $con\ \tau$,

By the induction hypothesis either C is a final step or else some C' exists that $C \rightsquigarrow C'$. If C is a final step, in the following LCC rewriting rule in Fig 7-2:

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}} c(M \Rightarrow A \leftarrow C) \qquad if\ satisfied(C),$$

either the evaluation of *satisfied*(C) is true, so $M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}} c(M \Rightarrow A \leftarrow C)$ or else it returns *false*, which indicates $M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_i, P, \emptyset} false$. In either case, L ends up in a *closed* state which means a final step.

If $C \rightsquigarrow C'$, it means that C is a compound constraint C' that is equal to $\neg C_1$, $C_1 \wedge C_2$ or $C_1 \vee C_2$, so based on one of the following rewrite rules in Fig 7-2:

$satisfied(\neg C_1) \leftarrow \neg satisfied(C_1)$,

$satisfied(C_1 \vee C_2) \leftarrow satisfied(C_1)\ \vee\ satisfied(C_2)$,

$satisfied(C_1 \wedge C_2) \leftarrow satisfied(C_1)\ \wedge\ satisfied(C_2)$,

then we have $L \rightsquigarrow M \Rightarrow A \leftarrow C'$.

We showed only a subset of the cases; other cases are similar. □

Proof of Theorem 6.2:

By induction on the structure of $\Gamma \vdash L : \varphi$ and proceed by case analysis (similar to the proof of Theorem 6-1).

Case *Seq*: $L = A_1\ then\ A_2$ and L: $op\ \tau$

We know that L is well-typed, so we have A_1: $op\ \tau\ and\ A_2$: $op\ \tau$. According to the following rewrite rules:

$A_1\ then\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} E\ then\ A_2 \qquad if\ A_1 \xrightarrow{R_i, M_i, M_o, P, O} E)$

$A_1\ then\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} A_1\ then\ E \qquad if\ closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i, M_i, M_o, P, O} E$

the transition $L \rightsquigarrow L'$ happens either by $A_1 \xrightarrow{R_i, M_i, M_o, P, O} E$ or when A_1 is a final step (closed(A_1)), by $A_2 \xrightarrow{R_i, M_i, M_o, P, O} E$.

If ¬closed(A_1), the $A_1 \xrightarrow{R_i, M_i, M_o, P, O} E$ can be derived by any of the clause expansion rewrite rules, some of the cases are shown; others are similar:

1) Subcase $A_1 = a(R, I) \leftarrow C$

By the induction hypothesis, we have $a(R, I) \leftarrow C : op\ \tau$, $A_1 \rightsquigarrow A_1'$ and $A_1' : op\ \tau$. The following rewrite rule, which deals with recursion in LCC, is the only rule that expands A_1:

$$a(R, I) \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \emptyset} a(R, I) :: B \qquad if\ clause(P, a(R, I) :: B) \wedge satisfied(C).$$

So we have $A_1' = a(R, I) ::$ B. Consequently, $A_1\ then\ A_2 \rightsquigarrow a(R, I) ::$ B $then\ A_2$ and $A_1\ then\ A_2 : op\ \tau$.

2) Subcase $A_1 = M \Rightarrow A$

By the induction hypothesis, we have $M \Rightarrow A : op\ \tau$, $A_1 \rightsquigarrow A_1'$ and $A_1' : op\ \tau$. The rewrite rule that handles A_1 is only $M \Rightarrow A \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}} c(M \Rightarrow A)$. So we have $A_1' = c(M \Rightarrow A)$. Consequently, $A_1\ then\ A_2 \rightsquigarrow M \Rightarrow A\ then\ A_2$ and $A_1\ then\ A_2 : op\ \tau$.

Other subcases are similar.

Case *If1*:

We know that $L \equiv M \Rightarrow A \leftarrow C$ is well-typed; $L : op\ \tau$, so $M \Rightarrow A : op\ \tau$ and $C : con\ \tau$, we also have $L \rightsquigarrow L'$.

Based on the LCC rewriting rule (9) in Fig 7-2 and the definition of transition $\rightsquigarrow$, the possible expansions of *L* to *L'* are:

1) If $C \rightsquigarrow C'$, it means that C is equal to a compound constraint C' that might be $\neg C_1$, $C_1 \wedge C_2$ or $C_1 \vee C_2$. Then we have $L' \equiv M \Rightarrow A \leftarrow C'$. By the induction hypothesis, $C' : con\ \tau$, hence, based on the type rule *If1*, $M \Rightarrow A \leftarrow C' : op\ \tau$.

2) If $satisfied(C)$ returns *true*, then it is a final step: $M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \{M \Rightarrow A\}} c(M \Rightarrow A \leftarrow C)$ and based on the type rule *Close*, the typing of the LCC expression will not be changed:

$$\frac{\Gamma \vdash M \Rightarrow A \leftarrow C : op\ \tau}{\Gamma \vdash c(M \Rightarrow A \leftarrow C) : op\ \tau} Close$$

3) If *satisfied* (C) returns *false* which means: $M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \emptyset} false$, then based on the type rule *False* we have: *false: op h,* i.e. *L'*: *op h*

We showed only a subset of the cases; other cases are similar. □