Securing Open Multi-agent Systems Governed by Electronic Institutions

Shahriar Bijani

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2013

Abstract

One way to build large-scale autonomous systems is to develop an open multi-agent system using peer-to-peer architectures in which agents are not pre-engineered to work together and in which agents themselves determine the social norms that govern collective behaviour. The social norms and the agent interaction models can be described by Electronic Institutions 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 are expected to have many applications in the near future as large scale distributed systems become more widespread, e.g. in emergency response, electronic commerce and cloud computing. A major practical limitation to such systems is security, because the very openness of such systems opens the doors to adversaries for exploit existing vulnerabilities.

This thesis addresses the security of open multi-agent systems governed by electronic institutions. First, the main forms of attack on open multi-agent systems are introduced and classified in the proposed attack taxonomy. Then, various security techniques from the literature are surveyed and analysed. These techniques are categorised as either prevention or detection approaches. Appropriate countermeasures to each class of attack are also suggested.

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 then suggest two frameworks to detect insecure information flows: *conceptual modeling* of interaction models and *language-based information* flow analysis. A novel security-typed LCC language is proposed to address the latter approach.

Both static (design-time) and dynamic (run-time) security type checking are employed to guarantee no information leakage can occur in annotated LCC interaction models. The proposed security type system is then formally evaluated by proving its properties. A limitation of both *conceptual modeling* and *language-based* frameworks is difficulty of formalising realistic policies using annotations.

Finally, the proposed security-typed LCC is applied to a cloud computing configuration case study, in which virtual machine migration is managed. The secrecy of LCC interaction models for virtual machine management is analysed and information leaks are discussed.

Acknowledgement

I am especially grateful to my supervisor David Robertson for his insight, supervision, time and support throughout my PhD study; even though he has been extremely busy as the Head of the School of Informatics. He has shown me qualities beyond that of academic nature, which I aspire to. I would like to thank David Aspinall, my second supervisor, for sharing his technical virtuosity and his valuable feedback to my work. Thanks to my examiners Michael Rovatsos and Maurizio Marchese for their valuable suggestions, especially to Michael for his detailed comments, which were really helpful. I would also like to thank my wife, for the constant source of unwavering support and her patience throughout the duration of my Ph.D, without her this thesis would not have been possible. Finally, thanks to my parents for supporting me spiritually throughout my life and nurturing me in the right trajectory.

Declarations

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- S. Bijani, D. Robertson, D. Aspinall, “Probing Attacks on Multi-agent Systems using Electronic Institutions”, *AAMAS 2011: Declarative Agent Languages and Technologies Workshop (DALT)*, Taipei, Taiwan, 2-3 May 2011.
- S. Bijani, D. Robertson, "Intrusion Detection in Open Peer-to-Peer Multi-agent Systems", *5th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2011)*, Nancy, France, June 13-17, 2011.
- S. Bijani, D. Robertson, “A Review of Attacks and Security Approaches in Open Multi-agent Systems”, *Artificial Intelligence Review Journal*, pp. 1-30, Springer, May 2012.
- P. Anderson, S. Bijani, A. Vichos, “Multi-Agent Negotiation of Virtual Machine Migration Using the Lightweight Coordination Calculus”, *6th International KES Conference on Agents and Multi-agent Systems, Technologies and Applications, KES AMSTA 2012*, June 2012.
- P. Anderson, S. Bijani, H. Herry, “Multi-agent Virtual Machine Management Using the Lightweight Coordination Calculus”, *Transactions on Computational Collective Intelligence (TCCI)*, Springer, 2013.

(Shahriar Bijani)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Open Multi-agent Systems (MASs) are experiencing growing popularity in the Multi-agent Systems community and are expected to have many applications in the near future, e.g. in emergency response, electronic commerce and cloud computing. An o*pen system* is a system that allows new components, which may have been created by different parties or for different objectives, and are not known at design time, to interact at run-time with existing components (Poslad and Calisti 2000). An open *multi-agent system* (MAS) is an o*pen system* in which agents can join and leave freely (Demazeau and Rocha Costa 1996). We focus on open MASs with dynamic interactions, in which *electronic institutions* (Esteva, et al. 2001) are used to form the interaction environment by defining social norms for group behaviour.

Considering the agents' interaction model, open MASs can be divided into three types, each with different security issues. In the first category it is assumed that agents are completely autonomous and that there is no assumption about protocols except that messages are passed (maximum autonomy and flexibility). In this case, since neither information is available about protocols nor about agents, we cannot ensure the security of interactions beyond assuming that agents individually may manage or just rely on the security of lower layers.

In the second type, there is one fixed protocol (an electronic institution) for agent communication. This is closer to traditional security models than the first type and is more likely to be standardised by means of conventional security mechanisms and reasoning about protocol security properties but is too rigid for many applications of knowledge sharing in MASs, e.g. in emergency response and distributed healthcare protocols. This is because of the necessity of global semantic agreement between agents.

The third class of MASs utilises multiple protocols for different applications so that agents may play roles in various electronic institutions. In the most general case, agents may invent the protocols themselves and share them with others or use other (unknown) agents' protocols; e.g. the *OpenKnowledge* system (Robertson, Giunchiglia, et al. 2008). If the first and second types are assumed as two extremes, the third type will be a solution somewhere in the middle that is quite flexible. We refer to this type as *open MAS governed by electronic institutions* or *open MAS with dynamic interactions*. In this thesis, henceforth, the notion of open MAS is taken to stand for *open MAS governed by electronic institutions*.

This thesis investigates security issues in these open MASs and proposes a security analysis framework for open MASs with dynamic interactions. The reminder of this chapter introduces the problem, lists the contributions of this work and describes the overall structure of the thesis.

1.1 The Problem

Security is a major practical limitation to the advancement of open MAS. Although openness in open MASs makes them attractive for different new applications, new problems emerge, among which security is a key issue. The more these systems are used in the real world, the more the importance of their security requirements will be obvious to users and system designers. Unfortunately there remain many potential gaps in the security of open MASs and little research has been done in this area, leaving systems vulnerable.

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). In open MASs the autonomy, openness and independence of agents from human users brings about new threats, undetectable using traditional security mechanisms that consider the lower network layers. Open MASs are particularly difficult to protect, because we can provide only minimum guarantees about the identity and behaviour of agents.

[1] Open Systems Interconnection

Moreover, many traditional security solutions may not be completely appropriate for open MAS without further adaptations. Conventional security mechanisms such as access control and encryption do not provide end-to-end security for agents' interactions and are at best a small (though necessary) part of the solution.

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.

Cloud Computing is an emerging application of open MAS that is a common target for confidentiality attacks. Clouds raise new privacy and confidentiality concerns as data are usually managed by external parties on remote data centres and various services have access to them. These concerns grows in new cloud computing paradigms such as *hybrid* and *community clouds*, which enlarge their capabilities through engaging different parties through employing non-centralised approaches and peer-to-peer technologies. Information leakage in cloud computing is a vulnerability that may cause serious privacy issues.

The following section provides a brief introduction to the proposed approach.

1.2 The Proposed Approach

We have completed a programme of research that addresses security of open MASs governed by electronic institution by analysing potential attacks against such systems, reviewing possible security countermeasures and proposing a formal framework for deploying end-to-end security policies. The proposed security framework of this thesis is aimed primarily at addressing the information leakage problem by analysing information flows in agents' interactions. Secure information flow is an end-to-end security policy, where not only access to information is restricted, but also the use of information. We have also investigated this issue in a cloud configuration management example used as a case study.

We use the OpenKnowledge architecture (Dupplaw, Kotoulas and Siebes 2007) as a portable platform of electronic institutions, whenever our security analysis depends on the infrastructure. To illustrate agents' interactions, we use the Lightweight Coordination Calculus (LCC), a suitable language to implement social norms. Our security analysis focus is on general aspects of the OpenKnowledge system and the LCC language (as much as possible), so it is inclusive enough to be generalised to other open MASs governed by electronic institutions.

1.3 Contributions

The main contribution of this thesis is proposing a formal framework for analysing information flow in open MASs governed by electronic institutions through proposing the security-typed LCC language. This framework provides end-to-end confidentiality in open MASs according to the defined security policy; i.e. an open MAS prevents information leaks through the agents' interactions by employing security-typed LCC. Confidentiality is guaranteed by proofing security properties of the security-typed LCC language.

Other contributions of this thesis are as follows:

- Categorising potential attacks against open MASs governed by electronic institutions and presenting a three-layer taxonomy of attacks, which helps understand the implications of attacks and the potential defence mechanisms against them.
- Reviewing security countermeasures to the attacks and classifying security solutions that follow detection and prevention approaches.
- Proposing a framework for information leakage analysis using the *conceptual modeling* approach, in which LCC interaction models are converted into another formalisation, and then an existing tool analyses the new formalism to find information leaks.
- Implementing a prototypical LCC interpreter that employs dynamic information flow analysis to protect confidential information at run-time. This demonstrates that the proposed framework is feasible and can be automated.

- Evaluation of the proposed security-typed LCC by proofing *progress* and *preservation* of the security type system. This guarantees *type safety* in the security typed LCC.
- Formal definition of non-interference for the security-typed LCC. Non-interference is a popular information flow property, which enforces all executions of an interaction model look identical to an observer (adversary) and guarantees absence of any information flow between secret and public systems activities in well-typed LCC interaction models.
- Applying the language-based security framework to cloud management scenarios for information flow analysis in a cloud computing case study. Insecure information flows in two scenarios of virtual machine migration management using the LCC interaction models are analysed.

1.4 Structure of the Thesis

This thesis is organised as follows:

Chapter 2 provides the background needed to obtain a better understanding of the discussions and techniques presented in the next chapters. It introduces the OpenKnowledge system and the LCC language.

Chapter 3 explores potential attacks against open MASs and defines a taxonomy of attacks. It also applies a risk assessment technique on these attacks, in order to facilitate prioritising response or preventive measures against them. This chapter is based on the work published in (Bijani and Robertson 2012) and (Bijani, Robertson and Aspinall 2011).

Chapter 4 surveys and analyses various techniques to secure open MASs and suggests appropriate prevention and detection techniques for the attacks introduced in Chapter 3. This chapter is mainly based on the work published in (Bijani and Robertson 2012) .

Chapter 5 addresses the information leakage problem in open MAS and develops two secrecy analysis frameworks for LCC interaction models: *conceptual modeling* and *language-based information flow analysis*. This chapter is partially based on the work published in (Bijani, Robertson and Aspinall 2011)

Chapter 6 develops a novel security type system for the LCC language as a language-based information flow analysis technique. This security-typed LCC avoids the information leaks problem by static or dynamic security checking. Several properties of this security type system are proven and some extensions to the proposed technique are discussed.

Chapter 7 investigates information leakage analysis in an application of open MAS in cloud computing using the security framework proposed in Chapter 6. Some of this chapter's material has been published in the following papers: (Anderson, Bijani and Vichos 2012) and (Anderson, Bijani and Herry 2013).

Chapter 8 finally, concludes this thesis by providing a summary of the contributions and discussing some future directions for research.

1.5 Summary

This chapter has introduced this thesis by presenting the problem, briefly describing the overall approach, listing the contributions and outlining the structure of the thesis. The next chapter provides a short introduction to the OpenKnowledge framework and describes the LCC language used throughout this thesis.

Chapter 2

Background

2.1 Introduction

This chapter provides the background needed to obtain a better understanding of the discussions and techniques presented in the next chapters. It briefly introduces the LCC language and the OpenKnowledge system.

2.2 Electronic Institutions

An electronic institution (Esteva, et al. 2001) is an organisation model for multi-agent systems 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. Lightweight Coordination Calculus, LCC (Robertson 2005), is a declarative language to execute electronic institutions in a peer to peer style. In LCC, electronic institutions are called *interaction models.* While electronic institutions 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 this chapter shows. In this thesis, the terms electronic institutions and interaction models are used interchangeably.

2.3 Lightweight Coordination Calculus (LCC)

In our security analysis, Lightweight Coordination Calculus (LCC) is used to implement interaction models and formulate attacks. LCC is a compact executable specification language used to describe the notion of social norms and is 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 IM 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:

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

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).

The LCC rewrite rules are in the form of $X \xrightarrow{M_i, M_o, S, O} Y$, where Y is the expansion of X, 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 S is the interaction model. As the result of the thirteen rewrite rules in Fig. 2-2, one clause of an interaction model is expanded. The first rule starts unpacking a clause by expanding the body of it (B) and the rules (2) to (13) expand different parts of the clause body. c(X) is the notion of closing an LCC term X and based on the *closed* rules (10) to (19), an LCC term X is decided to be closed.

$$a(R,I) :: B \xrightarrow{R_i,M_i,\,M_o,P,O} a(R,I) :: E \qquad if\ B \xrightarrow{R_i,M_i,\,M_o,P,O} E \tag{1}$$

$$A_1\ or\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} E \qquad if\ \neg closed(A_2)\ \wedge\ A_1 \xrightarrow{R_i,M_i,\,M_o,P,O} E \tag{2}$$

$$A_1\ or\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} E \qquad if\ \neg closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} E \tag{3}$$

$$A_1\ then\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} E\ then\ A_2 \qquad if\ A_1 \xrightarrow{R_i,M_i,\,M_o,P,O} E \tag{4}$$

$$A_1\ then\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} A_1\ then\ E \qquad if\ closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O} E \tag{5}$$

$$A_1\ par\ A_2 \xrightarrow{R_i,M_i,\,M_o,P,O_1 \cup O_2} E_1\ par\ E_2 \qquad if\ A_1 \xrightarrow{R_i,M_i,\,M_n,P,O_1} E_1\ \wedge\ A_2 \xrightarrow{R_i,M_n,\,M_o,P,O_2} E_2 \tag{6}$$

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i,M_i,\,M_i-\{M \Leftarrow A\},P,\emptyset} c(C \leftarrow M \Leftarrow A) \qquad if\ (M \Leftarrow A) \in M_i \wedge satisfy(C) \tag{7}$$

$$M \Leftarrow A \xrightarrow{R_i,M_i,\,M_i-\{M \Leftarrow A\},P,\emptyset} c(M \Leftarrow A) \qquad if\ (M \Leftarrow A) \in M_i \tag{8}$$

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i,M_i,\,M_o,P,\{M \Rightarrow A\}} c(M \Rightarrow A \leftarrow C) \qquad if\ satisfied(C) \tag{9}$$

$$M \Rightarrow A \xrightarrow{R_i,M_i,\,M_o,P,\{M \Rightarrow A\}} c(M \Rightarrow A\ ) \tag{10}$$

$$null \leftarrow C \xrightarrow{R_i,M_i,\,M_o,P,\emptyset,} c(null \leftarrow C) \qquad if\ satisfied(C) \tag{11}$$

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

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

$$closed\big(c(X)\big) \tag{14}$$
$$closed(false) \tag{15}$$
$$closed(A\ or\ B) \leftarrow closed(A)\ \vee\ closed(B) \tag{16}$$
$$closed(A\ par\ B) \leftarrow closed(A)\ \wedge\ closed(B) \tag{17}$$
$$closed(A\ then\ B) \leftarrow closed(A)\ \wedge\ closed(B) \tag{18}$$
$$closed(X :: B) \leftarrow closed(B) \tag{19}$$

$$satisfied(\neg C_1) \leftarrow \neg satisfied(C_1) \tag{20}$$
$$satisfied(C_1 \vee C_2) \leftarrow satisfied(C_1)\ \vee\ satisfied(C_2) \tag{21}$$
$$satisfied(C_1 \wedge C_2) \leftarrow satisfied(C_1)\ \wedge\ satisfied(C_2) \tag{22}$$
$$satisfy(\neg C_1) \leftarrow \neg satisfy(C_1) \tag{23}$$
$$satisfy(C_1 \vee C_2) \leftarrow satisfy(C_1)\ \vee\ satisfy(C_2) \tag{24}$$
$$satisfy(C_1 \wedge C_2) \leftarrow satisfy(C_1)\ \wedge\ satisfy(C_2) \tag{25}$$

Fig. 2-2: The LCC rewrite rules for expansion of one clause P in an interaction model.

In the rewriting rules (Fig. 6-4), $satisfy(C)$ is true if the agent's state of knowledge can be made such that C is satisfied and $satisfied(C)$ is true if C can be solved from the agent's current state of knowledge. Whenever the condition of each rule is not fulfilled, it returns false; e.g. in the rule 8, if no message is received, $M \Leftarrow A \xrightarrow{R_i,M_i,\, M_i,P,\emptyset} false$. Similarly in the rule 9, if the condition $satisfied(C)$ is false, then $M \Rightarrow A \leftarrow C \xrightarrow{R_i,M_i,\, M_o,P,\emptyset} false$.

In (Robertson, Barker, et al. 2009) the behaviour of agents coordinated through LCC definitions is defined in terms of traces produced via application of rewrites to LCC clauses.

Here, S is the state of an interaction, M_i is the initial set of messages, p is a unique identifier for a peer. $i(S, M_i, S_f)$ is true when the sequence of interactions and an initial set of messages M_i change the initial state of the interaction model S and security environment Δ to the state S_f and the security environment Δ'. $S \overset{s}{\supseteq} S_p$ (26) selects a clause S_p from the interaction state S. $S_p \overset{s}{\cup} S$ merges specific clause S_p to S and generates a new interaction state S'.

$$i(S, M_i, S_f) \leftrightarrow (S = S_f) \vee \begin{pmatrix} S \overset{s}{\supseteq} S_p & \wedge \\ S_p \xrightarrow{M_i,\, M_o,S,O} {S'}_p & \wedge \\ S_p \overset{s}{\cup} S = S' & \wedge \\ i(S', M_o, S_f) & \end{pmatrix} \qquad (26)$$

$$S \overset{s}{\supseteq} S_p \leftrightarrow \exists\, R, B. (S_p \in S \quad \wedge \quad S_p = a(R, I) :: B) \qquad (27)$$

Fig. 2-3: definition of a trace through an LCC interaction model *S*.

2.4 The OpenKnowledge System

As part of our security analysis, we have studied security of the OpenKnowledge (OK) system as a portable engineering style of electronic institutions. OpenKnowledge can also be considered as a multi-agent system in which agents can interact without any prior agreements or knowledge of other agents or interactions. OK, which has a strong peer-to- peer capability, is a knowledge sharing method based on sharing knowledge in the context of the interactions, in which the knowledge plays a role. OK interprets transmitting knowledge between peers in a similar way to electronic institutions in multi-agent systems (Robertson, Giunchiglia, et al. 2008). Agents must have the OK *kernel* to be a member of the OK system.

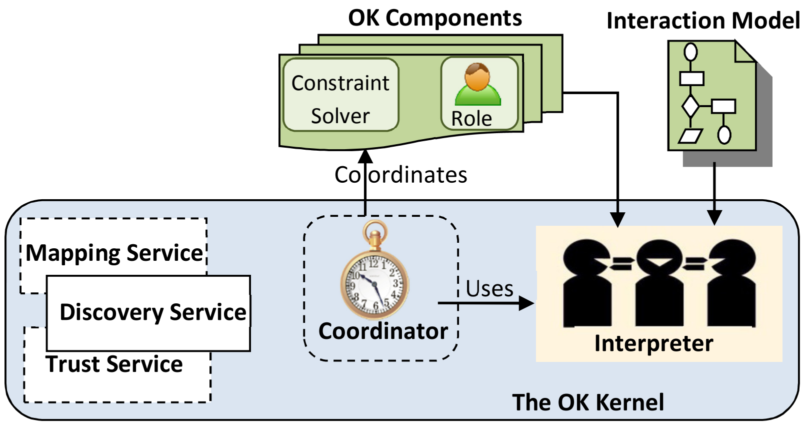


Fig. 2-4: An OpenKnowledge peer architecture

The OK *kernel,* which is shared amongst autonomous peers (agents), is responsible for interpreting and executing the specifications of interactions. It includes an **interpreter**, a **coordination unit** (which is only active in coordinator peers) and some **service modules** (Fig. 2-4). Two basic modules of the OK system are interaction model (IM) and OK component (OKC). IM is a formal specification of roles and interactions between them and OKCs implement roles and contain methods to solve the IM's constrains. The interpreter interprets and executes the IM and when it reaches a constraint, it uses the assigned OKC to fulfil it. A **coordinator** is a randomly selected peer who coordinates all the interacting peers, allocates the roles to the peers and simulates all message passing locally within the current interaction to find and assign the appropriate OKCs. OK also has a main service called ***Discovery and Team Formation*** (***DTS***), and two optional services: *Trust and Reputation* service (a symmetric reputation system which determines trustfulness of the OKCs and peers) and *Mapping Service* (ontology mapping for the IMs and OKCs). The main responsibilities of the *DTS* are the following (Dupplaw, Kotoulas and Siebes 2007): publishes, discovers, retrieves and stores IMs and OKCs, subscribes peer roles when the peers request, selects and initiates the coordinator peer.

OK components provide a way to extend an interaction’s functionality to include more sophisticated tasks. The use of components is integrated in the LCC interaction models in the form of constraints. This makes LCC a powerful and flexible tool to implement many applications and to integrate easily with existing systems.

Chapter 3

Attacks on MAS

3.1 Introduction

This chapter introduces and classifies important attacks on open MASs governed by electronic institutions. In our investigation, the attacker model, which describes the attacker's capabilities, is built on the assumption that the interpreter of the interaction models and the machines that run them are trusted. The attacker model also does not include attacks against the physical machine, where the agent run on. The adversary may perform an attack by creating or publishing a malicious interaction model.

Studying all possible attacks on such systems is too broad an area, so the scope of the topic needs to be limited. In this study, we do not emphasise dealing security issues that are specific to the mobility of mobile agents i.e. attacks from hosts on agents and vice versa. We assume that the interface between the agent platform and the agent is secure to focus on attacks by agents on other agents rather than on the infrastructure. Moreover, we assume that some confidential and important information exists in the open MAS, in order for some attacks to be worthwhile.

In this chapter, we first introduce an attack taxonomy, then discuss each attack, next propose a risk analysis of the attacks and finally summarise this chapter. The LCC language is used to formulate attacks in this study.

3.2 Attack Taxonomy

As a part of our security analysis, we categorised various attacks on open MAS. Attack or vulnerability taxonomies are designed for different purposes (Igure and Williams 2008): 1) to develop automated tools for performing security assessment, 2) to provide a way to explore unknown attacks and 3) to understand the attacks' implications and the defence mechanism against them. The latter is the main goal of our attack classification, which is valuable for the prevention, detection and response to potential attacks.

Attacks on (open) MASs can be categorised in different ways, however we suggest a three-layer taxonomy as illustrated in Fig. 3-1. The first level of our classification is based on violations against the four main security properties i.e. confidentiality, integrity, availability and accountability. In the second level we categorise attacks based on the novelty of the technique used to run an attack, namely, traditional and modern attack techniques. The third level of attack classification is grounded on the attack target, which could be sender agents, recipient agents and transferred information.

Some dimensions to attack classification might be to some extent fuzzy, for example, in the second level, some modern methods could be categorised as traditional. However it does not affect our classification objective. We do not intend to have mutually exclusive attack classes, so there might be some attacks that could be categorised in more than one class. This taxonomy might be tailored for each open MAS with regards to its implementation and application.

The four classes in the first level of taxonomy are: *1) disclosure attack, 2) modification attack, 3) denial of service attack* and *4) fake identity attack.* We describe the main forms of attack introduced in the taxonomy in the following sub-sections.

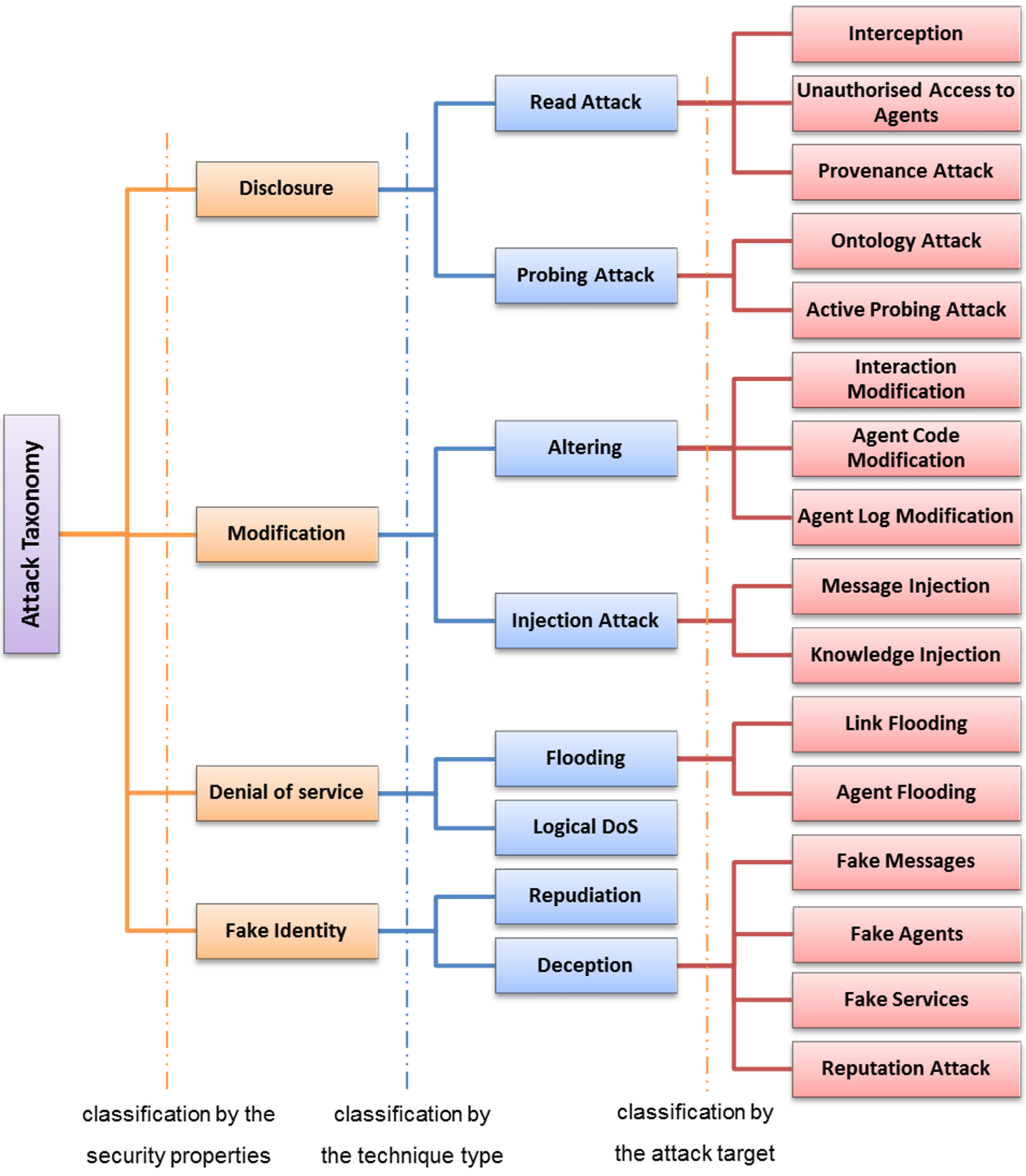


Fig. 3-1: Taxonomy of potential attacks on open MASs. The first level corresponds to confidentiality, integrity, availability and accountability respectively. The second level represent the traditional (the top box) and the modern (the bottom box) techniques in the attacks. The third level is grounded on the attack target: sender agents, recipient agents and transferred information.

3.3 Disclosure

Although open MASs are usually open to all, this does not mean that there is no confidential information in which malevolent people are interested. In the *disclosure* attack, the attacker tries to access confidential information. These attacks are of two types: Firstly, conventional *read attacks* similar to those in the traditional computer security literature. Based on the attacker target we have various read attacks:

(a) *Interception*: unauthorised access to confidential interaction information; e.g. exploiting information leakage in message passing in agents interactions (Fig. 3-2), which is described below.

(b) *Unauthorised access to agent*: the agent's information (including its state and code) and the agent's local knowledge could be disclosed to an adversary. In mobile agent systems there are threats from the host like reverse engineering and theft of mobile agent information (Bierman and Cloete 2002).

(c) *Provenance attack*: attack on provenance information of agents and their communications. An example of attack on provenance information is when agent *A* sends message *M1* to receiver B in an open interaction*,* maybe *A, B* and *M1* were not private to a malicious agent, who is monitoring the interaction, but probably the history of the sent messages from *A* or the agents' names who has interacted with *A* would be confidential information. Disclosure of a mobile agents' itinerary information to an adversary is another example of threat against open MAS.

Fig. 3-2 shows an example of information disclosure in a simple proteomics lab interaction model. This is a modified version of the DNA sequencing interaction model (Abian, et al. 2008) in the OpenKowledge project and it is used for knowledge sharing between researchers and proteomics labs. In this scenario a query is passed on to each laboratory in a list of proteomics labs and the results are sent back to the researcher to be analysed. It could be a case that proteomics labs may not wish to share all information related to DNA sequencing with researchers. An adversary could share the interaction model, in which he/she (in the role of *researcher*) could ask an implicit query from a proteomics lab agent (*omicslab*) to access commercially important information.

```
1.  a(researcher(LabList), R) ::
2.   ( ask(X)=>a(omicslab, H) <-  LabList=[H|T]  then
3.     tell(Y)<= a(omicslab,H) then
4.     null <- processResult(X,Y,H)
5.     then a( researcher(T), R)
6.   ) or
7.   null <- LabList = []

8.  a(omicslab, O) ::
9.    ask(X)<= a(researcher,R)then
10.   tell(Y)=> a(researcher,R)<- Combine(X,Y)
11.   then a(omicslab, O)
```

Fig. 3-2: An example of an *interception attack* using an *implicit query*, in which a malicious researcher could access confidential information of proteomics lab agents. An implicit query `Combine(X,Y)` is asked as a constraint in the `omicslab` clause (line 10), receiving the `tell(Y)` message by the researcher agent informs it that the constraint holds.

An implicit query could be asked by placing a query as a constraint in LCC, rather than sending a message. In other words, an adversary could not only infer information directly from a received message, but also from analysing the constraints in an interaction model. An example of confidential information in proteomics lab could be the combination (binding potential) of two publicly known proteins that activate a particular gene. In Fig. 3-2, the relation between two pieces of public information is confidential; i.e. X and Y are not confidential but a malicious *researcher,* R, could recognise whether proteins X and Y could combine by putting a combine(X,Y) constraint in line 10. When O sends the non-confidential tell(Y) message to R it will implicitly inform R that X and Y could combine together because R knows that combine(X, Y) had to be satisfied before the tell(Y) message could be sent.

Another example of an unauthorised access to confidential information is illustrated in Fig. 3-3. This is a modified fragment of an interaction model in the MIAKT project (Hu, et al.

2009), which aims to support multidisciplinary meetings for the diagnosis and management of breast cancers. The `dataHandler` retrieves a patient's private data based on a request submitted by an authorised domain `specialist` (line 3) and sends it to the `specialist`. The `specialist` then forwards this information to a `nurse`, who is not authorised to access the information, without any permission check (line 5).

```
1. a(dataHandler,H) ::
2.   credentials(SID,Pass,PID) <= a(specialist,E)   then
3.   authorised => a(specialist,E) <- is_authorised(E,SID,Pass,PID) then
4.   request_patient_record(Patient) <= a(specialist,E)   then
5.   inform(Patient,Record) => a(specialist,E) <-
6.                        get_patient_record(Patient,ID, Record) then
7.    ...
8. a(specialist,E) ::
9.    credentials(SID,Pass,PID) => a(dataHandler,H)   then
10.   authorised <= a(dataHandler,H) then
11.   request_patient_record(Patient) => a(dataHandler,H) then
12.   process(Record) <- inform(Patient,Record) <= a(dataHandler,H) then ...
13.   request_patient_record(Patient) <=  a(nurse, N) then
14.   inform(Patient,Record) => a(nurse, N)  ...
```

Fig. 3-3: An interaction model to support specialists meetings for the diagnosis and management of breast cancer. The confidential patient's `Record` **is propagated to an unauthorised nurse agent by a specialist agent (line 14).**

Secondly, *probing attack*s, in which an adversary accesses confidential information by analysing the results of the sent queries. Whether the attacker target is an ontology of a MAS or not, we can classify the probing attack into two types: (a) ontology attack and (b) active probing attack.

Use of ontologies in MASs is currently popular especially in knowledge-intensive applications. In an *ontology attack*, the attacker accesses the whole ontology by sending many queries to a semi-open ontology-based system. In the latter attack, it is assumed that the whole ontology is confidential but it is also open to questions about small parts of itself. The attacker may find the entire information and relations by asking intelligent and complementary queries based on its knowledge of the semantics of the ontology representation language being used by the attacked agent (Fig. 3-4).

```
a(ontologyAttacker(S, Os, Of), A) ::
       ( ask(subclasses(C)) => a(ontologyService, O) <- S = [C|Sr] then
       tell(subclasses(C,Sc)) <= a(ontologyService, B) then
       a(ontologyAttacker(Sn,On,Of),A) <-
                               merge(Sr, Sc, Sn)and merge(Os, Sc, On)
       or
       null <- S = [] and Of = Os


a(ontologyService, B) ::
       ask(subclasses(C)) <= a(ontologyAttacker(_,_,_), A) then
       tell(subclasses(C,Sc)) => a(ontologyAttacker(_,_,_), A) <-
       known_subclasses(C, Sc) then
       a(ontologyService, B)
```

Fig. 3-4: A sample interaction model of an ontology attack: an attacker `A` starts from the ontology root (`S`) and rebuilds the ontology (`Of`) by asking for subclasses of each node from the ontology service `B`. Initially `S = Os` = the ontology root. The merge function merges the two first arguments and into the third argument. Finally, the copied ontology `Os` will save into `Of`.

Another type of probing attack is *active probing* attack, in which an adversary accesses the private local knowledge (e.g. decision rules and policies) of the victim agent. The *active probing* happens by injection of some facts into the agent's knowledge-base, asking queries before and after the injection and analysing results. We described an example attack on an LCC interaction model in (Bijani, Robertson and Aspinall 2011).

3.4 Modification

This class of attack is against the integrity of the system. An integrated system guarantees that information has not been tampered with during handling (Mitchell 2003). A *modification* attack on an open MAS happens when a malevolent agent modifies a piece of information in the system. Based on the novelty of the attack technique, the *modification* attack on open MASs could be classified as an *altering* and *injection* attack.

We classified *altering* attacks based on the attack target, which could be the interaction, the agent itself, or the agent's log files, into: (a) modification of the agents' interactions by altering the transferring information (e.g. messages), (b) altering the agent code (e.g. interaction model), data and configuration and (c) altering the event logging system of a MAS. A malicious agent can exploit general vulnerabilities like *buffer overflow* to perform the attack. In *buffer overflow* attacks, an adversary tries to corrupt the execution stack of the agent code in order to take over the agent controller and to run arbitrary code.

An example of an *interaction modification* attack is similar to the one in peer to peer systems called the *colluded truncation attack* (Yue, et al. 2009) in which two or more malicious agents, who surround a sender agent, collude to modify a sent message. Silei et al. (2008) explained the same idea in an attack from a host to mobile agent's code. *Modification* attacks are usually dependent upon intruding on low level network communication layers.

Another type of *modification* attack is the *injection* attack, which is similar to *active probing attack.* The objective of the adversary in injection attack is twofold: to infuse forged information into MAS and to inject some facts into the system to be able to infer confidential information. *Injection* attack is classified into *message injection* and *knowledge injection*. In the *message injection* attack, an adversary injects fake messages or malicious interaction models into an open MAS to change or control the interaction between agents while in *knowledge injection* untruthful facts are added into an agent's knowledge base to affect the agent decisions (Bijani, Robertson and Aspinall 2011).

There also could be an active probing attack on an open MAS that use ontology merging, e.g. an intelligent adversary can build a well-designed ontology and add (inject) it to a confidential ontology during the ontology merging procedure. As a result of merging, a new ontology emerges that may leak confidential information. This is because the adversary knows some parts of the new ontology, so he/she can access confidential parts of it by sending carefully crafted queries.

3.5 Denial of Service (DoS)

In a *denial of service* attack on MASs the attacker attempts to prevent the system to provide the intended services to its legitimate users. The goal of *DoS* may be wasting other agents' resources, delaying the service, making real users forsake the system or ruining the system's reputation. A malicious agent may attack just one agent or a group of agents. When more than one attacker collaborates in a *DoS attack*, this is called a *distributed denial of service (DDoS).* As in computer networks (but here, with a different meaning) a *DoS* attack can be divided into two types (Traynor, McDaniel and Porta 2008): *flooding* and *logical* attacks.

Flooding DoS happens when too many messages are sent to one or more agents to overwhelm the victim agent or the connection to an agent by consuming the agent or network resources (e.g.: the agent's message buffer or the communication bandwidth).

```
a (attacker(PeerList), A) ::
       null <- PeerList = []
  or (
         m(PeerList) =>  a(zombie,Z) <-PeerList=[H|T]
          then  a(attacker(T), A)

        )

   a(zombie, Z)::
       m(PeerList)<= a(attacker(_), A) then
a(attacker(PeerList),Z)
```

Fig. 3-5: An example of a simple interaction model for *logical DoS*: An attacker sends a message 'm' including the list of agents to all agents. Each agent who receives this message changes its role to an attacker and starts sending messages to others.

While *flooding* is almost a blind attack, a *logical DoS* uses more sophisticated methods to exploit software or system bugs. In open MASs with dynamic interaction models, any user can publicise an interaction model, a malevolent agent may publish an interaction model to make other agents send many messages to each other (Fig. 3-5), do worthless tasks (Fig. 3-6) or remain in infinite loops. Another kind of *logical DoS* is that one hostile agent does not play its role correctly to terminate or interrupt a popular interaction. In (Sit and Morris 2002) a similar attack called *store and retrieve* is introduced in peer to peer systems. *Inconsistent behaviour* (Sit and Morris 2002) of a malicious peer in behaving with other peers also can be categorised as a *logical DoS* attack.

```
a (attacker(PeerList, K), A) ::
       null <- PeerList = []
  or (
        trigger(K) =>  a(zombie, Z) <- PeerList=[H|T]
        then  a(attacker(T,K), A)

        )


   a(zombie, Z)::
     trigger(K) <= a(attacker(_), A) then  a(in-loop(K),Z)

   a(in_loop(K),Z)::

   null<- K = []      or

             a(in_loop(K1),Z) <- K1 =[a|L]
```

Fig. 3-6: Another example of a simple interaction model for *logical DoS*: An attacker sends a trigger message including a number 'K' to all agents. Each agent who receives this message, starts looping for 'K' times ('K' may be a very large number).

These malicious behaviours could be camouflaged as part of a real interaction model in a way that zombie peers cannot be guaranteed to detect them via static analysis of the interaction (even when the specification of this interaction is explicit, as in our examples)

3.6 Fake Identity

Fake identity is a reinterpretation of a *Sybil* attack (Douceur 2002) in open MAS. In peer to peer networks, a *Sybil* attack is an attack in which an attacker subverts the reputation system by generating a huge number of pseudonymous peers to abuse the resources or to affect the system in a way that he/she wants. The *fake identity* attack is divided into two classes: *deception* and *repudiation* attacks.

Different versions of the *deception* attack are as follows:

a) Fake messages: Having a fake identity makes it easy for an attacker to send deceptive information to others. An example of the sort of system exposed to this vulnerability is the OKOmics system (Sierra, et al. 2008), by which proteomic researchers can send queries to different laboratories and compare the results. If an attacker could pretend to be a legitimate proteomics laboratory, then it could deceive researches by returning wrong answers to their queries about sequence identification.

b) Fake agents: An attacker plays many roles in a genuine interaction. e.g.: in an auction, many fake bidder agents may be created by an attacker to demolish the auction.

c) Fake services: An attacker creates an entirely fake interaction with fake agents for deception or fraudulence. e.g.: a completely fake auction interaction with a pseudo auctioneer and bidders to attract real agents and deceive them.

d) Reputation attack: A group of attackers combine to deceive the reputation mechanism; e.g. many malicious agents may collude to increase their own rank and thus deceive a trust service into believing that they are trusted agents.

Another class of fake identity attack is repudiation, which may be a result of an attack on the trust service in an open MAS. When benign agents believe in the reliability of a malicious agent by relying on a deceived trust service, they will probably interact with it. Then the malicious agent may send requests (e.g.: buying) to others (e.g.: resellers), but then repudiate its requests at the final stages (e.g. by logging wrong data to log files) to reach its goal (e.g.: to bring the resellers into disrepute amongst sellers or to defame the whole system to all benign users). An attacker may also deny its pervious actions by repudiation of its identity and changing the authoring information of his/her actions.

It is mainly the responsibility of the trust service of the MAS to prevent the *fake identity* attacks, but most trust services are based on a *symmetric reputation* algorithm, which is proven not to be Sybil-proof (Cheng and Friedman 2005), so cannot guarantee against these attacks.

3.7 Risk Assessment

Attacks bring about risks that have to be studied and modelled to determine and prioritise effective countermeasures against them. To have a better understanding of potential attack risks in an open MAS we present an example of the DREAD risk assessment methodology (Microsoft 2010). In DREAD the threat (attack) is approached and weighed from the different aspects of Damage, Reproducibility, Exploitability, Affected users and Discoverability. The weights indicating the risk level of each category are then averaged to calculate an overall risk assessment of the threat. The categories of risk in DREAD are:

- ***D****amage potential* indicates the damage caused if an attack occurs,
- ***R****eproducibility* is the ease of attack reproduction,
- ***E****xploitability* shows the simplicity of the attack exploitation,
- ***A****ffected users* is a rough estimate of the number of affected agents,
- ***D****iscoverability* points out the ease of discovering the vulnerability exploited in the attack.

It has to be noted that each open MAS has a different set of risks with regard to its implementation and application, so the weighing mechanism has to be customised accordingly to achieve more precise results. To give a sense of this risk assessment technique an example of a generic MAS is presented that utilise the following risk levels: high (3), medium (2), and low (1) (Table 3-1). The values are based on the reported attacks in the surveyed literature and our experience of real world security attacks.

In our risk analysis, each DREAD category is evaluated as low, medium or high with regard to the following explanation. In *Damage potential*, *high* may indicate the risk of an adversary getting full trust authorisation (e.g. become an administrator), *medium* could mean the leak of sensitive information and low risk might be in revealing unimportant information. In *Reproducibility*, *high* risk is where attacks are reproduced easily, *medium* reproducibility is when the attack is confined to specific conditions and *low* reproducibility means the attack is very hard to replicate. The *Exploitability* risk would be rated *high* if an amateur can take advantage of the vulnerability of the system, *low* if it takes a highly skilled and experienced adversary to carry out the attack. The percentage of affected agents in a MAS would denote the risk level of the *Affected users* category. The *Discoverability* of an attack would consider the information available on the vulnerability: a highly discoverable attack has published information about it, while an obscured vulnerability would claim a low risk assessment.

According to the risk analysis in Table 3-1, interception, link flooding, fake services and reputation attacks are the highest risked threats, while message injection has the lowest risks.

Table 3-1: A sample DREAD risk assessment for an open MAS. 3: high risk, 2: medium risk, 1: low risk.

Attack	DREAD risk	Damage Potential	Reproducibility	Exploitability	Affected Users	Discoverability	Risk (Max=3)
Read attack	Interception	3	3	2	2	3	**2.6**
	Access to agents	3	2	2	1	2	**2**
	Provenance	1	2	2	1	2	**1.6**
Probing	Ontology attack	1	2	2	1	2	**1.6**
	Active probing	2	2	1	2	3	**2**
Altering	Interaction modification	3	2	2	2	2	**2.2**
	Agent code modification	3	2	2	2	2	**2.2**
	Agent log modification	1	2	1	1	3	**1.6**
Injection	Message injection	1	2	1	1	1	**1.2**
	Knowledge injection	3	2	1	2	1	**1.8**
Flooding	Link flooding	2	3	3	3	2	**2.6**
	Agent flooding	2	3	2	2	2	**2.2**
DoS	Logical DoS	2	2	2	3	2	**2.2**
	Repudiation	2	2	2	2	2	**2**
Deception	Fake agents	1	3	2	1	2	**1.8**
	Fake services	3	3	2	2	2	**2.4**
	Reputation attack	2	3	3	2	2	**2.4**

3.8 Summary

In this chapter, we have introduced and categorised the main forms of attack on open MASs. A three-layer attack taxonomy has been proposed to help understand the implications of attacks and the defence mechanisms against them. In the first layer of our taxonomy we have classified attacks based on violations against confidentiality, integrity, availability and accountability. In the second and third layer we have categorised attacks based on the novelty of the attack technique and the attack target. In total, we have introduced sixteen attack classes and described a number of them, which were independent of the MAS's platform, in LCC code. A few simple examples have been given to clarify the attack concepts. In addition, we have performed a risk assessment using the DREAD technique on the attacks, as an example to facilitate prioritising response or preventive measures against them. The next chapter discusses potential countermeasure to these attacks.

Chapter 4

A Review of Methods to Secure Open MASs

4.1 Introduction

In this chapter, we survey and analyse various techniques to prevent and detect the attacks introduced in Chapter 3 to secure open MASs. To the best of our knowledge, this is the first review of security solutions for open MASs, which has been published in (Bijani and Robertson 2012). We first categorise security solutions following prevention and detection approaches. Then, we suggest which security technique is an appropriate countermeasure for which classes of attack introduced in Chapter 3.

This chapter presents research relevant to the security of open MASs, without focusing on security issues specific to the mobility of mobile agents i.e. attacks from hosts on agents and vice versa. Security of open MAS has been explored greatly in the literature, although only a few have focused on open MAS itself and most research has dealt with mobile agents security issues either directly or indirectly; so, many of the security solutions have been proposed for threats from agents to hosts or from hosts to agents.

There have been many attempts to protect mobile agents from the host platform in the literature (Bierman and Cloete 2002) (Jansen and Karygiannis 2000) and (Oey, Warnier and Brazier 2010); 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), 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 the security issues coming from the mobility of agents should not diminish the importance of many other security threats in open MASs and we will not concentrate on them any further.

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[1]) are used. Most research on secure MASs follows this approach. (Poslad and Calisti 2000), (Wong and Sycara 1999), (Wang, Varadharajan and Zhang 1999) and (Poslad, Charlton and Calisti 2002) 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[2]. Other prevention methods for secure MASs are: policy driven and secure development methodologies, e.g. (Mouratidis, Giorgini and Weiss 2003). 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).

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., (Endsuleit and Wagner 2004), (Page, Zaslavsky and Indrawan 2005), (Jansen and Karygiannis 2000) and (Bierman and Cloete 2002). 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.

In the following, we describe the contributions of related work to security of open MAS. Some of the systems described, although appropriate to open MAS, were developed with other architectures in mind. Therefore, the shortcomings we identify for them might not be flaws in those other applications.

[1] Public Key Infrastructure

[2] Virtual Private Network

4.2 Prevention Approach

"*Prevention is better than cure*", so the first step against security threats is trying to avoid attacks using prevention methods. Various prevention methods with different attitudes, at different levels of abstraction and for different goals have been proposed and implemented in the literature of MAS security. As one possible way of classification, we have categorised these prevention methods as follows: encryption and certificate driven systems, policy-based methods and secure agent development techniques.

4.2.1 Encryption and Certificates

Most existing security solutions for MASs suggest the use of encryption to fulfil confidentiality, integrity and non-repudiation in these systems. Symmetric encryption[1], public key (or asymmetric) encryption[2], digital signature and certificate management are the popular methods in communication security (Table 4-1). We discuss specific elements of Table 4-1 below.

Wang et al. in (1999) have suggested an *asymmetric encryption* scheme, which is simple and lightweight compared to well-known encryption algorithms such as DES and RSA. The encryption algorithm consists of compressing the message, N-bit grouping, subtracting from the secret key saved in the secret code file and ungrouping. The authors have argued that to guess the secret codes, hackers face a combinational explosion problem and also the compression adds another security layer. The disadvantages of this method, as the authors indicate, are the weakness of the algorithm for short messages and *secret key* management difficulties, which includes producing, transferring and saving large secret files in agents.

[1] In symmetric cryptography, a secret key, shared between both parties, is used for encryption and decryption. Some examples of common symmetric encryption algorithms are AES, DES, Triple DES, RC6 and Blowfish.

[2] In asymmetric cryptography a pair of public key and private key are used and everything that encrypted by a public key can be decrypted by private key and vice versa. To send a secret message, the sender codes it by the public key of receiver and the receiver can decode the message by their private key. Diffie-Hellman, RSA are ElGamal are some examples of well-regarded asymmetric encryption schema.

Table 4-1. Examples of encryption and certificate based methods to securing MASs

References	Description	Agent Platform
Foner **(1996)**	PGP-based solution using symmetric and public key encryption	Yenta
He et al. **(1998)**	Security agents as certificate authorities (CAs) and public key cryptography	KQML
Wang et al. **(1999)**	A lightweight asymmetric encryption scheme	-
Wong & Sycara **(1999)**	Unique agent IDs and SSL to provide agent communication security. Agent Certification Authority (ACA) to certify the binding of agents' ids to their public keys and DCA for deployer keys (for authentication)	RETSINA
Poslad & Calisti **(2000)**	Symmetric and public key encryption and Simple Public Key Infrastructure for authentication	FIPA model
Karnik & Tripathi **(2001)**	El-Gamal public key for encryption (and a DSA public key for digital signatures). Signed certificates using the agent owner's private key	Ajanta
Novak et al. **(2003)**	Symmetric and public key encryption and Security Certification Authority (SCA) for authentication	FIPA model
Borselius and Mitchell **(2003)**	XML encryption service to secure ACL messages using Open PGP	FIPA model
van 't Noordende et al. **(2004)**	SSL encryption for agent communication and an encoded SHA-1 hash of a public RSA key called Self-certifying Identifier *(ScID)*	Mansion
Park et al. **(2006)**	ID-based cryptography and Local PKI	-
Vila et al. **(2007)**	TLS/SSL for agent communication security. IMTPoverSSL certificates to provide confidentiality, data integrity and mutual authentication	JADE-S
van 't Noordende et al. **(2009)**	Public key encryption and Self-certifying Identifiers (ScIDs) for end-to-end authentication	Agent Operating System (AOS)

Broadly speaking, the two important problems of using symmetric cryptographic algorithms in open MASs are the need of a separate *secret key* for each pair (or group) of agents and sharing *secret keys*. The first problem may lead to a scaling hurdle for a large number of agents and the second problem particularly affects symmetric cryptosystems in some open MAS, in which unknown agents appear and re-appear frequently. In other words, in open MASs with large (possibly unbounded) number of agents it seems impractical to allocate a separate encryption key for each pair of agents and manage them. The secret key is the security basis of symmetric encryption methods, because if it is discovered, all messages can be decrypted, so it should be protected securely. A common solution to these problems is combining a *public key cryptography* schema with the symmetric encryption which many, such as Foner (1996), Wong and Sycara (1999), and Borselius and Mitchell (2003) have done.

Wong and Sycara (1999) have proposed a security infrastructure to address the security and trust of the RETSINA framework (Sycara, et al. 2003), a reusable multi-agent infrastructure, and provide solutions for secure communication, integrity of system level services (such as naming and matchmaking services) and accountability. They have used unique agent IDs and the Secure Socket Layer (SSL)[1] protocol, beneath their agent communication layer, to provide agent communication security. Use of SSL encryption is based on the assumption that agents' deployers have public and private keys binding their physical identities and they should be made responsible for the actions of their agents. The authors also suppose that ANSs[2] and Matchmakers are trusted. In a different research with a similar approach, Vila et al. (2007) have introduced various security services for the JADE framework by integrating existing JADE-S security features into their own mechanism, IMTPoverSSL. IMTPoverSSL provides confidentiality, data integrity and mutual authentication using a certificate-based container-to-container structure. In this framework, each container (group of agents) securely stores other containers' certificates and the above security features are provided using the TLS/SSL protocols. In both methods agent communication security relies on the security of TLS and SSL.

Broadly speaking, relying on widely-used existing security mechanisms has two sides. The disadvantage is that various hackers from different communities (network, web, etc.) are able to exploit its vulnerabilities. In the case of TLS/SSL, there is a reported TLS *Renegotiation* attack (Ray 2009), in which an attacker may be able to tamper with messages[3], and even in some

[1] SSL and its successor, TLS (Transport Layer Security), run on layers beneath application protocols such as HTTP and SMTP and above the TCP transport protocol

[2] Agent Name Servers

[3] A plaintext injection attack against SSL and all current versions of TLS

situations might break the encryption, which is a *man-in-the-middle* attack. On the other hand, the positive points are: (1) implementation of security protocols is not the responsibility of MAS developers meaning that implementation security flaws will be avoided; (2) it is a feasible resolution in some open MAS where agent platforms are created by different authorities; (3) these mechanisms are widely used and tested, so they are more reliable; (4) corresponding organisations will eliminate security vulnerabilities e.g. for the TLS *Renegotiation* attack, IETF[1] suggested a solution (E. Rescorla Feb. 2010).

Although using public key cryptography solves the problem of secret key management, authenticity and integrity of the other agents' public keys are still an open question. The question is how an agent X can be confident that the claimed public key of agent Y belongs to Y and the key has not been tampered with or replaced by a malevolent agent. Generally, applying a public key cryptosystem without using an authentication mechanism may amount to man-in-the-middle attacks. Using a certificate-based schema i.e. Public Key Infrastructure (PKI) and Web of Trust in Open-PGP (a defacto public key encryption system) are common solutions to this. Douceur (2002) proves that trusted certification is the only approach to prevent Sybil attack (or *fake identity attacks* in case of MASs) and without a logically centralised authority, there is no solution to Sybil attacks.

The following are some examples of approaches using certificate-based encryption.

- In (Foner 1996), in which the main concern is privacy, security issues of Yenta (a decentralised, p2p matchmaking agent) such as gathering other agents' private data and discovering users' profiles from their agents have been addressed. The author has implemented a security system to protect the integrity of MASs and has suggested a PGP-based solution using symmetric and asymmetric encryption for the agents' communications.
- He et al. (1998), have nominated autonomous agents, called security agents, as certificate authorities (CA) in PKI (instead of static hierarchy) to scheme an authentication foundation for MAS security, design scalable authentication systems and make certificate management customisation possible. They have suggested adding new

[1] Internet Engineering Task Force

speech acts to KQML to publish various certificates including apply-certificate, issue certificate, renew-certificate, update-certificate and revoke-certificate.

- Wong and Sycara (1999) have proposed a PKI-based solution for the RETSINA model of MAS. There are two certification authorities in their security infrastructure: *Agent Certification Authority* (ACA) for agent keys as a part of their security infrastructure and *Deployer Key Certificates* (DCA), a certification authority for deployer keys which lies outside their infrastructure.
- Borselius and Mitchell (2003) have developed an approach for secure agent communication based on using Open PGP to encrypt and sign ACL[1] messages. They have also recommended using the XML encryption service to secure ACL messages.
- Novak et al. (2003) proposed the X-Security package to secure the agent communication layer of FIPA based agent systems. X-Security supplies a secure model for inter-platform communication and agent activities even in the case of inaccessibility of a CA. A Security Certification Authority (SCA) is introduced as an independent agent who can renew, suspend and withdraw agents' digital certificates.
- Park et al. (2006) have proposed an algorithm to enhance the security of MASs in distributed computing environments through using *ID-based Cryptography* (ID-C) to solve the scalability problem of PKI in such a way that no infrastructure is necessary to authenticate public keys and manage directories to store certificates. They have suggested an ID-based threshold decryption method without *key escrow*[2] providing a key recovery scheme and key update strategy for dynamic group membership.

Unfortunately, using public key cryptography infrastructure, like PKI and Open PGP, does not completely solve the underlying problem of public key authenticity in open systems. In other words, correctly identifying the public key of the user (agent owner) and ensuring that this key belongs to the real one is still a problem. In an open system, in which there is no previously established contact to legally commit the agent owners, repudiation of the signature and keys is possible. A malicious user may also pretend to be a famous organisation so as to gain the

[1]Agent Communication Language

[2] In the *key escrow* schema, private keys are stored in an escrow for key recovery under certain circumstances by an authorized third party.

confidence of users in accepting its certificate. Furthermore, some general security challenges of using PKI are mentioned in (C. E. Schneier 2000). Additionally, in some applications of open MASs (for example in emergency response or crisis management), applying certificates, which requires pre-negotiation or physical contact between agent owners and the certificate authorities, may not be very helpful. Furthermore, almost all the proposed solutions using encryption and certificate-based methods have focused on securing agent communications in low level transactions, securing agent messages only at platform level, while the issue of security in MASs extends to higher levels of system architecture.

A fundamental limitation of encryption-base techniques is that they may guarantee the origin, confidentiality and integrity of information, but not its behaviour. They just prevent information from being released, not from being propagated.

Generally, cryptography and certificate-based solutions are suggested to prevent *read* and *altering attacks,* although openness in MASs might cause some difficulties in practice. Encryption of sensitive data and authentication are the first and most effective steps toward countering *interception*, *unauthorised access to agents*, *provenance attacks* and *agent log modifications*. To avoid *interaction modification* attacks, besides encryption, SSL, digital signatures (which are not content-aware) and integrity checking (e.g. based on SHA-1 and MD5) are also necessary. There is still more need to impede *interaction modification* attack by avoiding a single point of responsibility; not relying on just one (even trusted) agent information, which is the responsibility of the trust service in a MAS. To prevent agent modification attacks, employing *encrypted function* and other code security mechanisms are recommended. As an additional level of protection, proof-carrying code (G. C. Necula 2002) is a promising technique to detect any modification in agent code.

4.2.2 Policy-based Methods

Policy-driven mechanisms have been applied widely in a variety of security applications. Security policies, as a prevention approach to secure MAS, can be used for access control, definition of acceptable behaviour or confidentiality in adversary environments. Some examples of these methods are shown in Table 4-2.

Access control using security policy is quite common in computer systems and MASs are no exception. Wagner (1997) has shown how the database concept of multi-level security can be applied to inter-agent communication in order to protect confidential information. A knowledge system of *MSL*[1] databases and some basic inter-agent communication rules have been defined. The communication rules implement security classifications and the *MSL* database assigns a security classification (e.g.: unclassified, confidential, secret and top secret) to all information items and allocates an authorisation policy to all users. The whole system then enables agents to comply with a defined security policy.

In (Quillinan, et al. 2008), it is argued that well defined and easily configurable security policies better address the security of MAS. Several agent middleware systems provide access control architectures such as *JADE-S, SeMoA* and *AgentScape. JADE-S* supplies a style of decentralised access control that is not flexible enough to define default security policies. *SeMoA* has a centralised access control and does not allow users to define their own policies. *AgentScape* provides a hybrid access control method, a combination of centralised and decentralised policy enforcement. The architecture used in the *AgentScape* middleware uses Role Based Access Control[2] (RBAC) and while it has a set of default security policies, it also allows users to customise them (Quillinan, et al. 2008). Vitabile et al. (2008) extended the JADE-S framework with strong user authentication, a reputation-based trust system and an access control mechanism based on policy files. In the proposed framework, FPGA biometric sensors provide secure and fast authentication of the agent owner in a MAS.

[1]*Multi-level secure*

[2]*Role-based Access Control* or *role-based security* (Ferraiolo, Kuhn and Chandramouli 2007) is a predominant access control mechanism, in which all access is through roles (collections of permissions).

Table 4-2. Examples of policy-based methods

References	Type	Description
(Wagner 1997)	Access Control	Use of multi-level security concept
(Quillinan, et al. 2008)	Access Control	Role Based Access Control
(Tekbacak, Tuglular and Dikenelli 2009)	Access Control	Decentralised ontology and XACML
(Tan, Poslad and Xi 2004)	Definition of acceptable behaviour	Profile-based reasoning model (for dynamic security reconfiguration)
(Vazquez-Salceda, et al. 2003).	Definition of acceptable behaviour	Electronic institutions as police norms
(Paruchuri, Tambe, et al. 2006)	Policy randomisation	Linear and non-linear programming algorithms to randomise single-agent policies to avoid an agent's action being easily predictable
(Paruchuri, Pearce, et al. 2009)	Policy randomisation	A non-linear program with non-convex constraints ensuring the communication constraints are met
(Tekbacak, Tuglular and Dikenelli 2011)	Role-based Policy	A policy model for agents when ontologies in the environment can change

Although access control is necessary in many systems, defining a suitable security policy, which does not cramp users and is not very open is non-trivial, especially when there are heterogeneous agent systems with different ontologies. Malicious agents may also exploit vulnerabilities in access governance systems by masquerading. A common issue in many access control mechanisms for MASs is that they are usually applied at the level of agent middleware and concern access to low-level objects (e.g. files and IP address). This could be considered a weakness, because they might not be able to detect high-level access violations (such as probing attacks and fake identity attacks).

In (Tekbacak, Tuglular and Dikenelli 2009), a decentralised ontology and XACML[1]-based access control architecture for MASs has been proposed. The agent domain ontology and access control parameters in the agent security ontology have been combined within a common XACML policy document that is used through different MAS applications through translation of XACML and OWL to description logic (DL) concepts. This formalisation using DL concepts allows defining and effectively implementing an array of policy analysis services and helps the verification of policies under a common point. One limitation of this approach is that it is not completely compatible with open MASs without further upgrades in their implementation and this is mainly because of the centralised architecture of this model.

The second type of policy driven methods is more general than just controlling access to information and defines acceptable behaviour of the system. An example is (Tan, Poslad and Xi 2004), in which Tan et al. have proposed a policy-based infrastructure for dynamic security reconfiguration in open and heterogeneous systems to address the end-to-end security interoperability problem. They have developed a profile-based reasoning model for a dynamic security reconfiguration system, which detects and analyses policy conflicts and the need for security reconfiguration and then resolve them at a meta-level without the changing underlying systems' implementation. A similar approach may be used to detect attacks at run-time and automatically reconfigure the system for a suitable response.

Definition of acceptable behaviour for an agent by means of electronic institution is another way of attack prevention in MAS. An electronic institution provides a set of rules that define what agents are permitted and forbidden to do and the consequences of their actions (M. Esteva, et al. 2004). An electronic institution, as a collection of norms implemented by security policies, can be devised as a framework to define police norms that guide, control and regulate behaviour of other agents in an open MAS (Vazquez-Salceda, et al. 2003).

The third approach in policy-based methods utilises policy randomisation to prevent adversaries guessing the agents' next actions (Paruchuri, Tambe, et al. 2006) and (Paruchuri, Pearce, et al. 2009). The worst-case assumptions here are; first, agents act in an adversarial environment which cannot be modelled (because there may be unseen adversaries whose actions and capabilities are unknown); second, the adversary can observe the agent's state; third, the adversary knows the agent policy; forth, there is no or limited communication among agents.

[1]XACML is a declarative access control policy language implemented in XML.

The authors have introduced a randomised policy making to avoid an agent's action being easily predictable, using a decision-theoretic model based on the Multi-agent Constrained Markov Decision Problem (MCMDP).

Paruchuri et al. (2006) have provided three linear and non-linear programming algorithms, to randomise single-agent policies. They also have implemented a new algorithm, Rolling Down Randomisation (RDR), which efficiently generates randomised policies via the single-agent linear programming method. With similar assumptions in their prior work, the authors in (Paruchuri, Pearce, et al. 2009), have developed a non-linear program with non-convex constraints that randomises agent team policies while ensuring that the communication constraints of the team are met and the miscoordination arising due to randomised policies is countered.

Although preventing adversaries from guessing an agent's next actions can be useful in some applications, in open MAS with shared interaction models, we might not always be interested in randomised behaviour with all peers. In scenarios where we can group the counterpart agents into trusted and untrusted, the policy randomisation will be a countermeasure against untrusted agents.

Loulou et al. (2007) have proposed a formal approach to prevent attacks on MASs by verifying security policies. They proved theorems describing in which circumstances security policies are successful and could overcome a given kind of attack on mobile agents. They used Z-Eves tools for type checking and theorem proving. In a different study on credential-based authorisation policies, Becker (2010) has introduced a formal framework to analyse secrecy of policy languages. Becker redefined two information flow properties (non-interference and detectability) in credential systems and proposed an inference system that informs us what an adversary can detect from our system. Although his work is not directly related to MASs, his formal approach can be used to protect MASs against probing attacks (Bijani, Robertson and Aspinall 2011).

The security policy approach, especially using low-level and high-level access control methods, can help the prevention of read attacks (*interception, unauthorised access to agents, provenance attack*). Defining acceptable behaviour (e.g. limit the number of queries from one agent) can assist counter *denial of service* and *ontology attacks*.

Generally, access control mechanisms on their own are insufficient to ensure the secrecy of computational systems, because they cannot prevent the propagation of confidential information after it has been released.

4.2.3 Secure Agent Development

Considering security concerns during the development and implementation of MASs is another method to prevent security threats. To fulfil this goal, we consider two approaches: agent-oriented software engineering and language-based security for agents.

4.2.3.1 Agent-oriented Software Engineering

Adding security to agent-oriented software engineering (AOSE) has become an important area within the agent research community (Mouratidis, Giorgini and Manson 2003). (Liu, Yu and Mylopoulos 2002) and (Yu and Cysneiros 2002) are examples of addressing security issues within the requirements engineering process and (Mouratidis and Giorgini 2009), (Xiao 2009) and (Rojas and Mahdy 2011) are examples of integrating security throughout the agent development process (Table 4-3). We can also benefit from UML extensions as generic tools for secure software development e.g. UMLsec (Jurjens 2002) that incorporate security requirement analysis in software design.

Many efforts have been made to provide appropriate methods for integrating functional requirements and non-functional security requirements during the whole software development stages using *Tropos*, an agent-oriented software development methodology (Bresciani, Perini, et al. 2004); e.g. (Mouratidis, Giorgini and Manson 2003), (Mouratidis 2007), (Mouratidis and Giorgini 2009), (Massacci, Mylopoulos and Zannone 2010). Mouratidis et al. (2003) have extended *Tropos* by adding security concepts into the design methodology, with concepts such as Security Diagrams, Security Constraints, Secure Entities and Secure Capabilities, that enable MAS developers to describe security requirements. They illustrate their new methodology using a case study from the health care sector. In (Bresciani, Giorgini, et al. 2004) the degree of criticality and complexity of the parts of the agent system has been analysed to identify possible security bottlenecks of the system. This analysis facilitates decision making of agent system developers, about probable trade-offs between functional requirements and security. The authors also have suggested an algorithm to reduce the complexity or criticality of the security requirement analysis process. Other efforts to suggest a methodology for integrating functional and security requirements in *Tropos* are (Mouratidis, Giorgini and Weiss 2003), (Mouratidis and Giorgini 2009) and (Rojas and Mahdy 2011). Massacci et al. (2010) have proposed the use of *Secure Tropos* methodology with SI*, a modelling language to deal with security and trust. Their goal was to derive privacy policies from requirements to bridge the gap between policy specification and requirements analysis.

Xiao (Xiao 2009) has merged the concept of agent role in AOSE and in Role Based Access Control to produce a model-driven architecture for building adaptive and secure MAS. The idea is that interaction models, containing the agent's role, obligations and security policy rules, which define constraints derived from agent social roles, identify rights of agent. Therefore, functional requirements and security constraints are connected by the notion of role. This method aims to deal with a complete MAS development process from requirements analysis and the early design phase to the implementation phase. The proposed methodology is compatible with open multi-agent systems.

Xiao et al (2008) have proposed multilevel secure LCC interaction models for health care multi-agent systems. A security architecture for the *HealthAgents* system and a security policy set using LCC have been suggested in (Xiao, Lewis and Dasmahapatra 2008). Hu et al. (2009) have also developed a system to support data integration and decision making in the breast cancer domain using LCC and briefly addressed some security issues. They have all used constraints and message passing in LCC interaction models to implement security solutions for access control and secure data transfer, but they have not addressed confidentiality issues such as information leakage.

Table 4-3. Examples of adding security to agent-oriented software engineering

References	Phase	Description
(Liu, Yu and Mylopoulos 2002)	Requirements engineering	Modelling of relationships among strategic actors in order to extract, identify and analyse security requirements
(Yu and Cysneiros 2002)	Requirements engineering	A framework to model the way agents interact to achieve privacy

(Mouratidis, Giorgini and Weiss 2003)	Throughout the development process	Adds security concepts into the design methodology in *Tropos*
(Bresciani, Giorgini, et al. 2004)	Requirements engineering	An algorithm to reduce the complexity or criticality of security requirement analysis
(Mouratidis and Giorgini 2009)	Throughout the development process	A methodology for integrating functional and security requirements in *Tropos*
(Xiao, An adaptive security model using agent-oriented MDA 2009)	Throughout the development process	A model-driven architecture by merging the concept of agent role in AOSE and Role Based Access Control
(Massacci, Mylopoulos and Zannone 2010)	Requirements engineering	An attempt to bridge the gap between policy specification and requirements analysis by deriving privacy policies from requirements
(Rojas and Mahdy 2011)	Throughout the development process	An extension to *Tropos* methodology by integrating threat modelling in software application development

Attack modelling, during the design phase, allows software engineers to clearly understand and analyse design flaws and mitigate security vulnerabilities. As a result, various security attack modelling techniques and tools have been introduced in the computer security literature. We could name attack trees (B. Schneier 1999), attack graphs (Lippmann and Ingols 2005), statecharts (El Ariss and Xu 2011) and Petri Net-based attack nets (McDermott 2000) as a few of these techniques. These techniques can also be customised and applied to agent-oriented software engineering to prevent known attacks on open MASs. For example in (Rojas and Mahdy 2011), the Tropos methodology is extended by integrating threat modelling in software application development.

4.2.3.2 Language-based Security

Access control is insufficient for the secrecy of computational systems because it does not prevent the propagation of confidential information and is not suitable for encrypted information. Encryption and digital signature also cannot prevent attacks that exploit information flow (e.g. *injection* attack). Information flow analysis is a complement to those approaches to ensure some security properties hold.

Using information flow theory for software security analysis is an old idea e.g. by tagging confidential data and analysing tagged data propagation (D. E. Denning 1976). Sabelfeld and Myers (2003) argue, in their survey of language-based information flow security, that sound type systems could be a promising language-based technique to specify and enforce an information flow policy.

Two lines of work on *static program certification* initiated by Denning (1976) and *security specification* by Cohen (1977) are merged and pursued by Volpano and Smith (1997) proposing a security type system that enforces security specifications. In recent years, there has been a great deal of research into secure information flow analysis. Type-based secrecy has been studied in the context of imperative programming languages e.g. (Heintze and Riecke 1998), functional programming e.g. (Austin, Flanagan and Abadi 2012) and process calculi e.g. (Honda, Vasconcelos and Yoshida 2000) and (Focardi, Rossi and Sabelfeld 2005).

There are two approaches to analysing information flow security: static and dynamic (run-time) analysis. Static techniques prove program correctness with reasonable computation cost, conservatively detect implicit and explicit information flows and provide stronger security assurance than the dynamic techniques (Sabelfeld and Myers 2003).

Information flow techniques can also be applied to MAS. Halpern and O'Neil (2008) provided a general framework for analysing the secrecy of MASs that can handle probability and non-determinism in synchronous or asynchronous systems. In their approach, a logic that includes modal operators for reasoning about knowledge and probability is used to syntactically characterise secrecy. However their approach is very abstract and theoretical and needs adaptation and redefinition of security properties for any MAS.

To avoid a*ctive probing* and *injection* attacks in an open MAS, we advocate the implementation of static and dynamic language-based security systems that use security types to prevent information leakage. For this purpose, the agent needs to be implemented by a language that supports information flow analysis. Not much research has been done on applying language-based security techniques to agent development languages. We will discuss this approach in Chapters 5 and Chapter 6 in detail.

4.3 Detection Approach

Although prevention methods usually take precedence over detection methods, it is not always possible or feasible to impede all kind of attacks. In practice, there is a continual battle between defenders and attackers struggling to break the security defences. Hence, the design and implementation of powerful detection and response mechanisms against possible attacks as a second stage of defence are essential. In the detection approach, the sooner an attack is detected, the less the impact of the attack on the victim agents would be. In the case of open MAS, in which there is a minimum guarantee about the agent's identity and behaviour, detection seems to be at least as important as prevention.

4.3.1 Monitoring

Monitoring is a general method of attack detection in hostile environments. The goal of monitoring-based techniques in an open MAS is to detect the misbehaviour of agents or to find anomalies in the system. For misbehaviour detection, we need to define the specifications of agent communication protocol in order to detect interactions that exploit it. As agent communications do not follow a common consented standard, misbehaviour should be defined separately for each type of MAS.

In the anomaly detection approach, patterns in data that do not conform to the expected behaviour are detected (Chandola, Banerjee and Kumar 2009). Classification, clustering or statistical methods are a few examples of anomaly detection techniques that can be applied in open MASs for attack detection. Anomalies are detected when the current MAS state differs from the trained model (classifier). The openness in open MASs might hinder successful anomaly detection. Openness in the sense that new agents can freely join the system is not a problem, but when the open MAS allows new component to be added to the system at run-time, e.g. the OpenKnowledge system (Robertson, Giunchiglia, et al. 2008), this leads to a dynamic behaviour that resists anomaly detection. In these more dynamic systems, anomaly detection techniques generate many false positives rendering the techniques ineffective.

We will now discuss *peer monitoring*, *information monitoring* and *policy monitoring* as three misbehaviour detection monitoring techniques and *activity monitoring* as an anomaly detection technique.

4.3.1.1 Peer Monitoring

Page et al. (2005) have extended their previous research on the *Buddy* model to a more general security mechanism for mobile agent systems. This model adds a separate security layer to the agents' business functionality. In the proposed centralised system, agents who monitor others are called *Buddy* and the others who are monitored are called *protected agents* (PA). A PA might also be Buddy of another PA. There are three basic rules ensuring the uniformity of the model: first, each PA has two *Buddy* agents; second, PA and *Buddy* agents must never be the same; and three, an agent can be a *Buddy* of only one *protected* agent. The home base host manage security of the whole MAS and in case of any malicious activity; it receives alerts from the monitoring system.

While the *Buddy* model has been designed for mobile agents, with minor changes it can be applied to non-mobile agent systems as well. The centralised nature of this model may be a limitation for scalability of open multi-agent systems, although use of local monitoring subsystems may overcome this shortcoming.

4.3.1.2 Information Monitoring

Monitoring published information in agent communications and keeping track of transferred messages facilitate the detection of some information-centred attacks: *provenance, active probing and ontology attacks*. For example, in the case of the *ontology attack*, monitoring users' queries to the ontology and defining some thresholds for the number and the scope of the queries may impede the attack. The information monitoring technique helps us to at least find what other agents (adversaries) have already found out about the agents and be more cautious in future communication with the suspected agents.

4.3.1.3 Policy Monitoring

Implementing agents on top of web services has been introduced as a potential market for agent systems and as approach to make them more popular (Petrie and Bussler 2003) and (Xiao 2009), so security solutions from the Web Service community may be helpful in open MAS. Clark et al. (2010) have introduced a framework for secure monitoring of a specific Service Level Agreement (SLA) and have implemented it in *AgentScape* system. The authors have modified WS-Agreement, an SLA specification for establishing agreement between parties in Web Services, for effective monitoring. They have designed a model for secure and reliable violation monitoring of SLAs and a method for specifying violation policies in a hostile environment. A centralised and a decentralised monitoring of contraventions were tested and the results showed slightly better performance in the decentralised version. As an intrusion can be considered a kind of violation, this system might also be applied to detect attacks, at least for those attacks for which we could specify their features.

4.3.1.4 Activity Monitoring

Activity monitoring is similar to the concept of *activity profiling* (Carl, et al. 2006) to detect the *denial of service* attacks in computer network literature. This is based on the calculation of the average traffic rate for the whole interaction between two agents. Each agent can measure the average traffic rate and whenever any counterpart agent behaves very differently, it can react by raising an alarm, decreasing the rank of that agent or filtering possible malicious agent. In a MAS with centralised management approach, the total MAS communication activities and the average rate of all inbound and outbound flows can be measured and be used to detect suspicious communications. To avoid the high-dimensionality problem[1], a MAS can be clustered into different classes, each of which has a monitoring agent.

To detect DoS and DDoS attacks, especially flooding attacks, *change-point detection* methods and *wavelet analysis* of agent traffic are recommended (Carl, et al. 2006). We can also employ existing anomaly detection techniques; i.e. Classification Based, Clustering Based,

[1] If each communication activity is considered as an attribute (a dimension), analysing all MAS communication flows will be a high-dimensional problem that often leads to large-scale computations.

Nearest-Neighbour-Based, Statistical, Information-Theoretic and Spectral techniques (Chandola, Banerjee and Kumar 2009).

4.3.2 Attack Modelling

Modelling attacks or attackers is a useful technique to detect some attacks on MASs. In the security literature one can find several formal/informal attack(er) modelling strategies for different purposes. Security modelling, in general, is an approach to analyse various aspects of security (e.g. confidentiality and integrity) in a system. While security modelling is more likely to be categorised as a prevention method (such as a number of attack modelling techniques in section 4.2.3.1), online modelling can also be considered as a detection appraoch.

In the following sub-sections, two attack modelling approaches to attack detection in open MASs are introduced: coordination graphs and statistical modelling.

4.3.2.1 Coordination Graphs

Given that some attacks on agent systems are taking advantage of social behaviour between attackers, Braynov and Jadliwala (2004) have studied detection of coordinated attacks on MAS. A group of attacker agents may collude to stage a large attack on the victim agent system by dividing the tasks into separate subtasks among themselves. Conventional intrusion detection systems just detect the last section of the attack chain, not the attacker assistants who prepare attack prerequisites. Assuming that there is a security mechanism to detect a single malicious action, the authors have defined formal metrics on the coordination graph to discover main and peripheral attacker agents. They have introduced a formal model of distributed monitoring and a formal method and an algorithm to detect maximal malicious group of attackers using a coordination graph (nodes are states and arcs are attacks) of all users.

The proposed methods can be applied to detect attacks at an early stage and is capable of being used online (for attack detection) or offline (for forensic analysis). The suggested method is a useful defence against many coordinated attacks, but the assumption that every single malicious action can be detected may not be valid in some attacks, such as an *ontology attack*, so it may not be appropriate for them.

Dove (2009) has proposed a platform for unmanned autonomous system testing (UAST) based on a socially aware team of autonomous agents. His platform can be merged with the attack detection method of Braynov and Jadliwala (2004) to employ socially aware security agents as a community watch in an open MAS. Both methods can be used to detect coordinated attacks such as *distributed denial of service*.

Khan et al. (2009) have taken a different approach and have suggested a technique that models malicious hosts behaviour against mobile agents. They have also proposed the Mobile Agent Graph Head Sealing (MAGHS) method to grantee the integrity of mobile agent system. MAGHS uses symmetric encryption algorithms with a shared secret key between every host and the home platform. It is assumed that a dynamic data structure like a graph can represent the resultant of mobile agents' execution on a host. The attacker behaviour is modelled as a time function, in which long time means unsuccessful attack, and in the case of any possible modification in the agent state, it can be detected by the agent owner. MAGHS is a solution to truncation and repudiation attacks in mobile agent systems.

4.3.2.2 Statistical Modelling

Statistical modelling is another approach, in which anomaly of an open MAS can be detected. In statistical anomaly detection techniques, it is assumed that anomalies occur in the low probability regions of the stochastic model (Chandola, Banerjee and Kumar 2009). There are two types of statistical anomaly modelling: (1) parametric techniques such as the Gaussian model (Aggarwal and Yu 2008) or regression model methods (Kadota, et al. 2003) and (2) non-parametric techniques such as histogram-based methods e.g. (Dasgupta and Majumdar 2002).

In both modelling techniques, a statistical model of the training data is generated and a statistical test is applied to the new data set. Anomalies are the resultant instances with low probability. Parametric techniques assume an underlying distribution in the statistical population, while non-parametric techniques do not assume any distribution and the number and nature of the parameters are not fixed in advance. These anomaly detection methods can be used to detect some *probing attacks*, *ontology attacks* and *denial of service attacks* on open MAS. A disadvantage of statistical techniques is the assumption that the data is generated from a particular distribution and it often does not hold for high-dimensional real data sets (Chandola, Banerjee and Kumar 2009).

4.4 Conclusions

In this chapter, we have reviewed and categorised security solutions intended to provide security in MASs and have suggested some solutions from network security literature as countermeasures to attacks against open MASs. The presented security solutions can be divided into prevention and detection approaches; *Encryption and certification, policy-based methods* and *secure agent development* are three prevention mechanisms, and *monitoring* and *modelling* are two detection approaches. Some of the presented security techniques did not aim at securing open MASs or even MASs, but they were found to be applicable to open MASs.

To illustrate links between various techniques a visual summary of security approaches in the literature proposed to secure MASs is provided as a chronological tree in Fig.4-1. This might not be a comprehensive phylogenetic tree but it shows key relations amongst different approaches from 1995 to 2012. This does not include approaches to security issues that are specific to the mobility of mobile agents; i.e. threats from mobile agent to platforms and from the platforms to agents.

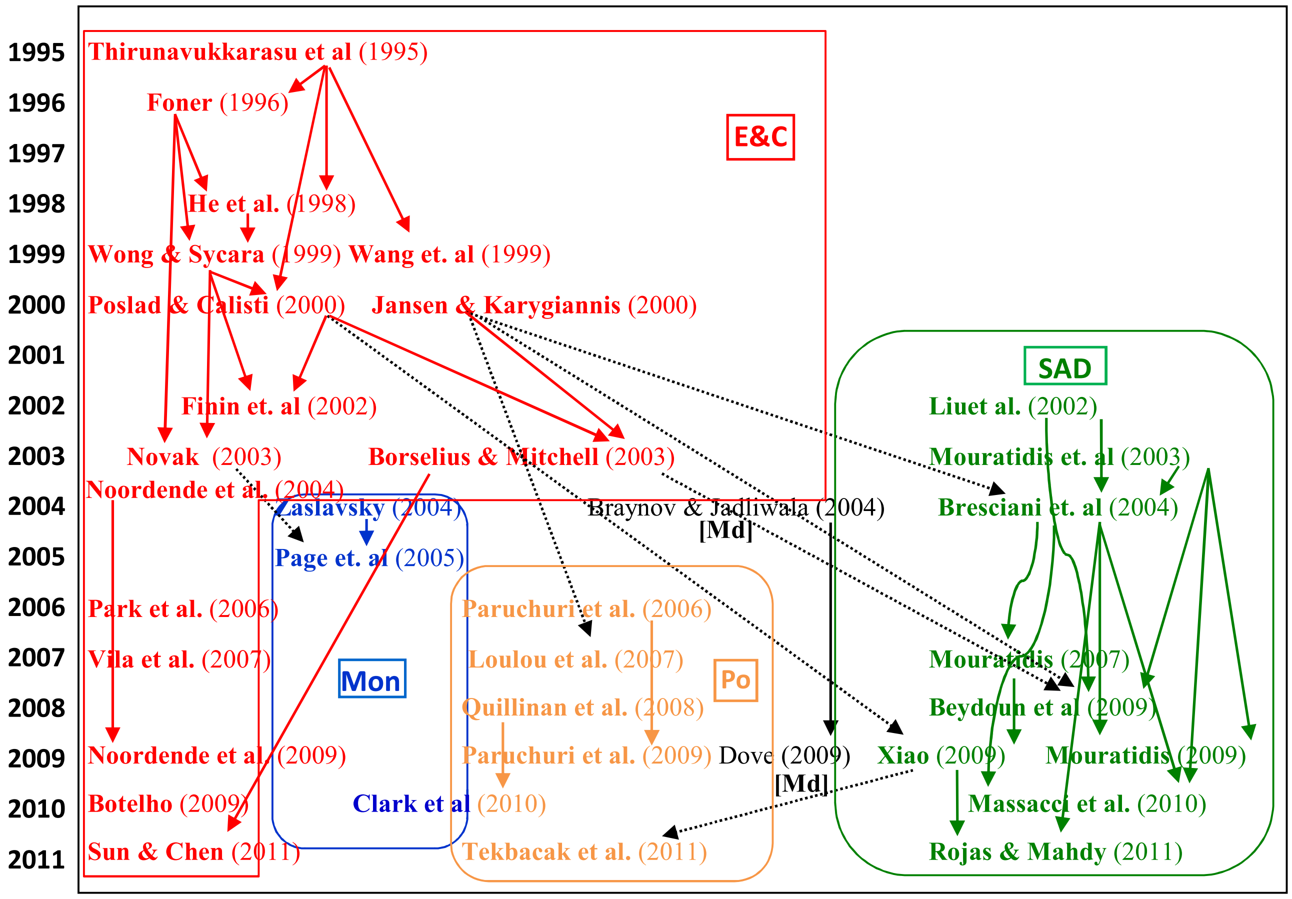


Fig.4-1: Chronological order of proposed security approaches to secure MASs. 'E&C' means encryption and certificate-based methods, 'Po' is policy-based methods, 'SAD' stands for secure agent development, 'Mon' is for monitoring approach and 'Md' means modelling technique.

The proposed appropriate security techniques for each attack category are summarised in Table 4-4. In this table, ● indicates an effective and feasible security technique to counter an attack, which is strongly recommended, although it may not completely solve the problem. ○ means that the attack cannot be prevented or detected with the specified security countermeasure and ⦿ shows that the security technique does not solve the problem, but helps us make the attack harder or discourage attackers. The feasibility and effectiveness of suggested solutions are imprecise because of variations in security sub-techniques and their dependency on the application and implementation of the open MAS. For example, variety access-control mechanisms have different capabilities in the prevention of ontology attacks, however in

Table 4-4, it is assumed that the most effective mechanism will be used to countermeasure each attack.

Table 4-4: Summary of security mechanisms to countermeasure different attacks on open MASs

		Prevention					Detection				
Attacks	Countermeasures	**Encryption and Certificates**	**Policy-based (Access Control)**	**Policy-based (Behaviour defin.)**	**Agent-oriented Software Eng.**	**Language-based Security**	**Peer Monitoring**	**Information Monitoring**	**Policy Monitoring**	**Activity Monitoring**	Attack Modelling
Read attack	Interception	●	●	○	●	●	○	○	◉	○	○
	Access to agents	●	●	○	●	◉	○	○	◉	○	○
	Provenance	●	●	○	●	◉	○	●	◉	◉	○
Probing	Ontology attack	◉	◉	◉	◉	◉	○	●	◉	◉	◉
	Active probing	◉	◉	◉	○	●	○	◉	◉	◉	◉
Altering	Interaction modification	●	●	○	●	●	●	○	◉	○	○
	Agent code modification	●	●	○	●	●	●	○	◉	○	○
	Agent log modification	●	●	○	◉	○	●	○	◉	○	○
Injection	Message injection	◉	◉	○	○	●	○	◉	◉	◉	◉
	Knowledge injection	◉	◉	○	○	●	○	◉	◉	◉	◉
Flooding	Link flooding	○	○	◉	●	○	◉	○	○	●	●
	Agent flooding	○	○	◉	●	○	◉	○	○	●	●
DoS	Logical DoS	○	○	◉	●	○	◉	○	○	●	●
	Repudiation	●	○	○	◉	○	○	○	◉	○	●
Deception	Fake agents	●	◉	◉	◉	○	○	○	○	●	◉
	Fake services	●	◉	◉	◉	○	◉	○	○	●	◉
	Reputation attack	●	○	◉	◉	○	○	○	○	●	●

In this thesis, hereafter, the term security refers to confidentiality and secrecy, unless otherwise stated. The next chapter deals with information leakage problem in open MASs.

Chapter 5

Information Leakage in Agent Interactions

5.1 Introduction

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. Revealing information may also help an adversary learn about the MAS and form a plan of attack.

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.

In this chapter, different types of insecure information flows in open MAS governed by LCC interaction models are introduced. Then, two approaches to avoid information leaks through insecure flows are proposed.

5.2 Security Levels

The first step in secure information flow analysis for LCC interaction models is defining security levels for LCC terms and components. 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 $\leq$. 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 thesis.

Definition 5-1 (Security Levels):

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

We need to ensure that information flows only upwards in the lattice (D. E. Denning 1976) e.g. when $l \leq h$, 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.

5.3 Insecure Information Flows

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.
```

5.3.1 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, PrivateAgent)::   ...
```

5.3.2 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[1]. 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 such as the illustrated examples of *interception* and *probing* attacks in Chapter 3. We need to prevent both *explicit* and *implicit* insecure information flows in order to ensure no information leaks to unauthorised parties.

5.3.3 Summary

Table 5-1 to Table 5-3 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 5-1, 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.

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

Table 5-1. 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 5-2 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 5-2. 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 5-3 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 5-3, `Max_Operation` is the maximum security level of `Operation1` and `Operation2`.

Table 5-3. 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**

The next section suggests countermeasures against insecure information leakage in LCC interaction models. It worth noting that, depending on the attack scenario or the type of the vulnerability, we sometimes investigate the whole interaction model and sometimes analyse a few clauses within it.

5.4 Solutions

Two approaches to address information flow problems in MASs governed by LCC interaction models are *conceptual modelling* by analysing the abstract models of LCC 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. In language-based analysis of LCC code, we employ security types for LCC terms and enforce a security policy by type checking.

5.4.1 Conceptual Modelling

In conceptual modelling, the idea is to convert LCC interaction models into another formalisation (e.g. pi-calculus or propositional logic) and use an existing information leakage analysis tool for the new representation (e.g. Counterdog[1], Secure Session[2], etc.).

In (Bijani, Robertson and Aspinall 2011), we have suggested a framework to detect *active probing* attacks that use electronic institutions to attack MASs. Although this example does not address all the information flow problems directly, it can describe the three main steps to conceptual modelling illustrated in Fig. 5-1. We have used *conceptual representation* to formulate LCC interaction models into a logical form and employed Counterdog as the analysis tool. In *active probing* attacks, a malicious interaction model designed by an adversary is deployed on a MAS. The adversary might then infer confidential facts about the agents by analysing the results of smart queries during interactions.

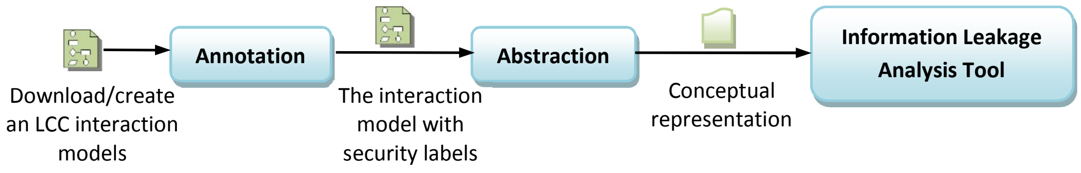

Download/create an LCC interaction models
Annotation
The interaction model with security labels
Abstraction
Conceptual representation
Information Leakage Analysis Tool

Fig. 5-1: The three steps to information leakage analysis using LCC conceptual modelling.

5.4.1.1 Annotation

The first step in analysing the secrecy of an LCC interaction model is defining the security policy through adding security labels to LCC terms. In the original LCC syntax there are no means of assigning security levels to information. Variables, constants and constraints are ultimately the most elementary causes of the described information leaks, so when we receive (e.g. download) an LCC interaction model, we need to annotate it to reflect the confidentiality level of the information.

[1] Becker, et al., Foundations of Trust Management (2012).

[2] Corin, et al., Secure Implementations for Typed Session Abstractions (2007).

We define the following annotation format:

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

in which `label` is a keyword, `Term` is any LCC term and `Level` is the security label (e.g. Fig. 5-2 (b)). Without loss of generality, security labels are defined as having two levels: high (`h`) or low (`l`). `l` Means low security or public information and `h` means high security or secret information. The default security level for a term without a label would be high security.

Definition 5-2 (Security Environment):

A security environment Π is a finite map from LCC terms to security levels and is defined by

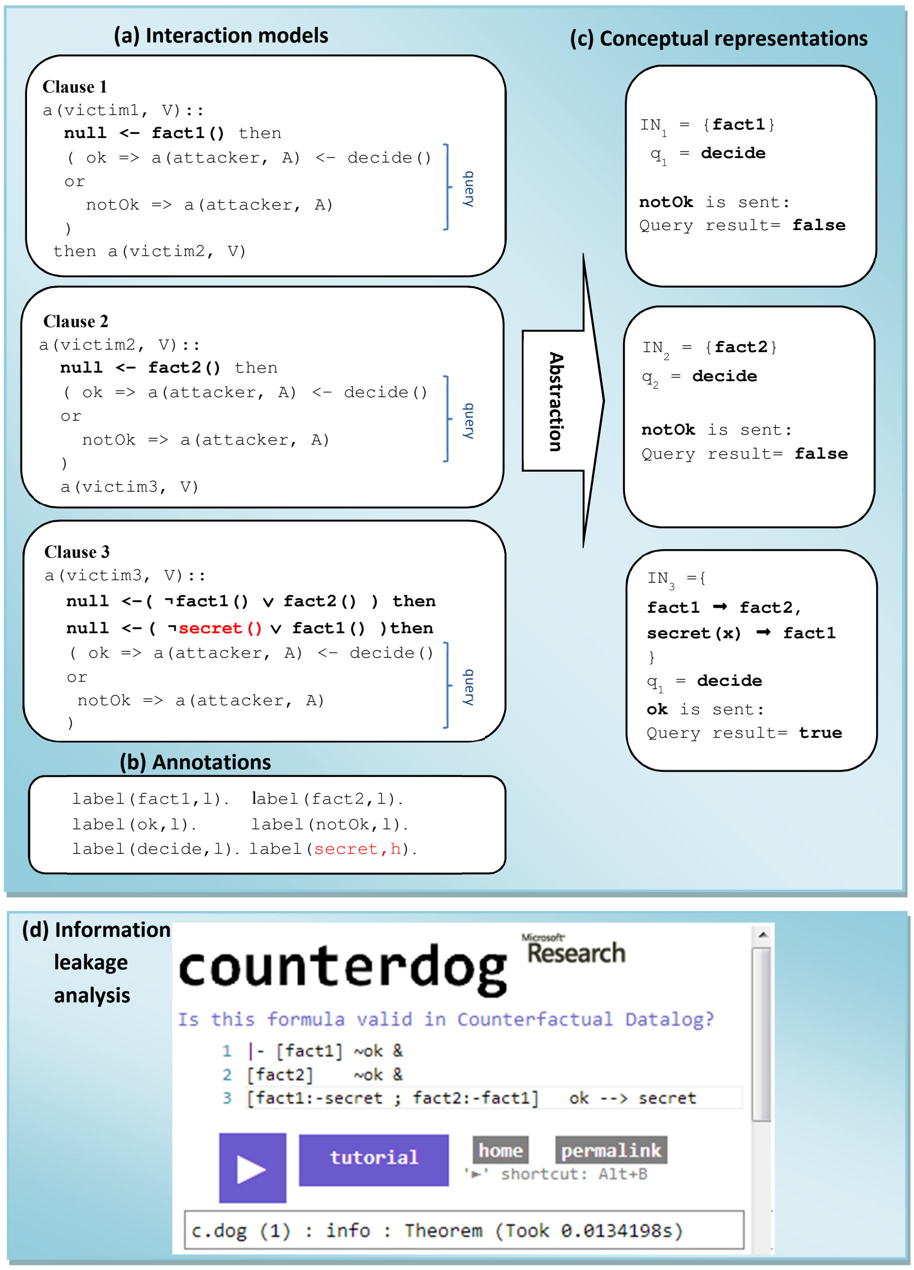
$$\Pi ::= \text{empty} \mid \Pi, \mathrm{T}: \{l, h\}, \tag{5-1}$$

in which Π is empty (with no binding) or an updated environment that contains a mapping of the LCC term T to the level *l* or *h*. As defined in section 2.3, an LCC term is either a constant, a variable or a structured Prolog term.

5.4.1.2 Abstraction

The next step in the security analysis is converting the annotated interaction models to simpler logical representations, which we call the *conceptual representation*, in order to identify only those parts of the LCC code related to the secrecy evaluation. In this conceptual representation, a more minimal interpretation of LCC code is represented, which reflects information leaks or helps to find information leakage. The conceptual representation may vary in different applications and scenarios. Hence, we cannot define a general-purpose conceptual representation that reflects all information leakages and can be perform automatically without need of manual supervision.

As an example, for the detection of active probing attacks, in (Bijani, Robertson and Aspinall 2011) we define the following conceptual representation: the *then* operator in LCC is translated in this case to a logical conjunction. That is because we use non-temporal logic and we ignore the effect of the actions' sequence on the information inferred by the adversary. LCC constraints are interpreted as queries or injections from the counterpart agent (an adversary). Hence the conditional operator (`<-`) is used only to find queries and injections and it does not appear in the representation. The sent message in the left of <– (if one exists), could be an answer to the query. The received message's parameters are also considered as new information for the receiver agent. We can legitimately interpret LCC interaction models in this way because we are not defining the semantics of the LCC specification but, instead, we are describing information that can be inferred to be true, as the result of our analysis. The abstractions of three simple LCC clauses are illustrated in Fig. 5-2(c).



Clause 1
```
a(victim1, V)::
  null <- fact1() then
  ( ok => a(attacker, A) <- decide()
  or
    notOk => a(attacker, A)
  )
 then a(victim2, V)
```

Clause 2
```
a(victim2, V)::
  null <- fact2() then
  ( ok => a(attacker, A) <- decide()
  or
    notOk => a(attacker, A)
  )
  a(victim3, V)
```

Clause 3
```
a(victim3, V)::
  null <-( ¬fact1() ∨ fact2() ) then
  null <- ( ¬secret() ∨ fact1() )then
  ( ok => a(attacker, A) <- decide()
  or
   notOk => a(attacker, A)
  )
```

```
label(fact1,l).  label(fact2,l).
label(ok,l).     label(notOk,l).
label(decide,l). label(secret,h).
```

```
IN_1 = {fact1}
q_1 = decide

notOk is sent:
Query result= false
```

```
IN_2 = {fact2}
q_2 = decide

notOk is sent:
Query result= false
```

```
IN_3 ={
fact1 ➡ fact2,
secret(x) ➡ fact1
}
q_1 = decide
ok is sent:
Query result= true
```

Is this formula valid in Counterfactual Datalog?

```
|- [fact1] ~ok &
[fact2]    ~ok &
[fact1:-secret ; fact2:-fact1]    ok --> secret
```

'►' shortcut: Alt+B

```
c.dog (1) : info : Theorem (Took 0.0134198s)
```

Fig. 5-2: An example of interaction modelling using the Counterdog tool to detect potential information leakage in an active probing attack. Finally, it shows that an adversary could infer that `secret` holds.

5.4.1.3 Information Leakage Analysis

After the conceptual representation of interaction models, in which injections and queries are defined, we analyse them to detect the possibility of information leakage. A leakage happens when a malicious agent infers any confidential fact about its counterpart's local knowledge.

In potential active probing attacks, we want to know when an agent plays a role in an interaction model designed by an adversary, in which some queries are asked, what secret information the adversary can infer. We then use a theorem prover such as Counterdog (Becker and Sultana 2012). Counterdog is an automated reasoning tool by Microsoft Research for a counterfactual meta-logic on propositional Datalog[1] programs. Counterdog is complete for this logic (i.e. can prove or disprove counterfactual statements) and accepts classical propositional logic statements as input. This prover employs an inference system for *detectability*[2] (Bryans, Koutny, et al. 2008) of a specific property in the input statements.

In Fig. 5-2 (a), a simple template of an active probing attack scenario is illustrated. The victim agent interacts with an attacker agent using three clauses of an interaction model published by an adversary. The constraints *fact1* and *fact2* and their combinations, which we call injections, are placed in the indication model. The same query (about the decision of the agent) is asked in the three interactions. In this attack, the assumption is that the injection could change the agent's knowledge state, which affects the agent's decisions. In other words, these constraints are information added to the knowledge-base of the *vendor* agent and could shape its decisions. If the attacker's query is unsuccessful the first two times (*notOk* message is sent) and successful for the third time (*ok* message is sent back to the attacker), after some analysis, the adversary can infer that the *secret* (which is a confidential fact) holds.

To check whether an adversary can deduce secret information form the interaction, we can formulate the attack in the following format accepted by the Counterdog prover:

```
|-  [fact1] ~ok  &  [fact2] ~ok  &  [fact1] ok  -->  secret
```

[1] Propositional Datalog is similar to Prolog, but all predicate parameters are constants and it does not have the "impure" features (such as *cut, not,* etc.) commonly presented in Prolog implementation.

[2] Detectability or non-opacity is an information flow property that denotes the ability to infer a specific predicate from a set of rules.

In this formula, the facts that hold (injections) are in bracket and the query result comes after that. The secret term to be investigated is placed after the --> operator. The prover determines if the formula is a valid theorem or a non-theorem.

As revealing secret information is important to us, we need to check the detectability of each secret LCC term to ensure high security information does not leak.

5.4.1.4 Updating the LCC Rewrite Rules

The proposed information leakage analysis can be performed before running the interaction model or at run-time. If we perform information leakage analysis before run-time, we need to check all possibilities of the query result. In the run-time analysis, the analyser is executed at each step of the LCC clause expansion, and if no leakage is detected, it allows the interaction to continue; so we would need an automatic translation of LCC code into the conceptual representation.

In order to integrate the automatic abstraction phase into the LCC interpreter, we upgrade the LCC clause expansion mechanism (as explained in Chapter 2) to detect probing attacks. The general format of the new rewrite rules is as following:

$$s(\Delta, \text{ReW}, \Delta'),$$

in which s is the notion of the new rewrite rules, Δ is the current security context, ReW is an LCC rewrite rule in Fig. 2-2 (page 21), and Δ' is the updated security context after expanding the rewrite rule. Δ= (Π, R, K), where Π is the mapping between LCC terms and security labels, R is the conceptual representation of interaction models and contains the set of injections and queries and K is the agents' current state of knowledge.

$$s\left(M \Rightarrow A \leftarrow C \xrightarrow{M_i, M_o, S, \{M \Rightarrow A\}} c(M \Rightarrow A), \Delta, \Delta'\right) \quad if \;\; extract\_rep\left(\Delta, \Delta^{'}, C\right) \wedge satisfied(C, \Delta')$$

where

C = the constraint to be satisfied

Δ = security context before abstraction of C,

Δ' = security context after abstraction of C, and *extract_rep* function extracts the conceptual representation of an LCC term. The updated LCC rewrite rules are shown in Fig. 5-3.

$$s\left(A :: B \xrightarrow{M_i, M_o, S, O} A :: E, \Delta, \Delta'\right) \qquad if\ s\left(B \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta'\right) \qquad (1)$$

$$s\left(A_1\ or\ A_2 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta'\right) \qquad if\ \neg closed(A_2) \wedge s\left(A_1 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta'\right) \qquad (2)$$

$$s\left(A_1\ or\ A_2 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta'\right) \qquad if\ \neg closed(A_1)\ \wedge\ s(A_2 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta') \qquad (3)$$

$$s\left(A_1\ then\ A_2 \xrightarrow{M_i, M_o, S, O} E\ then\ A_2, \Delta, \Delta'\right) \qquad if\ s(A_1 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta') \qquad (4)$$

$$s\left(A_1\ then\ A_2 \xrightarrow{M_i, M_o, S, O} A_1\ then\ E, \Delta, \Delta'\right) \qquad if\ closed(A_1)\ \wedge\ s(A_2 \xrightarrow{M_i, M_o, S, O} E, \Delta, \Delta') \qquad (5)$$

$$s\left(C \leftarrow M \Leftarrow A \xrightarrow{M_i, M_i - \{M \Leftarrow A\}, S, \emptyset} c(M \Leftarrow A), \Delta, \Delta'\right)$$

$$if\ (M \Leftarrow A) \in M_i\ \wedge\ extract\_rep\left(\Delta, \Delta', C\right) \wedge\ satisfied(C, \Delta') \qquad (6)$$

$$s\left(M \Rightarrow A \leftarrow C \xrightarrow{M_i, M_o, S, \{M \Rightarrow A\}} c(M \Rightarrow A), \Delta, \Delta'\right) \qquad if\ extract\_rep\left(\Delta, \Delta', C\right) \wedge\ satisfied(C, \Delta') \qquad (7)$$

$$s\left(null \leftarrow C \xrightarrow{M_i, M_o, S, \emptyset} c(M \Rightarrow A), \Delta, \Delta'\right) \qquad if\ \ extract\_rep\left(\Delta, \Delta', C\right) \wedge\ satisfied(C, \Delta') \qquad (8)$$

$$s\left(a(R, I) \leftarrow C \xrightarrow{M_i, M_o, S, \emptyset} a(R, I) :: B, \Delta, \Delta'\right)$$

$$if\ closed(P, a(R, I) :: B) \wedge\ extract\_rep\left(\Delta, \Delta', C\right) \wedge\ satisfied\left(C, \Delta'\right) \qquad (9)$$

$$closed\left(c(X)\right) \qquad (10)$$

$$closed(A\ or\ B) \leftarrow closed(A)\ \vee\ closed(B)$$

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

$$closed(X :: B) \leftarrow closed(B)$$

Fig. 5-3: The Updated LCC rewrite rules for expansion of one clause P in an interaction model.

The definition of LCC traces (Robertson, Barker, et al. 2009) is extended to include secrecy analysis (Fig. 2-3). As described in Chapter 2, *S* is the state of an interaction, M_i is the initial set of messages, *p* is a unique identifier for an agent. The $\overset{s}{\supseteq}$ operator in $S \overset{s}{\supseteq} S_p$ selects a clause S_p from the interaction state *S*. $S_p \overset{s}{\cup} S$ merges specific clause S_p to *S* and generate a new interaction state *S′*. *i*(S, M_i, S_f, Δ, Δ′) is true when the sequence of interactions and an initial set of messages M_i change the initial state of the interaction model *S* and security context *Δ* to the state S_f and the security context *Δ′*. *LA* is responsible for the information leakage analysis, described in section 5.4.1.3, through calling the Counterdog theorem prover. Counterdog can also be called as a web service, as it has a web interface. If it detects an attack, it will prevent the expansion of the rest of the interaction model and will generate an alert.

$$i(S, M_i, S_f, \Delta, \Delta_f) \leftrightarrow \begin{pmatrix} S = S_f \\ \wedge \\ \Delta = \Delta_f \end{pmatrix} \vee \begin{pmatrix} S \overset{s}{\supseteq} S_p & \wedge \\ s(S_p \xrightarrow{M_i, M_o, S, O} S'_p, \Delta, \Delta') \wedge LA(\Delta', p) & \wedge \\ S_p \overset{s}{\cup} S = S' & \wedge \\ i(S', M_o, S_f, \Delta, \Delta_f) & \end{pmatrix}$$

$$S \overset{s}{\supseteq} S_p \leftrightarrow \exists R, B. (S_p \in S \quad \wedge \quad S_p = a(R, I) :: B)$$

Fig. 5-4: Revised definition of a trace through an LCC interaction model *S* that supports secrecy analysis. The *LA* function calls the Counterdog prover.

5.4.1.5 Discussion

Conceptual modelling of the agent's interaction models as a countermeasure to information leakage vulnerabilities has been described by an example addressing active probing attacks on LCC protocols.

The main advantages of this approach are its independence from the LCC implementation and the flexibility in using existing secrecy analysis tools. Conceptual modelling is not depend on the LCC interpreter, which might have been implemented in Prolog, Java, etc., as the model of the LCC interactions are analysed not the LCC codes. We employed Counterdog, while other automated reasoning tools such as ProVerif (Blanchet 2001) can be used to prevent information leakage.

The main drawback of the conceptual modelling approach is that we check the encoded LCC protocols, rather than the LCC code itself. Hence, it might be difficult to pin-point the exact cause of the found leakages in the original LCC code. Furthermore, automatic abstraction, which is the key to implement an automatic information leakage analysis in the conceptual modelling method, is not easy to generalise. This is because there is no unique way to abstract different exploitation scenarios. In the given example, in the process of defining queries, a proper result is very much dependant on user opinion.

It is important to note that the suggested detection example is intended to detect only one type of information leakage, i.e. as a result of active probing attacks. To detect all insecure information flows, the abstraction and analysis phases need to be extended.

If we want to prevent information leakage, it is necessary to simulate the interactions that an agent undertakes, before run-time, and to know the security level of each term, the constraints that hold and the query results in advance. This simulation might not be accurate, undermining the analysis based on it. In a more cautious approach, ambiguous interaction models may be rejected. In run-time analysis we cannot guarantee to prevent a leak before it happens and we may only detect it when it has already occurred, which would be too late.

As free variables in formulas are disallowed in Counterdog, conceptual representations with variables need to be bound to a quantifier or converted to ground expressions. Nevertheless, if the detection happens at run-time, finding a ground substitute might not cause loss of generality in practice. This is because variables normally have been replaced with constants, before the conceptual representation phase.

Language-based information flow analysis of LCC codes is another countermeasure for information leakage in open MAS that overcome most of the weaknesses. The next section describes this approach.

5.4.2 Language-based Information Flow Analysis

Information flow analysis is one of the main techniques for studying confidentiality (Gorrieri, Martinelli and Matteucci 2009). Conventional security mechanisms usually target the operating system or low-level network communications and treat programs as black boxes, so they do not guarantee end to end security. Secure information flow analysis restrains both *access* to information and *use* of information by enforcing an end-to-end security policy.

Sound type systems are a promising language-based technique to specify and implement an information flow policy (Sabelfeld and Myers 2003). This is because security type checkers can automatically enforce security rules in programmes. In the security type checking approach, every LCC term has a security type and security is enforced by type checking.

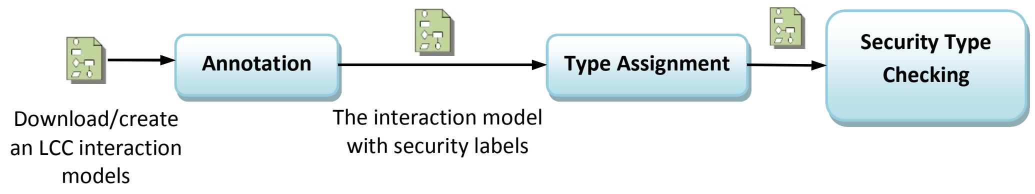

Annotation
Type Assignment
Security Type Checking
Download/create an LCC interaction models
The interaction model with security labels

Fig. 5-5: Three steps of information flow analysis using a security type system.

We propose a language-based information flow analysis framework for agent interaction models designed using the LCC language. In this framework, a security type system is designed for the LCC language to apply information flow controls on interaction models. The three steps for analysing information flow using the security type checking technique are shown in Fig. 5-5. To perform the secrecy analysis, the first step is defining the security policy (confidentiality policy) by annotation of the LCC code with security labels. This is similar to the annotation step in section 5.4.1.1. Security labels can be saved in the LCC interaction model file or as a separate file. So, the confidentiality policy can be created by the designer, programmer or the user of the interaction model.

The next step is automatic assignment of security types to LCC terms based on the syntactic structure of the terms. If there is any constraint that accepts different kinds of arguments (i.e. read-only, write-only, read-write) the annotation process is semi-automatic, i.e. the argument types need to be identified manually. This is due to the fact that the difference of arguments cannot be inferred from the syntax of the constraints.The third step is running the type checker against the code. We argue that type checking is a promising approach to prevent explicit and implicit insecure flows in order to guarantee secrecy. In the next chapter, this approach will be described in detail; both static and dynamic security type checking will be discussed.

5.5 Conclusions

In this chapter, we have addressed the information leakage problem in open MAS governed by electronic institutions and developed secrecy analysis frameworks for LCC interaction models. Explicit and implicit insecure information flows have been explained using a number of LCC examples. The two approaches to detect and prevent insecure information flows in choreography systems using the LCC language are conceptual modeling of interaction models and language-based information flow analysis.

The conceptual modeling method includes three steps namely: *annotation*, *abstraction* and *information leakage analysis*. In an example on the detection of active probing attack against LCC protocols, we have described these steps. In the *annotation* phase, we label LCC interaction models to reflect the secrecy level (high or low) of each term. For each interaction model, the *abstraction* generates a conceptual (logical) representation that helps to find information leaks. In the *information leakage analysis* phase, we have used the Counterdog theorem prover to detect the possibility of private information disclosure by an adversary. The exemplar information leakage analysis tells us whether an adversary could infer some facts from the local knowledge of an agent, by placing some constraints in an interaction model and asking some queries. The suggested attack detection system can be deployed on each agent as a part of the LCC interpreter. Hence, we have updated the LCC rewrite rules in the LCC interpreter to perform the abstraction and information leakage analysis.

The second approach is language-based information flow analysis using security types. Augmenting LCC terms with security types is a promising technique because then the security type checkers can automatically enforce security policies in agent interactions. The next chapter proposes the security-typed LCC and explains this approach.

Chapter 6

Information Flow Analysis in Lightweight Coordination Calculus

6.1 Introduction

In this chapter, we introduce the security-typed LCC language. 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).

This chapter is laid out as follows. First, a security type system is proposed by defining the *security levels*, which represent the confidentiality policy, *security types* and the *type inference rules*. Then, this security type system is evaluated by proving some properties of it such as *type soundness*, *simple security* and *confinement*. Next, the dynamic and the static approaches in type checking are reviewed and the LCC rewrite rules are upgraded to support security types. Furthermore, two more information flow properties (i.e. *non-interference* and *declassification*) and some extensions to the type system are discussed. Then, the implementation of the dynamic approach is briefly described. Finally, the Conclusions section summarises the content of this chapter.

6.2 Security Type System for LCC

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

6.2.1 Security Types

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

Definition 6-1 (Security Type Environment):

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

$$\Gamma ::= \text{empty} \mid \Gamma, \text{T}: \varphi, \qquad (6\text{-}1)$$

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 $\Gamma \vdash \text{T} : \varphi$, then T is called a well-typed LCC expression under the security context of Γ.

Definition 6-2 (Security Types):

The security types of our system are defined as following:

$$\varphi = \tau \mid \text{uTrm } \tau \mid \text{agent } \tau \mid \text{con } \tau \mid \text{op } \tau, \qquad (6\text{-}2)$$

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[1] 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:

[1] *Operational commands* are the *Def* keyword in the LCC syntax (Fig. 2-1): *Def := Role | Message | Def then Def | Def or Def | null<- C | Role <- C*

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 <u>higher</u> 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 <u>lower</u> than or equal to τ in context Γ.

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

d. $\Gamma \vdash C : con\ \tau$ means that the constraint name and every arguments within *C* has a security level τ or <u>lower</u> in context Γ.

e. $\Gamma \vdash D\!: op\ \tau$ means that every receiver of a message or any updated identifier in an operational command (i.e. *Def*) has a security level τ or <u>higher</u> 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 $\Gamma(t1) = h$ and $\Gamma(f1) = l$.

Security levels are directly assigned to LCC terms by annotations of the LCC code (similar to section 5.4.1.1) with the following format:

`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 then assigned based on the term definitions. All security types can be inferred from the term structure automatically, except constraints' arguments that need to be defined explicitly (by the user). By default, a constraint's arguments are assumed to be non-updatable and security type τ is assigned to them.

6.2.2 Type Inference for LCC

The proposed security type system for LCC programs is described by two sets of typing rules (Fig. 6-1) and subtyping rules (Fig.1-2). 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 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6-1: The security typing rules for LCC

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

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

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

Fig. 6-2: 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* ⇒ *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. It 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[1], 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* are regulating the composition of constraints in LCC. The rule *If1* states that the security type of constraint C and the message sending operation (*M* ⇒ *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*.

[1] Non-numeric constant

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. 6-2. The subtyping rules *AgentRule*, *uTrmRule* and *opRule* are contravariant[1] and the *conRule* is covariant[2].

6.3 Evaluation of the Type System

Two approaches to evaluate the proposed system are the analytical method using proofs and the experimental method by running the type system on several LCC interaction models. We choose the former approach and prove some of the type system properties. This is because proofs give stronger guarantee of the correctness of the type system and do not need large number of LCC interaction models with known security issues.

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 6-3 (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.

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

Definition 6-4 (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 $L \xrightarrow{R_i, M_i, M_o, P, O} L'$ (page 21, Fig. 2-2, rules 1-13) or as a structural expansion of a compound constraint using rules 20-25 in Fig. 2-2.

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 $L \rightsquigarrow L'$.

Proof: 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 2-2:

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

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 2-2:

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

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 2-2:

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

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

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

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

We showed only a subset of the cases; other cases are discussed in Appendix A. □

Theorem 6-2 (Preservation):

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

Proof: 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⇝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 τ.

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 τ.

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 2-2 and the definition of transition ⇝, 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 discussed in Appendix B. □

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* τ (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.

It 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.* $\Gamma \vdash T\text{: } agent\ \tau$ or $\Gamma \vdash T\text{: } op\ \tau$, 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.

It 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 by the LCC interpreter at run-time to verify whether an interaction model is secure.

6.4 Dynamic Information Flow Analysis

Dynamic (run-time) information flow analysis can appropriately be added to LCC language because of the dynamic nature of LCC language. Dynamic security checks may be accomplished via two similar approaches: monitors (Russo and Sabelfeld 2010) or dynamic security typing (Hennigan, Kerschbaumer, et al. 2011). We select the latter approach to use the proposed type system (section 6.9).

Based on the reaction policy, type checking could result in termination of the execution or breach detection and continuation of the clause expansion (Fig. 6-3).

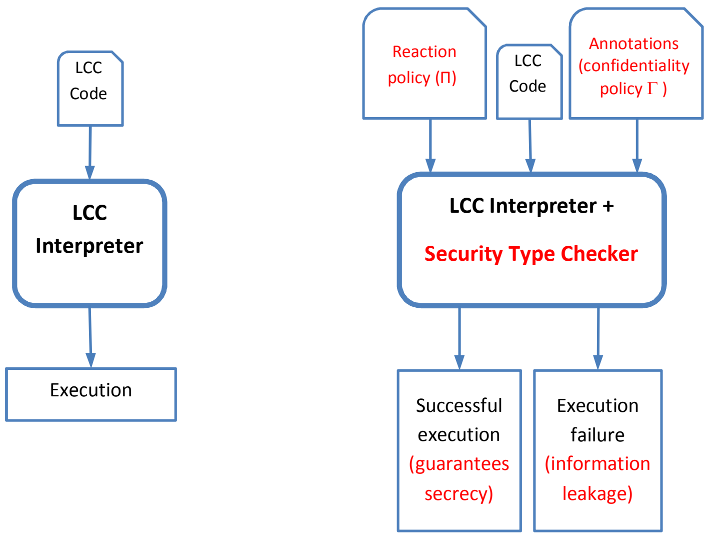

LCC Code
LCC Interpreter
Execution
Reaction policy (Π)
LCC Code
Annotations (confidentiality policy Γ)
LCC Interpreter +
Security Type Checker
Successful execution (guarantees secrecy)
Execution failure (information leakage)

Fig. 6-3: Upgrading the LCC interpreter (a) LCC interpreter executes LCC codes (b) Amended LCC interpreter performs the security type checking and executes the secrecy annotated LCC 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 2.3) has been upgraded by amending the LCC *rewrite rules*.

The extended LCC rewrite rules augmented with dynamic type checking are shown in Fig. 6-4. The updated rewrite rules in Fig. 6-4 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 $\Delta(\text{member})$; 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 6.2. 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. 6-4, 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. Based on the *closed* rules in (13) to (18), an interaction rule is decided to be 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 find another rewrite rule that matches the expression.

$$a(R,I) :: B \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} a(R,I) :: E \qquad if\ B \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \wedge typeChk(a(R,I)::E, \Delta) \qquad (28)$$

$$A_1\ or\ A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \qquad if\ \neg closed(A_2)\ \wedge\ A_1 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \qquad (29)$$

$$A_1\ or\ A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \qquad if\ \neg closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i, M_i, M_o, P, O}_{\Delta} E \qquad (30)$$

$$A_1\ then\ A_2 \xrightarrow{R_i,M_i,M_o,P,O} E\ then\ A_2 \qquad if\ A_1 \xrightarrow{R_i,M_i,M_o,P,O}_{\Delta} E \tag{31}$$

$$A_1\ then\ A_2 \xrightarrow{R_i,M_i,M_o,P,O}_{\Delta} A_1\ then\ E \quad if\ closed(A_1)\ \wedge\ A_2 \xrightarrow{R_i,M_i,M_o,P,O}_{\Delta} E \wedge\ typeChk(A_1\ then\ E, \Delta) \tag{32}$$

$$A_1\ par\ A_2 \xrightarrow{R_i,M_i,M_o,P,O_1\cup O_2}_{\Delta} E_1\ par\ E_2$$
$$if\ A_1 \xrightarrow{R_i,M_i,M_n,P,O_1}_{\Delta} E_1\ \wedge\ A_2 \xrightarrow{R_i,M_n,M_o,P,O_2}_{\Delta} E_2 \wedge\ typeChk(E_1\ par\ E_2, \Delta) \tag{33}$$

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i,M_i,M_i-\{M\Leftarrow A\},P,\emptyset}_{\Delta} c\big(C \leftarrow M \Leftarrow A, \Delta(L)\big)$$
$$if\ (M \Leftarrow A) \in M_i \wedge typeChk(C \leftarrow M \Leftarrow A, \Delta) \wedge satisfy(C) \tag{34}$$

$$M \Leftarrow A \xrightarrow{R_i,M_i,M_i-\{M\Leftarrow A\},P,\emptyset}_{\Delta} c(M \Leftarrow A, \Delta(L)) \qquad if\ (M \Leftarrow A) \in M_i \wedge typeChk(M \Leftarrow A, \Delta) \tag{35}$$

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i,M_i,M_o,P,\{M\Rightarrow A\}}_{\Delta} c(M \Rightarrow A \leftarrow C, \Delta(L)) \quad if\ satisfied(C) \wedge typeChk(M \Rightarrow A \leftarrow C, \Delta) \tag{36}$$

$$M \Rightarrow A \xrightarrow{R_i,M_i,M_o,P,\{M\Rightarrow A\}}_{\Delta} c(M \Rightarrow A, \Delta(L)) \qquad if\ typeChk(M \Rightarrow A, \Delta) \tag{37}$$

$$null \leftarrow C \xrightarrow{R_i,M_i,M_o,P,\emptyset,}_{\Delta} c(null \leftarrow C, \Delta(L)) \quad if\ satisfied(C)\ \wedge typeChk(null \leftarrow C, \Delta) \tag{38}$$

$$a(R,I) \leftarrow C \xrightarrow{R_i,M_i,M_o,P,\emptyset}_{\Delta} a(R,I) :: B$$
$$if\ clause(P, a(R,I) :: B) \wedge satisfied(C)\ \wedge typeChk(a(R,I) \leftarrow C, \Delta) \tag{39}$$

$$a(R,I) \xrightarrow{R_i,M_i,M_o,P,\emptyset}_{\Delta} a(R,I) :: B \qquad if\ clause(P, a(R,I) :: B) \tag{40}$$

$$closed\big(c(X,L)\big) \tag{41}$$
$$closed(A\ or\ B) \leftarrow closed(A)\ \vee\ closed(B) \tag{42}$$
$$closed(A\ par\ B) \leftarrow closed(A)\ \wedge\ closed(B) \tag{43}$$
$$closed(A\ then\ B) \leftarrow closed(A)\ \wedge\ closed(B) \tag{44}$$
$$closed(X :: B) \leftarrow closed(B) \tag{45}$$

Fig. 6-4: The amended LCC rewrite rules, which include security-related information and 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 a simple type checking (*typeChk*) is defined in Fig. 6-5. In Fig. 6-6, 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 (n*oChkMode*) 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 6-1: 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. 6-5: 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. 6-6: 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 6-1, 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:

$$\Pi = (Pre,\ Det,\ NoCk),$$

in which users can choose a policy by selecting one of the Boolean values *Pre*, *Det* and *NoCk* (Table 6-1). 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.

6.4.1 Drawbacks of Dynamic Type Checking

The main disadvantage of purely run-time information flow analysis similar to the one discussed above, is their false negative result as they cannot detect implicit information flows. This is because in dynamic security analysis of LCC, all execution paths of the program are not 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:

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

This is detected as a well-typed LCC command, which is wrong! This is because based on a high security constraint as described in section 5.3.3.

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 and 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).

6.5 Static Information Flow Analysis

In static information flow analysis, LCC interaction modes 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). 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. 6-7).

More examples of security type checking are given in the cloud computing case study in Chapter 7.

$$a(R,I) :: B \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: E \qquad if\ B \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(a(R,I)::E,\Delta) \tag{46}$$

$$A_1\ or\ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \qquad if\ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E \tag{47}$$

$$A_1\ or\ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \qquad if\ closed(A_1) \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(A_1\ or\ E,\Delta) \tag{48}$$

$$A_1\ then\ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E\ then\ A_2 \qquad if\ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E \tag{49}$$

$$A_1\ then\ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} A_1\ then\ E \qquad if\ closed(A_1) \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E \wedge typeChk(A_1\ then\ E,\Delta) \tag{50}$$

$$A_1\ par\ A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E_1\ par\ E_2 \qquad if\ A_1 \xrightarrow{R_i,P,\Delta}_{\Delta} E_1 \wedge A_2 \xrightarrow{R_i,P,\Delta}_{\Delta} E_2 \wedge typeChk(E_1\ par\ E_2,\Delta) \tag{51}$$

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c\big(C \leftarrow M \Leftarrow A,\Delta(L)\big) \qquad if\ typeChk(C \leftarrow M \Leftarrow A,\Delta) \tag{52}$$

$$M \Leftarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Leftarrow A,\Delta(L)) \qquad if\ typeChk(M \Leftarrow A,\Delta) \tag{53}$$

$$M \Rightarrow A \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Rightarrow A \leftarrow C,\Delta(L)) \qquad if\ typeChk(M \Rightarrow A \leftarrow C,\Delta) \tag{54}$$

$$M \Rightarrow A \xrightarrow{R_i,P,\Delta}_{\Delta} c(M \Rightarrow A,\Delta(L)) \qquad if\ typeChk(M \Rightarrow A,\Delta) \tag{55}$$

$$null \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} c(null \leftarrow C,\Delta(L)) \qquad if\ typeChk(null \leftarrow C,\Delta) \tag{56}$$

$$a(R,I) \leftarrow C \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: B \qquad if\ newClause(P, a(R,I) :: B) \wedge typeChk(a(R,I) \leftarrow C,\Delta) \tag{57}$$

$$a(R,I) \xrightarrow{R_i,P,\Delta}_{\Delta} a(R,I) :: B \qquad if\ newClause(P, a(R,I) :: B) \wedge typeChk(a(R,I) \leftarrow C,\Delta) \tag{58}$$

$$closed\big(c(X,L)\big) \tag{59}$$

$$closed(A\ or\ B) \leftarrow closed(A) \wedge closed(B) \tag{60}$$

$$closed(A\ par\ B) \leftarrow closed(A) \wedge closed(B) \tag{61}$$

$$closed(A\ then\ B) \leftarrow closed(A) \wedge closed(B) \tag{62}$$

$$closed(X :: B) \leftarrow closed(B) \tag{63}$$

Fig. 6-7: Static analysis of an LCC clause by expansion of an LCC clause.

6.5.1 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 genuine 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 annotate 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, while 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 cannot 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.

6.6 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 LCC 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 6-5 (Visibility): The set $\mathbf{visible}_{\mathbf{i}}(\Gamma)$ 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 6-6 (Alikeness[1])

$\Gamma 1 \approx_l \Gamma 2$: Two security contexts Γ1 and Γ2 are alike up to the level *l* iff: **visible$_l$ (Γ1) = visible$_l$ (Γ2)**.

For example, if we have the following two contexts: Γ1= { *m1: l*, *m2: l*, *m3: h* } and Γ2= { *m1: l*, *m2: l*, *m3: h* , *m4: h* }, then: ***visible$_l$*** (Γ1)={*m1*, *m2*} and ***visible$_l$*** (Γ2)={*m1*, *m2*}, which means other agents with security level of at least *l* can see these information. We also have $\Gamma 1 \approx_l \Gamma 2$.

Recall the LCC clause expansion mechanism of an original LCC clause C_i into C_n in terms of the interaction model P (introduced in section 6.4):

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

where security environment Δ=(*Γ, K, Π, L, Σ*) 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 6-7 (Observational Equivalence[2])

$O_{n1} \equiv_l O_{n2}$: 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 interpreted 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.

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

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

Definition 6-8 (Non-interference)

$$\forall\ \Gamma1, \Gamma2.\ (\Gamma1 \approx_l \Gamma2)\ \land\ C_i \xrightarrow{M_i, M_{n1}, P, \boldsymbol{O_{n1}}}_{\boldsymbol{\Gamma 1}} C_{n1}\ \land\ C_i \xrightarrow{M_i, M_{n2}, P, \boldsymbol{O_{n2}}}_{\boldsymbol{\Gamma 2}} C_{n2} \Rightarrow\ (O_{n1} \equiv_l O_{n2})$$

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 in section 6.2 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, which denotes it disregards information leaks due to the termination of the program (e.g. the last example in section 5.3.2). Thus, our type system cannot detect this type of insecure flows.

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.

6.7 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 (e.g. interaction model in Fig. 3-3). Rejection of a wrong 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 does not 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.

6.8 Extensions to the Typing Rules

The proposed type system may be extended to be more general, flexible, or fulfil different requirements. In this section we discuss some extensions to the typing rules that can expand the information flow analysis mechanism.

In the proposed security typing rules (Fig. 6-1), uTrm can only support variables, not structured terms; e.g. Only variables can be updated, not lists. This seems a reasonable assumption, if the term is an agent identifier but not for an argument in a constraint. To support data structures (such as lists) as updatable arguments in a constraint we need to add another typing rule:

$$\frac{\Gamma \vdash f\colon uTrm\ \tau,\ \Gamma \vdash T_i\colon uTrm\ \tau}{\Gamma \vdash f(T_i)\colon uTrm\ \tau} uStruct \quad i = 1, \dots, n$$

This helps the *Call* rule to guarantee that no low security list can be updated by high security information (no write down).

Another important issue in LCC are constraints. Constraints in LCC may have three functionalities: (1) conditions for other LCC operations (i.e. message passing and recursion) based on their returned value, (2) performing an action (e.g. doing intended jobs) and (3) updating some LCC terms (variables and structures).

The proposed type system supports these functionalities. As a condition, the main issue is the secrecy of the functor (the constraint's name) which is linked to the secrecy of readable arguments (τ type not uTrm τ type). So even if the secrecy level of the functor is not defined by the user, it can be inferred from these arguments (if any exist). As an action, we need to ensure that the performed operation has security less than or equal to the defined level of the functor.

In the proposed typing rules, we introduce the type rule *Call* for handling constraints, but we can extend it as below to support the updating functionality that accept read-write arguments; the type system needs to be modified slightly.

$$\frac{\Gamma \vdash this\colon agent\,\tau,\ \Gamma \vdash f\colon \tau,\ \Gamma \vdash T_i\colon \rho}{\Gamma \vdash f(T_i)\colon con\,\tau} Call2 \quad \rho = \tau |\ uTrm\,\tau | rwTrm\,\tau \quad i = 1, \dots, n$$

Γ⊢ *T: rwTrm* τ is a new type that means the read-write (input-ourput) identifier T in a constraint has a security level τ .This argument behaves as both a read-only identifier (type τ) and a write-only identifier (type uTrm τ), hence, the subtyping in rwTrm types is neither covariant nor contravariant and the security level of this argument can not be changed during type inference.

As LCC makes no commitment to the method of solving constraints, different agents might use different constraint solvers locally or a remotely. Hence, the assigned security types to the constraints limit the behaviour of (the external) constraint solver. We assume that the implementation of the constraint does not leak information.

The proposed type system can be merged with the security type system of the constraint solver to achieve a comprehensive information flow analysis; if a constraint is implemented by a version of Java that supports information flow, e.g. JIF (Myers, et al. 1997), security types and levels of the constraint's arguments can be translated to the equivalent types and values.

- **Extension of the Dynamic Approach**

The proposed dynamic security analysis can be extended to detect and prevent implicit flow. As discussed in section 6.4.1, the main problem of the dynamic approach is inability to prevent implicit flows due to ignoring some execution paths of the LCC program. The LCC interpreter selects one of the branches whenever it reaches to a committed choice operator. This problem can be solved by adding a run-time *control flow stack* (Hennigan, Kerschbaumer, et al. 2011). The *control flow stack* is implemented as a run-time shadow stack that records the history of *stack labels* attached to a controller at each control flow branch.

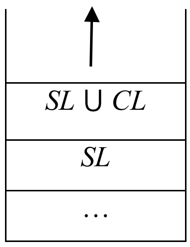


Fig. 6-8: The *control flow stack*. *SL* is the old stack label, *CL* is the security level of the constraint, the newly pushed stack label is *SL* ∪ *CL*.

The *control flow stack* is illustrated in Fig. 6-8; the newly pushed *stack label* when the control flow diverges is *SL* ∪ *CL*, where *SL* is the old *stack label* and CL is the security level of the constraint. The following two rules in a *control flow stack*, detects implicit flow:

1. If control flow diverges due to a conditional statement, a *stack label* is pushed onto the top of the *control flow stack* to indicate entry into the secure region.

2. If a control flow merges, the top of the *control flow stack* is popped and the previous *stack label* is restored to the level it had before the branch in control flow occurred.

We then need to merge the *control flow stack* mechanism with the *Choice* typing rule in Fig. 6-1. The following updated *Choice* rule incorporates the *stack level SL* in order to track the (nested) branches and detect implicit information flow.

$$\frac{\Gamma, SL \vdash A_1 : op\ \tau\ ,\ \ \Gamma, SL \vdash A_2 : op\ \tau\ , \tau \geq SL}{\Gamma \vdash A_1\ or\ A_2 : op\ \tau}\ Choice$$

Example: the following piece of LCC code leak secret information (the result of secretCondition) using implicit information flow:

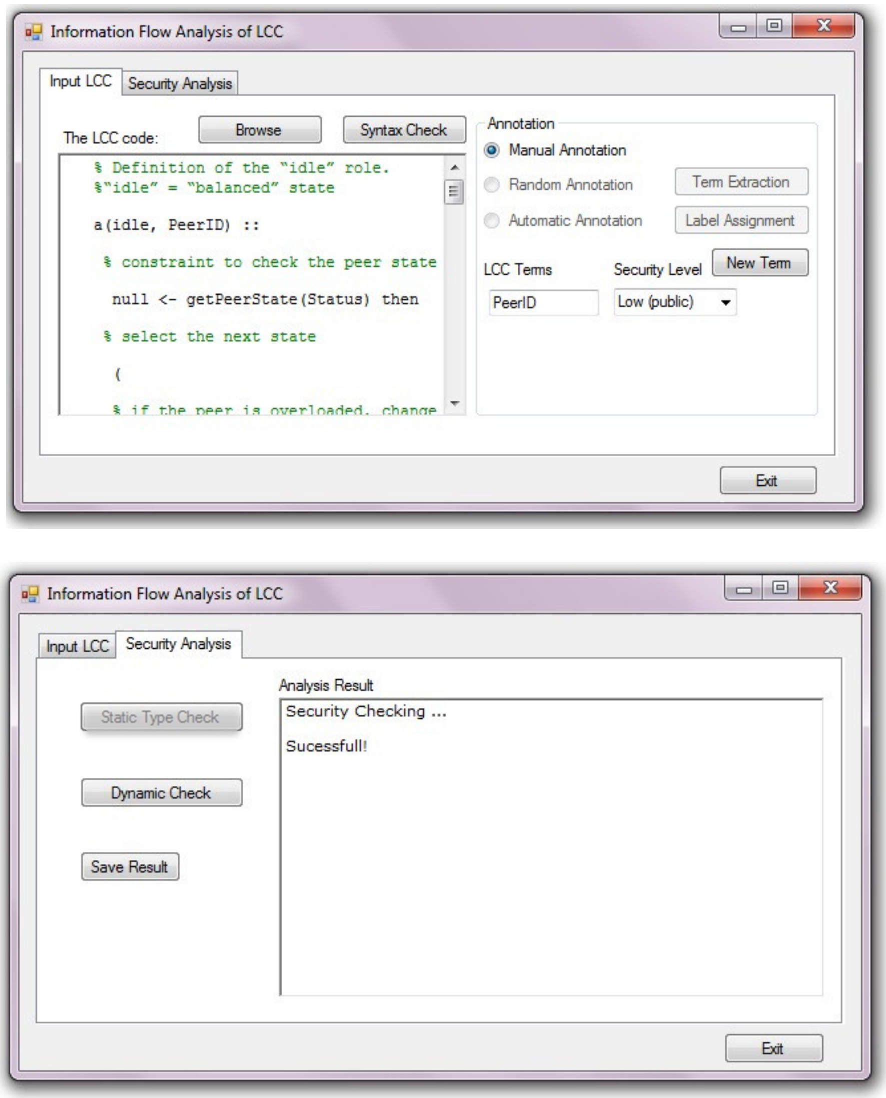
```
1.  (
2.      publicInfo1 ⇒ a(publicAgent, p)   % security type of this line is op l.
3.                                ← secretCondition()   % security type: con h.
4.              or
5.      publicInfo2 ⇒ a(publicAgent, p)   % security type of this line is op l.
6.  )
```

Let us assume the current *stack label* before running this code is *l*. *CL* is the security level of the constraint, so it is h. The control flow of diverges in line 1 (start of a choice operator), thus based on the rule 1, the new *SL* is *SL* ∪ *CL*, which is h, is pushed onto the top of *control flow stack*. Now, neither line 2, nor line 5 is allowed because in the typing rule we need $\tau \geq SL$ ($l \geq h$). Without using the *control flow stack* technique, line 5 is allowed in dynamic type checking, which is a security breach.

The next section briefly describes implementation of a prototype security-typed LCC.

6.9 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. There are several implementations of the LCC interpreter. However, the original version implemented in Prolog has been extended to support dynamic 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 C#.NET. This tool is designed for annotation of LCC interaction models with security labels and performing the security type checking.



```
   % Definition of the "idle" role.
   %"idle" = "balanced" state

   a(idle, PeerID) ::

    % constraint to check the peer state

     null <- getPeerState(Status) then

    % select the next state

     (

     % if the peer is overloaded, change
```

LCC Terms: PeerID

Security Level: Low (public)

```
Security Checking ...

Sucessfull!
```

Fig. 6-9: The user interface of the information flow analysis program for LCC codes

Fig. 6-9 illustrates the user interface of the security-typed LCC prototype program. The user can write an LCC interaction model or import an existing one from a file. In this first version of the security-typed LCC, security levels are defined manually by the user and terms which are not annotated are assigned to the highest security level (i.e. *h*). As a next step annotation may be done (semi)automatically by the matchmaker (at run-time).

An example annotation of an LCC interaction model that assigns security levels to LCC terms is shown in Fig. 6-10.

```
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. 6-10: Annotation of an LCC interaction model

6.10 Conclusions

In this chapter, we have proposed 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 and important properties; i.e. *type soundness*, *simple security* and *confinement*.

We have discussed two approaches for applying the security type system on the LCC interaction models; dynamic (run-time) and static type checking. The LCC rewrite rules in the LCC interpreter have then been updated to support information flow analysis.

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 section 6.8.

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.

A drawback of the proposed security type-checking approach is its dependency on the LCC language which makes hard to apply on other MAS platforms. However, adaptation of the proposed security type system for similar first-class agent protocol languages such as MAP[1] and RASA[2] is straightforward.

In this chapter, we have also formally defined the notion of non-interference in LCC interaction models and have shown that the proposed type system maintains this information flow property. The non-interference property guarantees that if an interaction model is run with

[1] MAP: Multi-Agent Protocol language (Walton 2004)

[2] RASA (Miller and McBurney 2007)

different secret inputs, while holding the public values fixed, the public output (outgoing messages), which is observable by the adversary, must not change.

Non-interference is considered as a restrictive property that may reject harmless interaction models for applications involving tasks such as authentication and data aggregation. Thus, sometimes we need declassification in order to release some secret information not worthwhile for the adversary. Another practical limitation of the language-based approach is difficulty to formalise intuitive policies using annotations. Realistic policies are complex and involve many stakeholders which makes hard to define conditions under which declassification may occur.

In the next chapter, as a case study, the proposed security type system is applied to cloud management scenarios governed by LCC interaction models.

Chapter 7

A Case Study in Cloud Computing

7.1 Introduction

In this chapter, we present an application of open MAS in cloud computing, i.e. cloud configuration management, and investigate information leakage analysis using the proposed security type system in this case study.

The cloud computing paradigm is a new class of network-based computing developed rapidly in the last few years to reduce administration and infrastructure costs. Cloud computing is predicted to revolutionise computing through reforming the management model of computing resource. It is also expected to be even more popular in the research community as well as industry in near future. Clouds can be viewed as a new generation of datacentres.

Virtualisation technology is the core to most cloud computing models and has transformed the availability and management of computer resources. In a cloud, each physical machine (PM) is capable of hosting several virtual machines (VMs). VMs can be migrated without considerable interruption to the running services or any noticeable impact on performance of running applications. As a result, we can have highly available services through dynamic load balancing on clouds.

The increasing popularity of clouds leads to new models and ideas for cloud computing; some examples are: "*hybrid clouds",* in which some resources are in-house and others provided externally by different providers, *"community clouds"*, where resources are offered collectively using P2P technology, or "*inter-cloud*" (Bernstein, et al. 2009) as an interconnected global cloud of clouds. The above cloud paradigms enlarge their capabilities through engaging different parties.

Existing commercial tools for VM management are usually based on centralised control, in which performance information is collected from all of the VMs by a central service. The more the size and the complexity of clouds increase, the less attractive is the centralised management architecture. In the case of *hybrid clouds, community clouds* or *inter-cloud* there may no longer be a single organisation with ultimate authority over the entire sources. Hence, the centralised approach is no longer appropriate.

This makes the MAS approach an appropriate solution to the problem of automatic cloud configuration management, especially where a globally optimal solution is not necessary or feasible. In this regard, we proposed a multi-agent VM management framework using LCC and the OpenKnowledge system in (Anderson, Bijani and Vichos 2012) and (Anderson, Bijani and Herry 2013). In this framework, agents negotiate to decide on VMs migration between themselves, without any need for a centralised authority.

Security is a challenging problem in cloud computing, especially when the number of interconnected and composed services increases. Clouds raise new privacy and confidentiality concerns as data are usually managed by external parties on remote data centres and various services have access to them. It is vital to deal with privacy and confidentiality issues of cloud users as one of the most important security concerns to encourage them to migrate from traditional local data centres to the cloud computing paradigm. Otherwise, users should either deprive themselves from the advantages of could-based services or take the risk of exposing their confidentiality.

Information leakage in cloud computing is a vulnerability that may cause serious privacy and confidentiality issues. This may happen in the scenario of virtual machine migration within a cloud (datacentre) or between clouds (datacentres). A case in which customers' sensitive information is leaked in a cloud when a service provider can access multiple virtual machines is discussed by Wu et al. (2010). Another example of potential information leakage is when a virtual machine of a high risk organisation (such as a bank) is hosted on a same physical machine that the adversary's virtual machine is resided. (Fig.7-1). Ristenpart et al. (2009) describe cross-VM side-channel attacks to extract information from a target virtual machine on the same physical machine.

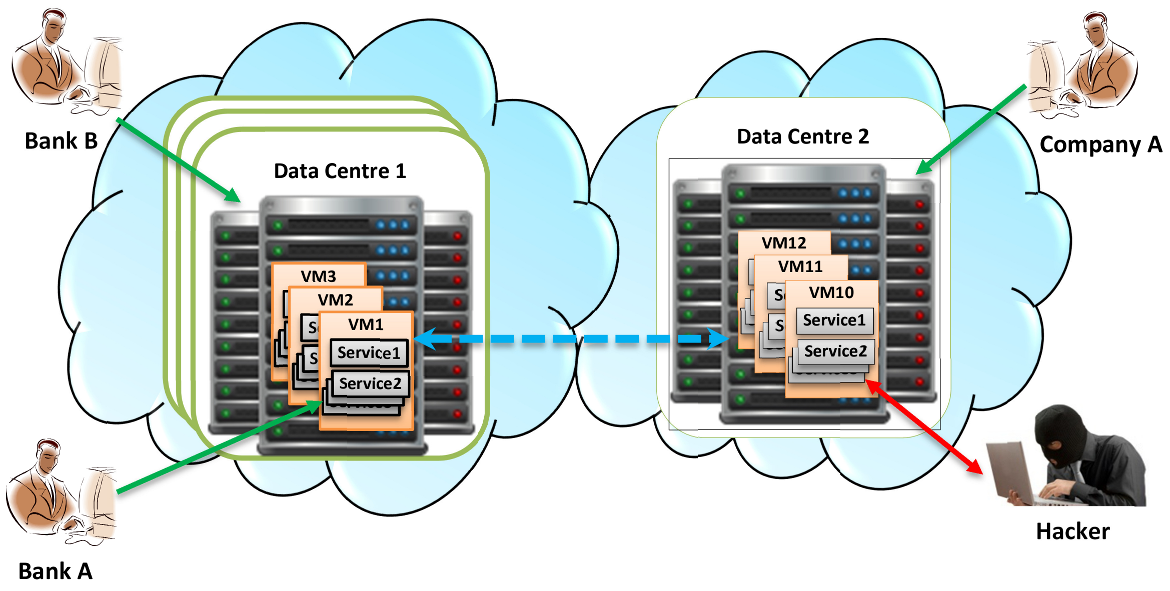


Fig.7-1: Vulnerable information flow in cloud computing caused by insecure virtual machine migration workflow

In this chapter, the information leakage problem in cloud management is addressed using the proposed information flow analysis in Chapter 6. As mentioned before, information flow analysis can be a complement to traditional security mechanisms, not an alternative. Encryption may be considered as a countermeasure to explicit leakage via message passing, not to other implicit or explicit information leaks. Access control mechanisms can also prevent some insecure explicit information flows, nevertheless they are not able to detect and prevent implicit flows.

Some of this chapter's material related to VM management using LCC has been published in (Anderson, Bijani and Vichos 2012) and (Anderson, Bijani and Herry 2013). The VM management scenario presented in this chapter has been simplified to maintain clarity in our explanation.

7.2 Autonomous Cloud Configuration Management

In this section our multi-agent solution to the VM management problem proposed in (Anderson, Bijani and Vichos 2012) and (Anderson, Bijani and Herry 2013), is briefly introduced. The agents use the OpenKnowledge system to find suitable LCC interaction models and to identify appropriate peers. The Interaction models define the behaviour of agents located on the physical or virtual machines. The agents make autonomous decision on VM migration based on local capabilities, free computing resources, the financial relationship of both parties, etc. The migration may be either *live*, which is transparent to the cloud service users and occurs within a datacentre, or *offline*, in which a more complex workflow handles cloning the VMs and services between different datacentres, and involves notifying clients.

7.2.1 Live Virtual Machine Management

An example of the LCC interaction model for managing the negotiation and transfer of virtual machines between two physical machines in the same datacentre (i.e. where live migration is possible) is shown in Fig.7-3.

In the first instance, we implement a policy which aims to migrate VMs from busy peers to underloaded peers in order to balance the load of each peer. There are three states: *idle*, *overloaded* and *underloaded*. The *idle* state is the initial and the goal state, in which the peer is balanced. Each peer is assumed to be balanced at the beginning of the interaction. It may then change state based on its load, or other factors. Fig.7-2 shows the corresponding state diagram, in which single-corner rectangles, diamonds and dashed-arrows represent agent roles, constraints, and message passing between agents, respectively.

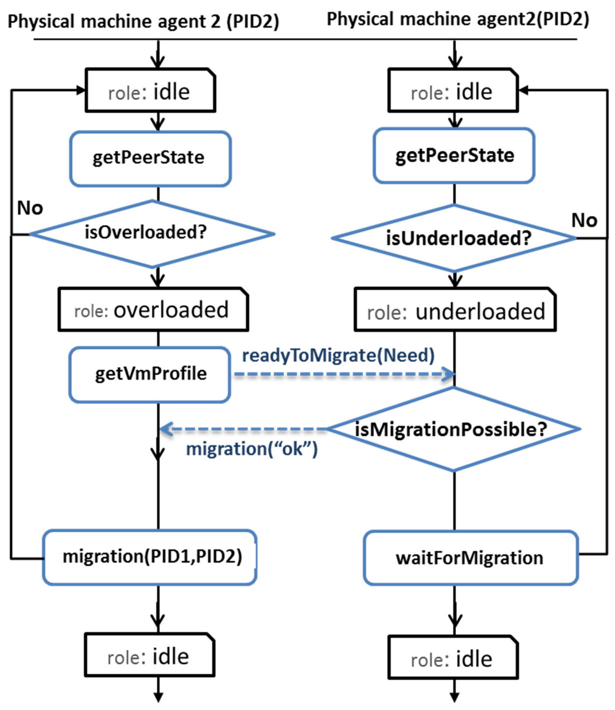


Fig.7-2: A simple Interaction diagram of a policy; unbalanced peers interact to balance their load

In this scenario, once a peer becomes unbalanced, it advertises its status to the discovery service where it will be matched with potential candidates for a transfer. The peer then negotiates with these candidates to find one which is prepared to participate in the transfer. The conditions for acceptance of the transfer and the complexity of the negotiation are completely determined by the interaction models of the individual peers - these may depend on, for example, security policies or cost considerations as well as the capabilities of the physical machine (processing power, network bandwidth, memory, etc.).

Fig.7-3 shows the interaction diagram of a simple implementation of the live migration scenario. The constraint *getPeerState(Status)* checks the state of the agent and selects the next state based on the agent's status. The constraint *isMigrationPossible* compares the *Need* of the virtual machine being migrated with the *Capacity* of the potential recipient. In our implementation we use a simple metric and a numerical comparison. In practice, this decision may be arbitrarily complex, and may be based on a number of parameters such as the available memory and load, or other characteristics. After an exchange of VMs, both agents revert to the *idle* role. If they are balanced, no further action takes place; otherwise, they continue to interact as an *underloaded* or an *overloaded* peer. This interaction model makes a single decision to accept or reject the proposed transfer. In practice, a more complex negotiation may be involved to find the most appropriate recipient.

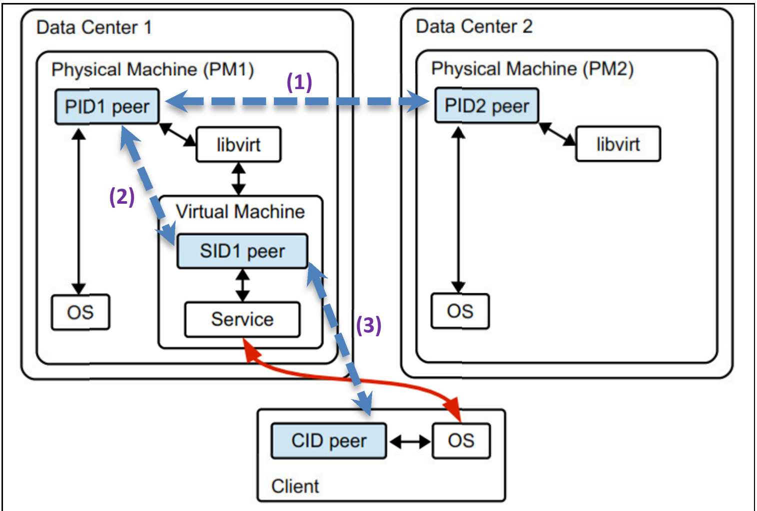
```
% Definition of the "idle" role. Here, "idle" means the "balanced" state
a(idle, PeerID) ::
  null <- getPeerState(Status) then %constraint to check the peer state
%select the next state based on the peer's status
( % if peer is overloaded, change its role to "overloaded"& pass the status
    a(overloaded(Status), PeerID)<- isOverLoaded()
  ) or
  ( % if the peer is underloaded, change its role to "underloaded"
    a(underloaded(Status), PeerID) <- isUnderLoaded()
  ) or
  a(idle, PeerID) % otherwise, remain in the idle role (recursion)
% "overloaded" role Definition. "Need" is the amount of required resources
a(overloaded(Need), PID1) ::
  % send the "readyToMigrate(Need)" message to an underloaded peer
readyToMigrate(Need) => a(underloaded, PID2) then
  % wait to receive "migration(ok)" from the underloaded peer
  migration(ok)<= a(underloaded, PID2) then
  % live migration: send VMs from this peer to the underloaded peer
  null <- migrateTo(PID2) then
  a(idle, PID1) % change the peer's role to "idle"
% "underloaded" role Definition. "Capacity"is the amount of free resources
a(underloaded(Capacity), PID2) ::
  % receive the "readyToMigrate(Need)" message from an overloaded peer
readyToMigrate(Need)<= a(overloaded, PID1) then
% send back the "migration(ok)" message, if the migration is possible, e.g.
% free "Capacity" of this peer >"Need" of the overloaded peer
(migration(ok)=> a(overloaded, PID1) <- isMigrationPossible(Capacity, Need))
then  null <- waitForMigration() )
    or  migration(notOk) => a(overloaded, PID1)  then
  a(idle, PID1) % change the peer's role to "idle"
```

Fig.7-3: The LCC interaction model of the live virtual machine migration.

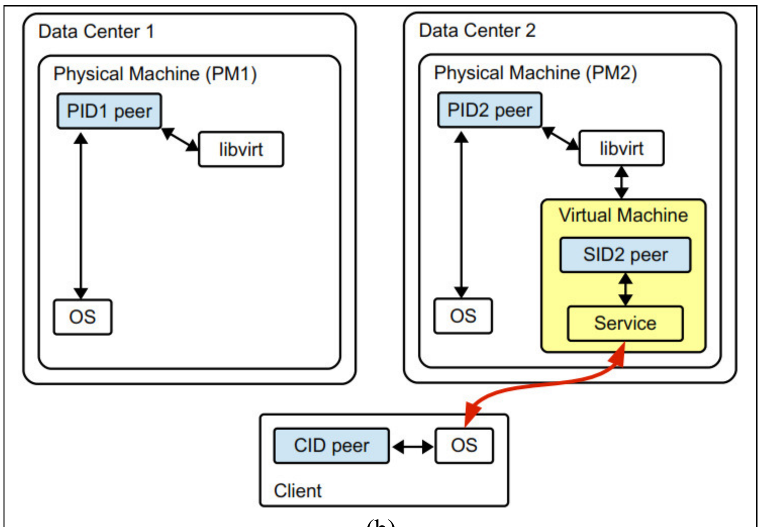
7.2.2 Virtual Machine Migration between Datacentres

Virtual machine migration within a datacentre (cloud) is not always possible, due to lack of resources, system support or cost.

Moving virtual machines between datacentres presents further challenges: in general it is not possible to perform a live migration, and a new virtual machine must be started in the target datacentre, and the services transferred, before stopping the original virtual machine. The new machine will also have a different IP address, and possibly other differences, which mean that the migration may not be transparent to clients of the service. In this case, the clients will need to be notified about the change, and a comparatively complex workflow may be needed to avoid any break in the service. Once again, there may be no obvious central authority to sequence this workflow, and this motivates an agent-based approach to the workflow execution. Fig.7-4 illustrates the workflow of offline virtual machine migration. For detailed information of workflow-based virtual machine migration and relevant LCC codes see (Anderson, Bijani and Herry 2013).



(a)



(b)

Fig.7-4: The workflow of offline virtual machine migration (Anderson, Bijani and Herry 2013); (a) the initial state and negotiations (dashed lines) between agents. (b) the final state where the VM has moved.

In the interaction diagram (Fig.7-5) of the LCC interaction model for offline virtual machine migration between datacentres, PID1 and PID2 are physical machine agents. PID1 starts with "initial" role, then by checking whether its VMs need to migrate, performs "emigrant" role. Similarly, PID2 changes its role from "initial" to "host", when it has enough resources, then if it is in a different datacentre, the offline migration process starts, by cloning new VMs. The next step is changing its role to "manage-services" and asking all new "service" agents (SID2s) to start the service. PID2 then informs PID1 that the destination is ready by sending "migration(ok)" message. Then, PID1 also changes its role to "manage-services" to stop current running services on datacentre1 by sending "do(shutdown)" message to SID1 service agents. Consequently, SID1 agents ask client agent CID to redirect from this services to newly lunched services on new VMs by sending "do(redirect)" message. CID then double check whether all necessary services are running in the new VMs by asking "check(running)" from each service agent SID2, then redirects to it.

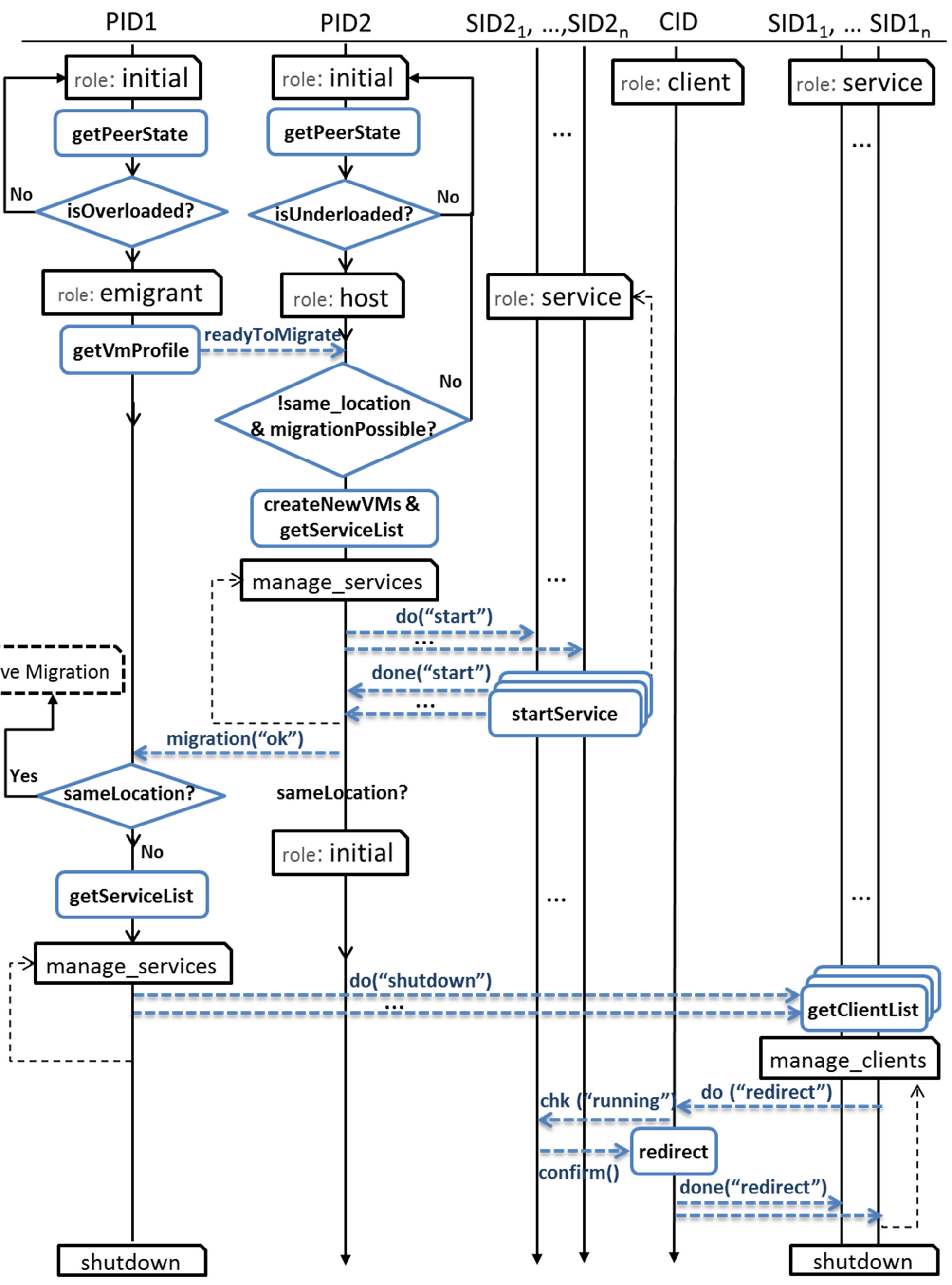


Fig.7-5: The interaction diagram of an LCC interaction model for offline virtual machine migration between datacentres.

Fig.7-6 illustrates some parts of the interaction model for offline virtual machine migration between datacentres. This is a simplified clause of an interaction model, in which some implementation details are omitted. The complete interaction model deals with exceptions and other practical issues.

```
 % Definition of the "emigrant" role.
a(emigrant, PID1) ::
% acquire the migrating virtual machine info and the list of services on it.
   null <- getVMprofile(VID1,V_Profile)and getServiceList(VID,SID_List) then
   % send a message to check availability of the required resources
   requiredResources(Load) => a(host, PID2) <- getPeerState(Load) then
   % receive the answer about the required services
   requiredResources(ok) <= a(host, PID2) then
   % send a message to check availability of the required services
   readyToMigrate(V_Profile, SID_List) => a(host, PID2) then
   % wait to receive "migration(ok)" from the host peer
   migration( ok ) <= a(host, PID2) then
   % VM migration from this peer (emigrant) to a "host" peer
   (null <-same_location(PID1, PID2) then
   null <- live_migration(PID2, PID1, VID1)
   ) or
   % next step of the offline migration
   (a(manage_services(SID_List, VID,  shutdown ), PID1) <-
                              not(same_location(PID1, PID2)) then
      null <- destroy(VID)
   ) then
a(shutdown, PID1).   % change the peer's role to "shutdown"
```

Fig.7-6: One of the LCC clauses of the virtual machine migration between datacentres.

7.3 Information Flow Analysis of Virtual Machine Management

The proposed security type system prevents and detects the information leaks. Having the security typing rules (section 6.2.2), the *overloaded* role definition in the LCC interaction model (Fig.7-3, lines 12 to 20) and the security annotations in Fig.7-7, we start our analysis. The security environment Γ corresponding to the annotations is as follows:

```
idle:  l,PeerID:  l,overloaded:  l,PID1:  l,  underloaded:  l,  PID2:h,
readyToMigrate: l,ok: l,notOk: l,migrateTo: l, ….
```

`Label(idle, l).`	`label(Capacity, h).`
`label(PeerID, l).`	`label(ok, l).`
`label(PID1, l).`	`label(notOk, l).`
`label(overloaded, l).`	`label(migrateTo, l).`
`label(PID2, h).`	`label(migration, l).`
`label(underloaded, l).`	`label(isMigrationPossible, l).`
`label(readyToMigrate, l).`	

Fig.7-7: Security label assignment to the LCC terms as annotations of the LCC code for the static security type check

Let us also assume the amount of resource shortage in the overloaded virtual machines is a secret piece of information; i.e. `Need: h`.

By static analysis of the role declaration in Fig.7-3, line 13;`a(overloaded(Need), PID1)`, we can prove that assigning the role `overloaded` to an agent `PID1` and passing the high argument `Need` to this role is not a permissible information flow:

$$\cfrac{\cfrac{\cfrac{\cfrac{overloaded: l \in \Gamma}{\Gamma \vdash overloaded: l}\, Id,\ l \le h}{\Gamma \vdash overloaded: h}\, Sub,\ \cfrac{Need: h \in \Gamma}{\Gamma \vdash Need: h}\, Id}{\Gamma \vdash overloaded(Need): h}\, Struct,\ \cfrac{PID1: l \in \Gamma}{\Gamma \vdash PID1: uTrm\ l}\, Id,\ l \neq h}{\Gamma \vdash a(overloaded(Need), PID1): agent\ ?}\, Agnt$$

Based on the annotations in Fig.7-7, `PID2` is a high security term; all the underloaded agents are assumed to be secret agents. Nevertheless, this is not always the case: e.g. a public physical machine in the datacentre with free resources for new virtual machines. If the secrecy level of `PID2` is set to low; `level(PID2, l)`, the secret term `Need` in line 15, reveals to a public agent `PID2` as a part of the sent message. This is an illegal information flow from agent `PID1` to agent `PID2` that must not be allowed.

This illustrates one of the limitations of the static approach (discussed in chapter 6) that does not have flexibility regarding run-time changes and can not handle variables having a different security label in each interaction. So, the security level of each term needs to be known and fixed before execution of the interaction model.

To solve the problem we may either revise the annotation or change the code. If `Need` has the low secrecy level; i.e. `label(Need, l)`,then line 13 will be a permissible flow:

$$\cfrac{\cfrac{\cfrac{overloaded: l \in \Gamma}{\Gamma \vdash overloaded: l}\,Id,\ \cfrac{Need: l \in \Gamma}{\Gamma \vdash Need: l}\,Id}{\Gamma \vdash overloaded(Need): l}\,Struct,\ \cfrac{PID1: l \in \Gamma}{\Gamma \vdash PID1: uTrm\ l}\,Id}{\Gamma \vdash a(overloaded(Need), PID1): agent\ l}\,Agnt$$

The next step is sending a message to the counterpart agent (line 16):

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

in which the requirements of the overloaded agent `PID1` is sent to an underloaded high agent `PID2`, using the `readyToMigrate` message. This is a flow from low to high, which is allowed:

$$\cfrac{\Gamma \vdash this: agent\ l,\ \cfrac{\cfrac{readyToMigrate: l \in \Gamma}{\Gamma \vdash readyToMigrate: l}\,Id,\ \cfrac{Need: l \in \Gamma}{\Gamma \vdash Need: l}\,Id}{\Gamma \vdash readyToMigrate(Need): l}\,Struct,\ \cfrac{\cfrac{underloaded: l \in \Gamma}{\Gamma \vdash underloaded: l}\,Id,\ \cfrac{\cfrac{PID2: h \in \Gamma}{\Gamma \vdash PID2: uTrm\ h}\,Id,\ uTrm\ h \leq uTrm\ l}{\Gamma \vdash PID2: uTrm\ l}\,Sub}{\Gamma \vdash a(underloaded, PID2): agent\ l}\,Agnt}{\Gamma \vdash readyToMigrate(Need) \Rightarrow a(underloaded, PID2): op\ l}\,Snd$$

`Label(idle, l).`	`label(capacity, h).`
`label(peerID, l).`	`label(need, l).`
`label(pid, l).`	`label(ok, l).`
`label(overloaded, l).`	`label(notOk, l).`

`label(pid2, h).`	`label(migrateTo, l).`
`label(underloaded, l).`	`label(migration, l).`
`label(readyToMigrate, l).`	`label(isMigrationPossible, l).`

Fig.7-8: Security label assignment to the LCC terms as annotations of the LCC code for the dynamic security type check

To have a clearer overview of the steps in the static and dynamic security checks, Table 7-1 shows an example that the dynamic checking fails to detect an illegal information flow, while the static checking prevents it (Table 7-2).

Table 7-1: An example of the dynamic type checking using Fig.7-8 annotations that ignores an illegal implicit information flow in the live virtual machine migration interaction model

The LCC code	The Action in the LCC Interpreter	Security Type Rules	Resul
a(underloaded(capacity),pid2):: ready <= a(overload,pid1) then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity,need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	typeChk(**a(underloaded(capacity),pid2)::**, Δ)	$\dfrac{\dfrac{\dfrac{\dfrac{underloaded: l,\ l \le h}{\Gamma \vdash underloaded: h}Sub,\quad \Gamma \vdash capacity: h}{\Gamma \vdash underloaded(capacity): h}Struct, \Gamma \vdash pid2: uTrm\ h}{\Gamma \vdash a(underloaded(capacity), pid2): agent\ h}Agnt}{\Gamma, this: agent\ h \vdash a(underloaded(capacity), pid2): agent\ h}Init$	OK
a(underloaded(capacity),pid2):: **ready <= a(overload,pid1)** then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity,need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	if (ready <= a(overload,pid1))∈ M_i∧typeChk(ready <= a(overload,pid1),Δ)	$\dfrac{\dfrac{\Gamma \vdash this: agent\ h, agent\ h \le\ agent\ l}{\Gamma \vdash this: agent\ l}Sub,\quad \dfrac{\Gamma \vdash ready: l,\ \Gamma \vdash\ need: l}{\Gamma \vdash ready(need): l}Struct,\quad \dfrac{\Gamma \vdash overloaded: l, \Gamma \vdash pid1: uTrm\ l}{\Gamma \vdash a(overloaded, pid1): agent\ l}Agnt}{\Gamma \vdash ready(need) <= a(overload, pid1): op\ l}Rsv$	OK
a(underloaded(capacity),pid2):: ready <= a(overload,pid1) then (**migrate(ok) =>a(overload, pid1)** **<- migratPossible(capacity,need)** then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	If ¬closed(migrate(notOk)=>a(overload,pid1)) satisfy(migratPossible(capacity,need)) **returns FALSE**	--	OK
a(underloaded(capacity),pid2):: ready <= a(overload,pid1) then (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity,need) then null <- wait()) or (**migrate(notOk)=>a(overload,pid1)**)	If ¬closed(migrate(ok)=>a(overload, pid1) <- migratPossible(…)then null<-wait()) ∧ typeChk(migrate(notOk)=>a(overload,pid1), Δ)	$\dfrac{\dfrac{\Gamma \vdash this: agent\ h, agent\ h \le\ agent\ l}{\Gamma \vdash this: agent\ l}Sub,\quad \dfrac{\Gamma \vdash overload: l,\ \ \Gamma \vdash pid1: utrm\ l}{\Gamma \vdash a(overload, pid1):\ agent\ l}Agnt,\quad \dfrac{\Gamma \vdash migrate: l,\ \ \Gamma \vdash notOk: l}{\Gamma \vdash migrate(notOk):\ l}Struct}{\Gamma \vdash migrate(notOk)\ => a(overload, pid1): op\ l}Snd$	OK
a(underloaded(capacity),pid2):: ready <= a(overload,pid1) **then** (migrate(ok) =>a(overload, pid1) <- migratPossible(capacity,need) then null <- wait()) or (migrate(notOk)=>a(overload,pid1))	if closed(ready <= a(overload,pid1))∧ typeChk(ready <= a(overload,pid1) then migrate(notOk)=>a(overload,pid1), Δ)	$\dfrac{\Gamma \vdash ready(need) \le a(overload, pid1): op\ l,\quad \Gamma \vdash migrate(notOk)\ => a(overload, pid1): op\ l}{\Gamma \vdash ready <= a(overload, pid1) then\ migrate(notOk)\ => a(overload, pid1): op\ l}Seq$	OK

`a(underloaded(capacity),pid2)::`
`ready <= a(overload,pid1) then`
` ( migrate(ok) =>a(overload, pid1)`
`<- migratPossible(capacity,need)`
`  then  null <- wait()`
` ) or (`
`  migrate(notOk)=>a(overload,pid1)`
` )`
	typeChk(`a(underloaded(capacity),pid2):: Def`, Δ)	$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{underload: l,\ l \le h}{\Gamma \vdash underload: h}Sub,\quad \Gamma \vdash capacity: h}{\Gamma \vdash underload(capacity): h}Struct, \Gamma \vdash pid2: uTrm\ h}{\Gamma \vdash a(underload(capacity),pid2): agent\ h \le agent\ l}Agnt,}{\Gamma \vdash a(underload(capacity),pid2): agent\ l}Sub \quad \Gamma \vdash ready <= a(overload,pid1)\ then\ migrate(notOk) => a(overload,pid1): op\ l}{\Gamma \vdash a(R,ID) :: Def :\ op\ l}Role$	OK

One possible implicit information leakage is in line 27 of the interaction model (Fig.7-3), in which a low message is sent to a low agent (`migrate(ok)=>a(overload,pid1)`), based on a high constraint (`migratPossible(Capacity,Need)`). Here, Some information about available resources of the physical machine 2 (`pid2`) may be revealed to an adversary (`pid1`). Another similar implicit leakage based on the same high constraint is in line 29: `migrate(notOk)=>a(overload,pid1)`.

The following static security type checking detects and prevents the illegal implicit information flows (Table 7-2). Then an alarm is raised that determines the corresponding line of the code.

Table 7-2: The static type checking (using Fig.7-7 annotations) detects and prevents implicit information flows in the live virtual machine migration interaction model

The LCC code	Security Type Rules	Result

`a(underloaded(Capacity),PID2)::`
`ready <= a(overload,PID1) then`
` ( migrate(ok) =>a(overload, PID1)`
`<- migratPossible(Capacity,Need)`
`  then  null <- wait()`
` ) or (`
`  migrate(notOk)=>a(overload,PID1)`
` )`
	$\dfrac{\dfrac{\dfrac{\dfrac{underloaded: l,\ l \le h}{\Gamma \vdash underloaded: h}Sub,\quad \Gamma \vdash Capacity: h}{\Gamma \vdash underloaded(Capacity): h}Struct, \Gamma \vdash PID2: uTrm\ h}{\Gamma \vdash a(underloaded(Capacity),PID2): agent\ h}Agnt}{\Gamma, this: agent\ h \vdash a(underloaded(Capacity),PID2): agent\ h}Init$	OK

`a(underloaded(Capacity),PID2)::`
`ready <= a(overload,PID1) then`
`( migrate(ok) =>a(overload, PID1)`
`<- migratPossible(Capacity,Need)`
`  then  null <- wait()`
`) or (`
`migrate(notOk)=>a(overload,PID1)`
`)`
	$\dfrac{\dfrac{\dfrac{\Gamma \vdash this: agent\ h, agent\ h \le\ agent\ l}{\Gamma \vdash this: agent\ l}Sub,}{} \quad \dfrac{\Gamma \vdash ready: l,\ \Gamma \vdash\ Need: l}{\Gamma \vdash ready(Need): l}Struct,\ \dfrac{\Gamma \vdash overloaded: l, \Gamma \vdash PID1: uTrm\ l}{\Gamma \vdash a(overloaded,PID1): agent\ l}Agnt}{\Gamma \vdash ready(Need) <= a(overload,PID1): op\ l}Rsv$	OK

`a(underloaded(Capacity),PID2)::`
` ready <= a(overload,PID1) then`
`( migrate(ok) =>a(overload, PID1)`
`<- migratPossible(Capacity,Need)`
`  then  null <- wait()`
`) or (`
`migrate(notOk)=>a(overload,PID1)`
`)`
	$\dfrac{\begin{array}{c}\dfrac{\Gamma \vdash this: agent\ h, agent\ h \le agent\ l}{\Gamma \vdash this: agent\ l}Sub, \\ \dfrac{\dfrac{\Gamma \vdash overload: l,\ \ \Gamma \vdash PID1: uTrm\ l}{\Gamma \vdash a(overload, PID1): agent\ l}Agnt, \dfrac{\Gamma \vdash migrate: l,\ \ \Gamma \vdash ok: l}{\Gamma \vdash migrate(ok): l}Struct}{\Gamma \vdash migrate(ok) => a(overload, PID1): op\ l}Snd, \\ \dfrac{\Gamma \vdash Need: l, l \le h}{\Gamma \vdash Need: l}Sub, \Gamma \vdash this: agent\ h, \\ \dfrac{\Gamma \vdash migratPossible: h,\ \ \Gamma \vdash Capacity: h}{\Gamma \vdash migratPossible(Capacity, Need): con\ h}Call\end{array}}{\Gamma \vdash migrate(ok) => a(overload, PID1) < -migratPossible(Capacity, Need): op\ ?}If2$	✗

The possibility of information leakage in the scenario of virtual machine migration between datacentres (clouds) is more than migration within a (known) datacentre. This is because of the complexity of the migration workflow and the enormous number of agents representing different parties. Most of the above mentioned insecure flows in live migration are also possible in the offline migration setting. An example of revealing some information to an adversary is in Fig.7-6, Line 10:

```
readyToMigrate(V_Profile, SID_List) => a(host, PID2)
```

in which information of a migrating virtual machine on PID1 and all of the required services running on it, are passed to any potential agent on another datacentre, probably on another cloud. Either the PID2 agent may itself be an adversary or it might send this information to a malicious agent, intentionally or unintentionally.

Let us assume the following security levels for the above piece of LLC code:

`this: agent h, readyToMigrate:l,V_Profile: h, SID_List:h, host:lPID2: l,`

so the bellow static type rule detect and prevent any direct or indirect disclosure of information:

$$\dfrac{\dfrac{\dfrac{\dfrac{readyToMigrate: l \in \Gamma}{\Gamma \vdash readyToMigrate: l}Id, l \le h}{\Gamma \vdash readyToMigrate: h}Sub, \dfrac{V\_Profile: h \in \Gamma}{\Gamma \vdash V\_Profile: h}Id, \dfrac{SID\_List: h \in \Gamma}{\Gamma \vdash SID\_List: h}Id}{\Gamma \vdash readyToMigrate(V\_Profile, SID\_List): h}Struct, \quad \Gamma \vdash this: agent\ h, \dfrac{\dfrac{host: l \in \Gamma}{\Gamma \vdash host: l}Id, \dfrac{PID2: l \in \Gamma}{\Gamma \vdash PID2: uTrm\ l}Id}{\Gamma \vdash a(host, PID2): agent\ l}Agnt}{\Gamma \vdash readyToMigrate(V\_Profile, SID\_List) \Rightarrow a(host, PID2): op\ ?}Snd$$

7.4 Summary

In this chapter, we have applied the proposed security-typed LCC to a cloud computing configuration case study, in which virtual machine migration is managed. The cloud computing paradigm is developed rapidly in the last few years and virtualisation technology is the core to most cloud computing models. Using MAS is an appropriate solution to the problem of automatic cloud configuration management, especially where a globally optimal solution is not necessary or feasible. Cloud security is one of the most important challenges of development of cloud computing. In cloud computing and consequently in applications such as VM management, information leakage is a vulnerability that may cause serious privacy and confidentiality issues.

We have analysed the secrecy of LCC interaction models for live and offline VM migration and discussed information leaks. We have also shown the differences between dynamic and static security type checking in preventing insecure information flows by looking at different scenarios.

Examples of information leakage in the cloud load balancing scenario are the amount of resource shortage in the overloaded VMs or the amount of free resources in the underloaded VMs. It is shown that this information may leak explicitly or implicitly. Both the dynamic and static security type checking techniques can detect explicit information flow, while the static approach can also prevent implicit leakage.

Applying the proposed language-based security framework on the live and the offline migration scenarios resulted in rejection of inappropriate security policies and approval of secure policies in the VM configuration management.

Chapter 8

Conclusions

Open MASs are growing in popularity in the Multi-agent Systems community, while there still remain many potential gaps in their security. The limitations in protecting open MASs, in particular minimal information on the identity and behaviour of agents, make them particularly difficult to protect, but their openness attracts new applications, making new problems emerge. These vulnerabilities have many effects rendering security issues indispensible and further research necessary. Information leakage is an important vulnerability in open MAS that threatens confidentiality of information.

A large body of work on information flow analysis has been developed in the area of information security, but the multi-agent community has not utilised it much. This research, therefore, represents an important step toward filling this security gap and equipping open MAS with techniques for preventing and diagnosing attacks.

This chapter summarises the contributions of this work and ends with some potential future directions.

8.1 Summary

This thesis investigates security issues of open MASs governed by electronic institutions through finding and categorising potential attacks and security countermeasures and proposing formal frameworks for information flow analysis.

This thesis proposes a taxonomy of attacks against open MASs, which can help understand the implications of attacks and their countermeasures. It also describes the various attacks possible in open MAS, some of which entail new concepts that cannot be found in conventional computer networks (e.g. ontology attack). Then, the methods to secure open MAS against the defined attacks are reviewed and classified.

Having focused on information leakage vulnerabilities in choreography systems using LCC, this thesis suggests two approaches to detect insecure information flows, *conceptual modeling* of interaction models and *language-based information* flow analysis. In *conceptual modeling*, interaction models are converted into another formalisation, then an existing tool analyses the new formalism to find information leaks.

The main advantages of conceptual modelling over language-based approach are the independence of modeling approach from the LCC implementation and its flexibility in using existing secrecy analysis tools. This modelling approach is not depend on the implementation of the LCC interpreter (i.e. in Prolog or Java), as the model of the interactions are analysed not the LCC codes. Existing automated reasoning tools can be used to prevent information leakage, however we employed Counterdog.

This thesis shows that the language-based security approach using static and dynamic type systems is applicable to open MAS. The static and dynamic analysis of LCC interaction models can detect insecurity and reveal where in the code it might arise. This thesis also introduces a prototype LCC interpreter that implements dynamic information flow analysis to protect confidentiality of a MAS.

In the proposed frameworks, the confidentiality policy is specified by annotations of the LCC interaction models. Integration of the security policy descriptions into the LCC interaction models potentially enforces a richer set of security policies than conventional access control mechanisms. However, the annotations are separate from the main LCC code and can be in a different file. The advantage of this separation is that the security policy can be decided upon either by the designer, the implementer (who creates and shares the interaction model) or the customer (who downloads and uses the interaction model).

Intuitively, a program is secure if all program runs look identical to an adversary. This well-studied notion of security is known as non-interference that guarantees absence of any information flow between secret and public systems activities. In this thesis, non-interference is extended to be applicable to LCC interaction models. The contribution is a formal definition of non-interference in terms of the LCC clause expansion mechanism that guarantees high-security input to the program never affects low-security output. The proposed security-typed LCC provides end-to-end confidentiality in open MAS. It means that it guarantees no insecure information flow through the agents' interaction based on the defined security policy. The proposed security type system is then formally evaluated by proving *progress* and *preservation* properties to guarantee *type soundness*.

The proposed security framework can be used by the designer, programmer or user of LCC interaction models without the need of any security expertise. Their main role would be defining the policy upon the detection of any leakage and specifying public and secret data, i.e. annotation of the LCC interaction model by identifying labels for LCC terms. Terms that have not been assigned any label have the default security label, which could also be defined by the user. However, inaccurate annotation may yield false positive results and wrong annotation may produce false negative outcomes.

The main limitation of the language-based approach is its dependency on the specific language i.e. LCC which makes hard to apply on other MAS platforms. However, adaptation of the proposed security type system for similar first-class agent protocol languages (such as MAP and RASA) is straightforward.

The proposed security frameworks focus on confidentiality to prevent information leakage, so they do not address security problems related to data integrity. Another practical limitation of both *language-based* and *conceptual modeling* approaches is difficulty to formalise intuitive policies using annotations; Realistic policies are complex and involve many stakeholders which makes hard to define conditions under which declassification may occur.

This thesis also applies the information flow analysis framework to cloud management scenarios to make the type-based security approach more comprehensible. Two scenarios of virtual machine migration management using the LCC interaction models are introduced, in which insecure information flows are analysed.

8.2 Future Work

The work in this thesis has yielded some insight into the security issues and solutions in open MASs, but there is still much room for improvement. The following are some suggestions for research:

- **Study of other Languages**

We focused on LCC as the language for designing electronic institutions in a MAS. This work can be extended to other languages such as Multi-Agent Protocol language (MAP) and RASA. MAP (Walton 2004) is a variant of LCC developed on the basis of distributed dialogue protocols and RASA (Miller and McBurney 2007) is an extended version of LCC, in which the relationship between the protocol specification language and the underlying constraint language is taken into consideration. We would suggest applying information flow analysis techniques to these languages.

- **Extending the Interaction Modelling Approach**

The abstraction module in the conceptual representation for other information leakage analysis can be adapted to be more general. We suggest the conversion of LCC code to a process calculus, i.e. pi-calculus, because information flow analysis of process calculus is a well-studied area and there are plenty of frameworks for this purpose.

Another direction for future research is the improvement of the proposed LCC-typed language:

- **Constraint Type Checking**

One possible extension of information flow analysis in LCC is combining the proposed security analysis of interaction models with security type checking of LCC constraints. Different tasks (e.g. the virtual machine migration policy in the cloud management case study) may be defined either in the LCC code or within the constraints, or in the combination of both. The choice here depends on whichever is most appropriate in each individual case. However in this study the focus of our security analysis is on LCC interaction models, not on the constraints, which usually are implemented in another programming language (e.g. Java in the OpenKnowledge system). To have thorough information flow analysis in LCC code and within constraints, our security analysis could be merged with an existing security Type system for Java components, e.g. Jif (Myers, et al. 1997).

- **Other Information Flow Properties**

Non-interference is too restrictive for many applications. It is also too coarse for some applications. Other information security properties such as *opacity* (Bryans, Koutny, et al., Opacity Generalised to Transition Systems 2008) can be investigated. *Opacity* (or *non-detectability*) is an information flow property that shows the inability of an adversary to infer the truth of a specific piece of information from the interactions.

This thesis has focused on one aspect of security, i.e. confidentiality. The other important aspect of security is integrity. In an information flow security approach, both confidentiality and integrity of information can be protected. In the case of integrity, instead of the notions of public and private information, we have trusted and untrusted information. In information flow analysis, integrity can be interpreted as trusted output does not depend on untrusted input. This implies that integrity is a dual to confidentiality (Hedin and Sabelfeld 2011). We would suggest defining security properties that guarantee integrity and studying agent's interactions in this regards.

- **Automatic Annotation and Variable Security Levels**

It is not easy to annotate all terms in interaction models manually, especially in larger pieces of code. Human mistakes in manual annotations also are likely to occur. It might be the case where the security levels of different clauses in an interaction model are annotated by different users; here, there is a high probability of mismatch in security labels.

Using variable security levels instead of constants is a solution to (semi)automatic annotation. In this method, the security levels are inferred from the structure of LCC terms in the form of constraints. The result of the annotation is a set of constraints showing the relationship between security levels. Regarding the automated annotation of constraints, as arguments in a constraint can be of different types, security types and levels can not be inferred from the structure of the constraint. So, the full automation of assigning types (and levels) to constraints is not possible and the user needs to be involved in the annotation process.

Flow sensitivity requires storing values of different sensitivity (low and high) over the course of interaction by the LCC terms. We leave the automatic annotation and flow-sensitivity analysis in LCC interaction models as topics for future research.

Bibliography

Abian, Joaquim, et al. “Deliverable 6.3: Bioinformatics Interaction Models.” the OpenKnowledge Project, 2008.

Aggarwal, C. C., and P. S. Yu. “Outlier Detection with Uncertain Data.” *SIAM International Conference on Data Mining (SDM).* 2008. 483-493.

Anderson, P., S. Bijani, and A. Vichos. “Multi-agent Negotiation of Virtual Machine Migration Using the Lightweight Coordination Calculus.” *Agent and Multi-Agent Systems.Technologies and Applications- 6th KES International Conference, KES-AMSTA 2012,.* Dubrovnik, Croatia, 2012. 124-133.

Anderson, P., S. Bijani, and H. Herry. “Multi-agent Virtual Machine Management Using the Lightweight Coordination Calculus.” *Transactions on Computational Collective Intelligence*, 2013.

Artikis, Alexander, Marek Sergot, and Jeremy Pitt. “Specifying norm-governed computational societies.” *ACM Trans. Comput. Logic* (ACM) 10 (2009): 1--42.

Austin, ThomasH., Cormac Flanagan, and Martín Abadi. “A Functional View of Imperative Information Flow.” In *Programming Languages and Systems*, 34-49. Springer Berlin Heidelberg, 2012.

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

Becker, Moritz Y., Alessandra Russo, and Nik Sultana. “Foundations of Trust Management.” *IEEE Symposium on Security and Privacy.* 2012.

Becker, Moritz Y., and Nik Sultana. Counterdog. Microsoft Research. 2012.

Bernstein, D., E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow. “Blueprint for the InterCloud - Protocols and Formats for Cloud Computing Interoperability.” *the Fourth International Conf on Internet and Web Applications and Services.* 2009.

Beydoun, G, G Low, H Mouratidis, and B Henderson-Sellers. “A security-aware metamodel for multi-agent systems (MAS).” *Inf. Softw. Technol.* (Butterworth-Heinemann) 51 (2009): 832--845.

Bierman, Elmarie, and Elsabe Cloete. “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, 2002. 141-148.

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

Bijani, Shahriar, and David Robertson. “A Review of Attacks and Security Approaches in Open Multi-agent Systems.” *Artificial Intelligence Review* (Springer), 2012: 1-30.

Blanchet, Bruno. “ProVerif: Cryptographic Protocol Verifier in the Formal Model.” Inria, 2001.

Borselius, Niklas, and Chris J. Mitchell. “Securing FIPA Agent Communication.” *Security and Management.* 2003.

Borselius, Niklas, and Chris Mitchell. “Securing FIPA Agent Communication.” *Security and Management*, 2003: 135–141.

Botelho, V., F. Enembreck, B.C. Avila, H. de Azevedo, and E.E. Scalabrin. “Encrypted Certified Trust in Multi-agent System.” *the 13th International Conference on Computer Supported Cooperative Work in Design.* 2009. 227-232.

Braynov, Sviatoslav, and Murtuza Jadliwala. “Detecting Malicious Groups of Agents.” *Proceedings of the 1st IEEE Symposium on Multi-Agent Security and Survivability (MAS&S) 2004.* Philadelphia, PA, USA: IEEE Computer Society, 2004. 90-99.

Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. “TROPOS: An Agent-Oriented Software Development Methodology.” *Autonomous Agents and Multi-Agent Systems* (Springer) 8 (2004): 203--236.

Bresciani, Paolo, Paolo Giorgini, Gordon Manson, and Haralambos Mouratidis. “Multi-Agent Systems and Security Requirements Analysis.” In *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004.

Bryans, Jeremy W, Maciej Koutny, Laurent Mazare, and Peter YA Ryan. “Opacity Generalised to Transition Systems.” *International Journal of Information Security*, 2008: 421-435.

Bryans, Jeremy W, Maciej Koutny, Laurent Mazare, and Peter YA Ryan. “Opacity Generalised to Transition Systems.” *International Journal of Information Security* (Springer) 7, no. 6 (2008): 421-435.

Carl, Glenn, George Kesidis, Richard R. Brooks, and Suresh Rai. “Denial-of-Service Attack-Detection Techniques.” *IEEE Internet Computing* 10, no. 1 (2006): 82-89.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. “Anomaly detection: A survey.” *ACM Computing Surveys* 41 (2009): 15:1-15:58.

Cheng, Alice, and Eric Friedman. “Sybilproof reputation mechanisms.” *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems.* Philadelphia, Pennsylvania, USA: ACM, 2005. 128--132.

Clark, K P, M Warnier, T B Quillinan, and F M Brazier. “Secure Monitoring of Service Level Agreements.” *Proceedings of the Second International Workshop on Organizational Security Aspects (OSA 2010).* IEEE,, 2010.

Cohen, Ellis. “Information Transmission in Computational Systems.” *ACM SIGOPS Operating Systems Review* 11, no. 5 (1977): 133-139.

Corin, Ricardo, P-M Denielou, Cedric Fournet, Karthikeyan Bhargavan, and James Leifer. “Secure Implementations for Typed Session Abstractions.” *20th IEEE Computer Security Foundations Symposium, 2007. CSF'07.* 2007. 170-186.

Dasgupta, D., and N. Majumdar. “Anomaly Detection in Multidimensional Data using Negative Selection Algorithm.” *the IEEE Conference on Evolutionary Computation.* Hawaii, 2002. 1039-1044.

Demazeau, Y, and A Rocha Costa. “Populations and organizations in open multi-agent systems.” *Proceedings of the I National Symposium on Parallel and Distributed AI (PDAI'96).* Hyderabad, 1996.

Denning, D. E. “A Lattice Model of Secure Information Flow.” *Communications of the ACM* 19, no. 5 (1976): 236-243.

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

Denning, Dorothy E., and Peter J. Denning. “Certification of Programs for Secure Information Flow.” *Communications of the ACM* 20, no. 7 (1977): 504-513.

Douceur, John R. “The Sybil Attack.” *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems.* Springer-Verlag, 2002. 251--260.

Dove, Rick. “On detecting and classifying aberrant behavior in unmanned autonomous systems under test and on mission.” *Live Virtual Constructive Conference, International Test and Evaluation Association.* 2009.

Dupplaw, David, Spyros Kotoulas, and Ronny Siebes. “The OpenKnowledge Kernel.” *Proceedings of the XXI Intl. Conference on Computer, Information and Systems Science.* Vienna, Austria, 2007.

E. Rescorla, M. Ray, S. Dispensa and N. Oskov. *Transport Layer Security (TLS) Renegotiation Indication Extension.* Internet Engineering Task Force (IETF), Feb. 2010.

El Ariss, Omar, and Dianxiang Xu. “Modeling Security Attacks with Statecharts.” *the joint ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium.* ACM, 2011. 123--132.

Endsuleit, Regine, and Arno Wagner. “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, 2004. 221-227.

Esteva, M, D de la Cruz, B Rosell, J L Arcos, J A RodrÃguez-Aguilar, and G CunÃ. “Engineering open multi-agent systems as electronic institutions.” AAAI Press / The MIT Press,, 2004. 1010--1011.

Esteva, M., D. de la Cruz, B. Rosell, J. Ll. Arcos, J. A. Rodriguez-Aguilar, and G. Cuni. “Engineering open multi-agent systems as electronic institutions.” *19th national conference on Artifical intelligence (AAAI 04).* AAAI Press, 2004. 1010--1011.

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

Ferraiolo, David, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based Access Controls.* Artech House Boston, 2007.

Finin, T, A Joshi, and Anupam Joshi. “Developing Secure Agent Systems Using Delegation Based Trust Management.” *In Security of Mobile MultiAgent Systems (SEMAS 02) held at Autonomous Agents and MultiAgent Systems (AAMAS}.* 2002. 200--2.

Focardi, Riccardo, Sabina Rossi, and Andrei Sabelfeld. “Bridging Language-Based and Process Calculi Security.” In *Foundations of Software Science and Computational Structures*, 299-315. Springer, 2005.

Foner, Leonard N. “A security architecture for multi-agent matchmaking.” In Proceedings of the Second International Conference on Multi-Agent Systems, pages 80-86, 1996.

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

Gorrieri, Roberto, Fabio Martinelli, and Ilaria Matteucci. “Towards Information Flow Properties for Distributed Systems.” *Electronic Notes in Theoretical Computer Science* (Elsevier) 236 (2009): 65-84.

Halpern, Bowen, and Fred B Schneider. “Recognizing Safety and Liveness.” *Distributed Computing* 2, no. 3 (1987): 117-126.

Halpern, J Y, and K R ONeill. “Secrecy in Multiagent Systems.” *ACM Trans. Inf. Syst. Secur.* (ACM) 12 (2008): 5:1--5:47.

He, Q., K. P. Sycara, and T. W. Finin. “Personal security agent: KQML-based PKI.” *the Second international Conference on Autonomous Agents.* 1998.

—. “Personal security agent: KQML-based PKI.” *the Second international Conference on Autonomous Agents.* 1998.

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

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

Heintze, Nevin, and Jon G. Riecke. “The SLam Calculus: Programming with Secrecy and Integrity.” *the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 98.* 1998. 365-377.

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

Hennigan, E, C Kerschbaumer, S Brunthaler, and M Franz. *Tracking Information Flow for Dynamically Typed Programming Languages by Instruction Set Extension.* Technical Report 11-01, Dept of Information and Computer Science, University of California Irvine, 2011.

Honda, Kohei, Vasco Vasconcelos, and Nobuko Yoshida. “Secure Information Flow as Typed Process Behaviour.” In *Programming Languages and Systems*, by Gert Smolka, 180-199. Springer Berlin Heidelberg, 2000.

Hu, Bo, Srinandan Dasmahapatra, Paul Lewis, David Dupplaw, and Nigel Shadbolt. “Facilitating Knowledge Management in Pervasive Health Care Systems.” In *Networked Knowledge-Networked Media*, 285-304. Springer, 2009.

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

Igure, V., and R. Williams. “Taxonomies of attacks and vulnerabilities in computer systems.” *Communications Surveys & Tutorials* (IEEE) 10, no. 1 (2008): 6-19.

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

—. “Mobile Agent Security.” *National Institute of Standards and Technology (NIST) Special Publication 800-19.* 2000.

—. “Mobile Agent Security.” *National Institute of Standards and Technology (NIST) Special Publication 800-19.* 2000.

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

Jurjens, Jan. “Using UMLsec and Goal Trees for Secure Systems Development.” *The 2002 ACM Symposium on Applied Computing.* Madrid, Spain: ACM, 2002. 1026--1030.

Kadota, K., D. Tominaga, Y. Akiyama, and K. Takahashi. “Detecting Outlying Samples in Microarray Data: A critical Assessment of the Effect of Outliers on Sample Classification.” *Chem-Bio Informatics* 3 (2003): 30-45.

Karnik, Neeran M., and Anand R. Tripathi. “Security in the Ajanta Mobile Agent System.” *Software - Practice and Experience*, 2001: 301-329.

Khan, Abid, Qasim Arshad, Xiamu Niu, Zhang Yong, and Muhammad Waqas Anwar. “On the Security Properties and Attacks against Mobile Agent Graph Head Sealing (MAGHS).” *the 3rd International Conference and Workshops on Advances in Information Security and Assurance (ISA 09).* Seoul, Korea: Springer-Verlag, 2009. 223-228.

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

Lippmann, R. P., and K. W. Ingols. *An Annotated Review of Past Papers on Attack Graphs.* Linoln Lab, MIT, 2005.

Liu, L., E. Yu, and J. Mylopoulos. “Analyzing Security Requirements as Relationships Among Strategic Actors.” *2nd Symposium on Requirements Engineering for Information Security (SREIS 2002).* 2002.

Loulou, M, M Tounsi, A H Kacem, M Jmaiel, and M Mosbah. “A Formal Approach to prevent Attacks on Mobile Agent Systems.” *SECUREWARE '07: Proceedings of the The International Conference on Emerging Security Information, Systems, and Technologies.* Washington, DC, USA: IEEE Computer Society, 2007. 42--47.

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

Massacci, Fabio, John Mylopoulos, and Nicola Zannone. “Security Requirements Engineering: The SI* Modeling Language and the Secure Tropos Methodology.” *Advances in Intelligent Information Systems* 265 (2010): 147-174.

McDermott, J. P. “Attack Net Penetration Testing.” *The 2000 Workshop on New Security Paradigms (NSPW'00).* Cork, Ireland, 2000. 15-21.

Microsoft. *Threat Risk Modeling.* 2010. https://www.owasp.org/index.php/Threat_Risk_Modeling.

Miller, Tim, and Peter McBurney. “Using Constraints and Process Algebra for Specification of First-class Agent Interaction Protocols.” In *Engineering Societies in the Agents World VII*, 245-264. Springer, 2007.

Mitchell, Chris. *Security for Mobility.* Institution of Electrical Engineers, 2003.

Mouratidis, Haralambos. “Secure Tropos: A Security-Oriented Extension of the Tropos methodology.” *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* (World Scientific Publishing) 17, no. 2 (2007): 285-309.

Mouratidis, Haralambos, and Paolo Giorgini. “Enhancing Secure Tropos to Effectively Deal with Security Requirements in the Development of Multiagent Systems.” (Springer-Verlag) 2009: 8--26.

Mouratidis, Haralambos, and Paolo Giorgini. “Enhancing Secure Tropos to Effectively Deal with Security Requirements in the Development of Multiagent Systems.” (Springer-Verlag) 2009: 8--26.

Mouratidis, Haralambos, and Paolo Giorgini. “Enhancing Secure Tropos to Effectively Deal with Security Requirements in the Development of Multiagent Systems.” (Springer-Verlag) 2009: 8--26.

Mouratidis, Haralambos, and Paolo Giorgini. “Enhancing Secure Tropos to Effectively Deal with Security Requirements in the Development of Multiagent Systems.” (Springer-Verlag) 2009: 8--26.

Mouratidis, Haralambos, Paolo Giorgini, and Gordon Manson. “Modelling secure multiagent systems.” *AAMAS 03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems.* New York, NY, USA: ACM, 2003. 859--866.

Mouratidis, Haralambos, Paolo Giorgini, and Michael Weiss. “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).* 2003.

Myers, A, N Nystrom, S Zdancewic, and L Zheng. “Jif: Java Information Flow.” 1997.

Necula, George C. *Proof-Carrying Code. Design and Implementation.* Springer, 2002.

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

Novak, P, M Rollo, J Hodik, and T Vlcek. “Communication Security in Multi-Agent Systems.” *The 3rd Central and Eastern European Conference on Multi-agent Systems (CEEMAS 03).* Springer-Verlag, 2003. 454-463.

Novak, P, M Rollo, J Hodik, and T Vlcek. “Communication Security in Multi-Agent Systems.” *The 3rd Central and Eastern European Conference on Multi-agent Systems (CEEMAS'03).* Springer-Verlag, 2003. 454-463.

Odubiyi, J B, and Abdur R Choudhary. “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, 2007. 49-55.

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

Page, John P, Arkady B Zaslavsky, and Maria T Indrawan. “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, 2005. 59-75.

Park, Haeryong, Haksoo Ju, Kilsoo Chun, Jaeil Lee, Seungho Ahn, and Bongnam Noh. “The Algorithm to Enhance the Security of Multi-Agent in Distributed Computing Environment.” *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems.* Washington, DC, USA: IEEE Computer Society, 2006. 55--60.

Paruchuri, Praveen, Jonathan P Pearce, Janusz Marecki, Milind Tambe, Fernando Ordonez, and Sarit Kraus. “Coordinating randomized policies for increasing security of agent systems.” *Information Technology and Management* (Kluwer Academic Publishers) 10 (2009): 67--79.

Paruchuri, Praveen, Milind Tambe, Fernando Ordonez, and Sarit Kraus. “Security in multiagent systems by policy randomization.” *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS 06).* Hakodate, Japan: ACM, 2006. 273--280.

Petrie, Charles, and Christoph Bussler. “Service Agents and Virtual Enterprises: A Survey.” *IEEE Internet Computing* (IEEE Computer Society) 7 (2003): 68-78.

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

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

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

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

Ray, M. *Authentication Gap in TLS Renegotiation.* http://extendedsubset.com/?p=8, 2009.

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

Ristenpart, T., E. Tromer, H. Shacham, and S. Savage. “Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds.” *the 16th ACM Conference on Computer and Communications Security.* ACM, 2009. 199-212.

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

Robertson, David, et al. “Models of Interaction as a Grounding for Peer to Peer Knowledge Sharing.” *Advances in Web Semantics I* (Springer-Verlag) 4891 (2009): 81--129.

—. “Open Knowledge - Coordinating Knowledge Sharing through Peer-to-Peer Interaction.” *Languages, Methodologies and Development Tools for Multi-Agent Systems. First InternationalWorkshop, LADS 2007. Revised Selected and Invited Papers.* 2008. 1-18.

Robles, Sergi. *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, 2008.

Rojas, Diana M., and Ahmed M. Mahdy. “Integrating Threat Modeling in Secure Agent-Oriented Software Development.” *International Journal of Software Engineering (IJSE)* 2 (2011): 23 - 36.

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

Sabelfeld, A., and A. C. Myers. “Language-based Information-flow Security.” *IEEE Journal on Selected Areas in Communications* 21, no. 1 (2003): 5-19.

Sabelfeld, A., and A.C. Myers. “Language-Based Information-Flow Security.” *IEEE Journal on Selected Areas in Communications* 21, no. 1 (2003): 5-19.

Sabelfeld, Andrei, and Alejandro Russo. “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, 2010.

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

Schneier, B. “Attack Trees.” *Dr. Dobb's Journal of Software Tools* 24 (1999): 21-29.

Schneier, C. Ellison and B. “Ten risks of PKI: What you're not being told about Public Key Infrastructure.” (Computer Security Journal,) 16(1):1-7 (2000).

Sierra, C, et al. “Report on Bioinformatics Case Studies.” techreport, 2008.

Silei, Lei, Zhang Rui, Liu Jun, and Xiao Junmo. “A Novel Security Protocol to Protect Mobile Agent against Colluded Truncation Attack by Cooperation.” *International Conference on Cyberworlds* (IEEE Computer Society), 2008: 186-191.

Sit, Emil, and Robert Morris. “Security Considerations for Peer-to-Peer Distributed Hash Tables.” *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems.* Springer-Verlag, 2002. 261--269.

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

Sun, Bo, and Hua Chen. “Communication Security in MAS with XML Security Specifications.” *Applied Mechanics and Materials*, 2011: 251-254.

Sycara, Katia, Massimo Paolucci, Martin Van Velsen, and Joseph Giampapa. “The RETSINA MAS Infrastructure.” *Autonomous Agents and Multi-Agent Systems* (Kluwer Academic Publishers) 7 (2003): 29--48.

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

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

Tekbacak, Fatih, Tugkan Tuglular, and Oguz Dikenelli. “An Architecture for Verification of Access Control Policies with Multi Agent System Ontologies.” *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference.* IEEE Computer Society, 2009. 52--55.

—. “Policies for Role based Agents in Environments with Changing Ontologies.” *The 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 11).* Taipei, Taiwan, 2011. 1335-1336.

Thirunavukkarasu, C, T Finin, and J Mayfield. “Secret Agents - A Security Architecture for the KQML Agent Communication Language.” *Intelligent Information Agents Workshop (CIKM'95)*, 1995.

Traynor, Patrick, Patrick McDaniel, and Thomas L Porta. *Security for Telecommunications Networks: Future Directions and Challenges.* Springer US, 2008.

van 't Noordende, G. , F. M. T.   Brazier, and A. S. Tanenbaum. “Security in a Mobile Agent System.” *the First IEEE Symposium on Multi-Agent Security and Survivability.* 2004. 35-45.

Van't Noordende, Guido J, Benno J Overeinder, Reinier J Timmer, and Frances MT Brazier. “Constructing Secure Mobile Agent Systems using the Agent Operating System.” *International Journal of Intelligent Information and Database Systems (IJIIDS)* 3 (2009): 363-381.

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

Vila, X., A. Schuster, and A. Riera. “Security for a Multi-Agent System based on JADE.” 2007.

Vitabile, Salvatore, Vincenzo Conti, Carmelo Militello, and Filippo Sorbello. “An extended JADE-S based framework for developing secure Multi-Agent Systems.” *Computer Standards and Interfaces* 31 (2008): 913-930.

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

Wagner, Gerd. “Multi-Level Security in Multiagent Systems.” *In Proceedings of the First international Workshop on Cooperative information Agents.* London, UK: Springer-Verlag, 1997. 272--285.

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

Walton, Christopher. “Multi-agent Dialogue Protocols.” *the 8th International Symposium on Artificial Intelligence and Mathematics.* 2004.

Wang, Hongzue, Vijay Varadharajan, and Yan Zhang. “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, 1999. 174--185.

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

Wu, R., G.J. Ahn, H. Hu, and M. Singhal. “Information Flow Control In Cloud Computing.” *6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom).* IEEE, 2010. 1-7.

Xiao, Liang. “An adaptive security model using agent-oriented MDA.” *Information and Software Technology* (Butterworth-Heinemann) 51 (2009): 933--955.

Xiao, Liang, Paul Lewis, and Srinandan Dasmahapatra. “Secure Interaction Models for the HealthAgents System.” In *Computer Safety, Reliability, and Security*, 168-180. Springer, 2008.

Yu, Eric, and Luiz M Cysneiros. “Designing for Privacy and Other Competing Requirements.” *2nd Symposium on Requirements Engineering for Information Security (SREISâ€™02).* Raleigh, North Carolina , 2002.

Yue, Xiaowen, Xiaofeng Qiu, Yang Ji, and Chunhong Zhang. “P2P attack taxonomy and relationship analysis.” *ICACT'09: Proceedings of the 11th international conference on Advanced Communication Technology.* IEEE Press, 2009. 1207--1210.

Zaslavsky, A, and M Indrawan. “A buddy model of security for mobile agent communities operating in pervasive scenarios.” *Proc. Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation.* 2004. 17--25.

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

Appendix

A) Proof of Theorem 6-1 (Progress) in page 89 (cont.):

Case *Choice*: $L = A_1 \ or \ A_2$ 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 one of them is a final step (*closed*), based on the following LCC rewriting rule in Fig 2-2:

$$closed(A \ or \ B) \leftarrow closed(A) \ \vee \ closed(B)$$

($A_1 \ or \ A_2$) is a final step. If both A_1 and A_2 are not final steps, based on the following LCC rewriting rules:

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

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

We have either ($A_1 \ or \ A_2$) $\rightsquigarrow A_1'$ or ($A_1 \ or \ A_2$) $\rightsquigarrow A_2'$.

Case *Par*: $L = A_1 \ par \ A_2$ 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 2-2:

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

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

$$A_1 \ par \ A_2 \xrightarrow{R_i, M_i, M_o, P, O} E_1 \ par \ E_2 \qquad if \ A_1 \xrightarrow{R_i, M_i, M_n, P, O_1} E_1 \ \wedge \ A_2 \xrightarrow{R_i, M_n, M_o, P, O_2} E_2,$$

$A_1 \ par \ A_2 \rightsquigarrow A_2'$. If $A_1 \rightsquigarrow A_1'$ and A_2 is a final step, $A_1 \ par \ A_2 \rightsquigarrow A_2'$.

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 above LCC rewriting rule, $A_1 \ par \ A_2 \rightsquigarrow A_1' \ par \ A_2'$.

Case *If2*: $L = C \leftarrow M \Leftarrow A$ and $L: op \ \tau$, so $M \Leftarrow A : op \ \tau$ 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 2-2:

$$C \leftarrow M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{M \Leftarrow A\}, P, \emptyset} c(C \leftarrow M \Leftarrow A) \qquad if\ (M \Leftarrow A) \in M_i \wedge satisfy(C),$$

When message M is recived from A, either the evaluation of the *satisfied*(C) is true, so $C \leftarrow M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{M \Leftarrow A\}, P, \emptyset} c(C \leftarrow M \Leftarrow A)$ or else $C \leftarrow M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{M \Leftarrow A\}, P, \emptyset} false$ (it returns *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 either 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 2-2:

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

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

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

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

Case *If4*: $L = a(R, I) \leftarrow C$ and $L: op\ \tau$, so $a(R, I)$: $agent\ \tau$ 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 2-2:

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

If $a(R, I)$ is a valid clause, in the case that the evaluation of *satisfied*(C) is true, so $a(R, I) \rightsquigarrow a(R, I) :: B$ or else $a(R, I) \leftarrow C \xrightarrow{R_i, M_i, M_o, P, \emptyset} false$ (it returns *false*).

If C is a compound constraint, progress is guaranteed as it is shown in *If2.*

Cases *If3, And, Or, Not*, *Call* and *Struct* are similar to case *If2.*

Cases *Snd* and *Rsv* are subsets of cases *If1* and *If2* respectively. □

A) Proof of Theorem 6-2 (Preservation) in page 90 (cont.):

Case *Choice*: $L = A_1\ or\ A_2$ and $L: op\ \tau$

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

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

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

the transition L⇝L′ happens either by $A_1 \xrightarrow{R_i, M_i, M_o, P, O} E$ or when A_1 is not a final step (¬closed(A_1)), by $A_2 \xrightarrow{R_i, M_i, M_o, P, O} E$.

If ¬closed(A_2), 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 $a(R, I) \leftarrow C\colon op\ \tau$, A_1⇝A_1' and $A_1'\colon op\ \tau$. 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, $(A_1\ or\ A_2)$ ⇝ $a(R, I) ::$ B $or\ A_2$ where $A_1\ or\ A_2$: op τ.

2) Subcase $A_1 = M \Rightarrow A$

By the induction hypothesis, we have $M \Rightarrow A\colon op\ \tau$, A_1⇝A_1' and $A_1'\colon op\ \tau$. 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, $(A_1\ or\ A_2)$ ⇝ $(M \Rightarrow A\ or\ A_2)$ where $A_1\ or\ A_2$: op τ.

3) Subcase $A_1 = M \Leftarrow A$

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

Other subcases are similar.

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 one of them is a final step (*closed*), based on the following LCC rewriting rule in Fig 2-2:

$$closed(A\ or\ B) \leftarrow closed(A)\ \vee\ closed(B)$$

(A_1 or A_2) is a final step. If both A_1 and A_2 are not final steps, based on the following LCC rewriting rules:

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

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

We have either (A_1 or A_2) $\rightsquigarrow A_1'$ or (A_1 or A_2) $\rightsquigarrow A_2'$.

As we showed in the discussed cases, transition of L $\rightsquigarrow$L' preserve the type. Other typing rules are similar due to the fact that no rule allows the inference of any un-typed LCC expression.