Secure information sharing in social agent interactions using information flow analysis

Shahriar Bijani\textsuperscript{a,}\textsuperscript{*}, David Robertson\textsuperscript{b}, David Aspinall\textsuperscript{b}

\textsuperscript{a} Computer Science Department, Shahed University, Persian Gulf Highway, Tehran, Iran
\textsuperscript{b} Informatics School, University of Edinburgh, 10 Crichton St, Edinburgh, UK

\section{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\textsuperscript{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 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

\begin{footnotesize}
\textsuperscript{*} Corresponding author.
\end{footnotesize}

\begin{footnotesize}
E-mail addresses: bijani@shahed.ac.ir (S. Bijani), dr@inf.ed.ac.uk (D. Robertson), david.aspinall@ed.ac.uk (D. Aspinall).

\end{footnotesize}

\begin{footnotesize}
\textsuperscript{1} Open Systems Interconnection.
\end{footnotesize}
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.

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 types 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:

\[ C_1 \xrightarrow{M_i,O_i,P.O_i} C_{i+1} \circ \ldots \circ C_{n-1} \xrightarrow{M_{n-1},O_{n-1},P.O_{n-1}} C_n \]

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 \( M_i, O_i \) 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 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 et al., 2010; Ngereki, 2015); some are based on cryptography while others are not; e.g. code obfuscation (Majumdar and Thomborson, 2005), function encryption (Lee et al., 2004; Zhu and Xiang, 2011), environmental key generation (Riordan and Schneier, 1998), execution tracing (Tan and Moreau, 2002), and agent monitoring (Page et al., 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 et al., 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 MAS 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 PKI2) are used. Most research on secure MAS follows this approach. Wong and Sycara (1999), Idrissi et al. (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 et al. (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 et al. (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 et al. (2004).

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 Oggunusi 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 indicates “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 L with two security levels, low l and high h, security level l ∈ (L, ≤), where l ≤ h and ≤ is a partial order relation.

We need to ensure that information flows only upwards in the lattice (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 l).

**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)

\[ \text{\ldots} \]

\[ \text{\ldots} \]

---

2 Public Key Infrastructure.

3 Virtual Private Network.
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.

\[
\text{PublicMessage} \Rightarrow a(\text{secretAgent}, \text{S})
\]

and

\[
\text{PublicMessage} \Leftarrow a(\text{secretAgent}, \text{S})
\]

(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:

\[
\text{null} \Leftarrow \text{assign}(\text{PublicVariable}, \text{SecretID})
\]

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:

\[
a(\text{secretAgent}, \text{PublicAgent})::
\]

\[
\ldots
\]

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

\[
a(\text{publicAgent}, \text{Private Agent})::
\]

\[
\ldots
\]

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 \leq 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.

\[
\text{PublicMessage} \Rightarrow a(\text{publicAgent}, P) \Leftarrow
\]

\[
\text{lessOrEqual(SecretID, 10)}
\]

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.

\[
\text{publicMsg}(X) \Rightarrow a(\text{publicAgent}, P) \Leftarrow
\]

\[
\text{lessOrEqual(SecretID, X)}
\]

or

\[
\text{publicMsg}(1) \Rightarrow a(\text{publicAgent}, P)
\]

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:

\[
a(\text{myAgent1}(X), Q)::
\]

\[
\{
\text{a(\text{myAgent1}(X1), Q) \Leftarrow lessOrEqual (SecretID, X) \land increase(X1,X,1)}
\]

or

\[
\text{publicMsg(X)\Rightarrow a(\text{publicAgent}, P)}
\]

\[
\% \text{ when X equals SecretID + 1}
\]

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.

\[
a(\text{myAgent2}(X,R), Q)::
\]

\[
\ldots
\]

\[
\{
\text{publicMsg(R)\Rightarrow a(\text{publicAgent}, P) \Leftarrow lessOrEqual(SecretID, X) \land increase(X1,X,1)}
\]

\[
\text{then a(\text{myAgent2}(X1,R), Q)}
\]

\[
\}
\]

or

\[
a(\text{myAgent2}(X,X), Q) \Leftarrow equals(SecretID, X)
\]

Information may leak because of the termination behaviour of the interaction model. 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.

\[
4 \text{ This is also called information leaks via the termination channel.}
\]
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 et al., 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.

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 $\Gamma$ is a type environment that maps term $T$ to type $\varphi$. Here are some definitions:

**Definition 5.1 (Security Type Environment).** A security type environment (context) $\Gamma$ is a finite map from LCC terms to security types and is defined by

$$\Gamma := \text{empty} | \Gamma.T : \varphi,$$

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

**Definition 5.2 (Security Types).** The security types of our system are defined as follows:

$$\varphi = t | \text{uTrm } t | \text{agent } t | \text{con } t | \text{op } t,$$

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

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

a. $\Gamma \vdash X : \text{uTrm } t$ 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 $t$ in context $\Gamma$.

b. $\Gamma \vdash T : t$ 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 $t$ in context $\Gamma$.

c. $\Gamma \vdash \text{a(R,Id)} : \text{agent } t$ means in the agent definition, agent identifier Id, to which a role is assigned has a security level $t$ or higher in context $\Gamma$.

d. $\Gamma \vdash C : \text{con } t$ means that the constraint name and every argument within $C$ has a security level $t$ or lower in context $\Gamma$.

e. $\Gamma \vdash \text{D}: \text{op } t$ means that every receiver of a message or any updated identifier in an operational command (i.e. Def) has a security level $t$ or higher in context $\Gamma$.

$$\Gamma \vdash T : \varphi, \Gamma \vdash \text{Def op } t \Rightarrow \Gamma \vdash \text{Def op } t \Rightarrow \Gamma \vdash \text{Def op } t$$

Fig. 5.1. The security typing rules for LCC.

56

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

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-updateable and to have a security type, $t$, 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 \textit{AgentRule} states that in order to assign a role to an agent in form of \(\\langle R, ID \rangle\) that has security type of agent \(r\), we must first check whether the security type of the role \(R\) is \(r\) and then whether the security type of the agent identifier is \(uTrm\). The environment \(\Gamma\) 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 \textit{Id} and \textit{uld} explain if an LCC identifier (a constant or a variable) is defined in the environment \(\Gamma\), security types \(r\) or \(uTrm\) \(r\) may be assigned to it. Selection of \(r\) or \(uTrm\) \(r\) 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 \(\Gamma\) by the \textit{Init} rule.

The rule \textit{Snd} expresses that if the sender (this), the receiver \(A\) and the message \(M\) have security level \(r\), then the sending operation \((M \Rightarrow A)\) can have the security type \(op\ r\). The rule \textit{Rsv} is the same as \textit{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 message in \textit{Snd} and \textit{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 \textit{Agent}, \textit{Snd} and \textit{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 \textit{Call} rule states that when we call a constraint, the security level of itsfunctor,\footnote{Non-numeric constant.} the security level of the current clause (this) and the security level of either read-only arguments (\(\Gamma_i \vdash r\)) or write-only arguments (\(\Gamma'_i \vdash uTrm\ r\)) have to be the same. This ensures us that a public agent cannot access secret constraints and a public constraint may not reveal secret information to a public agent. The \textit{Struct} rule denotes that in structured non-updateable terms (such as messages, role names and read-only arguments) the security types and levels of the functor \(f\) and the arguments \(\Gamma\) must be the same. The rules \textit{And}, \textit{Or} and \textit{Not} regulate the composition of constraints in LCC. The rule \textit{It} 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 \textit{Seq} say that if two LCC expressions have the same security level, their composition has also that security level. The \textit{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 \textit{Role} checks whether the role definition \(\alpha(R,ID)\) 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 \textit{AgentRule}, \textit{uTrmRule} and \textit{opRule} are contravariant\textsuperscript{7} and the \textit{conRule} is covariant.\textsuperscript{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.

An example annotation of an LCC interaction model that assigns security levels to LCC terms is shown in Fig. 5.3.

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 (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.

\textbf{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.

\textbf{Definition 6.2 (Transition \(L \Rightarrow L'\)).} Transition of \(L \Rightarrow L'\) means \(L'\) is an expansion of LCC expression \(L\), either as a result of an LCC rewrite rule \(L \Rightarrow_{R,M,M_0,P,O} L'\) 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 \(C_1 \land C_2\), when \(C\) is unfolded into \(C_1 \land C_2\) then \(L' \equiv null \leftarrow C_1 \land C_2\) and we can write \(L \Rightarrow L'\).
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(sold, l). label{afford, l}. label{sold, l}. label{P, l}.

Fig. 5.3. Annotation of an LCC interaction model.

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.

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.

LCC clauses are well-typed by ensuring that every expansion of them is performed according to 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 $X \overset{R, M, M', PO}{\Rightarrow} Y$, where $Y$ is the expansion of $X$ performing role $R$, $M$ is the initial set of messages, $O$ is the output

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 $L \leadsto L'$.

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 $L \leadsto L'$, 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 cannot have security level higher than the message level or the constraint level. In other words, when a message (or a constraint) has a security level $\tau$, it assures us that it will not reveal any information with security level more than $\tau$.

'No Write Down' means having an operational command with the security level of $\sigma \tau$ (any operational command), any updatable identifier within it has a security level higher than or equal to $\tau$. 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 $\tau$, i.e. $\Gamma \vdash T : \tau$ or $\Gamma \vdash T : con \tau$, then $T$ contains only identifiers with security level not higher than $\tau$.

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 $\tau$.

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.
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.

\[ a(R, I) \vdash E \quad \text{if} \quad B \vdash a(R, I) : E \land \text{typeChk}(a(R, I) : E, \Delta) \quad (1) \]

\[ A_1 \lor A_2 \vdash a(R, I) : E \quad \text{if} \quad \neg \text{closed}(A_2) \land A_1 \vdash a(R, I) : E \quad (2) \]

\[ A_1 \lor A_2 \vdash a(R, I) : E \quad \text{if} \quad \neg \text{closed}(A_1) \land A_2 \vdash a(R, I) : E \quad (3) \]

\[ A_1 \text{ then } A_2 \vdash E \quad \text{then} \quad A_2 \vdash a(R, I) : E \quad (4) \]

\[ A_1 \text{ then } A_2 \vdash a(R, I) : E \quad \text{if} \quad \text{closed}(A_1) \land A_2 \vdash a(R, I) : E \land \text{typeChk}(A_1 \text{ then } A_2, E, \Delta) \quad (5) \]

\[ A_1 \text{ par } A_2 \vdash a(R, I) : E \quad \text{if} \quad A_1 \vdash a(R, I) : E_1 \land A_2 \vdash a(R, I) : E_2 \land \text{typeChk}(E_1 \text{ par } E_2, \Delta) \quad (6) \]

\[ C \leftarrow M \vdash \frac{a(R, I) : E}{\Delta} c(C \leftarrow M \leftarrow A, \Delta(L)) \quad \text{if} \quad (M \Rightarrow A) \in M_i \land \text{typeChk}(C \leftarrow M \leftarrow A, \Delta) \land \text{satisfy}(C) \quad (7) \]

\[ M \Rightarrow A \vdash C \quad \text{if} \quad (M \Rightarrow A) \in M_i \land \text{typeChk}(M \Rightarrow A, \Delta) \quad (8) \]

\[ M \Rightarrow A \vdash C \quad \text{if} \quad \text{satisfied}(C) \land \text{typeChk}(M \Rightarrow A \land C, \Delta) \quad (9) \]

\[ M \Rightarrow A \vdash C \quad \text{if} \quad \text{typeChk}(M \Rightarrow A, \Delta) \quad (10) \]

\[ \text{null} \leftarrow C \quad \text{if} \quad \text{null} \leftarrow C \land \Delta(L) \quad (11) \]

\[ a(R, I) \vdash C \quad \text{if} \quad \text{clause}(P, a(R, I) : E) \land \text{satisfied}(C) \land \text{typeChk}(a(R, I) : E, \Delta, C) \quad (12) \]

\[ a(R, I) \vdash C(x, L) \quad (13) \]

\[ \text{closed}(a) \leftrightarrow \text{closed}(A) \lor \text{closed}(B) \quad (14) \]

\[ \text{closed}(A \text{ or } B) \leftrightarrow \text{closed}(A) \land \text{closed}(B) \quad (15) \]

\[ \text{closed}(A \text{ par } B) \leftrightarrow \text{closed}(A) \land \text{closed}(B) \quad (16) \]

\[ \text{closed}(A \text{ then } B) \leftrightarrow \text{closed}(A) \land \text{closed}(B) \quad (17) \]

\[ \text{closed}(X : B) \leftrightarrow \text{closed}(B) \quad (18) \]

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 \( \Delta \), the interpreter only keeps a record of the confidentiality breaches and continues to expand the expression X. Selection of the no-detection policy (noChkMode) bypasses the information flow analysis and the LCC interpreter do not perform the type checking procedure. The false result from typeChk(X, \( \Delta \)) 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. 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...
Table 7.1
Different reaction policy modes in security type checking: prevention, detection and no-detection modes.

<table>
<thead>
<tr>
<th>Reaction policy modes</th>
<th>Priority (Pre, Det, NoCk)</th>
<th>Type checking</th>
<th>typeChk result when a leakage is found</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevMode</td>
<td>1 1 0 0</td>
<td>Yes</td>
<td>False</td>
</tr>
<tr>
<td>detectMode</td>
<td>2 0 1 0</td>
<td>Yes</td>
<td>True</td>
</tr>
<tr>
<td>noChkMode</td>
<td>3 0 0 1</td>
<td>No</td>
<td>True</td>
</tr>
</tbody>
</table>

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).

\[
\text{publicMessage1 => a(publicAgent, P) <- check(Secret) }
\]

or

\[
\text{publicMessage2 => a(publicAgent, P) }
\]

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

\[
M \Rightarrow A \leftarrow C \text{, } \Delta \text{c}(M \Rightarrow A \leftarrow C, \Delta(L))
\]

Let us assume the constraint does not hold; i.e. satisfied (check(Secret)) return false, so the first part of the conditional statement
Becker

\[ M \Rightarrow M_{\text{new}} \]

\( \text{if} \) \( \text{typeChk}(M = A, \Delta) \)

then the type checking is as below:

\[
\Gamma \vdash \text{thisAgent h} \leftarrow \text{agent l} \quad \text{Sub}
\]

\[
\text{publicMessage} := \text{e} \quad \Gamma \vdash \text{Id} \quad \text{publicAgent l} \quad \Gamma \vdash \text{Id} \quad \text{p1} \quad \Gamma \quad \text{output}
\]

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).

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; 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, 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:

\[
\text{SecretMessage} \Rightarrow \text{a(publicAgent, P)} \leftarrow \text{smallerThan(PublicVar, PublicVar)}
\]

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 below, 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.

\[
\text{(null} \leftarrow \text{equals(PublicVar, PublicVar)}
\]

or

\[
\text{SecretMessage} \Rightarrow \text{a(publicAgent, P)}
\]

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:

\[
\text{a(secretAgent, S):}:
\]

\[
\text{null} \leftarrow \text{notEqual(SecretID, 0)} \text{then}
\]

\[
\text{a(secretAgent, S)}
\]

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, likeness, and observational equivalence as prerequisites:
\[ a(R, I) : B \xrightarrow{\text{non-interference}} a(R, I) : E \quad \text{if} \quad B \xrightarrow{\text{non-interference}} E \land \text{typeCheck}(a(R, I) : E, \Delta) \]

\[ A_1 \text{ or } A_2 \xrightarrow{\text{non-interference}} E \quad \text{if} \quad A_1 \xrightarrow{\text{non-interference}} E \]

\[ A_1 \text{ or } A_2 \xrightarrow{\text{non-interference}} E \quad \text{if closed}(A_1) \land A_2 \xrightarrow{\text{non-interference}} E \land \text{typeCheck}(A_1 \text{ or } E, \Delta) \]

\[ A_1 \text{ then } A_2 \xrightarrow{\text{non-interference}} E \quad \text{if} \quad A_1 \xrightarrow{\text{non-interference}} E \]

\[ A_1 \text{ then } A_2 \xrightarrow{\text{non-interference}} A_1 \text{ then } E \quad \text{if} \quad \text{closed}(A_2) \land A_2 \xrightarrow{\text{non-interference}} E \land \text{typeCheck}(A_1 \text{ then } E, \Delta) \]

\[ A_1 \text{ par } A_2 \xrightarrow{\text{non-interference}} E_1 \text{ par } E_2 \quad \text{if} \quad A_1 \xrightarrow{\text{non-interference}} E_1 \land A_2 \xrightarrow{\text{non-interference}} E_2 \land \text{typeCheck}(E_1 \text{ par } E_2, \Delta) \]

\[ C \leftarrow M \leftarrow A \xrightarrow{\text{non-interference}} \langle C \leftarrow M \leftarrow A, \Delta(L) \rangle \quad \text{if} \quad \text{typeCheck}(C \leftarrow M \leftarrow A, \Delta) \]

\[ M \leftarrow A \xrightarrow{\text{non-interference}} C \langle M \leftarrow A, \Delta(L) \rangle \quad \text{if} \quad \text{typeCheck}(M \leftarrow A, \Delta) \]

\[ M \leftarrow A \xrightarrow{\text{non-interference}} C \langle M \leftarrow A, \Delta(L) \rangle \quad \text{if} \quad \text{typeCheck}(M \leftarrow A, \Delta) \]

\[ \text{null} \leftarrow C \xrightarrow{\text{non-interference}} C \langle \text{null} \leftarrow C, \Delta(L) \rangle \quad \text{if} \quad \text{typeCheck}(\text{null} \leftarrow C, \Delta) \]

\[ a(R, I) \leftarrow C \xrightarrow{\text{non-interference}} a(R, I) : B \quad \text{if newClause}(P, a(R, I) : B) \land \text{typeCheck}(a(R, I) \leftarrow C, \Delta) \]

\[ a(R, I) \xrightarrow{\text{non-interference}} a(R, I) : B \quad \text{if newClause}(P, a(R, I) : B) \land \text{typeCheck}(a(R, I) \leftarrow C, \Delta) \]

\[ \text{closed}(c(X, L)) \]

\[ \text{closed}(A \text{ or } B) \leftarrow \text{closed}(A) \land \text{closed}(B) \]

\[ \text{closed}(A \text{ par } B) \leftarrow \text{closed}(A) \land \text{closed}(B) \]

\[ \text{closed}(A \text{ then } B) \leftarrow \text{closed}(A) \land \text{closed}(B) \]

\[ \text{closed}(X \leftarrow B) \leftarrow \text{closed}(B) \]

---

**Definition 7.1 (Visibility).** The set \( \text{visible}_l(\Gamma) \) denotes the LCC terms in the context \( \Gamma \) that can be observed by other agents (or adversaries) with the security level \( l \) or higher:

\[
\text{visible}_l(\Gamma) = \{ T \in \Gamma | | \Gamma(T)| \leq l \}.
\]

**Definition 7.2 (Alikenes).** \( \Gamma_1 \models \Gamma_2 \): Two security contexts \( \Gamma_1 \) and \( \Gamma_2 \) are alike up to the level \( l \) iff: \( \text{visible}_l(\Gamma_1) = \text{visible}_l(\Gamma_2) \).

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: \( \text{visible}_l(\Gamma_1) = \{ m1, m2 \} \) and \( \text{visible}_l(\Gamma_2) = \{ m1, m2 \} \), which means other agents with security level of at least \( l \) can see this information. We also have \( \Gamma_1 \equiv_l \Gamma_2 \).

Recall the LCC clause expansion mechanism of an original LCC clause \( C_l \) into \( C_{n+1} \) in terms of the interaction model \( P \):

\[
C_l \xrightarrow{\text{M}_{l_1},M_{l_2},P,O_{l_1},O_{l_2}} C_{l+1}, \ldots, C_{n-1} \xrightarrow{M_{l_1},M_{l_2},P,O_{l_1},O_{l_2}} C_n, \Delta C_{n+1} \ldots \Delta C_n \]

where security environment \( \Delta = (\Gamma, K, H, L, \Sigma) \) and \( O_n \) is an output message set that can express 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).** \( O_{l_1} \equiv O_{l_2} \): 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 \( O_{l_1} \) and \( O_{l_2} \).

**Observational equivalence of** \( O_{l_1} \) and \( O_{l_2} \) 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 \( \Gamma \) is used instead of the security environment \( \Delta \). This is safe to do, because in our investigation, \( \Gamma \) only changes within \( \Delta \).

**Definition 7.4 (Non-interference).** For all \( \Gamma_1, \Gamma_2, (\Gamma_1 \equiv_l \Gamma_2) \land C_l \xrightarrow{M_{l_1},M_{l_2},P,O_{l_1},O_{l_2}} C_{l+1} \wedge C_{n+1} \ldots \Delta C_n \Rightarrow (O_{l_1} \equiv O_{l_2}) \).

This states that for any two contexts \( \Gamma_1 \) and \( \Gamma_2 \) which are alike up to level of \( l \), a successful expansion of the LCC clause \( C_l \) in one of the contexts with behaviour \( O_{l_1} \) and a successful expansion in the other context with behaviour \( O_{l_1} \) 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 \( C_l \) is a message sending operation \( M \Rightarrow A \). If the type of the agent A is agent \( h \), the typing rule \( \text{Snd} \) 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 \( l \), 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 \( C_l \) 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.

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 (Zdancewic and Myers, 2001). 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:

\[
\text{declassify}(h) \rightarrow 1.
\]

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 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 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.

Appendix A

Tables A.1 to A.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 A.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 A.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.

The sources of implicit information flows are conditional operations. Table A.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 A.3, MaxOperation is the maximum security level of Operation1 and Operation2.

Appendix B

Proof of Theorem 6.1 (Progress). By induction on the structure of \( F \vdash L : \varphi \); 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 \( L : \varphi \), so \( A_1 : \varphi \rightarrow A_2 : \varphi \)

By the induction hypothesis either \( A_1 \rightarrow A'_1 \). Similarly, either \( A_2 \rightarrow A'_2 \). If both \( A_1 \) and \( A_2 \) are final steps (closed), based on the following LCC rewriting rule in Fig. 7.2:

\[
\text{closed (A then B) } \rightarrow \text{closed (A) } \land \text{closed (B)}
\]

If \( A_1 \) then \( A_2 \) is a final step. If \( A_1 \) is a final step and \( A_2 \rightarrow A'_2 \), according to the following rewrite rule:

\[
A_1 \text{ then } A_2 \rightarrow_{\text{R,M,M}_{\varphi,\rho}} A_2 \text{ then } E \quad \text{if } \text{closed (A$_1$) } \land \text{closed (B)}
\]

\[11\] MAP: Multi-Agent Protocol language.
A₁ then A₂ → A'₂. If both A₁ and A₂ are not final steps, which means A₁ → A'₁ and A₂ → A'₂, based on the following LCC rewriting rule:

\[ A₁ \rightarrow A₂ \quad \text{if} \quad A₁ \rightarrow A'₁ \text{ and } A₂ \rightarrow A'₂ \]

then A₂ → A'₂. If A₁ → A'₁ and A₂ is a final step, it is not an acceptable state in LCC, so the result is false: A₁ then A₂ → false.

**Case If** L = M ⇒ A ↔ C and L : op τ, so M ⇒ A : op τ and C : con τ.

By the induction hypothesis either C is a final step or else some C' exists that C ↔ C'. If C is a final step, in the following LCC rewriting rule in Fig. 7.2:

\[ M \Rightarrow A \leftarrow C \quad \text{if} \quad \text{satisfied}(C) \]

either the evaluation of satisfied(C) is true, so M ⇒ A ⊸ C or else returns false, which indicates M ⇒ A ⊸ C ⊸ false. In either case, L ends up in a closed state which means a final step.

If C ↔ C', it means that C is a compound constraint C'' that is equal to ¬C₁ ∨ C₁ ∧ C₂ or C₁ ∨ C₂, so based on one of the following rewrite rules in Fig. 7.2:

\[ \text{satisfied}(¬C₁) \quad \text{if} \quad \text{satisfied}(C₁), \]
\[ \text{satisfied}(C₁ ∨ C₂) \quad \text{if} \quad \text{satisfied}(C₁) \land \text{satisfied}(C₂), \]
\[ \text{satisfied}(C₁ ∧ C₂) \quad \text{if} \quad \text{satisfied}(C₁) \land \text{satisfied}(C₂), \]

then we have L ⇒ M ⇒ A ↔ C'.

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

**Proof of Theorem 6.2.** By induction on the structure of Γ ⊢ L : φ and proceed by case analysis (similar to the proof of Theorem 6.1).

**Case Seq:** L = A₁ then A₂ and L : op τ

We know that L is well-typed, so we have A₁ : op τ and A₂ : op τ.

According to the following rewrite rules:

\[ A₁ \rightarrow A₂ \quad \text{if} \quad A₁ \rightarrow A'₁ \text{ and } A₂ \rightarrow A'₂ \]

the transition L → L' happens either by A₁ then A₂ or when A₁ is a final step (closed(A₁)), by A₁ then E.

If ¬closed(A₁), the A₁ then E can be derived by any of the clause expansion rewrite rules, some of the cases are shown; others are similar:

1. **Subcase A₁ : a(R, I) → C**
   
   By the induction hypothesis, we have a(R, I) → C : op τ, A₁ : A'₁ and A'₁' : op τ. The following rewrite rule, which deals with recursion in LCC, is the only rule that expands A₁:

   \[ a(R, I) \leftarrow C \quad \text{if} \quad \text{clause}(P, a(R, I) : : B) \land \text{satisfied}(C). \]

   So we have A'₁' = a(R, I) : : B. Consequently, A₁ then A₂ → a(R, I) : : B then A₁ and A₂ then A₂ : op τ.

2. **Subcase A₁ : M ⇒ A**
   
   By the induction hypothesis, we have M ⇒ A : op τ, A₁ : A'₁ and A'₁' : op τ. The rewrite rule that handles A₁ is only M ⇒ A : op τ, A₁ : A'₁ and A'₁' : op τ. Consequently, A₁ then A₂ → M ⇒ A then A₂ and A₁ then A₂ : op τ. Other subcases are similar.
Case If 1:

We know that $L \equiv M \Rightarrow A \rightarrow C$ is well-typed; $L : op r, \text{so } M \Rightarrow A : op r$ and $C : con r$, we also have $L \Rightarrow L'$. Based on the LCC rewriting rule (9) in Fig. 7.2 and the definition of transition $\Rightarrow$, the possible expansions of $L$ to $L'$ are:

1. If $C \rightarrow C'$, it means that $C$ is equal to a compound constraint $C''$ that might be $\neg C$, $C \land C$ or $C \lor C$. Then we have $L \Rightarrow M \Rightarrow A \rightarrow C'$. By the induction hypothesis, $C' : con r$, hence, based on the type rule $\Gamma \Rightarrow$, $M \Rightarrow A \rightarrow C' : op r$.

2. If satisfied ($C$) returns true, then it is a final step: $M \Rightarrow A \rightarrow C$ \begin{align*}
\Gamma \vdash c \Rightarrow \top \\
\Gamma \vdash \langle M, M, p, M \rangle \Rightarrow \top
\end{align*}

Based on the type rule $\Gamma \Rightarrow$, the typing of the LCC expression will not be changed: $\Gamma \Rightarrow M \Rightarrow A \rightarrow C : op r$.

3. If satisfied ($C$) returns false which means that $M \Rightarrow A \rightarrow C \begin{align*}
\Gamma \vdash \langle M, M, p, M \rangle \Rightarrow \top
\end{align*}$false, then based on the type rule False we have: $\text{false}: \text{op h}$, i.e., $L': \text{op h}$.

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

References


