Universitat Autònoma de Barcelona

Departament d'Enginyeria de la Informació i de les Comunicacions

CONTRIBUTIONS TO MOBILE AGENT PROTECTION FROM MALICIOUS HOSTS

SUBMITTED TO UNIVERSITAT AUTÒNOMA DE BARCELONA
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

by Carles Garrigues Olivella

Bellaterra, June 2008

Adviser:

Dr. Sergi Robles Martínez

© Copyright 2008 by Carles Garrigues Olivella

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bellaterra, June 2008

Dr. Sergi Robles Martínez
(Adviser)

Committee:

Dr. Josep Rifà - Universitat Autònoma de Barcelona
Dr. Nikos Migas - Napier University
Dr. Juan Ramón Velasco - Universidad de Alcalá
Dr. Antorio Moreno - Universitat Rovira i Virgili
Dr. Jordi Sabater - Institut d'Investigació en Intel·ligència Artificial (IIIA)

A la meva Laura

Abstract

Several advantages have been identified in using mobile agents in distributed systems. The most frequently cited advantages include: reduction of network load, decrease in communication latency, dynamic adaptation, and better support for mobile devices with intermittent connections, among others. However, the benefits offered by mobile agents have not been sufficient to stimulate their widespread deployment. The main reason why mobile agents have not been widely adopted yet, despite their technological benefits, is their inherent security risks. Many breakthroughs have been achieved in the security, reliability and efficiency of mobile agents, but there are security issues still remaining unsolved.

The core work of this thesis revolves around the protection of mobile agents against malicious hosts. In order to provide a solution to some of the current security issues, first of all, an itinerary protection protocol is presented that supports free-roaming agents. Itinerary protection protocols proposed to date limit the agent's ability to migrate at will. Therefore, this thesis presents a protocol that allows agents to discover new platforms at runtime, so that applications can take full advantage of the benefits provided by mobile agent itineraries.

Second, a protocol is presented that protects mobile agents against external replay attacks. External replay attacks are based on resending the agent to another platform, so as to force the reexecution of part of its itinerary. The proposed protocol counters this kind of attacks without limiting the agent's ability to visit certain platforms repeatedly.

The security solutions presented in this thesis are based on the fact that trusted platforms can be found in many, if not most, mobile agent-based scenarios. By incorporating

trusted platforms in the agent's itinerary, the proposed solutions provide a balanced trade-off between security and flexibility.

In order to promote the development of secure mobile agent-based applications, this thesis also presents a development environment that facilitates the implementation of the proposed agent protection protocols as well as other security solutions.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Sergi Robles, for the encouragement, sound advice and lots of good ideas he provided me throughout the duration of my PhD studies. None of this would have been possible without him.

I would also like to thank all the members of the Department of Information and Communications Engineering for providing a friendly environment in which to work. In particular, I would like to thank Joan Borrell for spreading his spirit and enthusiasm onto others, and for his helpful advice and support. Besides, a note of thanks is due to those with whom I shared so many enjoyable lunches and stimulating discussions. I would like to say a big 'thank you' to Jordi Cucurull, who had the misfortune to be my office mate during the last five months of thesis preparation, and endured my whining during all this time.

I would also like to express my gratitude to Dr. William Buchanan, who accepted me into his research group at Napier University, and to Dr. Nikos Migas, who offered me help, advice and discussion, and who deserves much of the credit for the research paper that was written during my stay at Napier.

I am also grateful to all my friends and family, and I especially wish to thank my parents for their continued support and faith in me.

Lastly, and most importantly, I would like to thank my wife Laura for helping me get through the difficult times, and for all the emotional support, entertainment, and care. This thesis is dedicated to her.

Contents

Abstract **vii**

Acknowledgements **ix**

1 Introduction **1**

1.1 Objectives 5

1.2 Contributions 6

1.3 Thesis structure 7

2 Mobile Agent Security **9**

2.1 Overview 9

2.2 Malicious host problem: unrealistic approaches 11

2.2.1 Execution tracing 11

2.2.2 Cooperating agents 12

2.2.3 Obfuscation 12

2.2.4 Computing with encrypted functions 13

2.2.5 Tamper-proof devices 13

2.3 Malicious host problem: realistic approaches 14

2.3.1 Protecting the agent computational results 14

2.3.2 Protecting the agent's itinerary 16

2.3.3 Self-protected mobile agents 18

2.4 Conclusions 21

3 **Defining the agent's itinerary** 23
3.1 Related work on defining agent's itinerary 24
3.2 Explicit itineraries for free-roaming agents 25
3.3 Node special properties 30
3.3.1 Unchanged location property 30
3.3.2 Dynamic location property 31
3.4 Implementation issues 33
3.5 Conclusions 35

4 **Securing dynamic itineraries** 37
4.1 Assumptions made with regard to trusted platforms 38
4.2 Notation 39
4.3 The protocol 40
4.3.1 Building the protected itinerary 42
4.3.2 Management of the protected itinerary 47
4.3.3 Discussion 54
4.4 Implementation 56
4.4.1 Simulation and tests 58
4.5 Conclusions 62

5 **External replay attack protection** 63
5.1 Related work on agent replay attacks 64
5.2 Replay attack protection 67
5.2.1 Requirements of the solution 69
5.2.2 The protocol 69
5.2.3 Discussion 79
5.3 Implementation 80
5.3.1 Simulation and tests 83
5.4 Conclusions 87

6 Promoting the development of secure mobile agents **89**
6.1 Related work on mobile agent software engineering 90
6.1.1 Mobile agent platforms 90
6.1.2 Simplifying agent development 90
6.2 Development environment 92
6.3 MACPL 96
6.3.1 MACPL types 98
6.3.2 Scope of variables 101
6.3.3 Built-in functions 102
6.3.4 Function libraries 105
6.4 Auxiliary Tools 106
6.4.1 Itinerary Designing Tool 106
6.4.2 Agent Launcher 107
6.5 Conclusions 108

7 Conclusions **111**
7.1 Future work 113

A MACPL language specification **117**
A.1 Data Types 117
A.2 Variables 119
A.3 Operators 119
A.4 Functions 120
A.5 Built-in functions 120
A.6 Keywords 126
A.7 Precompilation directives 127
A.8 Comments 127

B MACPL implementation example **129**

List of Tables

4.1 Execution times (ms) for agents with protected and unprotected itineraries . 61
5.1 Execution times (ms) for agents protected and unprotected against replay attacks . . . 85
5.2 Execution times (ms) for agents protected and unprotected against replay attacks with dynamically located nodes . . . 86

List of Figures

3.1 Example itinerary with three stages or nodes. 25
3.2 Example itinerary with an *if* node. 26
3.3 Example itinerary with a *switch* node. 27
3.4 Example itinerary with a *set* node. 28
3.5 Example itinerary with a *loop* node. 28
3.6 Example itinerary with a *discoverer* node and a dynamically located node. . 29
3.7 Example itinerary with the *unchanged location* property set on a node. . . . 31
3.8 Use of the *unchanged location* property on a *loop* node. 31
3.9 Itinerary of a shopping agent that includes two *discoverer* nodes and two dynamically located nodes 32
3.10 Itinerary of a completely free-roaming agent 33
3.11 Example itinerary with two dynamically located nodes associated with the same *discoverer* node. 33
4.1 Representation of a chain of digital envelopes. 41
4.2 Modified chain of digital envelopes to support dynamic itineraries. 42
4.3 Example itinerary with a *loop*, a *discoverer* and a dynamically located node. 45
4.4 Reconstruction of the itinerary of figure 3.9 when two platforms are discovered for node 2 56
4.5 Itinerary of an agent implementing a hotel reservation system 59
5.1 Simple itinerary where the same platform is visited twice 68
5.2 Itinerary containing a loop with three platforms that are visited repeatedly . 68

5.3 Authorisation entities (AE) and authorisation nodes (AN) assigned to the nodes of a complex itinerary 71
5.4 Components of an agent protected against replay attacks 74
5.5 Algorithm for checking trip markers in *sequence* nodes 75
5.6 Algorithm for trip marker handling in *loop* nodes 77
5.7 Components of an agent protected against replay attacks with an agent-driven implementation 81
5.8 Itinerary of the car purchasing service agent 84
5.9 Itinerary of the car purchasing service agent with dynamically located car dealers 86
6.1 Components of the Agent Builder with its main inputs and outputs 94
6.2 Overview of the mobile agent development environment 96
6.3 Itinerary Designing Tool 107

Chapter 1

Introduction

A mobile agent is a software that can move autonomously from one computer to another while executing [Whi94]. The migration of the whole running process, along with its state, code and resources is what makes mobile agents different from other kinds of distributed applications.

Several advantages have been identified in using mobile agents in distributed systems [LO99]. The most frequently cited advantages include: reduction of network load, by moving agents to the data servers instead of transferring large amounts of data through the network; decrease in communication latency, by interacting locally with the resources available at the remote servers; dynamic adaptation, for agents can react autonomously to the changes in their execution environment; and better support for mobile devices with intermittent connections, for mobile agents can operate asynchronously without requiring a continuously open connection, among others.

Numerous applications have been developed that demonstrate the benefits of mobile agent technology. One of the most promising application areas for mobile agents is e-commerce [GMM98]. In these applications, mobile agents can be exploited to compare and gather information on prices of goods, negotiate on behalf of their owners, select the best product according to the owners' preferences, and then proceed to the purchase.

Another application domain that exploits the advantages of mobile agents is the one

related to *sea-of-data* applications [Rob02]. In these applications, massive amounts of data are distributed among several servers, and these data cannot be sent across the network due to bandwidth constraints, legal restrictions, or other limitations. The use of mobile agents in these applications allows data to be processed locally, using data mining techniques, machine learning algorithms, or other methods. An example of this kind of applications can be found in [VMRC+06], where an agent-based information gathering system for healthcare institutions is presented. In this case, mobile agents are used to search for patient medical records that are spread across different institutions.

However, the benefits offered by mobile agents have not been sufficient to stimulate their widespread deployment. The advantages of mobile agents were initially overstated, and some authors referred to mobile agents as a unifying solution that could be used to implement any distributed application. This initial hype soon contrasted with the severe security threats that researchers associated with the use of mobile agent technology. These misunderstandings and concerns resulted in the development of few commercial mobile agent systems and fewer standards. Consequently, most distributed applications are developed nowadays using other paradigms that provide fewer advantages but also raise fewer issues.

Despite this daunting reality, new breakthroughs in the security, reliability and efficiency of this technology have been made [ARO04, ZOL04, BCG07]. Besides, a constant trickle of publications in international journals and conferences demonstrates that the topic has not been abandoned by the research community. Several studies have shown that mobile agents provide an intuitive and appealing abstraction that simplifies the design and implementation of many distributed applications [Mil99]. Furthermore, while most applications realised using mobile agents can be equally implemented using traditional approaches, there is no single alternative to all of the functionality supported by the mobile agent technology [HCK97]. Therefore, we believe that mobile agents can still play an important role in the future of distributed computing.

However, some of the security concerns raised by mobile agent technology still linger, and they represent one of the main obstacles currently hindering the wider acceptance of

mobile agents [Bor02]. In general, mobile agent security can be divided into two broad areas: host security, which implies protecting the host platform from a malicious agent; and agent security, which implies protecting the agent from a malicious platform.

With regard to host security, the problems related to the protection of the execution environment have significantly been mitigated. The most suitable proposed techniques include: using safe code interpretation [LOW97], where the set of available instructions prevents the agent from attacking the platform; using *Software-based Fault Isolation* [WLAG93], which is also known as *sandboxing*, and is based on limiting program accessibility to a closed domain, among others.

As for agent security, agents also need to be protected against tampering by the hosts they visit. This problem is clearly much harder than the previous one [GKCR98]. The platform that executes the agent must have access to its code and resources. Therefore, a malicious platform can easily examine and divert the intended execution of the agent, and any attempt to detect a wrong execution or tampering of data is subject to diversion, too.

The techniques proposed so far to provide a complete protection against malicious hosts have proved to be impractical, for they cannot be effectively implemented in real-life applications. Some of the better known approaches are the use of tamper-proof hardware [WSB98] or encrypted functions [ST98], but they have serious drawbacks. Regarding the first approach, the problem stems from the fact that it is extremely unlikely that tamper-proof hardware will be available on every platform in the near future. With regard to the second approach, the encrypted functions known to date can only be used to implement rational functions and polynomials, and are thus not suitable for general programming.

The mobile agent community has largely accepted that it is unlikely that there will ever be a complete solution to the malicious host problem [Zac03]. Because of this, recent proposals focus on providing partial solutions aimed at preventing a subset of the attacks that can be mounted against an agent. The most important proposals in this area are based on the protection of the agent's initial itinerary and its computational results.

Regarding the protection of the agent's computational results, satisfactory solutions have already been devised [MS03, ZOL04]. These are based on storing the agent's results

inside an append-only data structure, in such a way that only additions of new elements are allowed. Any modification or deletion of an element can be detected afterwards, and appropriate actions can be taken against the offending platform.

On the other hand, the solutions aimed at protecting the agent's itinerary are based on allowing platforms to access only their corresponding parts of the agent's code and data [MB03]. Thus, platforms cannot access or modify parts of the agent's itinerary intended for other platforms.

Although the current proposals for itinerary protection are technically valid, their utilisation involves an important loss of flexibility. First of all, itinerary protection protocols presented to date force programmers to define *static* itineraries, in which all platforms must be known in advance, and thus agents are not allowed to discover and visit new platforms at runtime. Secondly, some proposals also protect agents from external replay attacks [Yee03], in which an agent is forced to perform unintended migrations to the same platform, but these proposals preclude agents from visiting a certain platform several times. Thus, they limit the programmer's ability to define itineraries that contain round-trips.

The loss of flexibility introduced by current protocols is due to the fact that they always assume the worst possible scenario, in which every platform in the agent's itinerary is potentially malicious. This assumption, however, is hardly realistic [Rot99].

Let us examine the scenario of a shopping agent. Consider that a mobile agent is created to search for a flight plan that meets a costumer's requirements. The costumer creates an itinerary with several airline companies and the agent is dispatched to collect offers from all of them. After visiting all the airline companies, the agent compares the collected results and commits to the best offer. In order to develop this application, two different strategies could be implemented. The first strategy could be to compare the collected offers and proceed to the purchase in the last airline company visited by the agent. The second strategy could be to do so after migrating to a trusted platform, so that the costumer could be confident that a fair comparison of offers and purchase was conducted.

Clearly, the second strategy would be the one preferred by most costumers. Relying on

a trusted platform, such as the agent's home platform, is certainly much easier than allowing the agent to make sensitive decisions on a potentially malicious platform, regardless of the itinerary protection protocol used. Consequently, this example application shows that regarding all platforms as potentially malicious does not reflect the reality of many applications.

However, the lack of flexibility of current agent protection protocols is not the only concern currently impeding the wider use of secure mobile agents. The complexity of programming applications that make use of these security solutions is also an important issue to overcome [MY06].

Therefore, this thesis pursues two main objectives: First of all, to provide a solution to the limitations of current agent protection protocols; and second, to design new tools and methodologies that simplify the development of secure mobile agents. These objectives are detailed in the following section.

1.1 Objectives

The first objective of this thesis is to provide a convenient way to define explicit itineraries for free-roaming agents. Defining explicit itineraries implies storing the itinerary information in a separate data structure. By storing and maintaining this data structure outside the main agent code, the protection of the itinerary is significantly simplified.

The second objective is to define an itinerary protection protocol that supports free-roaming agents. Itinerary protection protocols presented to date sacrifice most of the flexibility provided by mobile agent itineraries, for they limit the mobile agent's ability to migrate at will. Therefore, this thesis aims to allow agents to discover new platforms at runtime, so that applications can take full advantage of the benefits provided by the mobile agent paradigm.

The third objective is to define a protocol for the protection of mobile agents against external replay attacks. External replay attacks are based on resending the agent to another platform so as to force the reexecution of part of its itinerary. The current solutions

proposed for this problem do not allow agents to visit the same platform several times. Therefore, this thesis intends to counter this kind of attacks without limiting the agent's ability to visit certain platforms repeatedly.

Finally, the last objective is to encourage the development of secure mobile agent applications. The implementation of the security mechanisms required by mobile agents can turn out to be more time-consuming than the implementation of the agent tasks. Therefore, new tools have to be created that simplify to the greatest extent possible the tasks carried out by both the designer of security protocols and the developer of new applications.

1.2 Contributions

The main contributions of this thesis are based on the fact that trusted platforms can be found in many, if not most, mobile agent-based scenarios. By incorporating trusted platforms in the agent's itinerary, this thesis presents, first of all, an itinerary protection protocol that supports free-roaming agents, and second, a replay protection protocol that allows agents to loop over a certain number of platforms an undetermined number of times. Thus, the resulting protocols provide a balanced trade-off between security and flexibility.

In order to promote the development of secure mobile agent-based applications, this thesis also presents a development environment that facilitates the implementation of the proposed agent protection protocols as well as other security solutions.

The contributions of this thesis have given rise to some publications in international journals and conferences. The main contributions made towards the first and second objectives have been presented in [GRB08a]. The protocol proposed to accomplish the third objective has been presented in [GMB+08]. Besides, a patent has been filed and is currently pending [GRB08b]. Finally, the development environment devised to attain the fourth objective has been submitted to the Journal of Networks and Computer Applications. Nevertheless, this thesis provides an integrated perspective on these contributions.

1.3 Thesis structure

The remaining chapters of this thesis are organised as follows.

Chapter 2 presents a literature review of mobile agent security issues and most relevant techniques proposed to address these issues. The chapter places special emphasis on the techniques proposed to address the malicious host problem. The described techniques are classified according to their suitability for practical applications.

Chapter 3 provides, first of all, an overview of the related work on defining the agent's itinerary. Then, it presents a new way of defining explicit itineraries for mobile agents. Some considerations regarding the implementation of these itineraries are discussed at the end of the chapter.

Chapter 4 describes an itinerary protection protocol for free-roaming agents, which is based on introducing some trusted platforms into the agent's itinerary. The chapter also discusses what assumptions are made with regard to trusted platforms. The implementation of an example application that shows the viability of the proposed protocol is presented at the end of the chapter.

Chapter 5 describes a protocol for the protection of mobile agents against agent replay attacks. A survey of the related work on agent replay attacks is also presented. After defining the proposed protocol, the chapter presents the implementation and experimentation work conducted in order to prove the validity of the proposed protocol.

Chapter 6 outlines, first of all, the related work on mobile agent engineering, focusing mainly on the approaches proposed to date to simplify the development of mobile agents. Then, this chapter describes a development environment aimed at aiding in the development of secure mobile agents. The key element of the proposed environment is the Agent Builder and the MACPL language, which are described in detail.

Chapter 7 summarises the work of the thesis and provides some hints of future research directions.

Appendix A provides a detailed description of MACPL features. The chapter analyses its types, variables, operators, special keywords, built-in functions and precompilation

directives.

Appendix B provides an example of MACPL programming, which shows the simplicity and utility of this language.

Chapter 2

Mobile Agent Security

In this chapter, we analyse the security threats that arise in a mobile agent system, and we provide a survey of the most relevant solutions proposed so far. Security threats are first classified into four categories, depending on the source of the attack and the entity being attacked. We then focus on the types of threats that are most difficult to deal with: the threats stemming from an agent platform attacking an agent. We outline the better known techniques available to address this type of attacks, presenting them in two different categories: First, those proposing methods that cannot be effectively implemented in real-life applications. Second, those providing feasible solutions that are suitable for practical applications.

2.1 Overview

Since the beginning of mobile agent research, many security issues have been identified. In [JK00], these issues were classified according to the source of the attack and the entity being attacked: agents against agents, agents against platforms, others against platforms, and platforms against agents.

In the first category—agents against agents—we can find attacks in which agents modify or access another agent's data, disguise their identity in order to falsify a transaction,

or repeatedly send messages to another agent in order to launch a denial of service attack, among others. The second category—agents against platforms—includes threats in which agents perform some malicious action on a resource they can access to (e.g., deleting a file), consume an excessive amount of system resources, gain access to a service to which they are not entitled, and so on.

With regard to these two first categories, in which the attacker is an agent, sound solutions have already been proposed. Among the solutions that provide an acceptable level of protection, the most efficient one is called *Software-based Fault Isolation* [WLAG93]. This mechanism, also known as *sandboxing*, is based on limiting program accessibility to a closed domain, in such a way that the program address space and available resources are confined within this domain.

Other mechanisms proposed for these kinds of attacks include: using safe code interpretation [LOW97], where the set of available instructions prevents the agent from attacking the platform; signing the code in order to authenticate the agent owner, together with some mechanism to determine the level of trust of this owner [Gra95]; sending logical demonstrations along with the code, in order to proof that the execution of that code is secure (Proof Carrying Code [NL96]), among others.

Regarding the third category—others against platforms—the source of the attack can be any external entity that is not part of the agent platform. This external entity can perform attacks against the platform resources (files, communication ports, etc.) or against the platform's communications with the outside. In these cases, security greatly depends on the mechanisms provided by the operating system. Additionally, a secure communication channel, established using mechanisms such as Transport Layer Security [DR06] or IPSec [KA98], can be used to secure the communication between the platform and other parties.

The last type of attack—platforms against agents—is the most difficult to prevent. It is obvious that if a platform is to execute an agent, it must have complete access to the agent code, state and data. There is nothing to prevent the platform from analysing the agent code, from corrupting its state or data, from manipulating its execution environment, or from executing it multiple times in order to, for example, generate multiple purchases in

a shopping scenario. If some agent data is to be kept secret from the platform, it must be stored in a way that even the agent itself cannot directly access (encrypted with the key of a different platform, for instance).

Several mechanisms have been proposed to address the malicious host problem. The next subsections discuss some of the better known approaches. First of all, we present those that have a limited applicability because they can only be used if certain assumptions hold. Then, we will present those approaches that can be effectively implemented in real world applications.

2.2 Malicious host problem: unrealistic approaches

Some of the better known solutions to the malicious host problem are impractical, for they have been designed for particular scenarios that are actually rarely found in real-life applications. The following is a discussion of the better known ones.

2.2.1 Execution tracing

Execution tracing [Vig98] is a technique that allows unauthorised modifications of an agent to be detected upon completion of the agent execution. The protocol proposed in [Vig98] is based on recording the agent's behaviour on each platform in order to build a trace of its execution. The trace is composed of a sequence of identifiers corresponding to the operations executed by the agent. Platforms must produce and maintain traces of all executed agents, so that agent owners can request these traces after the agent has terminated its execution, and verify that the agent code or state has not been maliciously modified.

This approach has several drawbacks, such as the size and the number of logs to be kept by platforms, or the possible lack of connection between the owner and the platforms once the agent has returned to the home platform. Besides, the verification mechanism is too expensive to be applied systematically, and can only be used when the owner has a suspicion that the agent execution has been corrupted.

2.2.2 Cooperating agents

In [Rot99], Roth describes a protocol for detecting manipulations of the agent execution through co-operating agents. Roth considers applications to be designed using two or more mobile agents that co-operate in order to achieve their goals. The itineraries of these agents must have no single platform in common, and platforms can collude with each other as long as the collusion does not involve platforms of different agent itineraries. The itinerary and the operations performed by every agent must be tracked by their co-operating agents, so that any tampering with the execution of an agent can be detected by its co-operators. In order for this protocol to be secure, the interaction between the agents must always take place over a secure authenticated channel.

Roth's protocol suffers from several limitations, the first one being the complexity of defining subgroups of platforms that will not collaborate with each other to attack the application. The second limitation is the need to establish a secure authenticated channel between the agent and its co-operators, which may not be possible to provide in all scenarios. Besides, this technique undermines the agent's autonomy, for it requires the agent to interact with other agents in order to carry out its tasks.

2.2.3 Obfuscation

Code obfuscation [Hoh98] aims at generating executable agents which cannot be attacked by reading or manipulating their code. This technique is based on transforming the agent code in such a way that it is functionally identical to the original one, but it is impossible to understand it. The approach also establishes a time interval during which the agent and its sensitive data are valid. After this time elapses, any attempt to attack the agent becomes worthless. In [HR99], a modification of this approach is presented to prevent hosts from repeatedly executing an agent in order to obtain different outputs and draw conclusions about its behaviour. This modification is based on recording every input event on a trusted third party.

The major drawback of these techniques is the difficulty in establishing the time required by an attacker to understand an obfuscated code. Similarly, no mechanism is currently known for quantifying the amount of time required by an agent to accomplish its task, especially in heterogeneous environments. As a result, restricting the lifetime of a mobile agent is not feasible in practise.

2.2.4 Computing with encrypted functions

Computing with encrypted functions is a technique proposed by Sander and Tschudin [ST98] to achieve code privacy and code integrity. Their technique is based on creating encrypted programs that can be executed without decrypting them. Supposing that a mobile agent has to execute a certain function f, then f is encrypted to obtain $E(f)$, and a program is created that implements $E(f)$. Platforms execute $E(f)$ on a cleartext input value x, without knowing what function they actually computed. The execution yields $E(f(x))$, and this value can only be decrypted by the agent owner to obtain the desired result $f(x)$.

The main problem of this technique is that the authors have only found encryption schemes for polynomials, using homomorphic encryption and function composition techniques. Thus, their proposal is not suitable for general programming.

2.2.5 Tamper-proof devices

The use of tamper-proof devices is based on performing part or the entire agent execution on a physically sealed environment, which can be trusted to execute the agent correctly. Tamper-proof devices can be provided by a trusted third party and, if necessary, they can be inspected periodically to verify that their security has not been compromised. Tamper-proof devices can be used to carry out cryptographic operations with a private key that must be kept secret from the remote host. They can also have their own private key, for example, to sign partial results generated by the agent.

Approaches such as [WSB98] or [Yee99] propose performing the entire agent execution on tamper-proof devices. The cost of these solutions is high because each platform must

be provided with an expensive tamper-proof device. Besides, these techniques are only suitable for closed environments, such as corporate networks or computational grids, where a tamper-proof device has been installed in every platform. As a result, these techniques imply a loss of agent autonomy.

In order to reduce the cost of the solution, other approaches based on smart cards have been proposed in [Kar00] or [FM99]. In these solutions, the tamper-proof device has limited computation capabilities, and is only used to execute security-sensitive operations. However, the security of these approaches is limited because the platform controls the communication between the agent and the tamper-proof device. Thus, the inputs or outputs that are provided to or produced by the device can be easily tampered with.

2.3 Malicious host problem: realistic approaches

In this section, we provide a survey of some of the most widely accepted techniques for mobile agent protection against malicious hosts.

2.3.1 Protecting the agent computational results

Researchers have devised several mechanisms aiming at protecting the results generated by agents during their execution. These protocols allow owners to verify the identity of the host in which a given result was obtained, and can also be used to check whether the agent has visited all the platforms initially specified in the itinerary.

In [Yee99], Yee proposes a mechanism that involves signing the data generated by the agent in each platform with the platform's private key. Once the agent returns to its originator, it can verify the integrity of the results using the public keys of the visited platforms. Yee also proposes a variation of this method based on Partial Result Authentication Codes (PRACs). According to this method, the owner generates a list of cryptographic key pairs suitable for asymmetric encryption, and the private keys $(k_1 \cdots k_m)$ are given to the agent. The agent then uses each k_i to sign the results computed on platform i. Before migrating

to platform $i + 1$, the agent destroys k_i. Again, once the agent returns to its originator, the owner can verify the integrity of the results with the corresponding public keys.

The problem of these proposals is that they do not achieve data integrity when the agent chooses freely the next platform to visit at each stage of its itinerary. In this case, a malicious platform could remove previously collected results without being detected.

In [KAG98], Karjoth *et al.* improve Yee's proposals in order to ensure the integrity and confidentiality of the data generated by free-roaming agents. Their approach is based on binding each result to all previously collected results and to the identity of the subsequent itinerary platform. The problem of this approach is that the agent results are not bound to the agent code, thus allowing malicious platforms to generate fake results and append them to the chain of previous results, as described by Roth in [Rot01a].

In [KT01], Karnik and Tripathi propose to create an append-only container to store agent results. According to their protocol, a cryptographic checksum is used to bind the current results to the previous ones. This allows agents to add new results to the container, while preventing malicious platforms from modifying results previously generated. However, this protocol suffers from some limitations. First, fake results can be added to the append-only container, using the attack described in [Rot01a]. Furthermore, a collusion of two malicious platforms can easily truncate the chain of results collected between these two platforms.

The attacks described in [Rot01a] were overcome by Roth in [Rot01b] and [Rot02]. His protocol is based on a combination of two ideas. First of all, platforms only use their private keys to decrypt agent data if they can verify that these data belong to the agent. For this purpose, the data must be securely bound to the agent code before being encrypted. Second, the results generated by the agent execution are bound to the agent code so that they cannot be reused for another agent.

Finally, in [MS03], Maggi and Sisto put together the underlying ideas of different proposals [Yee99, KAG98, KT01, Rot01b] to define a protocol that is configurable according to the protection level required by the application.

2.3.2 Protecting the agent's itinerary

The protection of the agent's itinerary is one of the most commonly used techniques for agent protection against malicious platforms. The proposed techniques aim at preventing platforms from accessing or manipulating parts of the agent's itinerary intended for other platforms. For this purpose, the proposed protocols are usually based on encrypting every platform-specific information using the corresponding platform's public key. As a result, each platform is only given access to its corresponding part of the itinerary, as well as the address of its predecessor and successor platforms. The most relevant proposed protocols are discussed below.

In [BRSR99], Borrell *et al.* secure the agent's itinerary by protecting the information intended for each platform as follows:

$$I = [e_k, e_l, \cdots, e_t],\ k, l, t\ \in\ (1, \cdots, n) \tag{2.1}$$

where

$$e_i = P_i(h_i, m_i, h_{i+1}) \tag{2.2}$$

I denotes the protected itinerary, which is composed of a series of entries e_i randomly sorted. P_i is an asymmetric encryption method using the public key of platform i. h_i is the address of platform i, and m_i is the information intended for platform i.

To ensure that the itinerary is traversed in the correct order, a non-repudiation protocol [ZG96] is executed whenever a mobile agent is migrated from one platform to the next. Thus, the owner can verify that the agent has followed the itinerary as expected. Obviously, this solution assumes the existence of a Trusted Authority to guarantee a secure execution of the non-repudiation protocol. It is worth noting that Borrell *et al.*'s protocol prevents platforms from modifying the entries of the protected itinerary, for these entries are randomly sorted and the identity of the target platform is hidden by public key encryption. However, if the agent itinerary had only two hops, the first platform could easily modify the information intended for the second.

In [WSUK00], Westhoff *et al.* propose a protocol based on creating an onion-like structure to encapsulate the agent's route information as follows:

$$\begin{aligned} I \;=\; e_1 &= P_1(home, h_2, S_o(home, h_1, h_2, t, e_2))) \\ e_2 &= P_2(h_1, h_3, S_o(h_1, h_2, h_3, t, e_3))) \\ &\vdots \\ e_n &= P_n(h_{n-1}, \emptyset, S_o(h_{n-1}, h_n, \emptyset, t, \emptyset))) \end{aligned} \tag{2.3}$$

where we follow the same notation used for equation 2.2. $home$ denotes the agent's home platform address. S_o is a digital signature using the owner's private key, and t is a trip marker used to prevent replay attacks [Yee03].

As can be seen, the protected itinerary includes the addresses of the platforms that must be visited, but not any platform-specific code or data. However, this protection scheme could be easily extended to include platform-specific information inside the protected itinerary.

In [KT01], Karnik and Tripathi propose the *targeted state* mechanism, as a way to protect parts of the agent that are *targeted* towards particular platforms. This mechanism is based on creating a vector of encrypted objects as follows:

$$S_o(P_1(m_1), \cdots, P_n(m_n)) \tag{2.4}$$

Karnik and Tripathi's protocol was not designed to protect the agent's itinerary, but to provide the agent with information that can only be revealed to some platforms. As a result, this protocol suffers from some limitations. First, it does not guarantee that the itinerary will be traversed in the correct order. A malicious platform might forward the agent to an incorrect platform by sending it to randomly chosen platforms until finding one that is member of the initial itinerary. Additionally, as described in [Rot01a], this protocol is vulnerable to interleaving attacks [MvOV97], which allow malicious platforms to obtain information intended for other platforms.

In [RMAB02], Robles *et al.* present a secure mobile agent platform called MARISM-A. MARISM-A provides secure migration, secure communication between agents, protection

of agents' results and itinerary protection. The itinerary protection provided by MARISM-A is flexible because different itinerary protection protocols can be used depending on the specific requirements of the application. The set of itinerary protection protocols supported by MARISM-A includes protocols such as [WSUK00] or [BRSR99], and this set can be extended to adapt to new application requirements.

Finally, in [MB03], Mir and Borrell present a protocol based on building the protected itinerary as a chain of digital envelopes, in such a way that they can only be opened in the correct order. First of all, a random symmetric key is generated for each itinerary platform $(r_1 \cdots r_n)$. Then, for each migration from a platform i to its successor j, the following expression is computed:

$$t_{ij} \; = \; h_j, P_j(S_o(h_i, h_j, t, r_j)) \tag{2.5}$$

The value t_{ij} is called *transition* from i to j. Once all transitions have been computed, the protected itinerary is constructed as follows:

$$I = t_{01}, S_o(t, [e_1, \cdots, e_n]) \tag{2.6}$$

where

$$e_i = E_{r_i}(m_i, t_{ij}) \tag{2.7}$$

and E_{r_i} is a symmetric encryption method using the secret key r_i.

The protocol presented by Mir *et al.* supports different types of itinerary stages. Due to the fact that itinerary stage types will be discussed later in chapter 3, we have only presented here a slightly simplified version of the protocol, which is equivalent in terms of security.

2.3.3 Self-protected mobile agents

Protecting the agent's itinerary prevents platforms from accessing parts of the agent code or data intended for other platforms. Most itinerary protection mechanisms presented so far only define a way to create the protected itinerary, and assume that platforms know how to extract information from this protected itinerary.

The problem of these approaches, usually called *platform-driven*, is that the set of platforms that the agent can visit is restricted to those supporting the specific itinerary protection scheme implemented by the agent. In addition, any change in the agent's structure will imply changes in all itinerary platforms. Furthermore, platforms are forced to support multiple protection schemes if different types of secure mobile agents need to be executed.

In order to overcome these limitations, a solution based on self-protected mobile agents was proposed by Ametller *et al.* in [ARO04]. According to Ametller *et al.*, the agent protection mechanisms are not managed by the platform, but by the agent itself. This approach is called *agent-driven*.

Itinerary protection mechanisms presented so far did not make a clear definition of the different components of an agent. In [ARO04], Ametller *et al.* define agents as a pair *C*, *D*, where *C* is the main agent code that will be executed on all platforms, and *D* is the protected itinerary that contains the code and data intended for each platform.

The protection of *D* usually involves encrypting each platform-specific code and data using the public key of the target platform. The code *C* then deals with the management of this protected itinerary, extracting the code and data intended for each platform, executing the local task, and migrating the agent to its next destination.

In order to decrypt the information included in D, the code *C* needs to use the platforms' private keys. However, giving agents direct access to private keys would involve serious security problems. Consequently, Ametller *et al.* require each platform to have a cryptographic service providing a public decryption function to agents.

The public decryption function decrypts the agent's data using the platform's private key, but the decrypted data is only returned to the agent for which it was encrypted. Thus, malicious agents cannot decrypt data stolen from other agents. In order to verify that the decrypted data really belongs to the requesting agent, platforms verify that such data includes an integrity token *A* computed as follows:

$$A = H(C) \tag{2.8}$$

where H is a cryptographic hash function.

Thus, agent developers must always bind any platform-specific code or data to the agent instance using a hash of its main code *C*. Otherwise, agents will not be able to decrypt their itinerary data. Each stage of the agent's itinerary is thus encrypted as follows:

$$d_i = P_i(m_i, H(C)) \tag{2.9}$$

where m_i is the part of the itinerary that can only be revealed to platform i.

This mechanism prevents any tampering with the mobile agent code *C* that deals with the decryption of the itinerary data. However, an attacker might still insert arbitrary data in the protected itinerary *D*. All it would require is the inclusion of a hash of *C* along with the bogus data to be encrypted and inserted in *D*.

To prevent this kind of attack, Ametller *et al.* propose verifying the validity of the data extracted from the protected itinerary as follows:

1. A random pair of cryptographic keys is generated: P_a, the public key, and S_a, the private key.

2. The public key P_a is inserted in the agent's main code *C* (e.g. as a static data member).

3. The private key S_a is used to sign every platform-specific code or data included in the protected itinerary. Thus, the previous equation 2.9 is replaced by

$$d_i = P_i(S_{S_a}(m_i), H(C)) \tag{2.10}$$

   where S_{S_a} is a digital signature using the private key S_a.

4. The agent's main code *C* is modified to verify the validity of the itinerary data as follows:

   (a) The platform's public decryption function is called to decrypt $P_i(S_{S_a}(m_i), H(C))$. This operation can only succeed if the code *C* has not been modified after the agent creation.

(b) The cryptographic service returns $S_{S_a}(m_i)$ to the agent, and the signature of m_i is verified using the public key P_a inserted in the main code *C*.

This mechanism prevents any manipulation of the agent's protected itinerary *D*. The main agent code *C* cannot be modified because its integrity is verified by the platform's cryptographic service, and *C* verifies the authenticity of the data extracted from *D*, which must have been signed by the agent developer.

2.4 Conclusions

Research efforts in the field of mobile agent security have been quite intense over the last decade. Regarding the protection of platforms from agent or external attacks, several sound solutions have been presented. The problem of malicious hosts attacking an agent is by far the most difficult to solve. Although achieving a complete solution is considered impossible, protocols have been presented that mitigate several problems. The most relevant proposals address the protection of the agent's itinerary and the protection of the results generated during the agent execution. These advances have permitted the use of mobile agent technology to implement multiple distributed applications. However, some problems still remain unsolved and need to be addressed. In the following chapters, we will cover the protection of the itinerary of free roaming agents, the prevention of external replay attacks, and the simplification of the development of secure mobile agents.

Chapter 3

Defining the agent's itinerary

In the previous chapter, we have seen that several protocols have been proposed for the protection of the agent's itinerary. These protocols are usually based on storing the itinerary information in a separate data structure, and then on using cryptographic mechanisms to protect this data structure. When the itinerary information is stored and maintained outside the main agent code, the itinerary is said to be *explicit*, and its protection is significantly simplified.

The itinerary protection protocols presented to date do not support the protection of free-roaming agents. Agents are thus forced to travel *static* itineraries, that is, itineraries in which all platforms are known in advance. However, most mobile agent-based applications are devised using *dynamic* itineraries, in which some platforms are discovered at runtime.

One of the goals of this thesis is to provide a solution to this problem. For this purpose, first of all, this chapter presents a convenient way to define explicit itineraries for free-roaming agents. As will be seen, using explicit itineraries to create free-roaming agents promotes the reuse of these itineraries and simplifies their protection. Then, in chapter 4, we will present a protocol aimed at protecting dynamic itineraries.

3.1 Related work on defining agent's itinerary

Mobile agent itineraries have been implemented in different ways depending on the complexity and flexibility required by the application. The first approach was to merge tasks and migration instructions into a single code, so that every task was followed by a migration instruction to move the agent to its next destination. When this approach is taken, we say that the itinerary is *implicit* in the agent code.

In order to improve the readability, reusability and protectability of code, *explicit* itineraries were later introduced in [WPW+97]. In this case, the agent code is divided into stages, where each stage is usually executed on a different platform, and the information of all itinerary stages is stored in a separate data structure. The agent's explicit itinerary may only contain information regarding only the locations to be visited by the agent. However, the itinerary usually includes every platform-specific code and data as well. From now on, we will use the term *node* to refer to a stage of the agent's itinerary which is associated with a specific task and platform.

The data structure that contains the agent's explicit itinerary can be managed by the platform (thus following a platform-driven approach), or it can be managed by the agent itself, by executing a code that is common to all platforms of the itinerary (agent-driven approach).

First explicit itineraries were sequential [WPW+97], which involves that all platforms were visited one after the other, in the order initially specified by the programmer. For example, a sequential itinerary could be that of an agent that orders some flowers first, then buys a ticket for the theatre and finally reserves a table in a restaurant. The disadvantage of sequential itineraries is their lack of flexibility. They do not allow the programmer to define alternative routes, or routes that can be travelled in any order, among others.

In order to overcome these limitations, *flexible explicit* itineraries were introduced in [SRM98]. Flexible itineraries are composed of different types of nodes. These node types allow agents to make decisions about their travel plan at runtime, based on their previous computations or on other parameters. As an example, three node types were defined in

[SRM98]: the *sequence*, where the agent has only one possible destination after the current node, so no decision needs to be made; the *alternative*, where the agent can choose its next destination from a set of platforms; and the *set*, where the agent has to visit all the platforms of a set in any order.

However, as can be seen, no proposal presented so far allows programmers to define explicit itineraries for free-roaming agents, which involves discovering the location of one or more platforms at runtime. As mentioned earlier, the use of explicit itineraries significantly simplifies the protection of every platform-specific code and data. Therefore, the next section presents a new set of node types for the definition of dynamic itineraries. In chapter 4, a protocol will be presented for the protection of these itineraries.

3.2 Explicit itineraries for free-roaming agents

As stated earlier, flexible itineraries are comprised of different types of nodes. It is worth noting that our usage of the term *node* is quite different from that found in computer networks terminology, where a node simply refers to a location in the network. Here, an itinerary node is a stage of the agent execution that is associated with a certain task and a certain platform. This platform, however, does not necessarily have to be specified at the time of creating the agent; it can be determined by the agent at runtime.

In order to represent the itinerary of a mobile agent, we use a graphical notation in which each node type is depicted by a different symbol. The set of symbols we use will be presented later. Figure 3.1 shows the representation of an example itinerary comprising a sequence of three nodes.

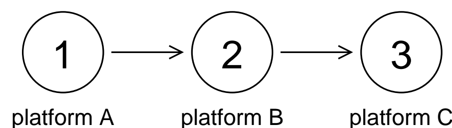

Figure 3.1: Example itinerary with three stages or nodes.

As shown in the figure, each node is associated with a numerical identifier, which appears inside the node's symbol. The name placed under each node is an identifier of the execution platform, which could be its hostname, for example. In this case, the agent first visits platform *A* and executes the task assigned to node 1. Then, it visits platform *B* and executes the task assigned to node 2; and so on.

This section presents a new set of node types devised to achieve two aims: First, enabling the definition of the agent's itinerary with the same flexibility that general purpose languages provide to control the execution flow of a program. Second, enabling the definition of nodes associated with platforms determined at runtime. The resulting set of node types is described next.

Sequence: In this type of node, the agent simply executes the local task and migrates to the platform associated with the next itinerary node. In figure 3.1, we have seen an example itinerary with three *sequence* nodes. As this figure shows, *sequence* nodes are depicted by the symbol ○.

If: The *if* node has a subitinerary associated with it, which is comprised of one or more nodes of any type. The local task of this node includes a method that is executed to decide whether or not the agent must traverse said subitinerary. Figure 3.2 shows an example itinerary with two *sequence* nodes and an *if* node. The *if* node is depicted by the symbol ◁. In this example, the subitinerary associated with the *if* node contains only one node.

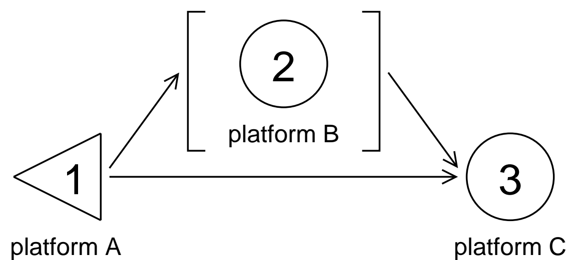

Figure 3.2: Example itinerary with an *if* node.

As shown in the figure, the agent first visits platform *A* and executes the task assigned

to node 1, which decides whether or not to enter the subitinerary associated with node 1. If the agent decides to do so, it will execute the tasks of nodes 2 and 3 on platforms *B* and *C* respectively. Otherwise, it will hop directly to platform *C* to execute the task of node 3.

Switch: The *switch* node has two or more subitineraries associated with it. The local task includes a method that chooses which subitinerary must be traversed next. Figure 3.3 shows an itinerary with a *switch* node and two subitineraries associated with it. The *switch* node is depicted by the symbol ⏢.

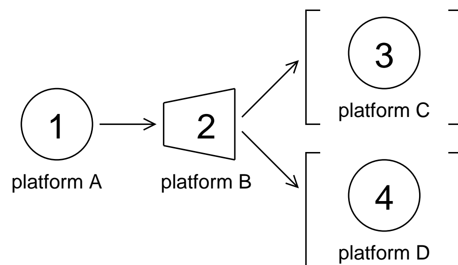

Figure 3.3: Example itinerary with a *switch* node.

As shown in the figure, when the agent visits node 2, it must choose which node must be visited next: 3 or 4. Depending on the choice, the agent will end its itinerary performing the task assigned to node 3 or the one assigned to node 4.

Set: The *set* node also has two or more subitineraries associated with it. In this case, however, after executing the local task assigned to this node, all subitineraries are traversed by the agent. This traversal can be done in sequence, one subitinerary after the other (in any order), or it can be done in parallel, sending a clone of the initial agent to each subitinerary. Whether this traversal is done in parallel or in sequence depends on the final implementation. Figure 3.4 shows an example itinerary with a *set* node associated with two subitineraries. In this case, one of the subitineraries has two nodes. The *set* node is depicted by the symbol ⬠.

As this figure shows, the agent can either visit nodes 2 and 3, and then visit node 4, or it can do the opposite: visit node 4 and then nodes 2 and 3. Depending on the

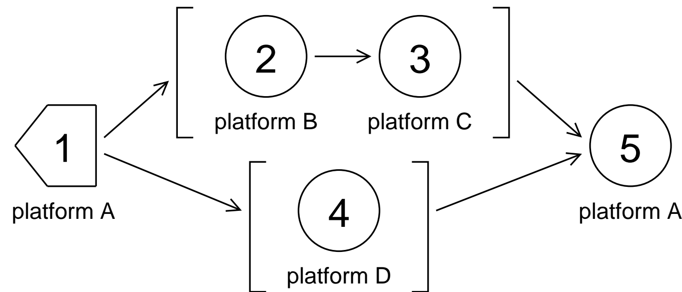

Figure 3.4: Example itinerary with a *set* node.

implementation, the agent could even clone itself in node 1, so that both subitineraries were traversed in parallel. This example also shows that the agent's itinerary can include platforms that are visited in more than one node. In this case, platform *A* is visited in nodes 1 and 5.

Loop: The *loop* node has only one subitinerary associated with it. The agent first visits the *loop* node, and then it traverses said subitinerary repeatedly. After each iteration, the agent returns to the *loop* node, where it executes the local task and decides whether or not to perform a new iteration. Figure 3.5 shows an example itinerary with a *loop* node and a subitinerary composed of two nodes. The *loop* node is depicted by the symbol ⬭.

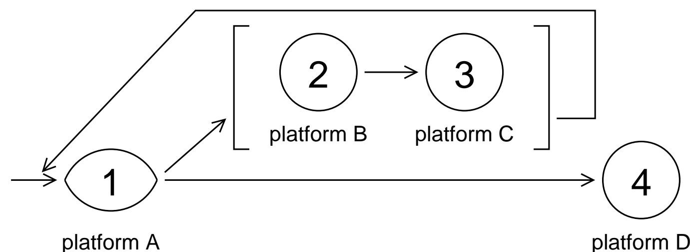

Figure 3.5: Example itinerary with a *loop* node.

As shown in the figure, the agent first executes the task assigned to node 1 and decides whether or not to enter the subitinerary starting at node 2. If it chooses not to do so, it will migrate directly to platform *D*, and such subitinerary will never be traversed. Otherwise, the agent will travel to platforms *B* and *C*, and then it will return

to *A*. In platform *A*, the agent will decide again whether or not to enter the following subitinerary, thus starting the whole process again.

Discoverer: The *discoverer* node also has a subitinerary associated with it. This subitinerary can contain nodes that do not have an assigned platform initially. Thus, these nodes are called *dynamically located* nodes, and they have no associated platform at the time of creating the itinerary. A dynamically located node is not a different node type. It is introduced into the itinerary using a special property that can be set on any type of node: the *dynamic location* property, which will be described in detail in the next section. The *discoverer* node determines where the dynamically located nodes will be visited.

Figure 3.6 shows an itinerary with a *discoverer* node and a dynamically located node. The *discoverer* node is depicted by the symbol ◁, and the dynamically located node is depicted by replacing the platform's name by the symbol "#". The *dynamic location* property is set on a *sequence* node in this example, but it could be set on any other type of node.

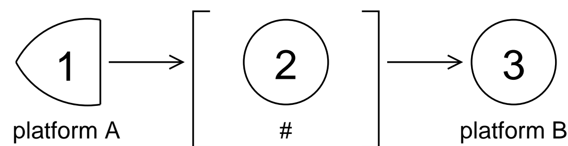

Figure 3.6: Example itinerary with a *discoverer* node and a dynamically located node.

In the example of figure 3.6, the platform where node 2 will be visited is not known at the time of creating the itinerary. Thus, node 2 has the *dynamic location* property set. In node 1, the agent will execute a task that will determine what platform is assigned to node 2.

The set of node types presented so far provides a flexible means of defining the agent's migration flow. In order to use these node types, the programmer should not introduce any migration from one platform to another within the local task of a node. Thus, for the

sake of clarity, migrations should only take place during the transition from one node to the next. In the next section, we will describe two node special properties that are necessary to introduce dynamism into the agent's itinerary: the *dynamic location* property, which has already been introduced, and the *unchanged location* property.

3.3 Node special properties

In this section, we describe the *unchanged location* property and the *dynamic location* property, which allow programmers to create itinerary nodes the task of which is executed on platforms that are determined at runtime.

3.3.1 Unchanged location property

The *unchanged location* property is used to specify that the task of a node will be executed on the previous platform visited by the agent. This implies that, after executing the local task, the agent performs no migration and resumes its execution on the same platform where it was being executed.

An example of the use of this property can be seen in figure 3.7. The *unchanged location* property is depicted by replacing the platform's name by a left arrow ($\leftarrow$), meaning "same as previous". In this itinerary, the agent visits node 4 after finishing any of the two subitineraries associated with the *switch* node 1. As node 4 has the *unchanged location* property set, this node will be visited in platform *B* or *C* depending on the decision made in node 1. The *unchanged location* property is set on a *sequence* node in this example, but it could be set on any other type of node.

This property is especially useful when it avoids performing unnecessary migrations. An example of this situation can be seen in figure 3.8. In this case, the programmer wants to evaluate the loop condition on platform *A* the first time, and then on platform *C* subsequent times. By setting the *unchanged location* property on *loop* node 2, the programmer is relieved of the need to introduce an additional platform into the agent's itinerary for the

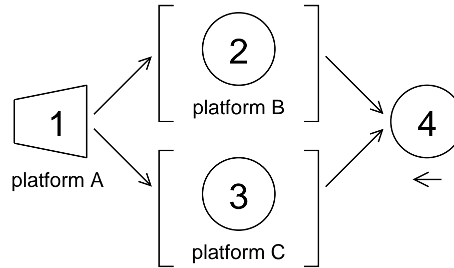

Figure 3.7: Example itinerary with the *unchanged location* property set on a node.

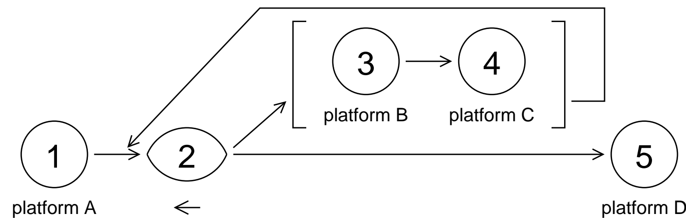

Figure 3.8: Use of the *unchanged location* property on a *loop* node.

evaluation of the loop condition. As a result, no migration is needed to reach node 2 and decide whether a new iteration has to be started.

3.3.2 Dynamic location property

The *dynamic location* property, as mentioned earlier, is used in combination with *discoverer* nodes in order to allow agents to visit platforms discovered at runtime. The *discoverer*'s local task determines what platform is assigned to the dynamically located nodes.

Figure 3.9 shows an example itinerary in which the agent has to visit two *discoverer* nodes and two dynamically located nodes. In this case, the *dynamic location* property has been set on an *if* node and a *sequence* node.

The itinerary of figure 3.9 could be that of an agent sent out to find a cheap flight and hotel for a business trip. First of all, the agent queries a remote search engine in order to find an airline company offering flights to a certain destination (node 1). Once the airline company is found, the agent migrates to this company's platform and searches for a cheap flight that fits within the budget (node 2). If such flight is found, the agent buys the ticket

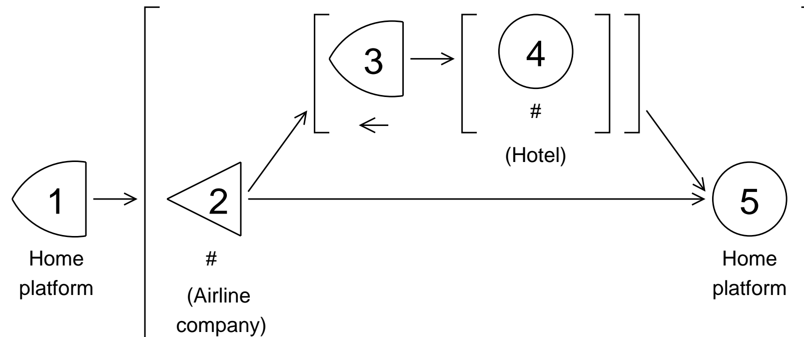

Figure 3.9: Itinerary of a shopping agent that includes two *discoverer* nodes and two dynamically located nodes

and queries the remote search engine again to find a cheap hotel in the destination (node 3). Once the hotel is found, the agent migrates to the hotel's platform to negotiate a good price for the stay. After booking the hotel, the agent returns to the home platform. As can be seen, the tasks of nodes 2 and 3 are executed on the same platform since node 3 has the *unchanged location* property set.

It is worth noting that, by placing all dynamically located nodes inside the subitinerary of a *discoverer* node, the programmer is prevented from defining faulty itineraries in which the agent would have to migrate to a dynamically located node that had no associated *discoverer* node.

The *dynamic location* property and the *discoverer* node can also be used in combination with other node types in order to define the itinerary of a completely free-roaming agent. An example of such itinerary is shown in figure 3.10. In this case, the agent developer only specifies the starting point of the itinerary, where the agent must be sent to start its execution. The rest of the agent execution takes place on platforms discovered at runtime, and the agent can visit as many platforms as necessary in order to accomplish its task.

As shown in the figure, the agent assigns a new platform to node 4 at each iteration of the *loop* node 2. On each of these platforms, the agent executes the task of node 4 and, in addition, the tasks of nodes 2 and 3. The task of node 2 checks whether the agent has finished its itinerary and, if that is not the case, the task of node 3 determines where to migrate next.

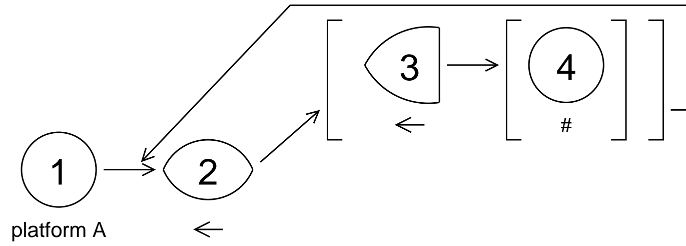

Figure 3.10: Itinerary of a completely free-roaming agent

In addition to the functionality presented so far, the *discoverer* node can also be used to assign multiple platforms to one or more dynamically located nodes. In the example itinerary of figure 3.9, the agent could decide that several hotels have to be visited in order to negotiate the best price for the stay. In this case, the same task would be executed in all hotels.

Furthermore, the itinerary associated with a *discoverer* node can contain several dynamically located nodes, as shown in the example of figure 3.11. In this case, the agent may assign two different sets of platforms to nodes 2 and 4.

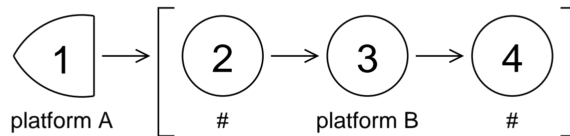

Figure 3.11: Example itinerary with two dynamically located nodes associated with the same *discoverer* node.

3.4 Implementation issues

Thus far, we have provided a theoretical description of a set of node types and properties aimed at allowing programmers to define dynamic itineraries. In this section, we will present some considerations regarding the implementation of these itineraries and their corresponding agents.

First of all, the information of the explicit itinerary can be stored inside the agent in multiple ways. If no security is required, the itinerary can be carried by the agent in a

data structure such as the one represented by the following expression. This expression corresponds to the itinerary of figure 3.5.

$$I = \Big[\ (1, m_1, loop, [(B, 2), (D, 4)]), (2, m_2, seq, [(C, 3)]), \\ (3, m_3, seq, [(A, 1)]), (4, m_4, seq, [])\ \Big] \tag{3.1}$$

As this expression shows, the explicit itinerary is structured as a list, where each element contains the following information: i, the node's numeric identifier; m_i, the task associated with node i; $type$, the node type; and $[[location_j, j] \cdots [location_k, k]]$, the list of locations and identifiers corresponding to the subsequent nodes $j \cdots k$.

The data structure that contains the explicit itinerary can be managed by the platform, following a platform-driven approach, or it can be managed by the agent itself, following an agent-driven approach.

In the first case, the platform accesses the agent's explicit itinerary to obtain the local task that has to be executed and the subsequent platform where the agent has to migrate next. In the second case, all these operations are performed by the agent by executing a code that is common to all platforms. We refer to this code as the agent's *control code*.

In order to implement agents using the set of node types presented in this chapter, the agent-driven approach is preferable, for it allows platforms to manage all agents in the same way, without knowing how they are internally structured. This is especially important as there may be different ways of storing the explicit itinerary inside the agent, and there may be multiple ways of protecting the data structure that contains such explicit itinerary.

Whether using a platform-driven or an agent-driven approach, different alternatives can be considered when implementing the behaviour associated with some itinerary nodes.

Regarding the *set* node type, the implementation can generate several clones to traverse the different subitineraries associated with this node, or the implementation can traverse all the subitineraries in sequence. The sequential approach can be used to implement fault-tolerant mechanisms, so that if the agent encounters problems starting a certain subitinerary, it can try with another and return to the problematic one later. The cloning approach can be used to parallelise the execution, thus increasing the performance of the application.

Regarding the *discoverer* node, it might be the case that the agent was not able to assign a platform to a certain dynamically located node. Different strategies may be implemented to deal with this situation, such as aborting the execution, or skipping that node and continuing with the subsequent one, among others.

3.5 Conclusions

The *dynamic location* and *unchanged location* properties, in combination with the *sequence*, *if*, *switch*, *set*, *loop*, and *discoverer* nodes previously presented, comprise a complete solution for the definition of flexible explicit itineraries for free-roaming agents.

Although this set of node types and properties has been devised to be a general purpose one, new node types or properties can be added in the future, for example, to meet the specific requirements of a certain application, and this will have no real effect on the protection protocols presented in chapters 4 and 5.

The use of flexible explicit itineraries divides the design of mobile agents into two different levels: global, where the programmer designs the migratory behaviour of the agent; and local, where the programmer defines the agent's behaviour at each of the steps of its travel. As a result, the implementation of mobile agent applications is significantly simplified.

Defining agents based on flexible explicit itineraries has other advantages as well. First of all, the itinerary is easier to reuse, for the different stages of the itinerary are clearly separated and can be easily modified or changed for those of other agents previously implemented. Second, the data structure that stores the agent itinerary can be protected using cryptographic mechanisms. Therefore, the set of node types and properties defined in this chapter provide a convenient way to define itineraries for free-roaming agents. The next chapter will be devoted to presenting a protocol for the protection of dynamic itineraries.

Chapter 4

Securing dynamic itineraries

The advances in mobile agent security have provided solutions to many of the problems initially identified, such as the protection of platforms against malicious agents, or the protection of the agents' computational results, among others. Nevertheless, as mentioned earlier in chapter 2, some issues still must be addressed, mostly regarding the protection of agents against malicious platforms [JK00].

Several proposals have been presented to prevent platforms from tampering with the code carried by the agent or with the results generated by its execution. Regarding the protection of the agent code, the most widely accepted proposals are based on using itinerary protection protocols [WSUK00, MB03]. These protocols use public key algorithms to encrypt the information intended for each itinerary platform. Thus, the information that becomes visible to a platform is reduced to a minimum: the agent's current task and the next destination.

However, itinerary protection protocols presented to date only support static itineraries because the public key of each itinerary platform must be known in advance. As a result, current developments still cannot take full advantage of the benefits provided by the mobile agent technology. This is compounded by the fact that most mobile agent applications are devised using dynamic itineraries. Examples of these applications can be found in many areas: grid computing [KBD02], data mining [KLM03], network routing [MV07], P2P

networks [LF06], sensor networks [WQ04], intrusion detection systems [HN05], ad-hoc networks [LCT+05], among others.

In order to support free-roaming agents, this chapter presents a protection protocol based on introducing trusted locations into the agent's route. Traditional approaches assume that all platforms are potentially malicious, but this makes the protection of dynamic itineraries unfeasible. By introducing some trusted platforms into the itinerary, the proposed protocol secures the information associated with dynamically located nodes. Before presenting such protocol, the following section discusses what assumptions are made when the programmer incorporates a trusted platform into the agent's itinerary.

4.1 Assumptions made with regard to trusted platforms

When the programmer introduces a trusted platform into the agent's itinerary, he believes that the agent's task will be executed on that platform as expected. The protocol presented in this chapter assumes that, if the programmer trusts a certain platform, then this platform will execute the agent's task honestly. For example, if the programmer decides to execute the task of a *loop* node on a trusted platform, then it is assumed that such platform will execute the loop condition included in this task honestly. As a result, the agent will perform the expected number of loop iterations.

Moreover, it is assumed that the agent platform and the computer it runs on are protected with appropriate mechanisms so as to prevent attacks from third parties that might alter the agent execution. In this case, security greatly depends on the mechanisms provided by the operating system and the good design of associated protocols. The study of what mechanisms are required to guarantee the secure execution of the agent platform is out of the scope of this thesis.

The proposed protocol also assumes the existence of a security infrastructure that allows agent developers and users to determine whether a platform is trustworthy or not. An example of such infrastructure can be found in [TM01]. In this work, the authors describe a security framework for a mobile agent system which incorporates a simple trust model.

Such model is based on establishing trust relationships in a manner similar to that used in public key infrastructures to handle distributed authentication.

The identification of trustworthy platforms can also be grounded on simpler mechanisms, such as relying on real-world trust relationships. For example, the platform associated with a bank where the user has an account, or the platform from which the agent was first launched, can be safely introduced into the agent's itinerary as trusted platforms.

4.2 Notation

Before presenting the proposed protocol, we will describe first the notation used.

- n denotes the total number of itinerary nodes.
- p_i denotes the platform assigned to node i.
- m_i denotes the agent's code and data to be executed in node i.
- h_i denotes the address of platform p_i.
- P_i denotes an asymmetric encryption function using the public key of platform p_i.
- S_i denotes a digital signature function using the private key of platform p_i.
- E_r denotes a symmetric encryption function using the secret key r.
- H denotes a cryptographic hash function.
- $a_i \cdots A_i$ denotes the set of direct successors of i. For example, if i is a *set* node associated with three subitineraries, then i has three direct successors, and thus the set $a_i \cdots A_i$ has three elements.
- $b_i \cdots B_i$ denotes the set of direct predecessors of i.
- If i is a *discoverer* node, then

  - $l_i \cdots L_i$ denotes the set of all nodes q such that h_q is discovered in node i.
  - $bl_i \cdots BL_i$ denotes the set of all direct predecessors of nodes $l_i \cdots L_i$.

- If i is a dynamically located node, then
  - d_i denotes the *discoverer* node where h_i is discovered.
  - h_{d_i} denotes the address of the platform assigned to node d_i.

- If i is not a dynamically located node, then
  - d_i denotes the agent owner.
  - h_{d_i} denotes the address or identifier associated with the owner.

- o denotes the agent owner. Note that the owner can be referred to as o or d_i. However, d_i denotes the owner only when i is not a dynamically located node. Otherwise, it denotes the *discoverer* node where h_i is discovered. On the other hand, o always denotes the agent owner.

4.3 The protocol

This section presents a protocol aimed at protecting flexible explicit itineraries constructed with the set of node types described in chapter 3. This protocol pursues three main objectives:

Integrity: Platforms must not be able to modify the agent's itinerary undetectably.

Confidentiality: Platforms must not be able to access itinerary information corresponding to other platforms.

Authenticity: Platforms must be able to verify the identity of the agent owner.

The proposed protocol is based on the protocol presented by Mir and Borrell in [MB03]. The main difference between their approach and the proposed one is the support for *discoverer* nodes, the *dynamic location* property and the *unchanged location* property, which enable the definition of free-roaming agents.

The general idea behind the proposed protocol is to construct a chain of digital envelopes, each of which containing two elements: the data, and the encrypted key that allows decrypting the following envelope. A representation of this scheme is shown in figure 4.1.

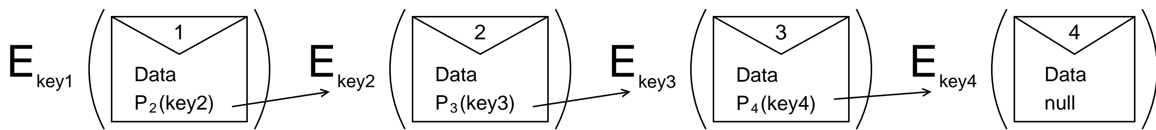

Figure 4.1: Representation of a chain of digital envelopes.

The envelopes shown in this figure represent the entries of the protected itinerary. Every envelope is encrypted using a random symmetric key, and this symmetric key is in turn encrypted using the public key of the platform entitled to open the envelope. Thus, each envelope can only be decrypted by the intended platform.

Additionally, the envelopes can only be opened in the correct order, for the symmetric key used to decrypt an envelope is protected inside the previous envelope.

The problem of protecting dynamic itineraries is that not all public keys are known in advance. This makes it impossible to build a chain of digital envelopes as the one previously described. More specifically, the platforms assigned to dynamically located nodes are discovered by the agent at runtime, in the corresponding *discoverer* nodes. Therefore, the public keys of the platforms assigned to dynamically located nodes are not available when the protected itinerary is initially created.

In order to solve this problem, the protocol proposed in this chapter is based on protecting the information of dynamically located nodes using the public keys of their corresponding *discoverer* nodes. Then, when the agent visits a *discoverer* node, the public keys of the newly discovered platforms are used to rebuild the protected itinerary. This scheme

involves changing the chain of digital envelopes, as shown in figure 4.2.

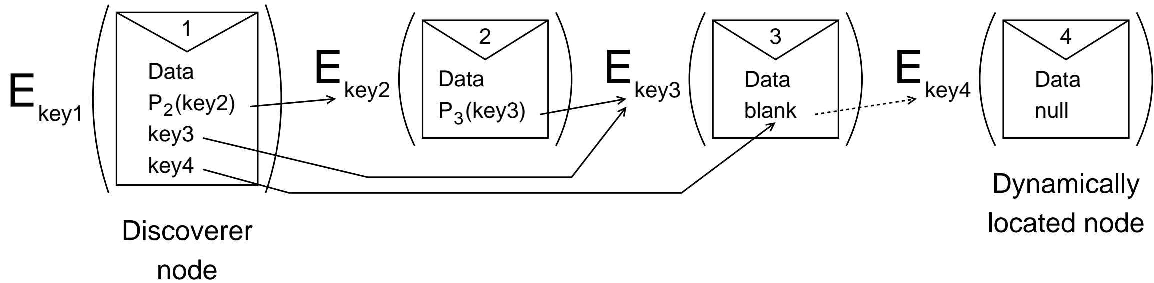

Figure 4.2: Modified chain of digital envelopes to support dynamic itineraries.

As this figure shows, the envelope associated with the *discoverer* node contains two additional symmetric keys: First, the one used to encrypt the dynamically located node (*key4*); and, second, the one used to encrypt the predecessor of the dynamically located node (*key3*). When the agent visits the *discoverer* node 1, it determines where to visit node 4, and it uses the public key of the newly discovered platform to encrypt *key4*. Thus, the agent generates $P_4(key4)$. Then, it uses *key3* to rebuild the envelope 3, in order to replace the *blank* with $P_4(key4)$.

As a result, the protocol presented in this chapter allows programmers to create protected itineraries with dynamically located nodes. This approach, however, requires agents to execute the tasks of *discoverer* nodes on trusted platforms, and also requires these trusted platforms to be known in advance.

The construction of the protected itinerary, and then the operations required to handle such itinerary are presented in the following sections. The combined actions comprise the itinerary protection protocol which supports free-roaming agents.

4.3.1 Building the protected itinerary

Regarding the construction of the protected itinerary, first of all, a random symmetric key is created for every itinerary node. Thus, the following set of keys is obtained:

$$r_1 \cdots r_n$$

Next, for each possible migration from a node i to any of its successors $j \in \{a_i \cdots A_i\}$, the *transition* from i to j (t_{ij}) is generated. The value of this transition depends on which property, if any, has been set on node j.

If neither the *unchanged location* nor the *dynamic location* property has been set on node j, then the transition t_{ij} is computed as follows:

$$t_{ij} = j, h_j, P_j(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_j)) \quad (4.1)$$

As shown in the above equation, the transition from i to j contains the random symmetric key r_j associated with node j. This random symmetric key will be used to encrypt the entry (or envelope) of the protected itinerary associated with node j. In order to ensure that only platform p_j has access to r_j, this symmetric key is encrypted using the public key of platform p_j (see figure 4.1).

Additionally, t_{ij} includes other information that is used to prevent malicious manipulations of this transition:

- First, it includes h_{d_j}, which is the owner's address or identifier because j is not a dynamically located node.

- Second, it includes $S_{d_j}(id, h_j)$, where d_j is the owner because j is not a dynamically located node. By introducing h_j into the transition, platform p_j will be able to verify that node j was certainly assigned to platform p_j.

- Finally, t_{ij} includes a unique agent identifier id that is used to prevent replay attacks. The prevention of replay attacks will be covered in detail in chapter 5. As can be seen, this trip marker is included twice, so as to bind together $S_o(id, r_j, h_{d_j})$ and $S_{d_j}(id, h_j)$, and prevent their reuse in other agents.

As can be seen, id, r_j and h_{d_j} are signed by the owner (S_o), so that platform p_j will be able to verify the identity of the agent owner and the integrity of this information.

The expression shown in equation 4.1 is used to build those transitions t_{ij} in which node j has neither the *unchanged location* nor the *dynamic location* property set. If node j

has the *dynamic location* property set, then the transition t_{ij} is computed as follows:

$$t_{ij} \;=\; j, blank \tag{4.2}$$

As can be seen, the expression $h_j, P_j(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_j))$ is replaced by the special value '$blank$'. This is due to the fact that h_j is not known at the time of creating the itinerary, and the public key needed to compute $P_j(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_j))$ is not available either (see envelope 3 of figure 4.2). As explained in the next section, this transition will be rebuilt at runtime in the *discoverer* node d_j associated with j.

If node j has the *unchanged location* property set, then the value of t_{ij} depends on which property, if any, has been set on node i. If neither the *unchanged location* nor the *dynamic location* property has been set on node i, then the transition t_{ij} is computed as follows:

$$t_{ij} \;=\; j, h_i, P_i(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_i)) \tag{4.3}$$

As shown in the above equation, the transition t_{ij} is equivalent to that shown in equation 4.1. The only difference is that h_j is replaced with h_i because the agent will visit node j in the same platform assigned to node i. For this same reason, this transition is encrypted using the public key of platform p_i.

If both nodes j and i have the *unchanged location* property set, then equation 4.3 must be replaced with

$$t_{ij} \;=\; j, h_{b_i}, P_{b_i}(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_{b_i})) \tag{4.4}$$

The equivalent change should be made in the above expression if node b_i had the *unchanged location* property set as well.

Finally, if node i has the *dynamic location* property set, and node j has the *unchanged location* property set, then the transition t_{ij} is computed as shown in equation 4.2. Note that, in this case, node j is treated as a dynamically located node because its task will be executed on the platform assigned to node i at runtime.

Once all transitions t_{ij} have been created, the protected itinerary is constructed as follows:

$$I \;=\; t_{01}, [e_1 \cdots e_n] \tag{4.5}$$

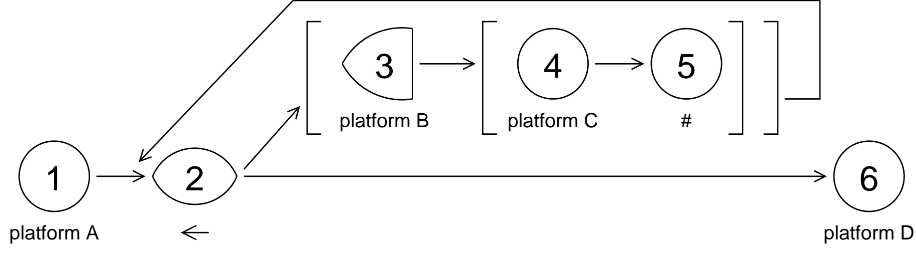

Figure 4.3: Example itinerary with a *loop*, a *discoverer* and a dynamically located node.

where $t_{01} = 1, h_1, P_1(S_o(id, r_1, h_{d_1}), S_{d_1}(id, h_1))$ is the transition needed by the agent to decrypt e_1 in the first platform of the itinerary. The value of each entry e_i depends on the type of node i. If i is not a *discoverer* node, then e_i is computed as follows:

$$e_i \ = \ E_{r_i}(S_o(id, i, m_i, type_i), [t_{i\,a_i} \cdots \ t_{i\,A_i}]) \tag{4.6}$$

As this equation shows, every entry e_i contains a unique agent identifier (id), a node identifier (i), a task (m_i), a node type ($type_i$), and a list of transitions from node i to its successors $a_i \cdots A_i$ ($[t_{i\,a_i} \cdots \ t_{i\,A_i}]$).

If i is a *discoverer* node, then e_i is computed as follows:

$$\begin{aligned} e_i \ = \ E_{r_i}\Big(& S_o\big(id, i, m_i, discoverer, \\ & [S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)], [r_{bl_i} \cdots r_{BL_i}] \big), [t_{i\,a_i}] \Big) \end{aligned} \tag{4.7}$$

As this equation shows, e_i contains only one transition $t_{i\,a_i}$ because i is a *discoverer* node and has only one direct successor. In addition, e_i includes two lists: $[S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)]$ and $[r_{bl_i} \cdots r_{BL_i}]$. If we draw an analogy between the contents of e_i and the first envelope shown in figure 4.2, then the first list is equivalent to *key4*, and the second list is equivalent to *key3*. As will be seen in the next section, the first list contains the symmetric keys needed to rebuild the transitions to the dynamically located nodes ($l_i \cdots L_i$), and the second list contains the symmetric keys needed to decrypt the entries of the protected itinerary $e_{bl_i} \cdots e_{BL_i}$, where the newly generated transitions will be included.

In order to exemplify the creation of a protected itinerary, figure 4.3 shows an example itinerary comprised of six nodes, one of which is a *loop*, another one is a *discoverer*, and

the remaining ones are *sequence* nodes. Applying the proposed protocol to this itinerary would yield the following protected itinerary.

$$\begin{aligned} I \;=\; t_{01}, \Big[& E_{r_1}(S_o(id,1,m_1,seq),[t_{12}]), E_{r_2}(S_o(id,2,m_2,loop),[t_{23},t_{26}]), \\ & E_{r_3}(S_o(id,3,m_3,discoverer,[S_o(id,r_5,B),S_o(id,r_2,B)],[r_4,r_5]),[t_{34}]), \\ & E_{r_4}(S_o(id,4,m_4,seq),[t_{45}]), E_{r_5}(S_o(id,5,m_5,seq),[t_{52}]), \\ & E_{r_6}(S_o(id,6,m_6,seq),[]) \Big] \end{aligned} \tag{4.8}$$

where

$$\begin{aligned} t_{01} &= 1, A, P_1(S_o(id,r_1,o),S_o(id,A)) \;,\; t_{12} \;=\; 2, A, P_1(S_o(id,r_2,o),S_o(id,A)) \;, \\ t_{23} &= 3, B, P_3(S_o(id,r_3,o),S_o(id,B)) \;,\; t_{34} \;=\; 4, C, P_4(S_o(id,r_4,o),S_o(id,C)) \;, \\ t_{45} &= 5, blank \;,\; t_{52} \;=\; 2, blank \;,\; t_{26} \;=\; 6, D, P_6(S_o(id,r_6,o),S_o(id,D)) \end{aligned}$$

As these expressions show, the proposed protection of the explicit itinerary attains the objectives initially established.

With regard to authenticity, the symmetric keys $r_1 \cdots r_n$, which are used to encrypt the entries of the protected itinerary, are digitally signed by the agent owner. Additionally, i, m_i, and $type_i$ are also signed to prevent their modification in dynamically located nodes. Thus, the authenticity of every itinerary entry can be verified.

The authenticity of the itinerary entries ensures that attackers can neither generate their own entries nor modify existing ones. Entries cannot be removed either because removing an entry e_i implies removing the key r_{a_i} that allows to decrypt the subsequent entry e_{a_i}. Furthermore, the unique agent identifier id prevents the reuse of entries previously generated by the same owner. Therefore, the integrity of the protected itinerary is also guaranteed.

As for the confidentiality of the itinerary information, each symmetric key r_i is encrypted in such a way that it can only be decrypted by platform p_i. Thus, platform p_i is only allowed to decrypt e_i, and all other itinerary entries remain hidden. The confidentiality of itinerary entries is thus guaranteed as well.

Additionally, every transition to a given node i includes the address h_i of the platform assigned to i. Thus, platforms can verify that they were certainly part of the initial itinerary.

Regarding the dynamically located nodes, their information is initially accessible to their corresponding *discoverer* nodes, and remains hidden to the remaining nodes. Therefore, provided that the platforms assigned to *discoverer* nodes are trustworthy, the authenticity, integrity and confidentiality of dynamically located nodes are achieved.

It is also worth noting that the platforms assigned to *discoverer* nodes must be known initially, for their addresses must be signed by the owner and introduced into the protected itinerary (see equation 4.7).

In order to understand how the protected itinerary is handled, the next section describes the operations required to (1) verify that the itinerary has not been tampered with; (2) extract and execute the local task; and (3) rebuild the protected itinerary at runtime to enable the execution of the tasks assigned to dynamically located nodes.

4.3.2 Management of the protected itinerary

The steps required to manage the protected itinerary at runtime depend on the type of node that the agent is visiting. The following is the set of operations that must be performed when the agent is not visiting a *discoverer* node.

1. Let i be the current node, and let $g \in \{b_i \cdots B_i\}$ be one of the predecessors of i. Then the transition t_{gi} placed at the beginning of the protected itinerary is decrypted using the platform's private key. Thus, $S_o(id, r_i, h_{d_i}), S_{d_i}(id, h_i)$ is obtained. For example, in the first platform of the itinerary, t_{01} is decrypted and $S_o(id, r_1, o), S_o(id, h_1)$ is obtained.

2. The signature of $S_o(id, r_i, h_{d_i})$ is verified, and if this verification does not succeed, the agent execution is discarded.

3. The signature of $S_{d_i}(id, h_i)$ is verified, and if this verification does not succeed, the agent execution is discarded. Note that, if i is not a dynamically located node, then d_i is the owner. Otherwise, d_i is the *discoverer* node associated with i.

4. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

5. h_i is compared with the current platform's address, in order to verify that the current platform is certainly part of the initial itinerary. Without this verification, the following attack would be possible: Let p_z be a malicious platform assigned to node z. p_z decrypts the transition $t_{b_z\,z}$ placed at the beginning of the itinerary. Then p_z encrypts $t_{b_z\,z}$ again using the public key of platform p_i, and sends the agent to p_i. Platform p_i decrypts $t_{b_z\,z}$, and ends up executing the task assigned to node z because it does not realise that $t_{b_z\,z}$ is actually intended for platform p_z.

6. e_i is decrypted using r_i, thus obtaining $(S_o(id, i, m_i, type_i), [t_{i\,a_i} \cdots t_{i\,A_i}])$ (see equation 4.6).

7. The signature of $S_o(id, i, m_i, type_i)$ is verified, and if this verification fails, the agent execution is discarded.

8. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

9. The agent executes the current task m_i. If i is not a *sequence* node, the agent chooses which of the following nodes must be visited next. Let us assume that the following node to be visited by the agent is a_i.

10. $t_{i\,a_i}$ is placed at the beginning of the protected itinerary, thus replacing the transition t_{gi} previously obtained in step 1.

11. The agent is migrated to platform p_{a_i}, where the execution of these steps is started again.

When the agent has to visit a *discoverer* node, the handling of the protected itinerary requires several additional operations. Let i be the *discoverer* node. Then, first of all, a platform must be assigned to each dynamically located node $l_i \cdots L_i$ associated with i. Second,

the transitions to nodes $l_i \cdots L_i$ must be rebuilt using the list $[S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)]$ included in e_i. Finally, the symmetric keys included in the list $[r_{bl_i} \cdots r_{BL_i}]$ must be used to modify the corresponding entries $e_{bl_i} \cdots e_{BL_i}$. Thus, the old transitions initialised using the special value '$blank$' must be replaced with the newly generated transitions.

The following is a detailed description of all the operations involved.

1. The transition placed at the beginning of the protected itinerary is decrypted using the platform's private key, thus obtaining $S_o(id, r_i, h_{d_i}), S_{d_i}(id, h_i)$.

2. The signature of $S_o(id, r_i, h_{d_i})$ is verified, and if this verification does not succeed, the agent execution is discarded.

3. The signature of $S_{d_i}(id, h_i)$ is verified, and if this verification does not succeed, the agent execution is discarded.

4. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

5. h_i is compared with the current platform's address, in order to verify that the current platform is really the one assigned to node i.

6. e_i is decrypted using r_i, thus obtaining the following information (see equation 4.7):

$$\Big(S_o\big(id, i, m_i, discoverer, [S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)], [r_{bl_i} \cdots r_{BL_i}] \big), [t_{i\,a_i}] \Big)$$

7. The signature of

$$S_o\big(id, i, m_i, discoverer, [S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)], [r_{bl_i} \cdots r_{BL_i}] \big)$$

   is verified, and if this verification fails, the agent execution is discarded.

8. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

9. The agent executes the current task m_i. As i is a *discoverer* node, the agent determines what platforms are assigned to the dynamically located nodes. The agent is thus discovering the platforms $p_{l_i} \cdots p_{L_i}$.

10. Using the symmetric keys included in the list $[r_{bl_i} \cdots r_{BL_i}]$, the entries $e_{bl_i} \cdots e_{BL_i}$ are decrypted and modified in order to replace the transitions to the dynamically located nodes $l_i \cdots L_i$. For this purpose, the following operations are performed:

    (a) Let r_c be one of the symmetric keys included in $[r_{bl_i} \cdots r_{BL_i}]$. Then r_c is used to decrypt e_c. Thus, $(S_o(id, c, m_c, type_c), [t_{c\,a_c} \cdots\ t_{c\,A_c}])$ is obtained (see equation 4.6).

    (b) For each transition $t_{cy} \in \{t_{c\,a_c} \cdots\ t_{c\,A_c}\}$ such that $y \in \{l_i \cdots L_i\}$, t_{cy} is rebuilt as follows:

    $$t_{cy} \;=\; y, h_y, P_y(S_o(id, r_y, h_i), S_i(id, h_y)) \tag{4.9}$$

    where $S_o(id, r_y, h_i)$ is included in the list $[S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)]$ that was obtained in step 6. h_y is the address of platform $p_y \in \{p_{l_i} \cdots p_{L_i}\}$, which was discovered in step 9. Note that, in this case, i is the *discoverer* node associated with node y and, therefore, $S_i(id, h_y)$ is signed using the private key of p_i.

    (c) The new transitions $t_{c\,l_i} \cdots t_{c\,L_i}$ are substituted for the previous ones inside e_c.

11. The transition $t_{i\,a_i}$ obtained in step 6 is placed at beginning of the protected itinerary, thus replacing the one previously obtained in step 1.

12. The agent is migrated to platform p_{a_i}.

The steps described above show how the transitions to dynamically located nodes are rebuilt when the agent is visiting their corresponding *discoverer* node. Thus, the agent can visit these dynamically located nodes performing the same operations carried out in any other node.

The lists $[S_o(id, r_{l_i}, h_i) \cdots S_o(id, r_{L_i}, h_i)]$ and $[r_{bl_i} \cdots r_{BL_i}]$, which are required to rebuild the protected itinerary, are only available to the *discoverer* node. Therefore, the information of dynamically located nodes is kept confidential at all times. However, the *discoverer* node must be associated with a trusted platform, in order to ensure that the information available to this platform is not maliciously used. Otherwise, a dishonest platform could easily alter the code and data assigned to dynamically located nodes and their direct predecessors.

In order to exemplify the management of a protected itinerary, the following steps show how the protected itinerary of equation 4.8 is handled during the agent execution.

1. The agent is migrated to platform *A*. The transition t_{01} is decrypted using *A*'s private key, and $S_o(id, r_1, o), S_o(id, A)$ is obtained.

2. The signature of $S_o(id, r_1, o)$ is verified.

3. The signature of $S_o(id, A)$ is verified.

4. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

5. A is compared with the current platform's address, in order to verify that the current platform is really the one assigned to node 1.

6. e_1 is decrypted using r_1, and $S_o(id, 1, m_1, seq), [t_{12}]$ is obtained.

7. The signature of $S_o(id, 1, m_1, seq)$ is verified.

8. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

9. The agent executes m_1.

10. The transition $t_{12} = 2, A, P_1(S_o(id, r_2, o), S_o(id, A))$ is placed at the beginning of the protected itinerary. Thus, the protected itinerary shown in equation 4.8 is replaced

by:

$$I = t_{12}, \Big[E_{r_1}(\cdots), \cdots, E_{r_6}(\cdots) \Big] \tag{4.10}$$

11. Node 2 has the *unchanged location* property set. As a result, the next platform of the itinerary, which can be obtained from t_{12}, is *A* again. Therefore, no migration is performed because the agent is already in *A*.

These steps are now repeated to execute the task assigned to node 2. In this case, however, step 9 requires an additional operation because node 2 is a *loop*:

9. The agent executes m_2. As the current node type is *loop*, m_2 includes a method that evaluates the loop condition and chooses which node must be visited next: 3 or 6.

Let us assume that the agent decides to visit node 3, thus entering the *loop* subitinerary. When the agent is migrated to platform B, the following steps are performed:

1. The transition t_{23} is decrypted using *B*'s private key, and $S_o(id, r_3, o), S_o(id, B)$ is obtained.

2. The signature of $S_o(id, r_3, o)$ is verified.

3. The signature of $S_o(id, B)$ is verified.

4. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

5. B is compared with the current platform's address in order to verify that the current platform is really the one assigned to node 3.

6. e_3 is decrypted using r_3, and the following information is obtained:

$$(S_o(id, 3, m_3, discoverer, [S_o(id, r_5, B), S_o(id, r_2, B)], [r_4, r_5]), [t_{34}])$$

7. The signature of $S_o(id, 3, m_3, discoverer, [S_o(id, r_5, B), S_o(id, r_2, B)], [r_4, r_5])$ is verified.

8. The agent identifier id obtained in the previous step is compared with that obtained in step 2. If they are not equal, the agent execution is discarded.

9. The agent executes task m_3. As the current node type is *discoverer*, m_3 includes a method that determines where the tasks of nodes 5 and 2 will be executed. Let us assume that the agent assigns nodes 5 and 2 to platform *E*.

10. Using the symmetric keys included in the list $[r_4, r_5]$, the entries e_4, e_5 are decrypted and modified in order to replace the transitions to the dynamically located nodes 5 and 2. For this purpose, the following operations are performed:

    (a) e_4 is decrypted using r_4, and $(S_o(id, 4, m_4, seq), [t_{45}])$ is obtained.

    (b) t_{45} is rebuilt as follows:

    $$t_{45} \;=\; 5, E, P_5(S_o(id, r_5, B), S_3(id, E)) \tag{4.11}$$

    Note that $S_o(id, r_5, B)$ has been obtained in step 6. S_3 is a digital signature using *B*'s private key, and P_5 is an asymmetric encryption function using the public key of platform *E*.

    (c) The entry e_4 is re-encrypted, replacing the previous t_{45} with the newly generated in the previous step.

    These same steps are now repeated using r_5 to decrypt e_5, and the previous t_{52} is replaced by $t_{52} \;=\; 2, E, P_2(S_o(id, r_2, B), S_3(id, E))$.

11. The transition t_{34} is placed at the beginning of the protected itinerary.

12. The agent is migrated to platform *C*.

The protected itinerary is managed in platform *C* by performing the same steps previously explained for node 1. Note that, once the transitions t_{45} and t_{52} have been rebuilt in node 3, the agent can visit nodes 5 and 2 as if they were not dynamically located nodes. The agent will eventually exit the loop (node 2) and will finish its execution on platform *D*.

As can be seen, the information associated with nodes 5 and 2 always remains confidential and immutable. e_5 and e_2 are always protected with their corresponding symmetric keys—r_5 and r_2—which are protected inside e_3.

Only platform *B* has access to $S_o(id, r_5, B)$ and $S_o(id, r_2, B)$, which are needed to rebuild the transitions t_{45} and t_{52}. Moreover, only platform *B* has access to the symmetric keys r_4 and r_5 that enable the modification of e_4 and e_5. Consequently, only platform *B* is allowed to determine where the tasks of nodes 5 and 2 will be executed, and to rebuild the itinerary accordingly. Obviously, platform *B* must be trusted by the owner, and the proposed protocol assumes that this platform will not modify the contents of e_4, e_5 or e_2.

It is worth noting that the contents of e_5 and e_2 are signed by the agent owner. Consequently, the platforms dynamically assigned to nodes 5 and 2 are not allowed to modify these entries of the protected itinerary. Otherwise, the following attack would be possible: Let us assume that nodes 5 and 2 were associated with platform *Z* in a given loop iteration, and then they were associated with platform *E* in the next loop iteration. As both platforms *Z* and *E* are accessing the same entries of the protected itinerary—e_5 and e_2—, platform *Z* could easily modify the tasks m_5 and m_2 executed later on platform *E*. However, this attack is not possible because these tasks are signed by the agent owner.

The operations previously described to construct the protected itinerary, together with the steps described above to manage this protected itinerary, comprise the complete protocol aiming at protecting flexible explicit itineraries for free-roaming agents.

4.3.3 Discussion

The protocol presented in this chapter ensures the integrity, authenticity and confidentiality of dynamic itineraries. In order to achieve these objectives, the protocol is based on introducing trusted platforms into the agent's itinerary.

The platforms assigned to *discoverer* nodes are allowed to access the information associated with dynamically located nodes, as well as the information of their direct predecessors. Therefore, the programmer must be confident that these platforms are not going to

alter the code or data associated with these nodes.

The agent itinerary, however, can be defined in such a way that all dynamically located nodes are placed immediately after their corresponding *discoverer* node. This allows *discoverer* nodes to rebuild the transitions to dynamically located nodes without needing to decrypt the entries of their predecessors.

Including trusted platforms in the itinerary might be difficult for some specific applications in which each host of the agent's itinerary was supposed to be potentially malicious. However, this is not the case of most applications. Moreover, several strategies can be used to reduce the number of trusted platforms that must be introduced in the itinerary. First of all, all *discoverer* nodes can be associated with the same trusted platform (e.g. the agent home platform). Secondly, the agent itinerary can be defined in such a way that the discovery of new platforms and the reconstruction of the itinerary are conducted in different platforms. Thus, the agent always migrates to the same trusted platform to rebuild the itinerary, regardless of where the new platforms are discovered.

The proposed protocol does not support assigning multiple platforms to the same dynamically located node. This would involve using the same symmetric key to encrypt several itinerary entries. As a result, this would allow platforms to decrypt and modify the entries corresponding to other platforms discovered at runtime. Additionally, rebuilding the transition to a node of any type other than *sequence* would involve replicating the sub-itinerary associated with this node. As a result, this might imply replicating most of the entries of the protected itinerary, and changing the successors associated with some nodes.

As an example, this problem could arise in itineraries such as the one shown in figure 3.9, where the *dynamic location* property is set on an *if* node. In this case, assigning two different platforms to node 2 would involve rebuilding the agent's itinerary as shown in figure 4.4.

As this figure shows, the reconstruction performed in node 2 would involve duplicating four entries of the protected itinerary, and changing the successor associated with node 4. For all these reasons, assigning multiple platforms to the same dynamically located node is not supported by the proposed protocol.

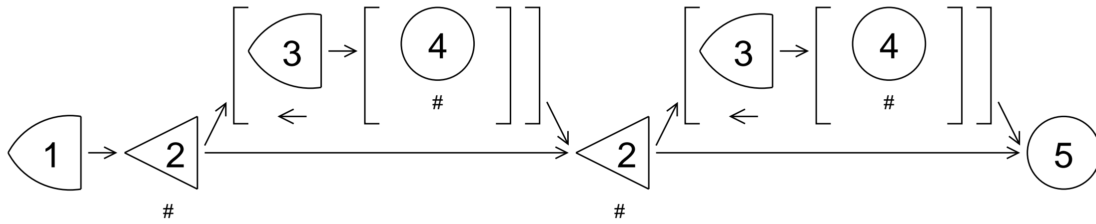

Figure 4.4: Reconstruction of the itinerary of figure 3.9 when two platforms are discovered for node 2

4.4 Implementation

The proposed protocol has been implemented and experimentation work has been carried out in order to prove its viability. The Java-based Jade platform [BCPR03] has been used as the agent execution environment.

Agents have been implemented following an agent-driven approach, as proposed by Ametller *et al.* in [ARO04]. This involves providing mobile agents with a code that is executed as soon as the agent is initiated on all platforms, and which deals with the management of the protected itinerary. We refer to this code as the *control code*. As mentioned earlier in previous chapters, the agent-driven approach has several advantages over the platform-driven one. First of all, platforms can handle the execution of any agent in the same way, regardless of how the agent is internally structured and protected. Moreover, the agent control code can be easily reused in different applications, and it can implement complex itinerary management algorithms that involve, for instance, rebuilding the agent's itinerary at runtime.

According to [ARO04], agents can decrypt their itinerary data using platforms' private keys without having direct access to these private keys. This decryption is performed using a call to a public decryption function provided by platforms. This public decryption function verifies that the decrypted data really belongs to the requesting agent. Thus, agents are not allowed to decrypt itinerary information stolen from other agents. A detailed description of Ametller *et al.*'s protocol is provided in section 2.

The agent-driven implementation of the proposed protocol has been carried out through

the following steps: First, generation of a random pair of cryptographic keys (k_p, k_s). Second, generation of the agent's control code, which contains the public key k_p as a compile-time constant that will never change during the lifetime of the agent. Third, generation of the protected itinerary.

With regard to the generation of the agent's control code, the implementation performs the steps described in section 4.3.2, so that the agent is able to manage its own itinerary and protection mechanisms. Some steps, however, have been modified slightly in order to adapt the implementation to the requirements of the agent-driven approach. The following are the changes performed:

- In step 1, the transition placed at the beginning of the protected itinerary must be decrypted using the platform's private key. In order to do so, the control code calls the platform's public decryption function. Additionally, the control code verifies the signature of the data returned by this function, using the public key k_p as described in [ARO04]. If this verification fails, the control code aborts the agent execution.

- When the agent is visiting a *discoverer* node, the transition to every dynamically located node must be rebuilt (see equation 4.9). The control code rebuilds this transition as follows:

$$t_{cy} = y, h_y, P_y(S_{k_s}(S_o(H(C), r_y, h_i), S_i(H(C), h_y)), H(C)) \tag{4.12}$$

  where $H(C)$ is a cryptographic hash of the agent's control code. As can be seen, the agent identifier id is computed as a hash of the agent's control code. Additionally, this hash value is appended to the data that is then encrypted by P_y.

With regard to the generation of the agent's protected itinerary, the implementation undertakes the operations described in section 4.3.1. However, the following modifications are required to allow the control code to use the platforms' public decryption function.

First of all, all public key encryptions are performed in such a way that the data to be encrypted are previously signed using k_s. Additionally, a hash of the agent's control code

$H(C)$ is appended to the result of this signature. As an example, the transition shown in equation 4.1 is computed as follows:

$$t_{ij} \;=\; j, h_j, P_j(S_{k_s}(S_o(id, r_j, h_{d_j}), S_{d_j}(id, h_j)), H(C)) \tag{4.13}$$

Attaching a hash of the agent's control code to the encrypted data allows platforms to verify that the control code remains unaltered. Signing the data to be encrypted with k_s prevents attackers from reusing transitions of previously executed agents. Performing this signature is not essential for the security of the proposed protocol because the unique agent identifier $H(C)$ is already included for this purpose. However, it has been implemented for the sake of consistency with the protocol proposed in [ARO04].

The second modification to the protocol described in section 4.3.1 involves constructing the entries e_i associated with *discoverer* nodes as follows:

$$\begin{aligned} e_i \;=\; E_{r_i}\Big(\, S_o\big(\, H(C), i, m_i, discoverer, [S_{k_s}(S_o(H(C), r_{l_i}, h_i)) \cdots \\ \cdots S_{k_s}(S_o(H(C), r_{L_i}, h_i))], [r_{bl_i} \cdots r_{BL_i}] \,\big), [t_{i\,a_i}] \,\Big) \end{aligned} \tag{4.14}$$

As can be seen, every element of $[S_{k_s}(S_o(H(C), r_{l_i}, h_i)) \cdots S_{k_s}(S_o(H(C), r_{L_i}, h_i))]$ is signed using k_s. These signatures, together with the hash of the control code, allow the agent to rebuild the transitions to the dynamically located nodes, as shown in equation 4.12.

All these modifications to the way the protected itinerary is constructed and managed allow the agent to handle its own protection mechanisms. As mentioned earlier, protecting agents using an agent-driven approach is preferable to using a platform-driven one, for it enables developers to customise the set of implemented security mechanisms depending on the specific requirements of their applications. In addition, platforms are not forced to support all existing security protocols, thus facilitating maintenance as well.

4.4.1 Simulation and tests

In order to prove the viability of the proposed protocol, experiments have been performed simulating a simple mobile agent-based application. This application implements a hotel search and reservation system.

The system allows an individual to find the cheapest hotel in a given destination, taking into account the user preferences with regard to room facilities and guest services. The application provides a graphical user interface where the search criteria are defined. After defining the search criteria, a mobile agent is started that, first of all, queries a remote hotel search engine to obtain a list of the five cheapest hotels in the destination. The agent then visits each one of these hotels and checks their room availability for the desired dates, their room facilities, services, etc. In addition, the agent can also negotiate a special discount for long stays.

Once all hotels have been visited, the agent returns to the its home platform, and decides whether there is any offer that fits the user's needs and budget. If there is none, the agent performs another query in order to find the next five cheapest hotels, and the whole process is started again. Otherwise, if the agent finds an acceptable offer, it proceeds with the reservation process. The agent's itinerary defined for the implementation of this example application is shown in figure 4.5.

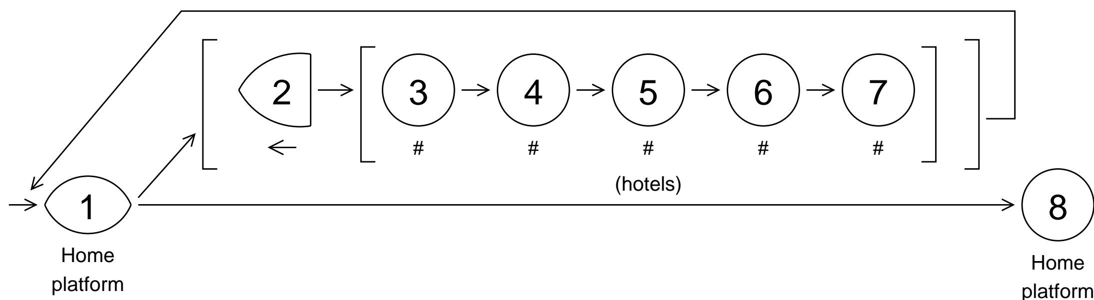

Figure 4.5: Itinerary of an agent implementing a hotel reservation system

By implementing this application using mobile agent technology, the search and negotiation process is automated, and the application can be used in devices with slow or intermittent connections, such as mobile phones, PDAs, etc.

Protecting the explicit itinerary using the protocol presented in this chapter keeps potentially malicious hotels from modifying the code executed on other hotels. Preventing modifications to the code intended for other hotels is of utmost importance in this application. Otherwise, it would be possible for a hotel to modify the agent's code in such a

way that the agent itself increased the offers obtained from other hotels. The following expression shows the initial protected itinerary:

$$\begin{aligned} I \ = t_{01}, \Big[\ & E_{r_1}(S_o(H(C), 1, m_1, loop), [t_{12}, t_{18}]), \ E_{r_2}(S_o(H(C), 2, m_2, discoverer, \\ & [S_{k_s}(S_o(H(C), r_3, HP)), S_{k_s}(S_o(H(C), r_4, HP)), S_{k_s}(S_o(H(C), r_5, HP)), \\ & S_{k_s}(S_o(H(C), r_6, HP)), S_{k_s}(S_o(H(C), r_7, HP))], [r_2, r_3, r_4, r_5, r_6]), [t_{23}]), \\ & E_{r_3}(S_o(H(C), 3, m_3, seq), [t_{34}]), E_{r_4}(S_o(H(C), 4, m_4, seq), [t_{45}]), \\ & E_{r_5}(S_o(H(C), 5, m_5, seq), [t_{56}]), E_{r_6}(S_o(H(C), 6, m_6, seq), [t_{67}]), \\ & E_{r_7}(S_o(H(C), 7, m_7, seq), [t_{71}]), E_{r_8}(S_o(H(C), 8, m_8, seq), []) \Big] \end{aligned}$$

where HP denotes the agent's home platform, and

$$\begin{aligned} t_{01} &= 1, HP, P_1(S_{k_s}(S_o(H(C), r_1, o), S_o(H(C), HP)), H(C)) \ , \\ t_{12} &= 2, HP, P_1(S_{k_s}(S_o(H(C), r_2, o), S_o(H(C), HP)), H(C)) \ , \\ t_{23} &= 3, blank \ , \ t_{34} \ = \ 4, blank \ , \ t_{45} \ = \ 5, blank \ , \ t_{56} \ = \ 6, blank \ , \ t_{67} \ = \ 7, blank \ , \\ t_{71} &= 1, HP, P_1(S_{k_s}(S_o(H(C), r_1, o), S_o(H(C), HP)), H(C)) \ , \\ t_{18} &= 8, HP, P_8(S_{k_s}(S_o(H(C), r_8, o), S_o(H(C), HP)), H(C)) \end{aligned}$$

It is worth noting that, in order to guarantee the confidentiality and integrity of previously collected offers, the proposed protocol must be used in combination with some other mechanism aimed at protecting the agent's computational results (such as [MS03]).

This example application shows an environment where platforms are discovered dynamically and might be interested in modifying the code executed by the agent for their own benefit. In order to validate the proposed protocol, this application has been simulated by introducing several malicious platforms into the agent's itinerary. These malicious platforms acted as dishonest hotels trying to access or modify the agent's code intended for other platforms. As expected, none of the attacks succeeded because every platform-specific code and data was encrypted using a symmetric key that was only available to the intended platform.

The experiments performed also compared the execution times of a protected agent with an unprotected one, in order to determine if the proposed protection protocol increased the execution times to overly high values. The agent was given an itinerary with different

number of hotels to visit, and a number of tests were conducted to determine whether or not the protection resulted in an exponential increase of the execution times. The evaluation setup used to make the tests was made up by 7 computers with 2 GHz Intel Pentium(R) IV processors and 256 MB RAM memory each. These computers were connected in a laboratory to a 100 Mbps Ethernet LAN. Table 4.1 shows the resulting execution times.

	Num. of iterations:	2	4	6	16
Unprotected agent	Exec time:	12416	24500	36644	96986
	Time/node:	776	816	832	850
Protected agent	Exec time:	19096	37166	54956	144836
	Time/node:	1193	1238	1249	1270
	Increase:	53.8%	51.7%	49.9%	49.3%

Table 4.1: Execution times (ms) for agents with protected and unprotected itineraries

As table 4.1 shows, the execution time increases linearly with the number of hotels visited. The execution time of a protected agent is approximately 50% higher than that of an unprotected agent. However, this increase depends largely on the specific application implemented. In our simulation, the time spent by the agent handling protection mechanisms significantly impacted the overall execution time. Other applications could require the agent to execute a lot more time-consuming tasks, and the time spent handling protection mechanisms would be negligible.

The increase in the execution time is readily understandable if we take into account the complexity of the cryptographic protection protocol presented in this chapter. The overhead introduced by the execution of this protocol is completely acceptable for the simulated application. However, this might not be the case for other applications, and the trade-off between computational cost and security level should be weighed.

4.5 Conclusions

Previous protocols aimed at protecting the agent's itinerary only allowed agents to travel to platforms known beforehand. As a result, these approaches limited the mobile agent's ability to migrate at will. Unfortunately, most mobile agent applications are devised using dynamic itineraries. Therefore, this limitation was a serious impediment to the deployment of mobile agent technology, for most of the flexibility provided by this technology had to be sacrificed.

In this chapter, a novel protocol has been presented for the protection of both static and dynamic itineraries. The protocol is based on adding trusted platforms to the agent's itinerary. This is not a limitation as it could seem at first sight because trusted platforms are usually found in real applications. By using the public keys of these trusted platforms, the information associated with dynamically located nodes is kept confidential. The protocol protects dynamic itineraries from tampering, impersonation and disclosure attacks, thus providing a balanced trade-off between security and flexibility.

An example application has also been presented, showing how the proposed protocol can be used to implement a hotel reservation system. In this application, agents are used to find the best offer for a hotel room. Hotels are discovered dynamically, and the itinerary is always rebuilt on a trusted platform: the agent's home platform. Experiments have been conducted using this implementation, and these experiments have proved that dynamic itineraries are effectively protected, and agents are executed in reasonable times.

The protocol presented in this chapter attains the objectives initially established, but it does not protect agents against replay attacks. In these attacks, the agent is captured in a platform and it is repeatedly sent to its next destination. This causes the agent to re-execute part of its itinerary. The next chapter is devoted to present a protocol to withstand these attacks.

Chapter 5

External replay attack protection

This chapter presents a protocol for the protection of mobile agents against agent replay attacks. Agent replay attacks can be classified into two different categories [Yee03]:

Internal replay attacks: These occur when the agent is forced to execute repeatedly on a single platform using different inputs, aiming to obtain different responses and analyse its behaviour.

External replay attacks: These are performed by malicious platforms by resending the agent to another platform, thus making the agent reexecute part of its itinerary.

Internal replay attacks are impossible to avoid because the platform has complete control over the execution, and can always reset the agent to its arrival state. On the contrary, external replay attacks can be avoided if platforms keep a record of previously executed agents.

The problem of current solutions [Yee03, WSUK00, WSB98, LSL00, Sue03, MB03, CLSY06, CAOR$^+$05] against external replay attacks is that they do not allow an agent to be executed *n* times on the same platform, especially if *n* is determined at runtime. However, the agent's itinerary often contains roundtrips that require the same platform to be visited several times. Thus, current solutions force programmers to sacrifice some of the inherent flexibility in mobile agent itineraries.

In order to enhance security and flexibility, this chapter presents a solution based on *authorisation entities*. The advantage is that these entities are entitled to generate new identifiers for the agent, thus allowing the repeatable migration of the agent to the same platform. Therefore, the proposed solution allows programmers to develop secure mobile agent applications and still maintaining the agents' intrinsic flexibility.

5.1 Related work on agent replay attacks

Replay attacks [Syv94] have traditionally been considered as a form of network attack. They are based on capturing some of the messages exchanged between two entities and sending them again at a later time. These attacks are usually performed during authorisation or key agreement protocols, in order to perform, for example, masquerade attacks.

Traditional mechanisms to prevent replay attacks are based on using nonces, timestamps, session tokens, or any other information that allows entities to bind their messages to the current protocol run [Aur97]. For example, an HTTP exchange between a browser and a server may include a session token that uniquely identifies the current interaction session. The token is usually sent as an HTTP cookie, and is calculated as a hash of the session data, user preferences, and so on.

Mobile agent systems are also exposed to traditional replay attacks. Any communication between two agents, or two platforms, or an agent and a platform is exposed to this kind of attacks. Mechanisms like the ones previously mentioned (based on nonces and session tokens) can be used to withstand these attacks.

In addition to traditional replay attacks, mobile agent systems are also exposed to *agent* replay attacks. Agent replay attacks are not based on replaying a message sent across the network, but on reexecuting an agent that has already been executed on a platform. In addition, these attacks are usually performed by platforms which are part of the agent's itinerary, and they are harder to prevent in this case. When agent replay attacks are performed by external platforms, they can easily be detected using mechanisms designed to prevent traditional replay attacks. Agent replay attacks can be divided into two classes:

internal and external replay attacks.

Internal replay attacks occur when a dishonest platform repeatedly runs an agent with the same or different set of inputs each time. A platform might execute an agent multiple times in order to understand its behaviour, or until the desired output is obtained. This type of attack is also known as *blackbox testing* [Hoh98], and is usually performed when the agent's code has been protected using some obfuscation technique.

This kind of attacks is performed inside a single platform, and cannot be externally observed by any other entity. Even if the agent tried to record all its actions on an external monitoring service, the execution environment could still interfere with these external communications, and route the messages to an incorrect recipient, or alter the contents of the messages, and so on. Moreover, agent attempts to store their state information in a secure external entity could be easily bypassed by malicious platforms altering the agent execution. In practise, internal replay attacks are impossible to prevent or detect [Yee03].

External agent replay attacks occur when a dishonest platform propagates an agent to a remote host, without this migration being defined in the agent's itinerary. This kind of attack is especially difficult to deal with, for it is difficult to distinguish between a legal migration of the agent to its next destination and a replayed migration that the agent did not intend to do. For example, supposing that the agent's itinerary includes a migration from platform A to B, then platform A is authorised to send the agent to platform B, and the agent is also authorised to be executed on platform B. As a result, no authentication mechanism can be used to prevent platform A from maliciously resending the agent to platform B multiple times. This kind of attack can be carried out against a shopping agent, for example, in order to generate unintended purchases.

In order to provide a solution to this problem, [Yee03] suggests considering a replay attack as an illegal state transition. Every platform within the itinerary implements a state transition inconsistency detection (STID) algorithm, which is able to determine when a migration from one platform to another is an illegal transition. The problem of this approach is that platforms must be aware of all possible illegal state transitions for every agent execution. In addition, illegal state transitions may be mistakenly identified if the agent is

performing a loop in which the same platform is visited repeatedly.

In [Vig98], a protocol is presented that allows replay attacks to be detected upon completion of an agent execution. The protocol is based on recording the agent execution on each platform. Platforms must keep a log of the operations performed by every agent, so that agent owners can detect any malicious manipulation of the agents' itineraries. This technique suffers from some drawbacks, such as the size of the logs that have to be maintained. In general, detecting replay attacks after they have been performed is useless in many occasions. For example, if the agent is buying a product, detecting a replay attack that leads to buying the product more than once is usually inappropriate, especially if the price of the product is too low to justify future legal actions.

Other work [WSUK00, WSB98, LSL00, Sue03, MB03, CLSY06] on mobile agent protection against malicious platforms suggests the use of trip markers for preventing replay attacks. A trip marker is an agent identifier that must be stored by platforms, which can be used to detect and prevent future attempts of reexecuting the same agent. Again, the problem of these solutions is that they do not take into account the case where the agent's itinerary includes one or more platforms that must be visited more than once. As a result, a legal reexecution of the agent on the same platform can be misinterpreted as a replay attack.

These issues were identified by [CAOR+05], who proposed a solution based on including counters inside the agent's trip marker. Every platform is assigned a different counter, which indicates the maximum number of times that an agent can be executed on a platform. Platforms keep a record of which agents have been executed, and the number of times they have done so. Before starting the execution of an agent, platforms check that the number of times that the agent has been previously executed does not exceed the number of allowed executions stored in the agent's trip marker.

The problem of this approach, however, is that the number of times that a given platform can be visited must be known in advance, specifically when the agent's itinerary is created, so that this information can be introduced inside the agent's trip marker. Consequently, this approach does not allow the agent to dynamically decide on the number of times a given

platform will be visited.

Preventing the agent from being executed more than once has also been referred in the literature as ensuring the *exactly-once execution property* [SRM98]. This property is usually considered when designing fault-tolerant mechanisms for mobile agents. Ensuring the exactly-once execution property implies that, when the agent is launched to do a certain task: first, the task will be eventually executed, regardless of host or communication failures that may occur; and second, the task will not be performed more than once.

Solutions presented to ensure the exactly-once execution property are based on using external entities that monitor the execution of the agent. When a failure prevents the agent from continuing its itinerary, another agent is launched to resume the execution at the point the original agent left it. The problem of these solutions is that the communications between the agent and the monitoring system lead to considerable traffic overheads. In addition, these protocols severely reduce the agent's autonomy because the agent has to constantly interact with the monitoring entity. Thus, they sacrifice one of the major advantages associated with the use of mobile agent technology.

To summarise, no solution presented so far against replay attacks allows an agent to dynamically determine the number of repeated executions of the same task on one or more platforms along its itinerary. Considering that one of the greatest appeals of mobile agents is their dynamism and flexibility, hard-coding the number of possible migrations to a platform in advance can be a serious impediment to the implementation of real-life applications. The next section describes the proposed protocol which solves this problem.

5.2 Replay attack protection

Protecting mobile agents against all kinds of replay attacks becomes a serious and complex problem to solve. The simplest case is where the itinerary does not contain any loops in its itinerary, that is, the agent does not need to repeatedly execute the same task on one or more platforms. In a loop-free scenario, if the agent has to migrate to a certain platform several times, it will do so in different nodes of its itinerary, which means that it will be

executing different tasks. Replay attacks are easy to prevent in this case. Platforms can store an identifier of the node visited by the agent, together with the agent's trip marker. Figure 5.1 shows an example of this kind of itinerary. In this case, platform *B* is visited in two different nodes of the itinerary: nodes 2 and 4.

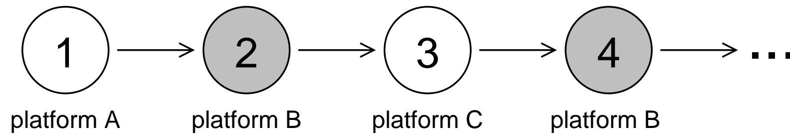

Figure 5.1: Simple itinerary where the same platform is visited twice

The most difficult case is where the agent has to visit the same node several times, especially if this number of times is determined at runtime. Figure 5.2 shows an example of this kind of itinerary.

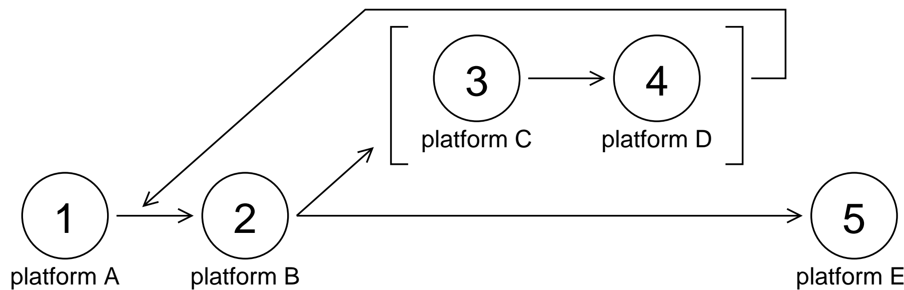

Figure 5.2: Itinerary containing a loop with three platforms that are visited repeatedly

As shown in the figure, the itinerary contains a *loop* node, which is associated with platform *B*. The detection of replay attacks is especially difficult, for example, when platform *C* resends the agent to platform *D*. How can platform *D* determine whether the new reexecution is a replay attack or a new iteration of the loop?

Before presenting the proposal for external replay attack prevention, the next section describes the requirements that any valid solution should fulfil.

5.2.1 Requirements of the solution

First of all, different agents must have different identifiers or trip markers, even if they perform exactly the same tasks. This also implies that any new instance of the same agent must carry a different trip marker.

Secondly, a valid solution to replay attacks has to focus on their prevention, rather than just on their after-the-fact detection.

Thirdly, platforms must store the trip markers of the agents previously executed, along with an identifier of the node visited by the agent, to allow agents to revisit the same platform in different nodes of its itinerary.

Finally, a valid solution to replay attacks should not involve the interaction of the agent with external entities. This allows the agent to run autonomously, without depending on the control or interaction with any monitoring service.

5.2.2 The protocol

The proposed protocol for the protection against replay attacks is based on using trip markers and authorisation entities. A trip marker is an *authorisation* that allows the agent to visit a certain set of nodes. Each node has an authorisation entity associated with it, and this entity is the only one entitled to generate the trip markers (authorisations) required to visit that node. Platforms must store the trip markers of previously executed agents, so that no trip marker can be used more than once to visit the same node.

The steps required to create a replay-safe mobile agent, and then the operations carried out to detect and prevent replay attacks are presented in this section. The combined actions comprise the protection protocol which tackles replay attacks in mobile agent environments.

In order to create a replay-safe mobile agent, the programmer must define the set of nodes that comprise the agent's itinerary, assigning the following information to each one of them:

- a node identifier

- a task
- a node type
- an authorisation entity
- an authorisation node
- a set of next destinations

Although different node types can be used to create the explicit itinerary, for the purposes of the proposed protocol only two types are considered: the *sequence* and the *loop*. Other node types can be introduced into the agent's itinerary, but they are treated in the same way as *sequence* nodes. Therefore, they will not be taken into account in the definition of the proposed protocol.

The programmer must assign an authorisation entity and an authorisation node to every node. The authorisation node is the node where the agent's trip marker must be generated. This implies that the agent's trip marker is only valid if it has been generated in the appropriate authorisation node. The authorisation entity is the corresponding platform or individual that must generate and sign the trip marker. In some cases, as explained next, a node can have no authorisation node associated with it: when the node is visited using an authorisation generated by the agent owner.

The authorisation node that is assigned to each node depends on whether or not the node is located inside a loop. If a node is located in the subitinerary of a *loop* node, then its authorisation node is the *loop* node itself. The corresponding authorisation entity is the platform assigned to the *loop* node. In all other cases, the node has no associated authorisation node, and the corresponding authorisation entity is the agent owner. As an example, figure 5.3 shows which authorisation entity and node are associated with each node of a complex itinerary. This itinerary contains two loops, one nested inside the other.

The itinerary shown in figure 5.3 has an outer loop starting at platform *A*. In this platform, the agent decides how many iterations of the outer loop have to be performed. If

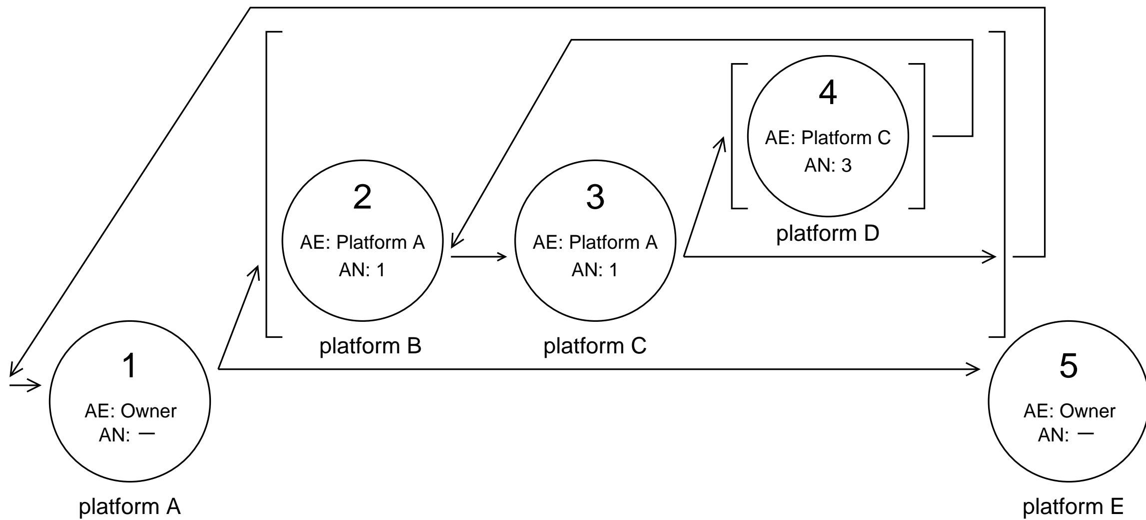

Figure 5.3: Authorisation entities (AE) and authorisation nodes (AN) assigned to the nodes of a complex itinerary

the agent decides to enter this loop, it will visit platform *B* and then platform *C*, which is the starting point of the inner loop. The inner loop contains just one node, associated with platform *D*. When the agent exits the inner and the outer loop, it migrates to platform *E*, where it reaches the end of its itinerary.

The itinerary shown in figure 5.3 has three different authorisation entities: the agent owner, platform *A* and platform *C*. Platform *A* is the starting point of the outer loop, and thus it is the authorisation entity of nodes 2 and 3. On the other hand, platform *C* is the authorisation entity of node 4 because this node belongs to the inner loop. The remaining nodes have no associated authorisation node, and their authorisation entity is the agent owner.

The proposed protocol requires authorisation entities to be trusted by the owner. Otherwise, no protection mechanism could be used to prevent a malicious platform from corrupting the agent execution, so the number of loop iterations could be easily altered as well. It is important to note that this protocol makes the same assumptions with regard to trusted platforms as the protocol presented in chapter 4. This means that if the programmer trusts a certain platform, then it is assumed that such platform will not subvert the agent execution.

In the example of figure 5.3, platform *A* and platform *C* must be trusted by the programmer, and thus the protocol assumes that these platforms will execute the tasks of nodes 1 and 3 correctly.

Once the aforementioned information has been assigned to every node, the programmer must secure the itinerary using a protection protocol. This protocol must satisfy the following properties:

- It must not allow platforms to access or modify any part of the itinerary which is intended for other platforms.
- It must not be possible to introduce new nodes in the itinerary.
- It must not be possible to traverse the itinerary nodes in an order different from the order initially defined.
- Every itinerary node must be uniquely bound to the agent it belongs to. As a result, it must not be possible to reuse any part of the agent's itinerary in a different agent.
- It must support flexible itineraries, which allow the agent to make decisions about its travel plan at runtime.

In order to define the proposed protocol, it will be assumed that a unique agent identifier is used to bind the itinerary nodes with the agent. This agent identifier can be simply a timestamp attached to a big random number, or any other information that uniquely identifies each agent instance. An example of a protocol that guarantees all the properties mentioned above is the one presented in chapter 4.

After defining all itinerary nodes and securing them using an itinerary protection protocol, the programmer must generate a trip marker for the agent. Agent trip markers must always contain the following information:

Agent identifier: This is the same identifier included in the protected itinerary, which ensures that any given agent instance can be uniquely identified.

Authorisation node: This is the identifier of the node where the trip marker is generated.

Expiry date: This is the date after which the trip marker can no longer be used. It allows platforms to remove expired trip markers from their tables, which is usually convenient but has no real effect on the protocol.

Loop counter: This counter is incremented by one unit every time the agent has to start a new loop iteration.

In the trip marker initially created by the programmer, the authorisation node is not set because the programmer has no associated itinerary node. This is consistent with what has been specified in the itinerary nodes. When the authorisation entity of a node is the agent programmer, then the node has no associated authorisation node.

The trip marker initially created must be signed by the agent programmer or owner. This trip marker must then be placed at the top of the agent's trip marker stack. As will be seen later, the trip marker stack stores trip markers previously used by the agent during its execution. The top of this stack always contains the trip marker currently used. The trip marker used in the first itinerary platform must always be signed by the owner, for this is the authorisation entity assigned to the first itinerary node.

Thus far, the steps required to create a replay-safe mobile agent have been described. Figure 5.4 shows a representation of the components of an agent resulting from this protection. Next, the operations conducted by platforms to prevent a mobile agent from being replayed are presented.

In order to start the execution of an agent, the information of the current node must be extracted from the protected itinerary. The operations carried out to do this extraction depend on the itinerary protection protocol used. In most cases, this implies using the platform's private key, which ensures that no other platform can access the contents of the current node. By extracting the current node from the protected itinerary, the following information is obtained: current node identifier, agent identifier, task, type, next platforms, authorisation entity and authorisation node.

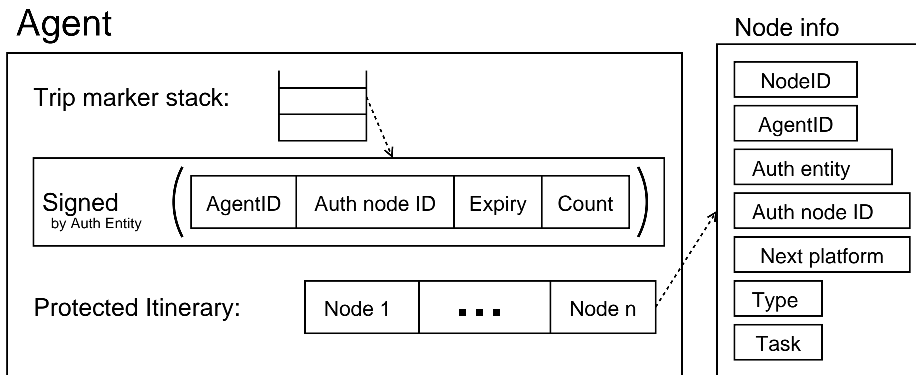

Figure 5.4: Components of an agent protected against replay attacks

In addition, the agent's trip marker must be retrieved from the top of the agent's trip marker stack. The trip marker contains the agent identifier, authorisation node, expiry date and loop counter.

Platforms must maintain a table with the trip markers of previously executed agents. Along with the trip markers, these tables must contain the node identifiers associated with the tasks executed by the agents. To effectively prevent replay attacks, platforms should only remove entries from their tables once they have expired.

In order to ensure that the agent is not being replayed, the agent's trip marker must be used to carry out some verifications, which depend on whether the current node type is *sequence* or *loop*. In case that the node type is *sequence*, the set of checks to perform is summarised in the algorithm of figure 5.5. The most relevant lines of this algorithm are described in detail below:

Line 1: The public key of the current authorisation entity is obtained from a public key server. With this public key, the trip marker signature is verified.

Line 2: The authorisation node extracted from the current node is used to verify that the trip marker has been generated in the appropriate itinerary node. This involves checking that the authorisation node included in the trip marker is equal to that extracted from the current node.

```
 1: verify trip marker (TM) signature
 2: check current auth. node = TM auth. node
 3: check agent's identifier
 4: check current date < expiry date
 5: check platform's TM table
 6: if agent was not executed before then
 7:    save TM in platform's table
 8: else
 9:    if node ident. is different then
10:       save TM in platform's table
11:    else
12:       if loop counter > previous one then
13:          replace previous TM from platform's table
14:       else
15:          discard agent execution
16:       end if
17:    end if
18: end if
19: run agent
```

Figure 5.5: Algorithm for checking trip markers in *sequence* nodes

Line 3: The agent identifier extracted from the current node is used to verify that the trip marker belongs to this very agent. Thus, the agent identifier included in the trip marker must be equal to the one obtained from the current node. This prevents a trip marker from being reused in a different agent, even if it was generated by the same owner.

Line 4: The trip marker expiry date is checked. If the trip marker has expired, the agent execution is discarded.

Line 5: The agent identifier is used to look for a previous trip marker of the same agent in the platform's trip marker table.

Line 7: If there is no trip marker with the same agent identifier, the new trip marker, along with the current node identifier, is stored in the platform's trip marker table.

Line 9: Otherwise, this means that the same agent was executed before on this platform.

In this case, the agent's current node identifier is compared with the node identifier of the previous execution.

Line 10: If the new node identifier is different, it means that the agent is visiting a different itinerary node. The new trip marker, along with the current node identifier, is stored in the platform's trip marker table.

Line 12: Otherwise, this means that the same itinerary node was visited before. The current loop counter is then compared with the one included inside the previous trip marker obtained from the trip marker table.

Line 13: If the current loop counter is greater than the previous one, it means that the agent is performing a new iteration of a loop. In this case, the previous trip marker from the trip marker table is replaced by the current one.

Line 15: Otherwise, this means that the current trip marker has already been used, and the agent execution is discarded.

Line 19: If the agent execution has not been discarded in any of the previous steps of the algorithm, the current node's task is executed.

The above algorithm is necessary in order to verify that the agent is not being replayed when its current node type is *sequence*. When the current node type is *loop*, basically the same checks are performed, but two additional operations are required.

Firstly, a new trip marker for the agent must be generated and signed. This new trip marker authorises the agent to visit all the nodes included inside the loop. The new trip marker is added to the top of the agent's trip marker stack, thus keeping the previous trip marker in the second position of this stack. This allows the agent to recover the trip marker used before entering the loop, which is essential because the trip marker generated by the current platform will no longer be valid when the agent exits the loop.

Secondly, if the signature verification performed in *line 1* of the previous algorithm fails, the trip marker signature must be verified using the current platform's public key. This

allows the agent to be executed on the *loop* node's platform using a trip marker generated by this very platform in a previous iteration.

As an example, in the itinerary of figure 5.3, platform *A* is the authorisation entity of nodes 2 and 3. For each iteration of these two nodes, platform *A* must generate and sign a new trip marker for the agent. In addition, it must verify the signature of the current trip marker with either its own public key or the owner's. Once the agent exits the loop, the trip marker signed by platform *A* must be removed from the top of the agent's trip marker stack, so that the original trip marker signed by the owner can be used again to continue the agent execution on platform *E*.

The algorithm shown in figure 5.6 summarises the operations carried out when the current node type is *loop*.

```
1:  verify TM signature with the pubkey of either the
    current platform or the current authorisation entity
2:  check TM auth. node = current node or current
    auth. node.
3 to 18: the same operations as algorithm of fig. 5.5
19: generate new TM by incrementing loop counter
20: sign new TM with current platform's private key
21: if this is the first iteration then
22:     add new TM to agent's TM stack
23: else
24:     replace previous TM from agent's TM stack
25: end if
26: run agent
27: if agent exits loop then
28:     Remove TM from the top of stack
29: end if
```

Figure 5.6: Algorithm for trip marker handling in *loop* nodes

The most relevant lines of the algorithm of figure 5.6 are described in detail below:

Line 1: The trip marker signature is verified using the current platform's public key of the public key of the current authorisation entity. If both verifications fail, the agent execution is discarded.

Line 2: The authorisation node included in the trip marker is used to verify that said trip marker has been generated in the appropriate itinerary node. This involves checking that this authorisation node is the current node or the authorisation node extracted from the current node. This allows the agent to use a trip marker generated in the appropriate authorisation node, as well as one generated in a previous visit to this same node.

Line 3 to 18: The agent's identifier, expiry date, loop counter and node identifier are checked performing the operations 3 to 18 of the algorithm of figure 5.5.

Line 19: Before the agent task is executed, a new trip marker is generated. This new trip marker contains the same information as the previous one, except for the authorisation node, which is set to the current node, and the loop counter, which is incremented by one.

Line 20: The resulting trip marker is signed with the current platform's private key.

Line 22: If the agent is visiting the current *loop* node for the first time, the new trip marker is added to the top of the agent's trip marker stack.

Line 24: Otherwise, the current trip marker, which was generated in this same platform in the previous iteration, is removed from the top of the agent's trip marker stack and is replaced by the newly generated one.

Line 26: The current task is executed, and the agent eventually decides whether or not a new iteration has to be performed.

Line 28: If no more iterations are required, the agent removes the trip marker from the top of the stack.

Thus far, the complete protocol aiming at preventing external replay attacks has been presented, describing the creation of a protected mobile agent and the operations required to withstand replay attacks. In the next section, the most important characteristics of the proposed protocol are discussed.

5.2.3 Discussion

The proposed protocol is based on using trip markers (authorisations) which are generated at runtime by authorisation entities. The agent itinerary is comprised of a set of nodes, where each one is associated with a certain authorisation entity. The task associated with a node is only executed if the trip marker has been signed by the appropriate authorisation entity. Every itinerary node is also associated with an authorisation node, which involves that the task of a given node is only executed if the trip marker has been generated in the appropriate authorisation node.

Using both an authorisation entity and node prevents the reuse of the agent trip marker in scenarios where the same authorisation entity is assigned to different nodes of the itinerary. In these cases, an attacker might try to replay an agent by reusing a trip marker signed by the proper authorisation entity but not generated in the appropriate authorisation node. However, this would be detected when checking if the trip marker has been generated in the appropriate authorisation node.

Every itinerary node contains an agent identifier, which is also included in the agent trip marker. This prevents a given trip marker from being reused in different agents, for this would be detected when comparing the agent identifier extracted from the current node with the one included in the trip marker.

In order to prevent replay attacks effectively, the agent's itinerary must be cryptographically protected, so that no attacker can change the information associated with a given node. Moreover, the itinerary protection must keep an agent task from being executed on a platform or itinerary stage different from that initially defined.

The proposed protocol does not involve the interaction of the agent with external entities, and only assumes that *loop* nodes will be associated with platforms trusted by the programmer. Because it is not possible to prevent platforms from tampering with the agent execution, it is legitimate to assume that the platform associated with a *loop* node is trusted by the programmer to evaluate the loop condition.

Platforms must prevent their trip marker tables from filling up by removing entries that

have already expired. Additionally, some other policy has to be implemented in order to enable the removal of entries when no more trip markers can be added to the table (e.g., removing older entries first).

The protocol described in this chapter can be used in combination with the itinerary protection protocol presented in chapter 4, as long as the platforms assigned to authorisation nodes are known at the time of creating the itinerary. Thus, the authorisation entities associated with these nodes can be specified inside every itinerary node.

The proposed protocol does not support agent cloning. In general, cloning mobile agents introduces the problem of resolving identities properly when the replicas communicate with other agents or platforms. With regard to the proposed protocol, cloning a mobile agent would imply generating a new trip marker for the replica, but this is not possible unless the new trip marker is generated by an authorisation entity. However, authorisation entities are designed to generate new trip markers only when new loop iterations have to be started. As a result, the proposed protocol does not support agent cloning. Further research will be conducted in the future to extend the proposed protocol in order to allow for agent cloning.

5.3 Implementation

In order to prove the viability of the proposed protocol, a prototype implementation has been created and experimentation has been carried out using the Java-based Jade platform [BCPR03] as the agent execution environment.

Agents have been implemented following an agent-driven approach, as proposed by [ARO04]. Figure 5.7 shows the main components of an agent resulting from this implementation. As this figure shows, the agent is made up by three components: the trip marker, the control code and the protected itinerary. These components are bound by an agent identifier, thus preventing from dishonest reuse in other agents.

The agent identifier is constructed as a hash of the agent's control code. As discussed

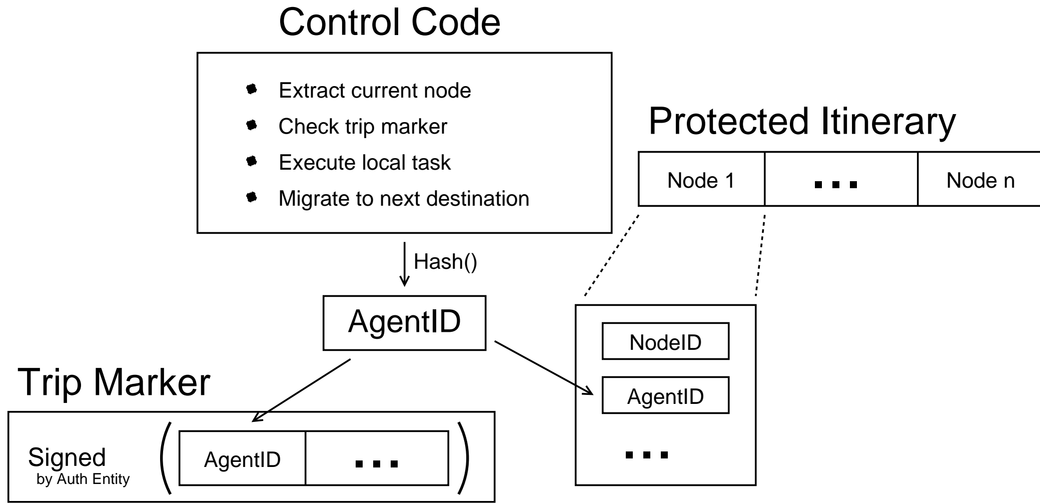

Figure 5.7: Components of an agent protected against replay attacks with an agent-driven implementation

in [ARO04], the control code is unique for each agent because it includes a random asymmetric key that is also unique. Therefore, the hash of the control code is a suitable agent identifier.

In order to create the protected itinerary, the implementation uses the protocol presented in section 4.3.1. The following is a summary of the operations involved.

First of all, a random symmetric key is generated for each itinerary node. Then, these symmetric keys are used to create the transitions t_{ij} from each node i to each of its successors $j \in \{a_i \cdots A_i\}$. The agent identifier id included in these transitions is computed as a hash of the agent's control code, as mentioned earlier.

Then, every entry of the protected itinerary is created as follows:

$$e_i \;=\; E_{r_i}(S_o(nodeInfo_i), [t_{i\,a_i} \cdots\; t_{i\,A_i}]) \tag{5.1}$$

where we use the same notation as in chapter 4. $nodeInfo_i$ denotes all the information associated with node i, which comprises the agent identifier, node identifier, task, node type, authorisation entity and authorisation node. If the node type is *discoverer*, then $nodeInfo_i$ also includes the lists $[S_{k_s}(S_o(H(C), r_{l_i}, h_i)) \cdots S_{k_s}(S_o(H(C), r_{L_i}, h_i))]$ and $[r_{bl_i} \cdots r_{BL_i}]$ (see equations 4.6 and 4.14).

With regard to the control code, the implementation is based on the same algorithm described in section 4.3.2. However, before executing the current task, the control code verifies that the trip marker extracted from the top of the trip marker stack is valid. For this purpose, the control code uses a call to a public trip marker verification function provided by platforms.

The trip marker verification function takes the following parameters: the trip marker extracted from the top of the trip marker stack, and the node identifier, node type, authorisation entity, authorisation node and agent identifier, which are extracted from the current entry of the protected itinerary. With all this information, the platform undertakes the operations described in the algorithms of figures 5.5 and 5.6.

As shown in these algorithms, the public trip marker verification function must verify that the trip marker provided as a parameter really belongs to the requesting agent. For this purpose, the platform computes a hash of the agent's control code, and compares it with the one included in the trip marker. In addition, the platform verifies the validity of the trip marker's signature, authorisation node, expiry date, loop counter and node identifier, performing the operations of the aforementioned algorithms. If all these verifications succeed, this function returns *true*, as well as a new trip marker when the current node type is *loop*.

The control code resumes or aborts the agent execution depending on the value returned by the trip marker verification function. In addition, if the current node type is *loop*, the control code modifies the top of the trip marker stack accordingly.

As can be seen, the trip marker check is triggered by the agent. Thus, the protection against replay attacks is completely optional, and will only be performed by those agents calling the trip marker verification function. Forcing every agent to support the replay protection protocol would unnecessarily increase the complexity of many applications where security is not a requirement.

It is also worth noting that the implemented agent-driven approach prevents any possible modification to the agent's control code, as discussed in [ARO04]. Thus, attackers are not allowed to modify this control code to bypass the trip marker check.

After creating the control code and the protected itinerary, the initial trip marker of the

agent is generated. This trip marker includes an agent identifier, an authorisation node, an expiry date and a loop counter, as described in section 5.2.2. The resulting trip marker is signed with the owner's private key.

Once the control code, the protected itinerary, and the agent trip marker have been created, the final executable mobile agent is generated.

5.3.1 Simulation and tests

In order to test the implementation of the proposed protocol, a simple mobile agent-based application has been simulated. This application shows the need for protection against replay attacks, and the results of the simulation prove that replay attacks can be effectively prevented, and agents can be executed in completely reasonable times.

This example application implements an automated car purchasing service. The service allows an individual to find the best price for a car, given its specific make and model, and the set of car dealers that have to be queried. The application carries out the purchase remotely from the user's financial institution, and this institution serves as the trusted location where the application decides which offer to accept.

Once the user has introduced all the required information in the application, a mobile agent is launched to visit every car dealer and obtain the price offered for the given model. After visiting all car dealers, a new round is started to negotiate an improvement on the best price previously obtained. This process is repeated until two consecutive iterations lead to the same best price. After each iteration, the agent visits the user's financial institution, to decide whether or not a new iteration has to be started.

When the best price is obtained, the agent proceeds with the car purchase. The purchase is always completed remotely from the user's financial institution, which ensures that this operation is performed securely in a trusted environment. Once the purchase has concluded, the agent returns to its home platform, and presents the resulting purchase contract to the user. The agent's itinerary defined for the implementation of this example application is shown in figure 5.8.

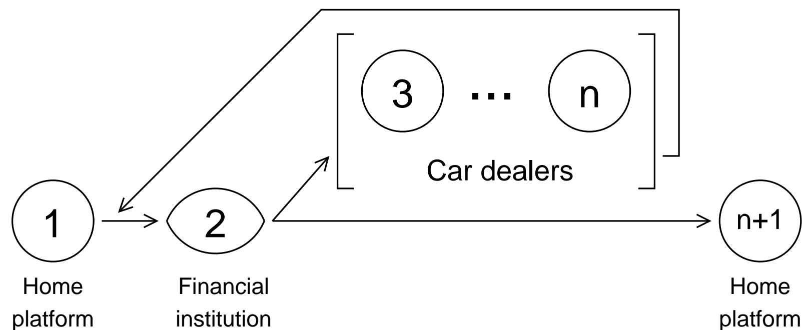

Figure 5.8: Itinerary of the car purchasing service agent

This application shows a simple scenario where the use of the mobile agent technology can introduce a number of advantages, such as reducing network load—by employing local communications—as well as automation of e-commerce processes. Considering that the negotiation can be a rather lengthy process, mobile agent technology enhances significantly the usability of this application in devices with intermittent, low bandwidth connections, for it eliminates the need for permanent connections with the remote sites.

This application also demonstrates the need of an agent to dynamically determine the number of repeatable visits required over a certain set of platforms. Unlike replay protection mechanisms previously presented, the proposed protocol allows the agent to visit the same platform as many times as necessary, without leaving it exposed to replay attacks. This is an essential part within the context of this application, for a malicious platform (e.g. an ill-intentioned car dealer) could easily resend the agent to its next destination to trigger additional car purchases.

In order to validate the proposed protocol, this application was simulated by introducing a malicious platform in the agent's itinerary. This platform acted as a dishonest car dealer trying to trigger more purchases every time it could provide the best offer for a car. As expected, none of these attacks succeeded because the next platform of the itinerary immediately detected that the trip marker of the replayed agent had already been used.

The experiments performed also compared the execution times of the replay-safe agent with the unprotected agent, in order to determine the overhead of the proposed protection

	Num. of iterations:	3	6	9	21
Unprotected agent	Exec time:	11365	22094	32837	75789
	Time/node:	757	818	841	871
Protected agent	Exec time:	16641	31682	46715	106848
	Time/node:	1109	1173	1197	1228
	Increase:	46.4%	43.4%	42.3%	40.9%

Table 5.1: Execution times (ms) for agents protected and unprotected against replay attacks

protocol. The agent created for the tests was given an itinerary with three car dealers. The number of iterations required to reach the best price was varied in order to identify the overhead in terms of the agent's execution time. It is important to note that no dynamically located nodes were introduced in the itinerary, so that the resulting execution times were not affected by the itinerary reconstruction process described in section 4.3.2.

The evaluation setup used to make the tests was made up by 6 computers with 2 GHz Intel Pentium(R) IV processors and 256 MB RAM memory each. These computers were connected in a laboratory to a 100 Mbps Ethernet LAN. Table 5.1 shows the resulting execution times.

According to table 5.1, the execution time of a replay-safe agent is 43% higher than the execution of an unprotected agent on average. This increase, however, depends greatly on the specifics of the application's requirements. In the context of this scenario, the agent negotiations with the car dealers took only a relatively short time. As a result, the time spent handling protection mechanisms significantly impacted the overall agent's execution time. In contrast, applications with high processing requirements may require the agent to execute time-consuming tasks, and thus, in that context, the time spent handling protection mechanisms would be negligible.

In order to evaluate the performance of the replay protection protocol in combination with the protocol presented in chapter 4, experiments were also carried out using a modification of the agent's itinerary. The modified itinerary, which is shown in figure 5.9, includes a *discoverer* node, where the agent discovers three new car dealers at each loop iteration.

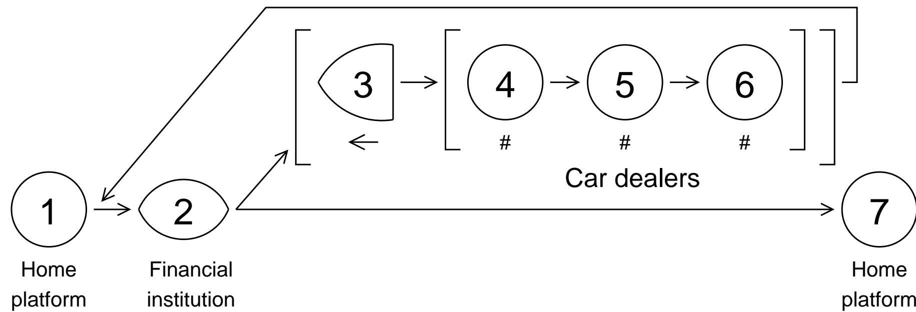

Figure 5.9: Itinerary of the car purchasing service agent
with dynamically located car dealers

In this case, only one round of negotiation is conducted with each car dealer. This modified itinerary was used to determine the impact on the execution time due to the itinerary reconstruction performed in node 3. The execution times are shown in table 5.2.

	Num. of iterations:	3	6	9	21
Unprotected agent	Exec time:	11365	22094	32837	75789
	Time/node:	757	818	841	871
Protected agent	Exec time:	18017	34517	50972	116462
	Time/node:	1201	1278	1306	1338
	Increase:	58.5%	56.2%	55.2%	53.6%

Table 5.2: Execution times (ms) for agents protected and unprotected
against replay attacks with dynamically located nodes

As shown in table 5.2, the execution time of a protected agent is 55% higher than that of an unprotected one on average. This increase in the agents' execution times is readily reasonable, taken into account the complexity of the proposed protocol, and the added complexity of the itinerary protection protocol presented in chapter 4. The overhead introduced by the execution of these protocols (around 465ms in absolute terms) is completely acceptable for this application. However, this may vary from one application to another. To conclude, it can be said that the implementation of the proposed protocols is advisable for any real-world application where security is an issue, and it is also perfectly feasible.

5.4 Conclusions

This chapter has presented a protocol aiming to protect mobile agents against external replay attacks. Previous published work on this area was mainly based on storing a trip marker, or some other kind of agent identification, inside agent platforms. This identifier was fixed for the whole agent execution, and, as a result, it was not possible to define itineraries where the same platform was visited more than once.

The proposed protocol protects agents against replay attacks, and allows agents to traverse itineraries that contain loops. The nodes that are part of a loop can be traversed repeatedly, an undetermined number of times. This allows programmers to define itineraries which take full advantage of the inherent flexibility of the mobile agent paradigm.

In order to make this possible, the proposed protocol is based on associating every node with an authorisation entity and node. The agent execution is only allowed if the agent trip marker has been generated and signed in the appropriate authorisation node, and by the corresponding authorisation entity.

The agent's itinerary is cryptographically protected, in such a way that the authorisation entity assigned to a given node cannot be changed. Additionally, the trip marker contains a unique agent identifier that is also included inside every itinerary node. Thus, the trip marker of an agent cannot be dishonestly reused in different agents.

The replay attack protection protocol presented in this chapter does not support agent cloning, for it cannot distinguish between a replayed agent and a legally replicated one. Further research will be conducted in order to support the generation of new trip markers for replicated agents.

In order to prove the validity of the proposed protocol, implementation and experimentation work has been carried out using the Jade agent platform and the itinerary protection protocol presented in chapter 4. The application simulated an automated, agent-based, car purchasing service with price negotiation and car purchase features. The simulation results prove that replay attacks can be effectively prevented, and replay-safe mobile agents can be executed in reasonable times.

The implementation of secure mobile agents, as can be seen, requires developing a number of protocols to prevent replay attacks, illegal modifications of the agent's itinerary or computational results, etc. The development of these mechanisms requires considerable work and expertise. In the following chapter, we will present a development environment that simplifies the implementation of these mechanisms and promotes the development of secure mobile agent applications.

Chapter 6

Promoting the development of secure mobile agents

The interest in mobile agent technology has recently increased after the progress made in some critical security aspects, such as the protection of the agent's computational results [ZOL04] or the agent's itinerary [MB03]. However, in order to achieve widespread deployment of this technology, providing security breakthroughs is not enough; development of secure mobile agent applications should be encouraged as well. New tools for the design and development stages have to be created [MY06], simplifying to the greatest extent possible the tasks carried out by both the designer of new cryptographic protocols and the developer of new applications.

There is very little literature about these specific aspects, primarily because the main work done on mobile agents has been focused on developing new agent protection mechanisms. More basic usability issues concerning the human developer have been left aside. As a result, the implementation of these mechanisms can turn out to be more time-consuming than the implementation of the agent tasks.

In this chapter, we present a development environment that simplifies the implementation of mobile agent protection protocols. This environment facilitates the implementation of protocols such as those presented in chapters 4 and 5, and allows programmers to reuse

these implementations in different applications. Thus, we aim to promote the use of mobile agent technology for the development of secure distributed applications.

6.1 Related work on mobile agent software engineering

In this section, we outline the relevant work done in mobile agent software engineering. First, we focus on mobile agent platforms and, then, we analyse some of the approaches that simplify mobile agent development, placing special emphasis on new programming languages created for this purpose.

6.1.1 Mobile agent platforms

Firstly, it must be noted that literally tens of mobile agent platforms have emerged since the appearance of this new paradigm (see [AGK+01] for a survey of agent platforms). Among these, we can highlight Telescript, ARA, D'Agents, Aglets, Concordia, Grasshopper, Ajanta, SeMoA, AgentScape and JaDE. Most agent platforms presented so far are prototypes that have only been used for research purposes. Few of them have users outside the academic or research centre where they were created. The platform that has more users at this present time is undoubtedly JaDE [BCPR03].

All these mobile agent systems have a similar purpose: to provide an execution environment to agents which allows them to use, search and provide services, such as sending messages to each other or moving to other platforms. Most of these systems, such as JaDE, are implemented in Java due to its reflection capabilities and the availability of a dynamic class loader. Much of the research conducted on mobile agents has been centred on defining new mobile agent platforms, usually leaving aside usability aspects.

6.1.2 Simplifying agent development

Research carried out on mobile agents has also given rise to multiple proposals to simplify the development of this kind of applications. These proposals can be divided into two main

groups: on the one hand, proposals based on using new agent programming languages that simplify the implementation of the agent's tasks; and, on the other hand, proposals aimed at aiding in the design of mobile agent-based applications.

Regarding the first group, most proposals suggest the use of new declarative languages, for their inherent high level of abstraction simplifies the implementation and readability of programs. Among these declarative languages, some proposals are based on logic languages and others on functional languages.

The number of proposed logic languages is quite large. Some allow using directly Prolog to express the agent reasonings or inferences [ZCM02]. Others enable programmers to define classes of objects in order to merge logic with object oriented programming [DELM00]. Other languages are based on formal systems like pi-calculus [MPW92], which allow expressing the mobility and the interactions among different agents [CG98]. Finally, some languages are appropriate to express the agent cognitive capabilities explicitly (reasoning, planning, decision making ...) [SF05]. With regard to the work based on functional languages, fewer proposals have been presented ([Kna95, KT04]). Among these, it is worth pointing out that [KT04] makes it possible to define object classes.

In general, all these languages have one feature in common: they provide a mechanism to allow the agent to move from one platform to another.

Regarding the proposals intended to aid in the design of mobile agent-based applications, most of them are based on the use of design patterns [MLH05, LMSF04, TOH99, TOH01]. Design patterns are proven solutions to recurring problems that arise within some contexts, thus enabling an easy reuse of good software design.

In conclusion, numerous proposals have been presented to simplify the implementation of agent tasks. However, these proposals ignore the specific problems related to the implementation of the security mechanisms required by most mobile agent applications. In order to solve this problem, the next section presents a development environment aimed at simplifying the implementation of secure mobile agent-based applications.

6.2 Development environment

This section presents a development environment aimed at aiding agent programmers in the development of secure mobile agents following an agent-driven approach. As discussed in chapter 4, this approach has a number of advantages. First, the control code can handle the protection of the explicit itinerary, the computational results, or any other agent management mechanism, for example, related to fault tolerance. Second, this control code can be easily reused because it does not depend on the tasks carried out by the agent. Therefore, agents with similar characteristics will usually execute the same control code. Finally, this approach relieves platforms of the need to deal with different protection protocols, which is especially important because different agents usually have different security requirements, and therefore, different protection mechanisms.

However, the agent-driven approach also entails a considerably more complex agent implementation. It can require, for example, obtaining platform certificates or performing cryptographic operations to encrypt and decrypt some parts of the agent's itinerary. Thus, the implementation can imply the use of a public key infrastructure, symmetric and asymmetric keys, cryptographic hashes and, in general, an extensive knowledge on cryptographic application programming. As mentioned earlier, none of the previous proposals on mobile agent security has addressed the difficulties faced by programmers when implementing these security mechanisms.

In order to relieve programmers of this burden, this section presents a development environment that simplifies the implementation of secure mobile agents. This environment is comprised of three main tools: the Itinerary Designing Tool, the Agent Builder and the Agent Launcher.

The Itinerary Designing Tool (IDT) is a graphical tool that can be used to design the agent's itinerary. This tool provides a graphical itinerary editor where the programmer can define the set of nodes that comprise the itinerary. Then, a task and an execution platform can be assigned to each node. This tool also provides a task editor, where new tasks can be created and compiled. With all the information provided by the programmer, this tool

produces an XML specification of the initial itinerary. More details about this tool will be given in section 6.4.

Once the XML itinerary specification has been produced by the IDT, the Agent Builder can be used to generate the final agent. In order to use the Agent Builder, programmers must define what protection mechanisms are required by their application. These mechanisms must be specified using the Mobile Agent Cryptographic Protection Language (MACPL), which is a new specification language specifically designed to simplify the implementation of agent protection protocols.

By using MACPL, the proposed development environment is not constrained to a specific set of protection protocols. This is essential because the use of one protocol or another depends on the specific requirements of the given application. Thus, the proposed environment simplifies the development of current protocols as well as others that may appear in the future.

MACPL is a high level language designed to facilitate the implementation of agent protection protocols. These protocols can address any security-related mobile agent concern, such as the protection of the agent's itinerary or its computational results. By using MACPL, implementations become more reusable and extensible, and thus programmers can often generate different mobile agents using the same pre-existing MACPL code, and they are relieved of any need to implement an agent protection protocol. In section 6.3, the main features of MACPL will be described in detail.

The Agent Builder is comprised of three main modules: the Agent Setup Module, the Control Code Module and the Agent Creator Module. These modules, together with their inputs and outputs, are represented in figure 6.1. As this figure shows, the Agent Builder has two main inputs: the MACPL specification and the XML itinerary specification. The XML itinerary specification is the document generated by the Itinerary Designing Tool. With regard to the MACPL specification, it is created by the agent developer or a security expert, and it is divided into two parts:

- The specification of the agent setup operations, where the data structures used by the

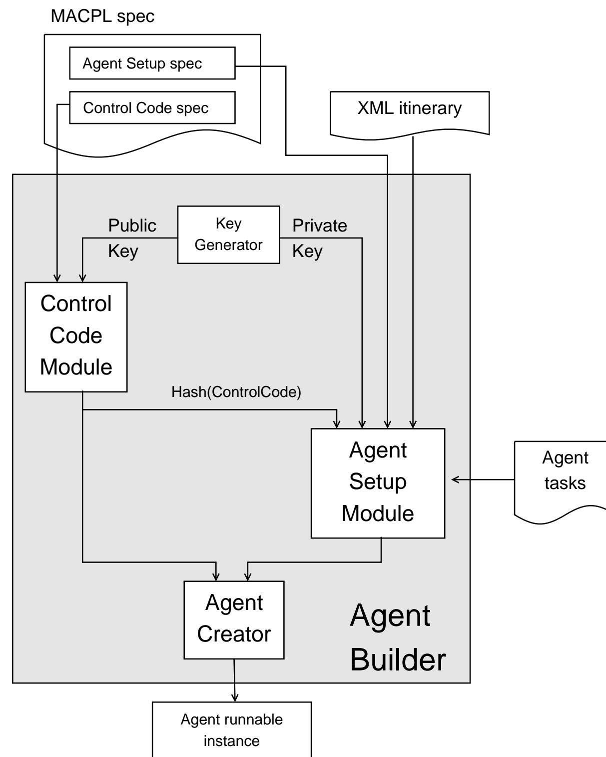

Figure 6.1: Components of the Agent Builder with its main inputs and outputs

agent during its execution are initialised (e.g. protected itinerary, trip marker, or any other).

- The specification of the operations performed by the control code.

Using the MACPL specification and the XML itinerary specification, the Agent Builder performs the following operations to generate a secure mobile agent.

First of all, it generates a random pair of asymmetric keys—a public key and a private key. This keypair is used to implement the mechanisms required by the agent-driven approach, as described in [ARO04].

Then, using the second part of the MACPL specification, the Control Code Module runs a MACPL compiler to generate the agent control code. This part of the MACPL specification must define how the control code manages the explicit itinerary, the trip marker, or any other agent component. The random public key previously generated is included in

the resulting control code as a compile-time constant.

Next, the Agent Setup Module runs a MACPL interpreter to execute the first part of the MACPL specification. This part of the MACPL specification must define how to initialise the data structures required by the agent. The protected itinerary is one of the data structures that must be always created during the agent setup. For this purpose, this module uses the XML itinerary specification provided to the Agent Builder and the agent tasks. This module also uses the random private key previously generated in order to sign every platform-specific code and data included in the protected itinerary.

Finally, the Agent Creator Module combines the outputs of the two previous modules to create the executable mobile agent.

Once the agent is obtained, the Agent Launcher (AL) is used to put the agent into execution on the first platform of the itinerary. More details about the Agent Launcher will be given in section 6.4.

At this point, it is worth noting that the agent's tasks are implemented by the programmer in the programming language supported by the execution environment, which can be Java, C++, or any other. Other protocols have been presented to simplify the implementation of the agent's tasks, usually providing new agent programming languages [ZCM02, KT04]. However, these protocols do not allow developers to implement any agent protection mechanisms. Because of this, the proposed development environment is focused on aiding the programmer in the implementation of the security protocols required by secure mobile agent applications. If necessary, the proposed environment can be combined with other proposals to simplify the implementation of the agent tasks, too.

Figure 6.2 shows a representation of all the components that comprise the proposed development environment. As this figure shows, different roles are involved in the development process. First, the agent programmer, who designs the explicit itinerary and generates the XML specification using the IDT. Second, the security expert, who implements the agent protection protocols using MACPL. Finally, the end user, who executes the agent and obtains its results without any knowledge about security or programming at all. The separation of these three roles shows the flexibility and ease of reuse that the proposed

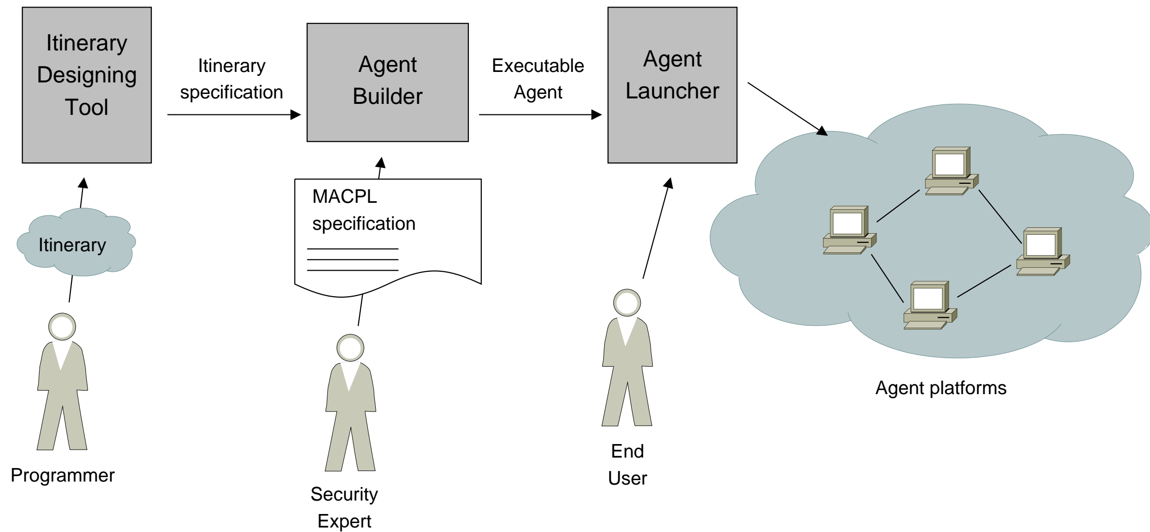

Figure 6.2: Overview of the mobile agent development environment

development environment brings to implementations. Thus, the development of a whole system is divided into independent components—XML and MACPL specifications—and independent tools that can be used by completely different people.

Thus far, the components of the proposed development environment have been presented. It is worth noting that this environment is not designed for any specific execution platform. It can be implemented to simplify the development of secure mobile agents regardless of their execution environment or programming language. The next section will be devoted to describe the main features of MACPL, and the simplification achieved as a result of its utilisation.

6.3 MACPL

MACPL is a domain-specific programming language intended to ease the development of agent protection protocols. The design of this language has pursued two main objectives:

- Simplifying the traversal of the initial itinerary, and its protection using cryptography.
- Simplifying the implementation of the control code, which handles the protected

itinerary and other agent security mechanisms.

The language resulting from these requirements is explained thoroughly in appendix A, where a detailed description of MACPL features can be found. Besides, appendix B contains an example of its use to implement an itinerary protection protocol.

MACPL code is divided into two clearly different parts: the part that defines how to create the explicit itinerary and any other data structures required by the agent, and the part that defines how the control code is generated. These two parts are separated by the `#control_code_begin` precompilation directive. The code placed above this directive is the agent *setup code*, and the code placed below is the agent *control code*.

MACPL provides four types of instructions: type declarations, assignment statements, function calls and function definitions. MACPL code is executed by evaluating all type declarations, assignment statements and function calls in order.

Functions are defined using the `fundef` keyword, and can take arguments which are always passed by value. In addition, MACPL functions always return a value, and the `return` keyword is used for this purpose. As will be seen, MACPL provides a broad set of built-in functions, which are intended to make it a powerful and easy-to-use language. The following code shows an example of a function definition.

```
fundef Task getTask(GraphNode node) {
  // function body
  return task;
}
```

In this case, this code defines the `getTask` function, which takes a GraphNode argument and returns a Task object. The different data types provided by MACPL, such as the GraphNode and the Task, will be described in the next section.

Type declarations are statements that specify the type of a variable. All variables must be declared before being used. The following is an example of a type declaration. In this case, a variable of type `Graph` is declared.

```
Graph initialItinerary;
```

Assignment statements assign a value to a variable using the '=' operator. A type declaration and an assignment statement can be combined in the same instruction. The following

example code shows a type declaration, an assignment statement, and a combination of the two.

```
String name;
name = "foo";
Integer id = 0;
```

6.3.1 MACPL types

The set of types provided by MACPL is quite small—only eight different data types. This is motivated by the fact that, first, MACPL is not a general purpose language and, second, MACPL is devised to be as simple as possible.

MACPL is statically typed since types are determined at compile time, not at runtime. In addition, MACPL is strongly typed because the language prevents the execution of code that uses types in an invalid way.

An important MACPL type is the List, for it allows programmers to create compound objects that can be protected using cryptographic mechanisms. Part of the MACPL syntax is conceived to facilitate the use of List objects. For example, lists are created by writing the elements in order, separated by ':' and surrounded by '[' and ']'. The '`< >`' operator allows to refer to individual elements of a list. Thus, `list<n>` refers to the *n*th element of `list`. The following is an example of the creation of a list.

```
List it = [getnexttrans(node,5,nhost):finalIt<5>];
```

In this case, a list of two elements is created and assigned to the `it` variable. The first element is the value returned by the `getnexttrans` function, and the second element is the fifth element of the `finalIt` list.

MACPL allows programmers to access the last element of a list using the `last` keyword. This keyword is often used in expressions like the one represented next.

```
[ExpressionWithIndex | IndexVariable,FirstIndex,LastIndex]
```

These expressions are used to evaluate `ExpressionWithIndex` from `IndexVariable=FirstIndex` to `IndexVariable=LastIndex`, and store the result in a list. For example, if `list` is a List object containing three elements, then the following code.

```
[fun(list<j>)|j,1,last]
```

is equivalent to this one

```
[fun(list<1>):fun(list<2>):fun(list<3>)]
```

MACPL provides two data types to facilitate the handling of the agent's itinerary: the Graph and GraphNode data types. The `getInitialItinerary` built-in function reads the XML itinerary specification provided to the Agent Builder, and returns a Graph representation of it (see section 6.3.3). This Graph object is composed of one or more GraphNode objects, which can be traversed and manipulated by the programmer using several built-in functions: `getNode`, `successors`, `predecessors`, `addNode`, `graph2List`, among others. The following code shows an example of graph manipulation using the Graph object returned by the `getInitialItinerary` function.

```
1:  Graph initItin = getInitialItinerary();
2:  List initItinList = graph2List(initItin);
3:  export List protectedItin = [protectNode(initItinList<i>)|i,1,last];
4:  fundef List protectNode(GraphNode node) {
5:    String platform = nodeData(node)<3><2>;
6:    String nextplatform = nodeData(successors(node)<1>)<3><2>;
7:    return [aencrypt(platform,graphNode2String(node)):nextplatform];
8:  }
```

The first line of the above code initialises `initItin` to the Graph object returned by `getInitialItinerary`. Line 2 introduces all the GraphNode objects of `initItin` into a list, using the `graph2List` built-in function. The resulting list is stored in the `initItinList` variable. Line 3 applies the `protectNode` function to every element of `initItinList`. As a result, a list of protected itinerary nodes is obtained and stored in the `protectedItin` variable. Finally, lines 4 to 8 define the `protectNode` function, which takes a GraphNode parameter (`node`) and returns a List object.

The `protectNode` function uses the `nodeData` built-in function to extract information from `node` and from the successor of `node` (more details about this function will

be given in section 6.3.3). The platform associated with `node` is stored in the `platform` variable, and the subsequent platform of the itinerary is stored in the `nextplatform` variable. Then, the `graphNode2String` built-in function is used convert `node` into a String object, and the result is encrypted using the public key of `platform`. The encrypted node and `nextplatform` are finally returned using the `return` keyword.

This short example shows that the protection of the initial itinerary can be significantly simplified. In this case, only eight lines of code are needed to traverse the agent's initial itinerary, encrypt each one of its nodes, and then introduce the result in a list.

Another important MACPL type is the RuntimeDefined. This type is used to deal with the data types provided by the agent programming language, which is the language supported by the agent execution environment. The data types of the agent programming language are not directly supported by MACPL, which means that type errors related with RuntimeDefined objects are detected at runtime, not at compile time.

An example of a built-in function that uses RuntimeDefined objects is the `sencrypt` function. This built-in function encrypts data using a symmetric key algorithm, and takes a secret key parameter which is a RuntimeDefined object. However, if the secret key provided to `sencrypt` is a RuntimeDefined object that does not encapsulate a proper secret key, then MACPL will issue an error at runtime, not at compile time. In general, most cryptographic functions provided by MACPL use RuntimeDefined objects.

MACPL also provides a data type associated with the tasks executed by the agent: the Task data type. In order to execute tasks, MACPL provides the `exec` built-in function, which takes a Task object and a String object as parameters. The String object specifies the name of the method that has to be executed, which must be implemented within the task. The type returned by this function is a String. The generation of Task objects in a format suitable for MACPL is performed using the Itinerary Designing Tool.

The String is also an important MACPL type. Apart from representing a sequence of characters (e.g. "foo"), the String type is used to encapsulate objects of other types. For example, the `sdecrypt` built-in function decrypts data using a certain secret key, and returns a String object encapsulating the decrypted data. In order to convert the resulting

String object into another data type, MACPL provides several conversion functions, such as `string2Task`, `string2List`, etc. The inverse operations can also be performed using the corresponding conversion functions (`task2String`, `list2String` ...).

In addition to the aforementioned data types, MACPL has also two more types: the Boolean and the Integer. The purpose of these types is equivalent to that of many other programming languages. They are used to evaluate conditional expressions, index elements of graphs and lists, etc.

6.3.2 Scope of variables

MACPL variables have two different types of scope:

Global: Variables with global scope can be accessed from anywhere within the entire MACPL code. They must be declared outside any function definition.

Function: Variables with function scope are only visible within the function in which they are declared.

Global variables may be referred to anywhere in the program, but they lose their value once the agent migrates from one platform to another. An example of this situation is shown in the following code.

```
Graph initItin = getInitialItinerary();
List protectedItin = protectItinerary(initItin);
List accumulatedResults =
  [signok(list2String(["Home":null:"Platform1"]))];
...
#control_code_begin
GraphNode currentNode = getCurrentNode(protectedItin);
List accumulatedResults =
  executeCurrentTask(currentNode,accumulatedResults);
...
```

The above code defines `protectedItin` and `accumulatedResults` as global variables. They are first initialised during the agent setup, and then they are used by the

control code in every platform of the itinerary. The problem of this example code is that the value assigned to these variables during the agent setup will never be available to the control code. Likewise, the value assigned to `accumulatedResults` in the control code will be lost when the agent migrates from its current platform to the next.

In order to allow the values of variables to be recovered after migrating from one platform to another, the `export` keyword must be used. This keyword must be placed at the beginning of the type declaration, as shown in the following example.

```
Graph initItin = getInitialItinerary();
export List protectedItin = protectItinerary(initItin);
export List accumulatedResults =
  [signok(list2String(["Home":null:"Platform1"]))];
...
#control_code_begin
GraphNode currentNode = getCurrentNode(protectedItin);
accumulatedResults =
  executeCurrentTask(currentNode,accumulatedResults);
...
```

The above code shows that `protectedItin` and `accumulatedResults` are now declared as *exportable* variables, and therefore their value is never lost during migrations. It is worth noting that the `export` keyword can only be used to declare global variables.

6.3.3 Built-in functions

MACPL provides a comprehensive set of built-in functions for the implementation of agent protection protocols. This section provides a brief description of the most important ones. As mentioned earlier, MACPL built-in functions are covered in detail in appendix A.

A subset of MACPL built-in functions is used to handle Graph objects. This subset includes: `successors` and `predecessors`, which return the successors and predecessors of a given GraphNode, respectively; `graph2List`, which returns a list containing all the GraphNode objects of a graph; `joinGraphs`, which returns the graph resulting from the union of two graphs, among others.

MACPL also provides functions for list management: `length`, to determine the size of a list; `remove`, to remove an element from a list; `join`, to concatenate two lists, among others.

In order to extract the information included in the XML itinerary specification, MACPL provides the `getInitialItinerary` built-in function. This function introduces all the information found in the XML document into a Graph object. In order to make this possible, the XML document must provide at least the following information for each itinerary node: task, type and platform. The following DTD document shows the structure of a valid XML itinerary especification.

```
<!ELEMENT ITINERARY (NODE+)>
<!ELEMENT NODE (TYPE,TASK,PLATFORM,(ATTRIBUTE*),(ITINERARY*))>
<!ELEMENT TYPE (#PCDATA)>
<!ELEMENT TASK (#PCDATA)>
<!ELEMENT PLATFORM (#PCDATA)>
<!ELEMENT ATTRIBUTE (NAME,VALUE)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT VALUE (#PCDATA)>
```

The following XML document shows a valid specification of an example itinerary that is comprised of a single node.

```
<ITINERARY>
  <NODE>
    <TYPE>Sequence</TYPE>
    <TASK>Task1.jar</TASK>
    <PLATFORM>ccd-pr2</PLATFORM>
  </NODE>
</ITINERARY>
```

The `getInitialItinerary` function introduces every itinerary node into a GraphNode object. In order to read the contents of a GraphNode object, MACPL provides the `nodeData` built-in function. This function returns a list of element-value pairs. Both the element names and the values are represented as String objects. As an example, the node defined in the above XML document would be returned by the `nodeData` function as follows.

```
[ ["TYPE":"Sequence"] : ["TASK":"Task1.jar"] : ["PLATFORM":"ccd-pr2"] ]
```

Apart from a task, type and platform, the XML itinerary can also specify other information for each itinerary node. This additional information can be specified using one or more `ATTRIBUTE` elements, each of which containing a `NAME` element and a `VALUE` element. For example, an XML itinerary could specify an authorisation entity and an authorisation node for every itinerary node.

MACPL also provides the `readFile` built-in function to read the contents of a file and introduce them into a String. A common use of this function is to read files that contain agent tasks. For example, `readFile` can be used to read the `Task1.jar` file specified in the previous XML itinerary example. The String object returned by `readFile` can be then converted into a Task object using the `string2Task` built-in function, and then this task can be executed using the `exec` built-in function.

MACPL also provides other built-in functions for the implementation of the control code: `move`, which allows agents to migrate from one platform to the next; `clone`, which allows agents to send a clone of themselves to other platforms; and `sendResults`, which allows agents to send their partial or final results to the owner.

One of MACPL's primary goals is to simplify the implementation of cryptographic protocols. For this purpose, it provides several cryptographic functions: `aencrypt` and `adecrypt`, to perform asymmetric encryption and decryption; `sign` and `verify`, to perform digital signatures and verifications; `skeygen` and `keypairgen`, to generate symmetric and asymmetric keys, among others. It is worth noting that the `adecrypt` function, which allows agents to decrypt data using the current platform's private key, is implemented as described in [ARO04], so that agents can never access platforms' private keys directly.

A common feature of all cryptographic functions is that they allow programmers to specify what they want to do, without specifying how they want to do it. For this purpose, the parameters taken by these functions never depend on any specific algorithm or implementation. This feature makes MACPL code more portable and easier to use. For example, when the `skeygen` function is used to generate a secret key for a symmetric algorithm,

the programmer does not specify if the key is intended for AES or 3DES encryption and decryption. The following section describes how the programmer can compile the agent selecting a specific set of algorithms or implementations, and how built-in functions are grouped into libraries.

6.3.4 Function libraries

MACPL built-in functions are designed to be independent of any algorithm or implementation. This makes MACPL code more generic and reusable. In addition, the Agent Builder supports different implementations of the built-in functions. Thus, programmers can compile the same MACPL code using different versions of these functions, depending on the requirements of the application.

Built-in functions are grouped into libraries. For example, all built-in functions related to list management are grouped into the same library. Each library implements an interface, so that different versions of the same set of built-in functions can be provided. For example, all built-in functions related to cryptography are defined in one interface. The Agent Builder may provide two different libraries implementing this interface, one based on PGP and another one based on X.509v3 certificates.

The set of interfaces and libraries provided by MACPL can be extended. Thus, programmers can develop new libraries by creating their own implementations of MACPL interfaces. Additionally, programmers can also create their own interfaces, and then provide one or more implementations of those interfaces. A new interface could be created, for example, to provide MACPL with networking capabilities. It is worth noting that libraries are implemented in the programming language supported by the agent execution environment, which essentially means that programmers can implement new libraries in a general purpose programming language.

Because each interface can be implemented by many different libraries, the Agent Builder provides command line parameters to select what specific libraries have to be used to compile the agent. In addition, if the MACPL code uses a certain interface provided by

the programmer, then the name of this interface must be specified inside the MACPL code, using the `#require` precompilation directive for this purpose. The programmer can then use command line parameters to select the library that implements his interface.

6.4 Auxiliary Tools

In the previous sections, we have described the Agent Builder and the MACPL language. In this section, we will present other auxiliary tools of the proposed development environment, which help programmers to generate the XML itinerary specification and allow them to launch the agent to the first platform of the itinerary.

6.4.1 Itinerary Designing Tool

The Itinerary Designing Tool (IDT) is used to aid the programmer in the generation of the XML itinerary specification. This tool provides a graphical interface that is organised in tabs, which allow the programmer to define the itinerary nodes, implement their tasks and see the messages generated by the agent compilation.

The *itinerary definition tab* is very similar to a drawing application. The left side of the window contains a node palette where the programmer can choose which type of node is included in the itinerary. Once a node has been placed in the drawing area, a task and a platform can be assigned to it. Figure 6.3 shows a screenshot of an example itinerary that is being edited in the IDT.

The task assigned to a node can be provided in precompiled form or it can be implemented and compiled in the *implementation tab*. When the programmer starts editing a new task in this tab, the IDT generates a skeleton of the methods that must be implemented. For example, if the programmer is editing the task assigned to a *loop* node, the skeleton includes the `jumpCondition` method, which decides whether or not the agent has to perform a new iteration.

Once the nodes and their corresponding tasks have been introduced in the itinerary

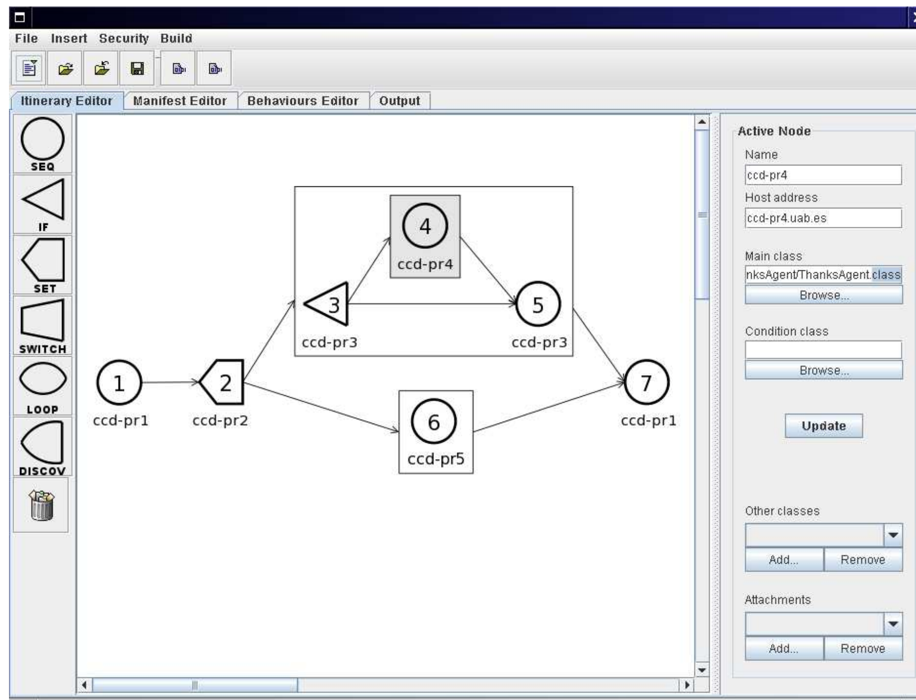

Figure 6.3: Itinerary Designing Tool

definition tab, the XML itinerary specification can be generated.

In addition to generating the XML itinerary specification, the IDT can also be used to create the executable mobile agent. For this purpose, programmers can choose which MACPL specification implements the protection mechanisms required by their application. Then, they can run the Agent Builder program from the IDT and obtain the executable agent. Thus, the IDT is designed as a development environment in which all the stages of the development process are integrated in the same tool.

6.4.2 Agent Launcher

The agents generated by the Agent Builder can be put into execution using the Agent Launcher (AL). The AL is a lightweight client application that allows agents to be launched to both local and remote platforms. This application should be able to run on any device, either a desktop computer or a handheld device, such as a PDA.

In order to start agents on remote platforms, the AL uses the immigration module of platforms' migration service [ARB03]. The communication with this migration service is performed using an Agent Communication Channel. The AL introduces the agent into an ACL message, and this message is sent to the remote platform. Then, the platform's immigration module extracts the agent from the ACL message, and puts the agent into execution.

6.5 Conclusions

In this chapter we have presented a development environment that simplifies the implementation of secure mobile agents. Most applications using mobile agent technology require the use of security mechanisms. These mechanisms, when possible, should be implemented following an agent-driven approach, thus allowing agents to manage their own security mechanisms.

Implementing agents that manage their own protection mechanisms has many advantages, but it also entails a quite more complex implementation. So far, no proposal has been presented to simplify the development of secure mobile agents. In this chapter, a development environment has been presented that simplifies the implementation of agent protection protocols, and promotes the reuse of these protocols for future developments.

The key element of the proposed development environment is the Agent Builder, which allows programmers or security experts to define protection protocols using the Mobile Agent Cryptographic Protection Language (MACPL). MACPL is a domain-specific language that is easy to learn and use. The main advantages of MACPL are:

- Availability of high level cryptographic functions that make it possible to quickly create security protocols. A subset of these functions allows agents to encrypt and decrypt itinerary data using platforms' private keys, as described in [ARO04].

- Integration of the agent control code, which manages the agent execution, and the agent setup code, where the protected itinerary, or any other initial data structure

required by the agent, is created.

- Easy code reuse, for MACPL built-in functions are generic and do not depend on any specific algorithm or implementation. Moreover, implementations are also independent of the agent's itinerary, the tasks executed on each platform, and the execution environment where agents run.

In addition to simplifying the implementation of agent protection protocols, our proposal also includes other tools intended for the end user, such as the Itinerary Designing Tool, which addresses the creation of the XML itinerary specification, and the Agent Launcher, for the introduction of new agents in remote platforms.

A proof-of-concept of the proposed development environment has been implemented using the Java language and the Jade agent platform [BCPR03]. The Agent Setup Module and the Control Code Module have been implemented using a MACPL to Java translator, which generates Java code that is then compiled to generate an executable bytecode. Nevertheless, further work is still required to complete this proof-of-concept, so that the protocols presented in chapters 4 and 5 can be implemented, and tests can be carried out to evaluate the performance of the resulting agents.

Chapter 7

Conclusions

Soon after mobile agents were first introduced in 1994 by White [Whi94], researchers envisaged that this technology would revolutionise the development of distributed applications. However, mobile agents have not met the expectations they raised in terms of widespread deployment and use.

The main reason why mobile agents have not been widely adopted yet, despite their technological benefits, is their inherent security risks. Many breakthroughs have been achieved in the security, reliability and efficiency of mobile agents, but there are security issues still remaining unsolved. Additionally, the complexity of developing the security solutions proposed to date has significantly hampered the deployment of this technology.

In order to provide a solution to these problems, this thesis has pursued two main objectives: First, to overcome some of the limitations of current agent protection protocols; and second, to simplify the implementation and use of security protocols for mobile agent-based applications.

More specifically, the first objective of this thesis has been to define an itinerary protection protocol that supports free-roaming agents. Protocols presented to date only allow agents to travel to platforms known beforehand. As a result, these approaches limit the mobile agent's ability to discover new platforms at runtime.

In order to achieve this goal, first of all, chapter 3 has presented a convenient way to

define explicit itineraries for free-roaming agents. Using explicit itineraries to create free-roaming agents promotes the reuse of these itineraries, for itineraries are defined in different stages that can be easily modified or changed for those of other agents. Besides, these itineraries are stored in separate data structures that are easier to protect using cryptographic mechanisms.

Next, chapter 4 has been devoted to present a novel protocol for the protection of dynamic itineraries. The proposed protocol is based on associating *discoverer* nodes with trusted platforms. The inclusion of trusted platforms in agent itineraries is a key characteristic of the contributions presented in this thesis. The public keys of *discoverer* nodes are used to protect the agent's code and data associated with dynamically located nodes. Thus, the information of dynamically located nodes remains protected during the whole agent execution.

The second objective of this thesis has been to define a protocol for the protection of mobile agents against external replay attacks. These attacks are performed by resending the agent to the same platform several times, so that the agent is forced to reexecute part of its itinerary. This can lead to, for example, unintended purchases in a shopping scenario. Solutions presented so far against replay attacks do not allow the agent to visit the same platform n times, especially if n is determined at runtime.

Chapter 5 has presented a protocol aimed at providing a satisfactory solution to external replay attacks. The proposed protocol is based on storing an agent trip marker inside platforms, which allows them to identify repeated attempts to execute the same agent. Unlike previous protocols previously presented, the trip marker is associated with an authorisation entity, which is trusted by the owner. The agent execution is only allowed if the agent trip marker has been generated and signed by the appropriate authorisation entity in the proper itinerary node.

In order to prove the validity of the proposed protocols, implementation and experimentation work has been conducted using the Jade agent platform [BCPR03]. Agents have been implemented following an agent-driven approach, which allows agents to manage their own itinerary and protection mechanisms. As a result, agents become more autonomous, and

platforms can easily support the execution of agents with different protection algorithms.

The results of the simulations have shown that the overhead introduced by the proposed protocols is around 55% of the execution time. This increase was completely acceptable for the simulated applications, but this may vary from one application to another.

The implementation of the proposed protocols has also demonstrated the complexity of providing security to mobile agent-based applications. As usual, there is a compromise between cost, security level and ease of use. For this reason, chapter 6 has presented a development environment aimed at simplifying the development of secure mobile agents. The key element of the proposed environment is the Agent Builder, which allows programmers to implement agent protection protocols using the Mobile Agent Cryptographic Protection Language (MACPL). MACPL has been designed to simplify the implementation of agent protection protocols, as well as to promote the reuse of these protocols in different applications.

In addition to the Agent Builder, other tools have been implemented to aid the programmer in the definition of the agent's itinerary, and in the introduction of new agents in their first itinerary platform.

The solutions proposed in this thesis represent a valuable contribution to the development of secure mobile agents. However, there are still problems remaining unsolved. In the next section, we outline some possible future research directions that could be explored to extend the results of this thesis.

7.1 Future work

The core work of this thesis revolves around the protection of mobile agents against malicious hosts. However, the opposite problem—the protection of platforms from malicious agents—is also an important issue that has hampered the adoption of this technology. Even though good solutions are already available to counter this problem, people are still reluctant to allow someone else's code to execute on their computers. Therefore, additional work needs to be undertaken to develop secure mobile agent frameworks in which the protection

from malicious hosts and from malicious agents are seamlessly integrated.

The itinerary protection protocol presented in chapter 4 fulfils the requirements initially established, but it does not support the protection of completely free-roaming agents. This is due to the fact that trusted platforms must be included in the itinerary, and they must be known by the owner in advance. Thus, investigating how the proposed protocol could be modified so as to support completely free-roaming agents is a possible avenue for future research.

Chapter 3 has presented a set of node types and properties which allow the definition of explicit itineraries for free roaming agents. One of these node types is the *set*, which is associated with two or more subitineraries that can be traversed in any order. One of the appealing uses of the *set* node type is to generate several clones of the agent to traverse the different subitineraries. Thus, programmers can parallelise the execution of different tasks and increase the performance of their applications.

However, mobile agent cloning introduces several problems, such as resolving identities properly when the replicas communicate with other agents or platforms. Regarding the replay protection protocol presented in chapter 5, cloning a mobile agent would imply generating a new trip marker for the replica. Otherwise, the agent and its replica would travel with the same trip marker, and this could lead to false replay attack detections. Nevertheless, new trip markers can only be generated by authorisation entities when new loop iterations have to be started. Therefore, the proposed protocol cannot be used if agents must be allowed to clone themselves.

A similar problem arises when the protocols presented in chapters 4 and 5 are implemented using an agent-driven approach. The agent has a unique agent identifier, which is used to bind together the control code and the protected itinerary. As the agent identifier is signed by the owner, new identifiers cannot be generated at runtime. Therefore, the agent and its replica cannot coexist in the same platform at the same time because they are using the same identifier.

Because agent cloning could be a desirable feature in some applications, further research should be conducted to explore how new agent identifiers and trip markers could be

generated at runtime when an agent is cloned.

The *discoverer* node presented in chapter 3 allows agents to discover multiple platforms for the execution of a single dynamically located node. This functionality, however, is not supported by the itinerary protection protocol presented in chapter 4. Assigning various platforms to the same node would involve using the same symmetric key to encrypt several itinerary entries. As a result, platforms would be able to use their symmetric keys to access or modify parts of the itinerary associated with other platforms. Consequently, further work could be conducted to improve the proposed protocol in this regard.

Finally, the development environment presented in chapter 6 facilitates the implementation and reuse of security protocols. However, only a proof-of-concept of this environment is currently available. Experiments have to be carried out in order to test the performance of the resulting agents, and more security techniques should be implemented in order to address other mobile agent security aspects, such as agent authentication or protection of computational results, among others. Thus, the development environment should provide a comprehensive set of security techniques that enabled the development of any kind of secure mobile agent-based application. Additionally, it would be interesting to provide a mechanism for selecting the appropriate technique or combination of techniques to use, depending on the execution environment and targeted application. Thus, this environment should be helpful for developers to better understand the design choices involved in the development of their secure mobile agent-based applications.

Appendix A

MACPL language specification

The Mobile Agent Cryptographic Protection Language (MACPL) is a domain-specific programming language used for the rapid development of mobile agent protection protocols. In the following sections, we will analyse its types, variables, operators, special keywords, functions and precompilation directives.

A.1 Data Types

MACPL is a language with strong, static typing, which means that type errors are found reliably at compile time. The following are the data types offered by MACPL:

Graph Graphs are created by surrounding GraphNode objects by '([' and '])'. GraphNode objects within a Graph are separated by commas. Graph objects are basically used to process the initial itinerary. The XML itinerary specification is translated into a Graph object using the `getInitialItinerary` function.

GraphNode GraphNode objects are created by surrounding its elements by parentheses. Each GraphNode contains the following information separated by commas: a node identifier, which is an Integer; the node data, which is an object of type List; and a list of node identifiers associated with the successor nodes.

List Lists objects are used to create compound structures of data, e.g. the protected explicit itinerary or the structure where agent results are stored. Lists are created by writing its elements in order, separated by ':' and surrounded by '[' and ']'. A List object can contain objects of any type.

Task Task objects encapsulate the tasks that are executed by the agent. MACPL provides no constructor to create new Task objects, for tasks are always read from files or from the XML itinerary specification directly. The `exec` built-in function is provided to execute a Task object.

Boolean Boolean objects have two possible values: *true* or *false*. These values are reserved keywords of the language.

String String objects contain a sequence of characters. Strings are created by surrounding the characters by double quotes. The String is an important MACPL type because it can encapsulate objects of other types. In order to convert a String object into another data type, MACPL provides several conversion functions, such as `string2Task`, `string2List`, etc. The inverse operations can also be performed using the corresponding conversion functions (`task2String`, `list2String`, ...).

Integer Integer objects are signed integers in the range of -2^{31} to $+2^{31} - 1$. Integers are basically used for indexing the elements of graphs and lists.

RuntimeDefined This type is used to deal with the data types provided by the agent programming language, which is the language supported by the agent execution environment. The data types of the agent programming language are not directly supported by MACPL, which means that type errors related with RuntimeDefined objects are detected at runtime, not at compile time.

A.2 Variables

The type of MACPL variables must be declared before using them. The type declaration can be made at the same time the first value is assigned to a variable. The assignment operator is the equal sign ('='). Variable identifiers can contain any sequence of alphanumeric characters and underscores. MACPL variables have 2 different types of scope:

Global scope Variables with a global scope can be accessed from anywhere within the entire MACPL code. They must be defined outside any function definition.

Function scope Variables with a function scope are only accessible within the function in which they are defined.

Global variables lose their value after an agent migration. However, the `export` keyword can be placed at the beginning of the type declaration so as to prevent this from happening.

A.3 Operators

The following is the list of operators used to perform integer arithmetic:

+ Addition of integers

– Subtraction of integers

***** Multiplication of integers

/ Multiplication of integers

% Modulus, returning the integer remainder.

The following is the list of operators used for list handling:

[ ] List creator. It creates a new list and introduces the elements between '[' and ']' in the list. List elements can be of any type, and are separated by ':'.

$< >$ List index operator. It allows expressions of the form varlist<i> to access the *i*th element of List varlist.

[|] Sublist generator. It allows expressions of the form:

```
[ExpressionWithIndex | IndexVariable,FirstIndex,LastIndex]
```

to evaluate ExpressionWithIndex from IndexVariable=FirstIndex to IndexVariable=LastIndex, and store the result in a list.

The following is the list of operators used for Graph creation:

([ ]) Graph creator. It creates a new Graph object and introduces the GraphNode objects between '([' and '])' in the graph. GraphNode objects must be separated by commas.

( ) GraphNode creator. Three elements separated by commas must be included inside this operator: the node identifier (Integer), the data (List), and the list of next node identifiers (List).

A.4 Functions

MACPL functions are defined using the fundef keyword. They can take a comma-separated list of parameters in parentheses, and return a value. The function code is surrounded by '{' and '}'. Function names must start with an alphabetic character. The rest of the identifier may include any sequence of alphanumeric characters and underscores. Function parameters are passed by value. Any variable defined within a function is only visible within that function.

A.5 Built-in functions

The following are the functions used for list handling:

- **length** (*List list*) : *Integer* Returns the number of elements in *list*.
- **join** (*List l1*, *List l2*) : *List* Returns a list that is the concatenation of *l1* and *l2*.
- **remove** (*Integer pos*, *List list*) : *List* Removes the element at position *pos* from *list*.
- **reverse** (*List list*) : *List* Returns the list resulting from reversing *list*.

The following are the functions used for graph handling:

- **firstNode** (*Graph graph*) : *GraphNode* Returns the first node of *graph*.
- **addNode** (*GraphNode n*, *Graph g*) : *Graph* Adds GraphNode *n* to Graph *g*.
- **removeNode** (*Integer id*, *Graph g*) : *Graph* Removes the GraphNode referenced by *id* from Graph *g*.
- **getNode** (*Integer id*, *Graph g*) : *GraphNode* Returns the GraphNode from *g* referenced by *id*.
- **graph2List** (*Graph graph*) : *List* Returns all the GraphNode objects of *graph* in a list.
- **joinGraphs** (*Graph g1*, *Graph g2*) : *Graph* Returns the graph resulting from the union of graphs *g1* and *g2*. If any node identifier is repeated in both graphs, this function raises an error.
- **nodeId** (*GraphNode n*) : *Integer* Returns the identifier of node *n*.
- **nodeData** (*GraphNode n*) : *Object* Returns the data contained in node *n*.
- **successors** (*GraphNode n*) : *List* Returns the list of node identifiers associated with the successors of node *n*.
- **predecessors** (*GraphNode n*) : *List* Returns the list of node identifiers associated with the predecessors of node *n*.

The following are the functions used to perform cryptographic operations:

- **aencrypt** (*String platform*, *String data*) : *String* Encrypts *data* using the public key of *platform*. This function implements the mechanism proposed in [ARO04], so that the agent will be able to decrypt the resulting encrypted data using the platform's private key.
- **adecrypt** (*String data*) : *String* Decrypts *data* using the current platform's private key. This decryption is performed by a call to the platform's public decryption function. In order for this call to succeed, *data* must have been encrypted using the mechanism proposed in [ARO04].
- **aencryptpk** (*RuntimeDefined publicKey*, *String data*) : *String* Encrypts *data* using an asymmetric key algorithm and the public key *publicKey*.
- **adecryptpk** (*RuntimeDefined privateKey*, *String data*) : *String* Decrypts *data* using an asymmetric key algorithm and the private key *privateKey*.
- **sign** (*String data*) : *String* Signs *data* using the current platform's private key.
- **signok** (*String data*) : *String* Signs *data* using the agent owner's private key.
- **signpk** (*RuntimeDefined privateKey*, *String data*) : *String* Signs *data* using the private key *privateKey*.
- **verify** (*RuntimeDefined publicKey*, *String data*, *String signedData*) : *Boolean* Verifies the signature of *signedData* using the public key *publicKey*.
- **verifyok** (*String data*, *String signedData*) : *Boolean* Verifies the signature of *signedData* using the owner's public key.
- **sencrypt** (*RuntimeDefined secretKey*, *String data*) : *String* Encrypts *data* using a symmetric key algorithm and the secret key *secretKey*.

- **sdecrypt** (*RuntimeDefined secretKey*, *String data*) : *String* Decrypts *data* using a symmetric key algorithm and the secret key *secretKey*.

- **skeygen** () : *RuntimeDefined* Generates a secret key for symmetric encryption and decryption.

- **keypairgen** () : *List* Generates a pair of a public and a private key. These keys are returned in a list with two elements: the first element is the public key, and the second is the private key.

- **getPublicKey** (*String platformAddress*) : *RuntimeDefined* Obtains the public key referenced by *platformAddress*.

- **hash** (*String data*) : *String* Returns a digest of *data* using a cryptographic hash function.

The following are the functions provided to perform conversions from and to String objects:

- **string2Graph** (*String str*) : *Graph* Converts *str* into a Graph object.

- **string2GraphNode** (*String str*) : *GraphNode* Converts *str* into a GraphNode object.

- **string2List** (*String str*) : *List* Converts *str* into a List object.

- **string2Task** (*String str*) : *Task* Converts *str* into a Task object.

- **string2Boolean** (*String str*) : *Boolean* Converts *str* into a Boolean object.

- **string2Integer** (*String str*) : *Integer* Converts *str* into an Integer object.

- **string2RuntimeDefined** (*String str*) : *RuntimeDefined* Converts *str* into a RuntimeDefined object.

- **graph2String** (*Graph graph*) : *String* Converts *graph* into a String object.

- **graphNode2String** (*GraphNode graphNode*) : *String* Converts *graphNode* into a String object.
- **list2String** (*List list*) : *String* Converts *list* into a String object.
- **task2String** (*Task task*) : *String* Converts *task* into a String object.
- **boolean2String** (*Boolean boolean*) : *String* Converts *boolean* into a String object.
- **integer2String** (*Integer int*) : *String* Converts *int* into a String object.
- **runtimeDefined2String** (*RuntimeDefined rd*) : *String* Converts *rd* into a String object.

In order to allow programs to perform different actions depending on a condition, MACPL provides the `if-else` statement. This statement can be used in two forms:

```
if (Boolean condition) {
   statements
}
```

or

```
if (Boolean condition) {
  statements
}
else {
  statements
}
```

The following are the functions used to evaluate conditions:

- **or** (*Boolean b1*, *Boolean b2*) : *Boolean* Returns true if *b1* or *b2* are true.
- **and** (*Boolean b1*, *Boolean b2*) : *Boolean* Returns true if *b1* and *b2* are true.
- **not** (*Boolean b*) : *Boolean* Returns true if *b* is false; otherwise it returns false.

The following are the functions used to test the equality of two objects:

- **eq** (*o1*, *o2*) : *Boolean* Returns true if *o1* and *o2* are equal. MACPL provides eight different versions of this function, one for each MACPL type.
- **ne** (*o1*, *o2*) : *Boolean* Returns true if *o1* and *o2* are not equal. MACPL provides eight different versions of this function, one for each MACPL type.

The following are the functions used for integer comparison:

- **gt** (*Integer int1*, *Integer int2*) : *Boolean* Returns true if *int1* is greater than *int2*.
- **ge** (*Integer int1*, *Integer int2*) : *Boolean* Returns true if *int1* is greater or equal to *int2*.
- **lt** (*Integer int1*, *Integer int2*) : *Boolean* Returns true if *int1* is lower than *int2*.
- **le** (*Integer int1*, *Integer int2*) : *Boolean* Returns true if *int1* is lower or equal to *int2*.

The following function is provided to allow MACPL code to terminate the execution in case of an unrecoverable error.

- **error** (*String message*) : *null* Causes the execution to be terminated, printing *message* to standard output.

The following are the functions provided to enable the implementation of the agent setup code. These functions can only be used in the first part of the MACPL specification, before the `#control_code_begin` precompilation directive.

- **getInitialItinerary** () : *Graph* Reads the explicit itinerary from the XML itinerary specification and returns a graph representation of it.
- **getControlCodeHash** () : *String* Returns a hash of the agent's control code, which is usually used as the unique agent identifier.

The following are the functions provided to enable the implementation of the control code of a mobile agent. These functions can only be used in the second part of the MACPL specification, after the `#control_code_begin` precompilation directive.

- **exec** (*Task t, String m*) : *String* Executes the method *m* of the task *t*.
- **move** (*String p*) : null Causes the agent to migrate to platform *p*. The migration is performed once the execution of the control code is finished.
- **clone** (*String p*) : null Sends a clone of the agent to platform *p*. The clone is sent to *p* once the execution of the control code is finished.
- **sendResults** (*String results*) : null Sends *results* to the agent owner.

The following are the functions provided to perform file input/output operations:

- **readFile** (*String path*) : *String* Reads the contents of file *path*.
- **writeFile** (*String path*, *String obj*) Writes *obj* to file *path*.

A.6 Keywords

nil Used to represent the empty list, the same as `[ ]`.

null Used to indicate that a given object has no assigned value.

true Used to represent one of the two legal values for a Boolean object.

false Used to represent one of the two legal values for a Boolean object.

last Used to address the last element of a list.

fundef Used to start a function definition.

return Used to return a value from a function.

export Used to declare a variable as exportable, so that it does not lose its value during migrations.

A.7 Precompilation directives

- `#control_code_begin` Separates the agent setup code from the agent control code.
- `#include filename` Enables the inclusion of file *filename* as a part of the MACPL specification.
- `#require interface` Extends the set of built-in functions available with those defined in *interface*.

A.8 Comments

Lines starting with '//' are ignored. End-of-line comments are not supported; comments must be on a line of their own.

Appendix B

MACPL implementation example

In order to demonstrate the simplicity and utility of MACPL, this chapter presents an example implementation of the itinerary protection protocol presented in [MB03], as described in section 2.3.2.

```
Graph initItin = getInitialItinerary();
List symmetricKeys = genKeys(1,[]);
String tripmarker = getControlCodeHash();
List initItinList = graph2List(initItin);
List protectedItin =
  [tripmarker:[protectNode(initItinList<i>)|i,1,last]];
String protItinString = list2String(protectedItin);
export List finalItin =
  [transition(0,1):protectedItin:signok(protItinString)];

fundef List genKeys (Integer nid,List list) {
  if (ne(getNode(nid,itinItin),null)) {
    genKeys(nid+1,join(list,[skeygen()]));
  } else {
    return list;
  }
}
fundef String protectNode(GraphNode node) {
```

```
  Integer i = nodeId(node);
  String data = list2String([getTask(node):transition(i,i+1)]);
  return sencrypt(symmetricKeys<i>,data);
}
fundef Task getTask(GraphNode node) {
  List taskPair = nodeData(node)<2>;
  String taskString readFile(taskPair<2>);
  return string2Task(taskString);
}
fundef List transition(Integer nid1, Integer nid2) {
  String platform1;
  if (ne(getNode(nid1,itinItin),null)) {
    platform1 = nodeData(getNode(nid1,itinItin))<3><2>;
  }
  else {
    platform1 = null;
  }
  if (ne(getNode(nid2,itinItin),null)) {
    String platform2 = nodeData(getNode(nid2,itinItin))<3><2>;
    List tr = [platform1:platform2:tripmarker:symmetricKeys<nid2>];
    String data = list2String([tr:signok(list2String(tr))]);
    return [platform2:aencrypt(platform2,data)];
  }
  else {
    return null;
  }
}

//-----------------
#control_code_begin
//-----------------

String protItinString = list2String(finalItin<2>);
if (not(verifyok(protItinString,finalItin<3>))) {
  error("Wrong owner signature");
```

```
}
List currentTransition = finalItin<1>;
List transitionList = string2List(adecrypt(currentTransition<2>));
List tr = transitionList<1>;
if (not(verifyok(list2String(tr),transitionList<2>))) {
  error("Wrong owner signature");
}
RuntimeDefined symmetricKey = tr<4>;
List protectedNodes = finalItin<2><2>;
List currentNode = getcurrentnode(1);
Task currentTask = currentNode<1>;
List nextTransition = currentNode<2>;
exec(currentTask,"main");
if (ne(nextTransition,null)) {
  finalItin = [nextTransition:finalItin<2>:finalItin<3>];
  move(nextTransition<1>);
}

fundef List getcurrentnode(Integer pos) {
  String decryptNodeString = sdecrypt(symmetricKey,protectedNodes<pos>);
  if (ne(decryptNodeString,null)) {
    return string2List(decryptNodeString);
  }
  else {
    getcurrentnode(pos+1);
  }
}
```

As shown in the above example implementation, the MACPL code is divided into two parts: the agent setup, where the protected itinerary is created; and the control code, where the management of this protected itinerary is carried out. These two parts are separated by the `#control_code_begin` precompilation directive.

Regarding the protection of the initial itinerary, first of all, the XML itinerary specification is translated into a Graph object using the `getInitialItinerary` built-in function, and the resulting object is stored in the `initItin` variable. Then, the `genKeys` function generates a random symmetric key for each itinerary node, and the resulting list of keys is stored in the `symmetricKeys` variable.

Next, a unique agent identifier is generated and stored in the `tripmarker` variable. This identifier is constructed as a hash of the agent's control code, which is obtained from a call to the `getControlCodeHash` built-in function.

Then, using the `graph2List` built-in function, `initItin` is converted into a list, and the `protectNode` function is applied to every element of this list. `protectNode` takes a GraphNode object as a parameter, and generates its corresponding entry of the protected itinerary. This entry contains the current task, which is extracted using the `getTask` function, and the transition to the subsequent node, which is generated by the `transition` function. Each itinerary entry is encrypted using the corresponding symmetric key (`symmetricKeys` variable), and the resulting protected itinerary is stored in the `finalItin` variable.

As can be seen, the `finalItin` variable is declared as exportable using the `export` keyword. This makes `finalItin` available to the control code after migrating to the first itinerary platform.

The operations carried out by the control code are defined in the second part of the MACPL specification, which is started by the `control_code_begin` precompilation directive.

First of all, the protected itinerary is obtained from the `finalItin` variable, and its signature is verified using the `verifyok` built-in function. Then, the `adecrypt` built-in function is used to decrypt the transition placed at the beginning of `finalItin`. The signature of the decrypted transition is verified as well, and the symmetric key associated with the current node is obtained.

Next, the `getcurrentnode` function is called to obtain and decrypt the current entry

of the protected itinerary. Thus, the current node is stored in the `currentNode` variable. The current task and the transition to the next itinerary node are extracted from `currentNode`. The current task is executed using the `exec` built-in function, and the agent is migrated to its next destination.

This MACPL specification shows that not many lines of code are needed to implement an itinerary protection protocol. Even though the protocol implemented in this example only supports sequential itineraries, because it is a reduced version of the one proposed in [MB03], the code shows that MACPL can significantly simplify the implementation of security protocols.

Bibliography

[AGK$^+$01] J. Altmann, F. Gruber, L. Klug, W Stockner, and E. Weippl. Using Mobile Agents in Real World: A Survey and Evaluation of Agent Platforms. In *Proceedings of the 2nd Int. Workshop on Infraestructure for Agents, MAS, and Scalable MAS*, Montreal, Canada, 2001.

[ARB03] J. Ametller, S. Robles, and J. Borrell. Agent Migration over FIPA ACL Messages. In *Mobile Agents for Telecommunication Applications (MATA)*, volume 2881 of *Lecture Notes in Computer Science*, pages 210–219. Springer Verlag, 2003.

[ARO04] J. Ametller, S. Robles, and J. A. Ortega. Self-Protected Mobile Agents. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04)*, pages 362–367. IEEE Computer Society, 2004.

[Aur97] T. Aura. Strategies against Replay Attacks. In *Proceedings of the Computer Security Foundations Workshop*, pages 59–68. IEEE Computer Society, 1997.

[BCG07] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

[BCPR03] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE - A White Paper. Technical report, Telecom Italia Lab, 2003. Available at http://jade.cselt.it.

[Bor02] N. Borselius. Mobile agent security. *Electronics & Communication Engineering Journal*, 14(5):211–218, 2002.

[BRSR99] J. Borrell, S. Robles, J. Serra, and A. Riera. Securing the Itinerary of Mobile Agents through a Non-repudiation Protocol. In *Proceedings of the IEEE Int. Carnahan Conf. on Security Technology*, pages 461–464. IEEE Computer Society, 1999.

[CAOR+05] J. Cucurull, J. Ametller, J. A. Ortega-Ruiz, S. Robles, and J. Borrell. Protecting Mobile Agent Loops. In *Mobility Aware Technologies and Applications*, volume 3744 of *Lecture Notes in Computer Science*, pages 74–83. Springer-Verlag, 2005.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[CLSY06] H. Che, D. Li, J. Sun, and H. Yu. A Novel Solution of Mobile Agent Security: Task-Description-Based Mobile Agent. *IJCSNS International Journal of Computer Science and Network Security*, 6(2B):121–125, 2006.

[DELM00] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications: Proceedings of ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. In *RFC 4346*. IETF, 2006.

[FM99] S. Fünfrocken and F. Mattern. Mobile Agents as an Architectural Concept for Internet-based Distributed Applications-The WASP Project Approach. In

Proceedings of Kommunikation in Verteilten Systemen (KiVS '99), pages 32–43. Springer-Verlag, 1999.

[GKCR98] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a Multiple-Language, Mobile-Agent System. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer Verlag, 1998.

[GMB+08] C. Garrigues, N. Migas, W. Buchanan, S. Robles, and J. Borrell. Protecting mobile agents from external replay attacks. *Journal of Systems and Software*, 2008. doi:10.1016/j.jss.2008.05.018.

[GMM98] R. H. Guttman, A. G. Moukas, and P. Maes. Agents as Meditors in Electronic commerce. *International Journal of Electronic Markets*, 8(1):22–27, 1998.

[Gra95] R. S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of CIKM'95 Workshop on Intelligent Information Agents*, 1995.

[GRB08a] C. Garrigues, S. Robles, and J. Borrell. Securing dynamic itineraries for mobile agent applications. *Journal of Network and Computer Applications*, 2008. doi:10.1016/j.jnca.2007.12.002.

[GRB08b] C. Garrigues, S. Robles, and J. Borrell. Método para la protección de plataformas de computación frente a ataques externos de repetición de agentes móviles y sistema de plataformas de computación protegidas. Spanish Patent Pending P200801492, filed 2008.

[HCK97] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer Verlag, 1997.

[HN05] A. Hijazi and N. Nasser. Using Mobile Agents for Intrusion Detection in Wireless Ad Hoc Networks. In *Proceedings of the 2nd IFIP Int. Conf. on Wireless and Optical Communications Networks (WOCN '05)*, pages 362–366. IEEE Computer Society, 2005.

[Hoh98] F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 92–113. Springer Verlag, 1998.

[HR99] F. Hohl and K. Rothermel. A Protocol Preventing Blackbox Tests of Mobile Agents. In *Proceedings of Kommunikation in Verteilten Systemen (KiVS '99)*, pages 170–181. Springer-Verlag, 1999.

[JK00] W. Jansen and T. Karygiannis. NIST Special Publication 800-19 - Mobile Agent Security, 2000.

[KA98] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. In *RFC 2401*. IETF, 1998.

[KAG98] G. Karjoth, N. Asokan, and C. Gülcü. Protecting the computation results of free-roaming agents. In *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 195–207. Springer Verlag, 1998.

[Kar00] G. Karjoth. Secure Mobile Agent-Based Merchant Brokering in Distributed Marketplaces. In *Proceedings of the 2nd Int. Symp. on Agent Systems and Applications and 4th Int. Symp. on Mobile Agents (ASA/MA '00)*, volume 1882 of *Lecture Notes in Computer Science*, pages 44–56. Springer Verlag, 2000.

[KBD02] H. Kuang, L. F. Bic, and M. B. Dillencourt. Iterative Grid-Based Computing Using Mobile Agents. In *Proceedings of the Int. Conf. on Parallel Processing (ICPP '02)*, pages 109–117. IEEE Computer Society, 2002.

[KLM03] M. Klusch, S. Lodi, and G. Moro. Agent-Based Distributed Data Mining: The KDEC Scheme. In *Intelligent Information Agents: The AgentLink Perspective*, volume 2586 of *Lecture Notes in Computer Science*, pages 104–122, 2003.

[Kna95] Frederick C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[KT01] Neeran M. Karnik and Anand R. Tripathi. Security in the Ajanta mobile agent system. *Software Practice and Experience*, 31(4):301–329, 2001.

[KT04] Y. Kambayashi and M. Takimoto. A Functional Language for Mobile Agents with Dynamic Extension. In *Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems KES'04*, volume 3214 of *Lecture Notes in Computer Science*, pages 1010–1017. Springer-Verlag, 2004.

[LCT+05] R. Levy, P. S. Carlos, A. Teittinen, L. S. Haynes, and C. J. Graff. Mobile agents routing - A survivable ad-hoc routing protocol. In *Proceedings of the IEEE Military Communications Conference (MILCOM '05)*, volume 5, pages 2903–2909. IEEE Computer Society, 2005.

[LF06] T. Lu and M. Fu. Using Mobile Agents for Object Sharing in P2P Networks. In *Proceedings of the 1st Int. Conf. on Innovative Computing, Information and Control (ICICIC '06)*, volume 1, pages 741–744. IEEE Computer Society, 2006.

[LMSF04] Emerson F. A. Lima, Patrícia D. L. Machado, Flávio R. Sampaio, and Jorge C. A. Figueiredo. An Approach to Modelling and Applying Mobile Agent Design Patterns. In *SIGSOFT Software Engineering Notes*, volume 29, pages 1–8. ACM Press, 2004.

[LO99] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[LOW97] Jacob Y. Levy, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. Technical report, Sun Microsystems, Inc., 1997.

[LSL00] T. Li, C. Y. Seng, and K. Y. Lam. A Secure Route Structure for Information Gathering Agent. In *Proceedings of the 3rd Pacific Rim Int. Workshop on Multi-Agents: Design and Applications of Intelligent Agents*, volume 1881 of *Lecture Notes in Artificial Intelligence*, pages 101–114. Springer-Verlag, 2000.

[MB03] J. Mir and J. Borrell. Protecting Mobile Agent Itineraries. In *Mobile Agents for Telecommunication Applications (MATA)*, volume 2881 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, 2003.

[Mil99] Deja Milojicic. Mobile Agent Applications. *IEEE Concurrency*, 07(3):80–90, 1999.

[MLH05] Vishal D. Modak, David D. Langan, and Thomas F. Hain. A pattern-based development tool for mobile agents. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 72–75. ACM Press, 2005.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[MS03] P. Maggi and R. Sisto. A configurable mobile agent data protection protocol. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03)*, pages 851–858. ACM Press, 2003.

[MV07] S. S. Manvi and P. Venkataram. Mobile agent based approach for QoS routing. *IET Communications*, 1(3):430–439, 2007.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[MY06] Qusay H. Mahmoud and Leslie Yu. Making Software Agents User-Friendly. *Computer*, 39(7):94–96, 2006.

[NL96] G. Necula and P. Lee. Proof-Carrying Code. Technical report, School of Computer Science, Carnegie Mellon University, September 1996.

[RMAB02] S. Robles, J. Mir, J. Ametller, and J. Borrell. Implementation of Secure Architectures for Mobile Agents in MARISM-A. In *Mobile Agents for Telecommunication Applications (MATA)*, volume 2521 of *Lecture Notes in Computer Science*, pages 182–191. Springer-Verlag, 2002.

[Rob02] S. Robles. *Mobile Agent Systems and Trust, a Combined View toward Secure Sea-of-Data Applications*. PhD thesis, Universitat Autònoma de Barcelona, 2002.

[Rot99] V. Roth. Mutual protection of co-operating agents. In *Secure internet programming: Security issues for mobile and distributed objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 275–285, 1999.

[Rot01a] V. Roth. On the robustness of some cryptographic protocols for mobile agent protection. In *Proceedings of the 5th Int. Conf. on Mobile Agents (MA '01)*, volume 2240 of *Lecture Notes in Computer Science*, pages 1–14. Springer Verlag, 2001.

[Rot01b] V. Roth. Programming Satan's agents. In *Proceedings of the 1st Workshop on Security of Mobile Multiagent Systems (SEMAS '01)*, volume 63 of *Electronic Notes in Theoretical Computer Science*, pages 124–139. Elsevier, 2001.

[Rot02] V. Roth. Empowering Mobile Software Agents. In *Proc. 6th IEEE Mobile Agents Conference*, volume 2535 of *Lecture Notes in Computer Science*, pages 47–63. Springer Verlag, 2002.

[SF05] A. Suna and A. E. Fallah-Seghrouchni. A mobile agents platform: architecture, mobility and security elements. In *Programming Multi-Agent Systems (ProMAS 2004)*, volume 3346 of *Lecture Notes in Computer Science*, pages 126–146. Springer-Verlag, 2005.

[SRM98] M. Straßer, K. Rothermel, and C. Maiöfer. Providing Reliable Agents for Electronic Commerce. In *Proceedings of the International IFIP/GI Working Conference*, volume 1402 of *Lecture Notes in Computer Science*, pages 241–253. Springer-Verlag, 1998.

[ST98] T. Sander and C. F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, page 44. Springer-Verlag, 1998.

[Sue03] A. Suen. *Mobile Agent Protection With Data Encapsulation And Execution Tracing*. PhD thesis, The Florida State University, 2003.

[Syv94] P. Syverson. A Taxonomy of Replay Attacks. In *Proceedings of the Computer Security Foundations Workshop*, pages 131–136. IEEE Computer Society, 1994.

[TM01] H. K. Tan and L. Moreau. Trust Relationships in a Mobile Agent System. In *Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2001.

[TOH99] Y. Tahara, A. Ohsuga, and S. Honiden. Agent System Development Method Based on Agent Patterns. In *Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems ISADS '99*. IEEE Computer Society, 1999.

[TOH01] Y. Tahara, A. Ohsuga, and S. Honiden. Mobile agent security with the IPEditor development tool and the mobile UNITY language. In *Proceedings of the Fifth International Conference on Autonomous Agents AGENTS '01*, pages 656–662. ACM Press, 2001.

[Vig98] G. Vigna. Cryptographic Traces for Mobile Agents. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 137–153. 1998.

[VMRC+06] P. Vieira-Marques, S. Robles, J. Cucurull, R. Cruz-Correia, G. Navarro, and R. Martí. Secure Integration of Distributed Medical Data using Mobile Agents. *IEEE Intelligent Systems*, 21(6):47–54, 2006.

[Whi94] J. E. White. Telescript technology: the foundation for the electronic marketplace. Technical report, General Magic, Inc., 1994.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216. ACM Press, 1993.

[WPW+97] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. In *Mobile Agents: First International Workshop*, volume 1219 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 1997.

[WQ04] X. Wang and H. Qi. Mobile agent based progressive multiple target detection in sensor networks. In *Proceedings of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP '04)*, volume 2, pages 285–288. IEEE Computer Society, 2004.

[WSB98] U. G. Wilhelm, S. Staamann, and L. Buttyan. On the problem of trust in mobile agent systems. In *Proceedings of the Symposium on Network and Distributed System Security*. Internet Society, 1998.

[WSUK00] D. Westhoff, M. Schneider, C. Unger, and F. Kaderali. Protecting a Mobile Agent's Route against Collusions. In *Proceedings of the 6th Annual Int. Workshop Selected Areas in Cryptography (SAC '99)*, volume 1758 of *Lecture Notes in Computer Science*, pages 215–225. Springer-Verlag, 2000.

[Yee99] B. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 261–273. Springer Verlag, 1999.

[Yee03] B. Yee. Monotonicity and partial results protection for mobile agents. In *Proceedings of the 23rd Int. Conf. on Distributed Computing Systems*, pages 582–591. IEEE Computer Society, 2003.

[Zac03] J. Zachary. Protecting Mobile Code in the Wild. *Internet Computing, IEEE*, 7(2):78–82, 2003.

[ZCM02] A. Zunino, M. Campo, and C. Mateos. Simplifying Mobile Agent Development through Reactive Mobility by Failure. In *Advances in Artificial Intelligence: SBIA'02*, volume 2507 of *Lecture Notes in Computer Science*, pages 163–174. Springer-Verlag, 2002.

[ZG96] J. Zhou and D. Gollmann. Observations on non-repudiation. In *Advances in Cryptology (ASIACRYPT '96)*, volume 1163 of *Lecture Notes in Computer Science*, pages 133–144. Springer-Verlag, 1996.

[ZOL04] J. Zhou, J. A. Onieva, and J. Lopez. Analysis of a free roaming agent result-truncation defense scheme. In *Proceedings of the IEEE Int. Conf. on e-Commerce Technology (CEC '04)*, pages 221–226. IEEE Computer Society, 2004.

Carles Garrigues Olivella

Bellaterra, June 2008