UNIVERSITY OF SOUTHAMPTON

INTERACTION TRACING FOR MOBILE AGENT SECURITY

by

Victor Hock Kim TAN

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Engineering and Applied Science
School of Electronics and Computer Science

March, 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

INTERACTION TRACING FOR MOBILE AGENT SECURITY

by Victor Hock Kim TAN

This thesis develops a new technique, interaction tracing, to address the security issue of protecting mobile agents from potentially malicious hosts. In this technique, a mobile agent is modeled as a black box whose behaviour can be captured through a trace of its inputs and outputs during the process of execution. Formalization of the activity of creating and verifying traces is detailed for a simple agent programming language using operational semantics. An interaction protocol is developed to enable secure exchange of traces between entities in the system that are responsible for verifying the validity of the traces. This protocol is formally modeled and verified for specific security properties using a finite-state model checker. The protocol is extended to allow for the activity of trace reconciliation, which protects inter-agent communication between mobile agents operating in a multi-agent context. Implementation of this secure protocol in conjunction with the interaction tracing activity is undertaken in a mobile agent framework and is quantitatively evaluated against a non-secure mobile agent system and standard client-server approach. A trust model is introduced in the context of the protocol that allows trust relationships to be formed between the various entities in the system, permitting a more flexible deployment of the protocol.

Contents

List of Figures

Acknowledgements

To my family in Malaysia, who have been a constant source of unwavering support throughout the duration of my PhD studies: this thesis is dedicated to them. My achievement here is as much theirs as it is mine. A big note of gratitude is also due to my supervisor, Dr. Luc Moreau, for his guidance and patience, particularly during the trying period of writing up the thesis.

I would like to thank all my friends whom I have been privileged to know during my stay at Southampton. Their camaraderie, support and laughter has made my stay at the IAM lab a very pleasant one. In particular, my numerous 'potluck' comrades have contributed to some interesting trips, gatherings and (sometimes inane !) conversations on every topic imaginable. The memories of these occasions will continue to be treasured for a long time to come.

A note of thanks is due to all my former class mates and ex-students in Malaysia who have, from time to time, maintained a lively correspondence through e-mail. It is edifying to know that friendships can be maintained over distance and time.

Last but not least, I owe a big debt to my friends: Wade, Doug and Hari for providing a ready ear and helping me work through some of my personal issues during my course of study. A friend in need is a friend indeed !

At times our own light goes out and is rekindled by a spark from another person. Each of us has cause to think with deep gratitude of those who have lighted the flame within us - Albert Schweitzer

Chapter 1

Introduction

The mobile agent paradigm has been proposed as a new paradigm for distributed computation that offers a more flexible alternative to traditional client-server computing environments, particularly on the Internet [80]. Several advantages have been identified for utilizing mobile agents in a distributed environment [85]. The most frequently cited advantage involves reduction of network traffic, achieved by moving the required computation (encapsulated in a mobile agent) to a server whereby all further interactions with the server become localized. Other advantages include support for mobile users utilizing mobile devices, asynchronous computations in disconnected operation as well as increased stability in dealing with unexpected network failures [113].

There have been numerous mobile agent systems (D'Agents [54], Aglets [139], NOMADS [137]) developed that have demonstrated positive results in favour of the performance of mobile agent systems when they are quantitatively evaluated against traditional client-server systems in specific application scenarios [55]. However, there is still an apparent unwillingness to adopt this new paradigm widely. One of the most frequently cited concerns regarding this reluctance to employ mobile agents on a wider scale is the issue of security [38]. In general, the two main areas of consideration are host security and code security.

Host security involves protecting the host platform (which provides the environment and resources for agent execution) from malicious agents that may attempt to gain unauthorized access to local resources or initiate malicious actions on the platform. This aspect of security is reasonably well researched, particularly in the operating systems community, and is roughly analogous to the well known security concern regarding viruses (as well as Trojan horses and worms) which arises whenever executing code downloaded from an untrusted source. Correspondingly, there are many viable mechanisms that have been developed to address host security for mobile agents. Examples of such mechanisms include sandbox security (used in Java to provide access control) [51], safe-typed languages [152] and proof-carrying code [108].

Code security is however more problematic, since this aspect has only come into prominence recently as a security problem unique to mobile agents. The general assumption prevalent in existing security research is that code that is distributed and executed in environments remote from the one that it was composed in (for example, Java applets) only represents a threat to those remote environments (host security). The need to examine the possibility of a host platform interfering with execution of code in its own environment does not arise at all, since it is assumed that any damage resulting from such interference would have a resulting adverse affect solely on the operators of that host platform. For the case of the mobile agent paradigm however, an agent representing a user's interest in some manner could be transported from the environment controlled by the user to another host platform not under the user's control, where it is subsequently executed. In the environment of this remote host platform, incorrect execution of the code could result in the compromise of the user's interests. In particular, the controllers of this remote host platform may stand to gain some advantage in a compromise of this nature.

As an example, consider the typical e-commerce shopping scenario for mobile agents [58]. This usually involves a mobile agent representing a user's specific interests being launched into a network where it migrates through an itinerary of host platforms that represent on-line shopping sites. On each hop, the execution of the mobile agent would constitute an interaction between the agent and the shopping site, generally taking the form of inquiry regarding the price of a particular item that the user is interested in. The action would be repeated on each of the platforms until the agent eventually migrated back to the user's environment, carrying pertinent information such as the list of prices for that item or the identity of the site with the best offer.

In such a scenario, there now clearly exists an economic gain to be achieved (or conversely, damage to be inflicted on the user) if the remote host platform executing the agent code could tamper with its execution in a specific manner. The most obvious method would be to modify the list of existing prices accumulated by the agent in its previous hops in order to convey the false impression to the user that the price offered by the tampering host platform is the best option available. In this context, code security attempts to address this risk either by providing some form of protection to the mobile agent that safeguards against such illegal tampering, or by making such tampering detectable by the user or some other trusted third party.

Some of the more well known code security mechanisms that have been proposed include code obfuscation [64], encrypted functions [128], execution tracing [148] and tamper resistant hardware [157], [163]. Most of the solutions proposed so far have been conceptual, and it is likely that this area will be crucial in determining the future viability of mobile agent applications in distributed e-commerce scenarios. The primary motivation of this thesis is to develop a practical solution to the code security issue in a systematic manner which enables the clear exposition and verification of the security properties achieved.

1.1 Contributions of the thesis

The primary contributions of this thesis revolve around the new code security technique, *interaction tracing*, as well as the security protocol developed to support it. The contributions are as follows:

1. The concept of interaction tracing is derived and developed from the original motivations of execution tracing [148]. Interaction tracing adopts the viewpoint that an agent can be considered as a black box whose behaviour is essentially described through the basis of its interactions (inputs and outputs) with its external environment. Hence an accurate description of its behaviour can be encapsulated in the form of a record (or trace) of these inputs and outputs. This approach preserves the core idea underlying execution tracing while offering some distinct advantages: a tractable version of execution tracing, a less complicated implementation as well as language independence. The motivation and the development of interaction tracing is described at the end of Chapter 3 and in the whole of Chapter 4.
2. The overwhelming majority of code security techniques developed so far (including execution tracing) discuss security in a purely qualitative manner.

A formal definition of the security properties achievable through the use of a given technique would permit a better understanding of the nature of security and its limitations in the given context. With this motivation in mind, an operational semantics is proposed to formalize the process of code execution, creation and verification of a trace in the interaction tracing technique. This allows a formal establishment of the meaning of verifying the execution of an agent in an untrusted environment within the context of interaction tracing.

3. In order for the traces produced by interaction tracing to be used properly, a protocol must be implemented to allow them, along with mobile agent code, to be transmitted securely between various entities in a mobile agent system. Such a protocol exists in the original execution tracing approach (as well as in other code security techniques), but they have never been modelled formally or verified to have attained the security properties that they were designed to achieve. With this motivation in mind, we develop a secure protocol for exchanging traces that attempts to take into account the need to safeguard against the possibility of interleaving attacks on secured communication protocols [90]. This is accomplished by formally modelling and verifying the new protocol utilizing a model checker. This contribution is detailed in Chapter 6.

4. The large majority of code security techniques aim to safeguard mobile agents that work in isolation (i.e. the agent does not interact with other agents during the course of traversing its network itinerary). For the case of mobile multi-agent systems, two mobile agents executing on untrusted host platforms may communicate with each other. In this instance, there is a possibility that either one or both of these host platforms will tamper with the outgoing or incoming communications. An agent that unwittingly accepts tampered information from a compromised agent will be subsequently affected as well since its future behaviour may very well depend on this tampered information. This issue has yet to be addressed significantly in code security research. In interaction tracing, this issue is addressed through the use of *reconciliation* of traces and is supported by an extension to the existing trace exchange protocol. This contribution is also described in Chapter 6.

5. There have been numerous quantitative studies performed to evaluate the performance of mobile agent systems against standard client-server configurations in various application scenarios ([55], [34]). However, all these stud-

ies involve mobile agent systems that do not incorporate any code security technique. Consideration of security in quantitative evaluation is important because it affects one of the fundamental arguments for using mobile agents in the first place (i.e. more efficient performance than client-server configurations). In particular, it is interesting to examine to what extent the additional costs incurred by the use of a code security technique would offset the original cost savings achieved by employing a mobile agent approach as opposed to a client-server approach in a given application context. This quantitative evaluation is described in Chapter 7.

6. The study of security in the research area of multi-agent systems tends to be focused on concepts like trust and reputation which revolve around the interaction of agents, while mobile agent security concerns itself with host and code security areas. It would be interesting to explore the possibility of incorporating security concepts such as trust from multi-agent systems research into the trace exchange protocol. Trust is useful here by providing a way to increase the flexibility of deployment of the trace exchange protocol. Chapter 8 examines how the trust is used in the trace exchange protocol.

1.2 Thesis structure

The next two chapters of the thesis present a literature review of mobile agents in general as well mobile agent security issues. The remaining chapters present the various contributions of the thesis that have been outlined in the preceding section.

Chapter 2. Mobile agents: Introduction. The second chapter outlines the advantages generally associated with the mobile agent paradigm, as well as some of the application areas that it is employed in.

Chapter 3. Mobile Agent Security. The third chapter discusses the specific techniques of host and code security and notes their attendant characteristics and drawbacks. A survey of some better known mobile agent systems is conducted with emphasis on how code and/or host security is achieved in these systems. An analysis of the various code security techniques is undertaken and arguments are presented to support our viewpoint that the execution tracing methodology [148] presents the most promising approach in this area. The existing drawbacks in execution tracing are identified and we outline the various approaches that are required to improve on these drawbacks.

Chapter 4. Interaction Tracing. The fourth chapter presents the motivation behind and the application of the new concept of interaction tracing. The use of interaction tracing in a typical on-line shopping scenario is demonstrated to illustrate its efficiency and suitability in comparison to the execution tracing approach.

Chapter 5. Formalizing Interaction Tracing. The fifth chapter is concerned with formalizing the activity of creating and verifying a trace in interaction tracing using an operational semantics approach. Two possible cases for agent code execution are considered in modelling: single threaded and multi-threaded with shared state. The evaluation of programs and verification of traces is defined, and lemmas that describe the specific security properties achievable are presented and proved.

Chapter 6. Security Protocol for Trace Exchange The sixth chapter details an interaction protocol for securely exchanging agent code as well as traces produced in the interaction tracing approach. The protocol is formally modelled and verified using the FDR finite-state model checker. The concept and the need for *reconciliation* of traces is introduced, and an additional protocol for achieving this is provided, modelled and verified in a similar manner.

Chapter 7. Evaluation of Protocol Implementation. The seventh chapter describes a practical implementation of the protocol in the SoFAR mobile agent system [106]. A quantitative evaluation of the performance cost of this secure mobile agent system in a sample application scenario is undertaken and contrasted against the performance of an unsecured mobile agent implementation as well as a standard client-server approach.

Chapter 8. Trust in the Trace Exchange Protocol. The eighth chapter describes how the concept of trust can be incorporated into the trace exchange protocol developed to support interaction tracing. Trust relationships between the various entities involved in the protocol are detailed and an algorithm is developed to show how these relationships can be combined to form new relationships which extend the flexibility of the trace exchange protocol.

Chapter 9. Conclusions and Future Work. The last chapter summarizes the work of the thesis and discusses future research directions that arise from it.

Chapter 2

Mobile Agents: Introduction

This chapter outlines the advantages of the mobile agent paradigm (section 2.1), explores some of the more widely employed applications for mobile agents (section 2.2) and concludes with a discussion of some of the issues affecting the development and deployment of mobile agent systems (section 2.3).

2.1 Advantages of the mobile agent paradigm

The history behind the development of the mobile agent paradigm is presented in detail in [113]. There are several advantages that have been associated with the use of mobile agents [85], which are briefly described below.

1. *Reducing network load.* Communication protocols in distributed systems may involve multiple interactions to accomplish a given task. In some applications such as information retrieval and filtering, large amounts of raw data may be transferred through the network. A series of interactions can be packaged as a mobile agent and shipped off to the remote site, where any further interactions with a data repository would be localized. The primary advantage to be gained here is based on the motivation that the network load arising from the transfer of agent code across the network to a remote site and back again is considerably less than the traffic generated from a client-server type interaction that a stationary agent would need to engage in with the same remote site. The use of mobile agents in this manner is particularly useful when working with mobile devices such as laptops and PDAs that have intermittent, low bandwidth connections to the main network.

2. *Overcoming network latency.* Critical real time systems, such as robots or sensors in manufacturing processes, need to respond in real time to changes in their environments. Controlling such systems via RPC calls in a large network may involve unacceptable network latencies arising from unpredictable and unbounded communication delays. Mobile agents offer a solution in that they can be dispatched from a central controller to act locally at a remote system. Executing asynchronously and autonomously, they can also monitor events on the remote machines and alert a user of the occurrence of specific events, hence negating the need for periodical checking.

3. *Providing dynamic adaptation and cooperation.* Mobile agents can sense changes in their execution environment and react autonomously to them. They are also capable of engaging in cooperative action with other mobile agents through appropriate communication mechanisms. This allows a complex task to be partitioned to different parts that can be allocated to individual agents that can process them in parallel. Mobile agents working in this manner can also distribute themselves among the hosts in a network so that they maintain the optimal configuration for solving a particular problem.

4. *Upgrading protocols and software.* In the exchange of data in a distributed system, each host owns the code that implements the protocols used in its interactions with other remote hosts. Upgrading protocols on the various hosts may be difficult for a large network, resulting in a possible legacy problem. A new protocol can be encapsulated within a mobile agent which, on the basis of specific network events, can migrate autonomously in the network to the appropriate hosts to initiate interactions using the new protocol. This eliminates the need for direct user intervention in the upgrading process. Software applications on remote distributed hosts can also be updated or reconfigured through the use of mobile agents in a similar manner.

2.2 Applications of the mobile agent paradigm

Here, we survey the various application areas that are generally associated with mobile agent usage, usually based on exploiting one or more of the advantages that were described in the previous section. It should be noted that some applications categorized under a particular area may demonstrate characteristics or features that would fit equally well in another area.

2.2.1 Distributed information retrieval and processing

Information retrieval and processing in a distributed system is likely to involve transferring large amounts of data across the network at some stage when using a standard client-server interaction paradigm. Mobile agents can be deployed in different ways to obtain the advantage of reduction in network load in such a situation. The Concordia mobile agent framework [159] has been utilized to enable cooperative information gathering, which involves combining information from multiple, heterogeneous information sources. The DAgents mobile agent system [54] has been employed in searching distributed collections of technical reports, where planning algorithms were used to select the best migration path through the network as well as permitting agents to efficiently detect changes within a changing set of documents [21]. Another application along a similar theme involves a compound document that can be dynamically composed of a hierarchy of mobile agents and is capable of migrating itself over a network as a whole. The mobile document can be used in structuring e-mail and workflow management systems in more flexible ways [129]. The mobile agent approach has also been utilized to improve the performance of web crawling in search engines by uploading agent code to web sites, where it crawls pages locally and sends a summary back to the search engine [146].

2.2.2 Network management

Traditional network management [27] revolves around the use of a client-server model, where a central administrator (the client) collects and analyzes data from other network components, which act as servers. More flexible network models [145] have enabled the development of decentralized network management models, where network management code can be decomposed into smaller modules whose execution can be delegated to the various network components themselves, allowing for greater scalability and dynamicity. Mobile agents could encapsulate such modules and migrate to specific network nodes, where they can then analyze and retrieve local information, install and execute code and make autonomous decisions that take into account localized network conditions [9], [14]. In this way, they exploit the advantages of overcoming network load and latency as well as providing protocol updating. Mobile agents are exploited in [126] to provide local interactions with fixed SNMP agents. Quality of service negotiation is delegated to mobile agents enriched with contracting capability in order to simplify the negotiation protocol and to reduce the complexity of network design [112]. Another area of interest involves using techniques such as organization modelling of agent relationships to enable mobile agents to perform complex tasks such as on-line state based quality of service routing [158]. The advantage of reconfiguring and updating systems offered by mobile agents is also exploited in other areas. In [78], mobile agents are used to dynamically reconfigure and customize different types of environment from PDAs to embedded systems within a CORBA object oriented framework.

2.2.3 Enhancing the use of mobile devices

The advantage of reduction of network load makes the use of mobile agents quite attractive when used in conjunction with various applications involving mobile devices that suffer from intermittent, low bandwidth connections. The Agent TCL mobile agent system [79] (a forerunner to the D'Agents system) was designed to support mobile computers through appropriate network sensing tools and a docking system that allows agents to move transparently between mobile computers independently of the timing of availability of network connectivity. Other mobile agent systems which have included support for mobile devices include Tacoma [69] (application for cellular phones based on the SMS protocol) and MAGENTA [127] (naming scheme for mobile agents that supports dynamic IP addresses). More recent work such as [130] focuses on developing a framework for building context aware applications in ubiquitous and mobile computing settings.These use location tracking systems to navigate mobile agents to locations of the entities and places to which these agents are associated with. This allows support for applications such as follow-me desktop applications [61] and user navigation systems. On the same theme, mobile agents can also be used as intermediaries to support collaborative information sharing environments [165]. Here, a mobile agent tracks the movement of a mobile user and a protocol is provided for coordinating and implementing these types of agents.

2.2.4 Real time control and event monitoring

Devices that operate under real time constraints will benefit from the advantage of reduction of network latency that mobile agent usage confers. The Concordia mobile agent system [159] has been utilized as part of a core technology in applications for the electrical power systems industry, where agents are used in remote monitoring and control of power units. A framework has been proposed for developing flexible data feeds in sensors grids where mobile agents are utilized to dynamically establish data flows, data transformations as well as to act as policy enforcers in the framework [26]. An application involving mobile agents that are used for application upload, remote configuration and maintenance in an object kernel running on an embedded Java processor has also been developed in [16]. This application has been deployed in an autonomous station to implement an on-demand bus stop.

2.2.5 Distributed collaboration

A mobile multi agent system can exploit the cooperative capability of agents to solve a particular task. In distributed information retrieval scenarios, some mobile agent systems such as Concordia [159] support cooperative information gathering where the information gathering task can be divided among multiple agents that can search the complex information space concurrently and semi-autonomously, coordinating with other agents to solve interdependencies as needed. Mobile agents can be used to implement workflows with the users interaction environment in a collaboration being modelled as a collection of mobile agents. In [147], a collaboration plan created in XML is used to plan a mobile agent-based middleware system that allows each user an interface to perform the tasks pertaining to the collaboration.

Another area that can benefit from the advantages of cooperation, network latency and dynamic adaptation of mobile agents is network intrusion detection systems [71]. Mobile agents provide better resilience than conventional intrusion detection systems in that they can relocate when sensing danger or suspicious activity at a remote sites, clone for redundancy or replacement and be self organizing by dynamically reconfiguring themselves to compensate for failure of key components in the network. Their autonomous and asynchronous capability allows them to respond to an intrusion in a far more timely manner, enhancing an agent's ability to trace the attacker and collect evidence about the attack. Examples of applications include an attack resistant architecture for defending intrusion detection systems from denial of service attacks [99] and a social, insect-based, mobile agent framework that relies on the raising and lowering of pheromone fields to guide simple agents towards collectively exhibiting complex problem solving behaviour in handling intrusion detection [40]. There have also been efforts that attempt to incorporate an auditing facility into an existing mobile agent system, Aglets [139], in order to support an intrusion detection system [149].

2.2.6 Electronic commerce

E-commerce has been identified as a major area of application for software agents as they can be designed and automated for optimised decision making which allows improved market efficiency when they intermediate on behalf of human users in economic transactions. This generally involves finding, comparing, negotiating and executing economic contracts for items that the user is interested in buying or selling ([110], [58]). Mobile agents enhance this application area by leveraging the advantages of reduction of network latency and traffic load [24]. We discussed a typical application scenario for mobile agents in e-commerce in Chapter 1 and noted that the biggest challenge here was protecting the migrating mobile agent from compromise by a malicious host. There have been numerous approaches proposed for securing the various protocols that mobile agents use in the e-commerce scenarios that they participate in ([73], [81], [82],[122]). The consideration of security is intrinsic to mobile agents regardless of the application area, however, and we shall consider this aspect in greater detail in the next chapter of this thesis.

2.2.7 Web technologies

Web Services [52] have been proposed as an important paradigm for supporting heterogeneous distributed computing by focusing on a core group of Internet standards such as XML, SOAP and WSDL in the development of web-centric applications, particularly for business-to-business (B-to-B) transactions. Web Services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. Recent developments in mobile agent systems are taking into account this trend; for example, [32] explores the porting of a toolkit for code mobility written in Java (most mobile agent systems are implemented in Java) to the .NET platform (Microsoft's platform for Web Service development and deployment). There have been suggestions for structuring mobile agents as Web Services as this would ease their wide scale deployment in the Internet, given the increasing support for Web Services standards [46]. In [29], a model for implementing Web Services is proposed with mobile agents moving between cooperating Web servers to implement applications such as a distributed searching service and a distributed parallel information filtering service that takes advantage of agent mobility. The Grid [45] is another paradigm for structuring worldwide distributed computations involving multi-site collaboration in order to benefit from combined computing and storage power. In [18], a Java based mobile agent model for a Grid infrastructure is proposed, which addresses issues such as customizable distribution of computation, security, billing and accounting. A mobile agent based middleware that allows remote computer users to mutually offer their desktop computing resources to other Internet group members in a Grid-like infrastructure is described in [48].

2.3 Issues in mobile agent systems

There are several issues that will influence the future development and deployment of mobile agent systems in different application scenarios.

2.3.1 Performance and scalability

Mobile agents are often implemented using a byte-code interpreted language such as Java to enhance their portability in heterogeneous platforms. Executing interpreted code however incurs a higher load on the execution environment in comparison to compiled code (for example, C++). Thus the time savings from avoiding intermediate network traffic might be less than the time penalties that result from slower execution and migration overhead. Efforts to overcome this include techniques to allow mobile code to execute nearly as fast as natively compiled code such as just-in-time (JIT) compilation for Java as well as developing a more efficient and optimized JVM to execute interpreted code. In [116], it is argued that Java performance can exceed that of C++ because dynamic compilation gives the Java compiler access to runtime information not available to a C++ compiler. A benchmark of Java against C and Fortran has also shown that the performance gap between these languages is small on the Intel Pentium platform [22] while [155] demonstrates that some applications perform more efficiently when implemented in Java as compared to C.

2.3.2 Portability and standardization

The lack of standardization among the early mobile agent systems meant that interoperability was a problem as agents constructed on one specific platform could not run on or migrate to a different platform. The OMG MASIF standard [101] was an initial effort to address this issue by standardizing agent management, transfer, naming conventions and location syntax as well as defining a common API. This was, however, only adopted by a few mobile agent systems such as Grasshopper [12] and Aglets [139]. Later standards include the OMG Mobile Agent Facility which builds on the CORBA naming, life cycle, externalization and security services [57] and the FIPA standards [43]. Neither of these two have been universally accepted. A more recent effort in this direction is the Grid Mobile Agent System [56] that supports agent migration between different mobile agent systems such as NOMADS [137], D'Agents [54] and EMAA [97] running on the CoABS Grid [72].

2.3.3 Security

As mentioned earlier in discussing the electronic commerce application for mobile agents, security becomes an important issue when a mobile agent acts as an intermediary on behalf of a user's economic interest. The next chapter of this thesis will cover the basic security issues involved in the use of mobile agents in general, how they are currently addressed in ongoing research and the motivation for the work in this thesis.

Chapter 3

Mobile Agent Security

In this chapter, an overview is provided of the security threats that arise in a mobile agent system based on a survey of the existing literature. We discuss the two most important aspects: host security (section 3.2) and code security (section 3.3), and the different techniques available to address these aspects. In section 3.4, the mechanisms that are used to address code security in e-commerce are described. This is followed by a brief survey of the security mechanisms that are implemented in current mobile agent systems in section 3.5. We then provide a series of arguments for using the execution tracing approach as the basis for future work in code security in section 3.6.

3.1 Overview

In general, security threats presented by a mobile agent system can be classified into four categories [70], the distinguishing criteria being the proponent and the source of the attack. In the discussion below, the term host platform is used to refer to the computational environment that executes mobile agent code. A host platform may support multiple meeting places where agents can interact with each other. We now discuss the threats associated with these four categories.

3.1.1 Agent against host platform

The host platform needs to execute the agent code as well as grant system resources (such as CPU time, files, communication ports, etc) to it in order to allow the agent to complete its tasks satisfactorily. The most obvious danger is the agent code initiating some form of malicious action explicitly on a system resource that it can gain access to (for example, erasing files or reformatting the hard disk). Sensitive information on the agent platform also becomes a potential target; this may be altered, erased or simply duplicated by the agent, who may later disclose it to a third party to obtain some benefit or advantage. This threat necessitates some form of access control to system resources. Such access control can be achieved by authenticating the agent in some manner (determining correctly the identity of the owner or the composer of the mobile agent) and providing the agent code authorization to access specific resources on the basis of this identity.

An agent that has been granted access to a specific system resource could still use it in an unexpected and disruptive fashion, for example hogging it to the extent that other potential users are denied equal or fair access (denial-of-service attack). In an extreme case, this may cause the termination of other services provided by the agent platform or the eventual shut down of the platform. Some constraints on the use of resources, particularly computational and network resources, are therefore also required. The requirement for such constraints is similar to the motivation of the work in [107], which describes methods to monitor and control computations, whether parallel or distributed, by mastering their resource consumption.

3.1.2 Host platform against agent

This type of threat is specific to mobile agents and has no existing equivalent in traditional computer security considerations. It is usually the case that the owner of the application and the operator of the system on which the application will be executed are the same. It is of course possible that occasionally the operator of the system hosting the application could be a system administrator/programmer rather than the actual application owner. In such instances, however, the identity of the system administrator/programmer is clear and the level of trust and corresponding responsibility invested in that role is well established and defined. In a typical mobile agent system, it is usually the case that the system hosting the mobile agent is unknown to the deployer of the agent in terms of identity or trustworthiness. A host platform will require access to the code and state of the agent (or least to a significant portion of it) in order to execute it properly. Such access allows a malicious host platform to conduct the following types of attacks:

1. The host platform could extract information such as electronic cash, private encryption keys, certificates or secrets that the agent uses for its own security requirements.
2. The host platform could interfere in the communications of the agent with its external environment, for example intercepting outbound messages of the agent or routing them onwards to incorrect destinations.
3. The host platform could corrupt the code, state or data of the agent. This would alter the behaviour of the agent so that it could potentially behave in manner that would benefit the host platform, or perhaps cause damage to other platforms that it would subsequently migrate to. The state of the agent could also be reinitialised so that previous actions performed and results retained by the agent are effectively erased. Alternatively, data embedded within the agent could be deleted or altered in some manner that will subsequently affect its ability to be utilized for its original purpose.
4. The host platform could deny requests for resources from the agent. Such denial might extend from system resources that the agent should be entitled to (resulting in substantial delay in the agent completing its required tasks) to the complete termination of the agent itself.

The above represents only a partial list of the types of attacks that might be possible, the general idea being that complete access to agent code, state and data allows the host to manipulate either the agent or its execution environment in a manner that will eventually benefit the host (this benefit will differ depending on the agenda of the host and the application of the mobile agent system).

3.1.3 Agent against other agents

An agent can target another agent using several general approaches which include actions to falsify transactions, eavesdrop upon conversations, or interfere with an agent's activity. For example, an attacking agent can respond incorrectly to direct requests it receives from a target or deny that a legitimate transaction occurred. An agent can gain information by serving as an intermediary to the target agent (e.g., through masquerade) or by using platform services to eavesdrop on intra-platform messages. If the agent platform has weak or no access control mechanisms in place, it might be possible for an agent to access and modify another agent's data or code. Even with reasonable control mechanisms in place, an agent can attempt to send messages repeatedly to other agents in an attempt to deny them the ability to communicate (denial of service attack).

3.1.4 Other external entities against agent system

Agent system in this context refers to all the mobile agents currently in operation as well as the collection of host platforms that provides the specific execution environment for them; together, they constitute the framework of a particular mobile agent system. The external entities are anything that are not included in this framework and may include normal workstations/servers, network devices or even hosts from another unrelated mobile agent system. A potential avenue of attack is the communication links between agents and between platforms; a potential interloper could attempt to masquerade as an agent or platform, or eavesdrop on an ongoing interaction or replay passively captured messages.

The vast majority of current research efforts are centred on the first two concerns (agents against host platform and host platform against agents); the next two sections of this chapter provide a survey of the work done in these areas. The third concern (agent against other agents) can be resolved satisfactorily by ensuring that the first threat (agent against platform) is resolved successfully and by ensuring that the platform has sufficient control in intermediating interactions between agents. The fourth type of threat (external entities against agent system) can be negated by providing some form of secure communication protocol that authenticates the identities of two communicating parties to each other correctly and ensures that information exchanged between them remains private. This eliminates the possibility of interaction, modification, and replay attacks that are common in all distributed systems [136]. There are many existing security mechanisms such as Secure Sockets Layer (SSL) [47] or IPSec [77] that can be used towards this end.

3.2 Protecting the agent platform from the agent (Host security)

There has been a significant amount of work done in studying the manner in which an agent platform can be protected from a malicious agent (since this issue is similar to other traditional security concerns such as resource control); this section provides a short overview of some of the better known approaches.

3.2.1 Software-based fault isolation

Software-based fault isolation [154] is a method of isolating application modules into distinct fault domains enforced by software. This is implemented in Java through the sandboxing technique [51]. In the JVM, a distinct name space is maintained by the code loader for untrusted downloaded code and linking of references between modules in different name spaces is restricted to public methods. A security manager can also be invoked to mediate all accesses to system resources.

3.2.2 Safe code interpretation

Safe code interpretation is based on the idea that commands in agent systems which are considered harmful can be made safe for, or denied to, an agent. Most mobile agent systems are developed using interpreted scripts or programming languages, since these provide better portability for mobile agent code that are to be deployed in heterogeneous systems. The run-time characteristics of the code can therefore be checked during the process of interpretation; for example, Java (a popular choice for developing mobile agent systems) enforces security through static type checking in the form of byte code verification. Verification of this nature is particularly applicable to downloaded code from untrusted sources to ensure that the code does not violate the integrity of the JVM when it is executed. In addition, some limited form of dynamic checking is also performed during runtime.

3.2.3 Signed Code

Signed code involves the attachment of a digital signature to a piece of code in order to provide authentication with regards to its origin or integrity. Because an agent operates on behalf of an end-user or organization, many mobile agent systems ([75], [53]) commonly use the signature of the user as an indication of the authority under which the agent operates. The use of signed code relies on the existence of a Public Key Infrastructure (PKI) [4] to permit wide scale distribution of public key certificates. Microsoft's Authenticode is a good example of code signing, it enables Java applets or ActiveX controls to be signed, assuring users that the software has not been tampered with or modified and that the identity of the author is verified.

3.2.4 Proof Carrying Code

The approach taken by Proof Carrying Code (PCC) [108] obligates the code producer (e.g., the author of an agent) to formally prove that the program possesses safety properties previously stipulated by the code consumer (e.g., security policy of the agent platform). The aim of the technique is to prevent the execution of unsafe code. The code and proof are sent together to the code consumer where the safety properties can be verified. The proof is structured in a way that makes it straightforward to verify without using cryptographic techniques or external assistance. Once verified, the code can run without further checking.

3.2.5 Evaluation

Most of the efforts described have been successful in addressing the problem of protecting the agent platform from a potentially hostile agent, and have been implemented in one form or another (e.g., Java's sandbox), although the actual methods employed would be application dependent. It is likely that a good solution will involve a combination of several of these methods.

3.3 Protecting the agent from the agent platform (Code security)

For wide scale application, the approaches to protecting an agent can be broadly classified into two main mechanisms [76]:

- Detection mechanisms attempt to detect unauthorized modification of code, state, or execution of a mobile agent.
- Prevention mechanisms try to make it impossible (or at least very difficult) to access or modify code, state, or execution of a mobile agent in a manner that is meaningful to the perpetrator.

We note it is possible that access or modification to agent data may be permissible or even desirable in a limited form. For example, a host platform may extract non-sensitive data or concatenate results to a list carried by the mobile agent. Prevention is thus only targeted at unauthorized access or modification (as determined by the security policy of the agent system). On that note, it is equally important to realize that it may sometimes be impossible to prevent a hostile host from such an unauthorized extraction or modification. In these situations, the key point to observe is that such extractions or modifications are assumed to be meaningful to the perpetrator. This essentially means that:

- Unauthorized access or modification is undertaken with the primary intention of obtaining some benefit on the part of the perpetrator. This benefit may be direct and immediately realizable or may be more subtle and require a longer time to attain.
- In order to modify or access agent code/data to accrue this benefit, there must be understanding of the structure of the data being extracted or the original functionality of the code being modified. Without such understanding, extractions or modifications merely degrade to random manipulations of the mobile agent, the results of which are unpredictable to the perpetrator or anyone else.

Therefore in situations where unauthorized access or modification of agent code/data is impossible to prevent, a prevention mechanism will instead seek to alter or obscure agent code/data in some manner so as to make it indecipherable to potential attackers. Detection mechanisms, by comparison, are unable to obscure agent code/data and are therefore limited to applications where the data of an agent do not have to be kept confidential. We discuss below some of the more well known detection and prevention mechanisms. A more comprehensive review of all the proposed mechanisms can be found in [70] and [76].

3.3.1 Data encapsulation protocols (Detection)

The approach was conceived in the context of a mobile agent accumulating partial results in its state as it executes on the various platforms that it visits. A good example would be the best price shopping scenario, where a mobile agent roams an itinerary of vendor platforms in search of some item specified by the user it represents. The offers available at each platform would become partial results that are accumulated within the agent's state. A chain of encapsulated partial results [74] is constructed that binds each partial result to all previous result entries and to the identity of the subsequent platform to be visited. Each platform digitally signs its result entry using its private key, and uses a secure hash function to link results and identities within an entry. The result entry is also encrypted using the public key of the owner of the agent. When the mobile agent returns, the agent owner can verify the entire chain to detect if it has been tampered with during the agent's execution. This provides the guarantee of *forward integrity*, which states that a malicious platform will not be able to compromise the partial results generated from the platforms executing the mobile agent prior to its migration to that malicious platform. However, a collusion of hostile platforms could compromise the integrity of the results accumulated from that point onwards, even if the mobile agent eventually migrates to a trusted platform prior to returning to the agent owner.

In a later protocol [73], trusted tamper-proof hardware is used in conjunction with an extended version of the protocol for encapsulation of partial results to protect mobile agents operating in a distributed marketplace. Each vendor platform that is visited by a mobile agent in its itinerary is fitted with a trusted device (a smart card or a secure co-processor) that is provided by an independent market authority, which plays the role of a trusted third party for both the agent owner as well as the vendor platforms.

The encapsulation protocols described above were later shown to be vulnerable to specific types of security flaws such as interleaving and oracle attacks [121]. These weaknesses were overcome via appropriate modifications to the protocols in question [122]. A more recent protocol improves on the ability to detect possible data truncation in an agent through a specific way of binding accumulated data to static agent code [92].

3.3.2 Replicating and co-operating agents (Prevention)

Approaches that consider replicating agents as a variation of a form of fault-tolerance have been proposed in [120] and [133]. The common idea of these approaches is to share some information among the agent replicas. The mobile agents, which are now part of some larger application, have to co-operate with each other in order to achieve the desired result. In [120], the approach assumes the existence of two independent subgroups of agent executors that will not collaborate with each other in attacking the application. The application is then realized with two co-operating agents, each of which migrates exclusively in one of the two subgroups. These two agents can then use secret sharing, remote authorization, and remote storage of commitments to protect the application. Since no potential aggressor ever has access to both mobile agents, and co-operation between subgroups is assumed not to occur, both agents cannot be compromised simultaneously.

Another possible method is to use the idea of itinerary recording between two mutually co-operating agents [133]. When moving between two platforms, an agent conveys the last platform, current platform and next platform information to its co-operating peer through a secure channel. The peer maintains a record of the itinerary and takes appropriate action when inconsistencies are noted. Attention is paid so that an agent avoids platforms already visited by its peer.

There are some drawbacks with both approaches. For one thing, a secure authenticated communication channel is required, and this may not be possible to provide in all domains. This conceptual approach is also only suitable for specialized applications where the task can be formulated as a multi-component computation. Cloning mobile agents raises the issue of resolving identities properly when these replicas engage in communication with other agents or platforms (i.e. ensuring messages are forwarded and received by the correct replica).

3.3.3 Obfuscation (Prevention)

Obfuscation [64] is a special form of protection that can be extended to mobile agents for a limited period of time. It is based on a mechanism that transforms an application into one that is functionally identical, but much more difficult to understand and therefore to attack. Since obfuscation cannot provide complete protection, there is the assumption of the existence of a certain minimal time interval (the agent protection interval) during which it is impossible for an attacker to discover relevant data or to manipulate the execution of the agent in a meaningful way. After this interval lapses, the agent and all the sensitive information it contains becomes invalid, which makes any manipulation worthless. A good example for this invalidation of information is electronic cash that contains an expiration time.

The major problem of this approach is the difficulty in establishing a reasonable agent protection interval during which the mobile agent can accomplish a meaningful task. There is yet to be a working implementation of a system that is capable of executing obfuscated code, so the potential application of this approach is questionable. More recently, theoretical investigation of obfuscation has shown that obfuscation is intractable even when it is only necessary for the obfuscated code to approximately preserve the functionality of the original program [11]. There has been some related effort that combines the theme of code obfuscation and co-operating agents by distributing the intent of a mobile agent into a group of collaborating agents, of which some could be bogus agents as well [109].

3.3.4 Tamper-resistant Hardware (Prevention)

This approach, which relies on the existence of powerful tamper-resistant hardware that can withstand potent physical attacks, was proposed in [157] and [163]. The basic idea is to encapsulate the entire execution environment within the tamper-resistant hardware, which is a physically sealed environment. It protects a mobile agent that is executing on this execution environment from any illegitimate access or modification by the agent executor. The major disadvantage is that this protection is only available within a closed environment, in which all platforms that an agent can potentially migrate have to be equipped with potentially expensive tamper-resistant hardware.

A more limited approach that only assumes the existence of a very restricted tamper-resistant hardware (e.g., a smart card) has been proposed in [49]. The idea is to identify a small part of a mobile agent that will be executed in an execution environment that is protected by tamper-resistant hardware. The security implications of this approach are rather complex since the agent executor can control the entire communication between the protected and the unprotected parts of the mobile agent (e.g., manipulate the parameters or the results of function calls between the two parts).

3.3.5 Computing with Encrypted Functions (Prevention)

The use of encrypted functions was first proposed in [128] and promises to provide a protection similar to the one that can be achieved with tamper-resistant hardware relying on software only. The idea is to encrypt a function f to obtain some other function *E(f)* that hides the function *f*. This encrypted function can then be implemented as a program *P(E(f))*, which is interpreted as a mobile agent and sent to the agent executor. The latter can execute the mobile agent *P(E(f))* on an input value x, which leads to the encrypted result: *P(E(f))(x)*. Since the agent executor does not know what function it actually computed, it cannot meaningfully tamper with the code or its execution and is restricted to random modifications. The encrypted result is returned to the agent owner who can apply the decryption function, that only he knows, to obtain the desired result of the function f applied to the input value x: $E^{-1}(P(E(f))(x)) = f(x)$.

In its original form, this approach is limited since it does not allow the evaluation of arbitrary functions and is restricted solely to polynomials and rational functions. Another limitation is that no allowance is made for continuous interaction between the encrypted function and its host environment, i.e., the function cannot compute a clear-text result that is then used by the executor to provide new inputs (all the outputs of the encrypted function are themselves encrypted).

The protocol described in the encrypted functions approach only provided protection for single hop agents (i.e. agents which only visited a single untrusted host and then returned to their owner platforms). In [25], a one-round secure computation protocol is developed to enhance this protocol to protect mobile agents that leave the originator and migrate to a number of remote, untrusted hosts without a predefined itinerary. This was further improved upon in [5], where some limited form of interaction between the mobile agent implementing the encrypted function and the untrusted host executing it is made possible. This is achieved through the use of a trusted third party (which could take the form of tamper proof hardware) which intermediates in the exchange in order to protect the privacy of information exchanged between the mobile agent and its untrusted host. A recent development [166] further improves on the privacy of the information interchanged between the mobile agent and its untrusted host by replacing the single trusted party with a group of threshold trusted servers.

3.3.6 Execution tracing (Detection)

Execution tracing [148] is a technique for detecting unauthorized modifications of an agent through the recording of the agent's behaviour during its execution on each agent platform. The technique requires each platform involved to create and retain a non-repudiable log or trace of the operations performed by the agent while resident there, and to submit a hash of the trace upon conclusion as a trace summary. A trace is composed of a sequence of identifiers corresponding to executed statements in the agent code and affixed with the digital signature of the executing platform. A secure protocol is also developed to convey agents and associated security related information among the various parties involved, which may include a trusted third party to retain the sequence of trace summaries for the agent's entire itinerary. If any suspicious results occur, the appropriate traces and trace summaries can be obtained and verified, and a malicious host identified.

This approach can only detect a manipulation after the agent has terminated its execution and returned to its owner. Then, the agent owner must have a suspicion (based on the final result of the mobile agent or the charges for the resources used by the agent during execution) to launch the verification mechanism, which is too expensive to be used systematically. The assumption of connectivity between the platform at which the owner of the mobile agent resides and the other platforms in the path itinerary of the mobile agent may not hold under all circumstances. Creation of traces also requires that statements of the agent code be in a form that is amenable to easy recording; the proposed approach involves using a higher-level agent language, SALTA, which is refined from Safe-TCL. Other drawbacks include the fact that this approach is presented only in the context of single-threaded mobile agents, which does not take into account the multi-threaded operation of some mobile agent applications. The size and number of logs related to traces that need to be retained by the hosts can be problematic as well over a period of time.

3.4 Code security in electronic commerce

As was previously pointed out in Chapter 2, electronic commerce is a specific application area for mobile agents where security is of a paramount concern. In particular, the area of code security must be considered as the mobile agents will be executing electronic transactions on potentially malicious host platforms. Here we outline the way in which some of the code security techniques we discussed in the earlier section are extended and tailored to fit the requirements of electronic transactions [28]. One of the crucial issues in many electronic transactions is the need for the mobile agent to access some private information (for example, its owner's private key) in order to complete a transaction (for example by generating a digital signature). Embedding this private information in the agent however leaves it potentially vulnerable to theft or exploitation by a malicious host.

The concept of replicating and co-operating agents is utilized in the k-out-of-n threshold scheme [134] to address this problem. The private information is distributed over n agents that are sent to different hosts. Confidentiality of the private information is provided because none of the agents has the complete secret. It is assumed that there are less than k agents running on malicious and possibly colluding hosts, with at least k out of n co-operating agents needed to complete a transaction. Another example on a similar theme is used in [31] in which multiple mobile agents carry encrypted electronic cash and a share of the key to decrypt the electronic cash.

The computing with encrypted functions approach could be used to implement a digital signature function with a hard coded private key which would essentially protect the private key but permit the generation of the digital signature. This approach is used in the undetachable signature scheme developed in [81]. A similar theme is found in the idea of blinded signature keys [41]. A blinded signature is produced using a blinded signature key, with the cryptographic process of blinding performed in such a way that while the resulting signature can be verified the blinded key cannot be discovered. Mobile agents will carry the blinded signature key and a signed policy that defines the restrictions under which the signature key may be used.

A digital signature scheme in which the public key is fixed but the private key is updated at regular intervals is described in [13] and [83]. The key update provides forward security: compromise of the current private key does not enable an attacker to forge signatures from the past. The public key is fixed, which is an advantage for all verifying entities. When a malicious host compromises a private key, previous keys are not exposed. Revocation is, however, still necessary, as the following private keys will be compromised too.

3.5 Security in mobile agent systems

Most mobile agent systems in use currently only implement some form of host security. Some of the more recent ones include:

1. *Aglets*. The Aglets mobile agent system [75] uses a security model that clearly defines the principals within the aglet mobile agent system with respect to their responsibilities and interests. This controls how aglets migrate and access local resources on their host platforms. The security architecture also includes a policy database and accommodates owner-specified preferences to allow for a more flexible specification of security policies in the area of host security. There is no support for any code security technique.
2. *D'Agents*: The D'Agents mobile agent system [54] is a multiple language system (supporting Tcl, Java and Scheme) and provides three mechanisms for host security: cryptographic authentication of an agents owner; resource managers that make policy decisions based on an owner's identity; and a secure execution environment for each language that enforces the decisions of the resource managers. A planned market based approach is used for protecting groups of host machines. There is no implementation of any code security technique, although there is discussion of how such mechanisms might be integrated into the system.
3. *NOMADS*: NOMADS [137] is a Java based mobile agent system that supports strong mobility and fine grained resource control as a form of host security. It provides an agent execution environment built on a modified Java VM (Aroma) which provides the ability to dynamically control the resources consumed by agents thereby facilitating guarantees of quality of service and protecting against denial of service attacks from hostile agents. No code security technique is supported.
4. *J-SEAL2*: J-SEAL2 ([150], [15]) is not a mobile agent system, but a Java runtime system that allows the reification of resource control through byte code rewriting as well as providing an accounting scheme to monitor CPU and memory usage. It can be used in conjunction with standard JVMs to provide host security support for mobile agent systems to control and monitor the resource consumption of agents, thus protecting against denial of service attacks. No support is provided for any code security technique.

5. *SEMOA*: The SEMOA mobile agent system [124] has been used to implement a scalable global tracking service for mobile agents that incorporates protocols that addresses agent impersonation, malicious location updates and privacy of the agent owners [125]. A more recent application involves the transparent implementation of several cryptographic protocols that protect partially accumulated results in agents using the idea of forward integrity [122]. This is the only agent system at the moment that has an implemented version of a code security technique.

3.6 Execution tracing: Future direction for mobile agent security

The issue of host security can be considered to have been successfully resolved as all the mobile agent systems we have just described either implement a host security technique in their system or rely on the existing mechanisms available in Java, the language of choice for constructing mobile agent systems. In [17], a survey is performed on the various aspects of Java that relate to code mobility such as object management, thread management and byte code verification. Deficiencies in these areas are highlighted and the need for accounting and control of resource consumption in order to provide a more secure environment for executing mobile code in future versions of Java runtime systems.

The current focus of research in the area of code security appears to be in the area of developing protocols for protecting encapsulated results in agents ([122], [92]) and in the area of computing with obfuscated functions ([5], [166]). Although execution tracing in its current form suffers from some limitations, we believe that in essence it provides the most promising way forward for code security. We provide some arguments below to support our proposition.

3.6.1 Inter agent communication

In multi agent systems, agents interact with each other through a variety of communicative acts in order to implement specific applications such as negotiations in electronic commerce scenarios ([86], [62]) or bidding on behalf of users in online auctions ([6], [7]). As agents are static in multi agent systems, the security concerns of these systems generally revolve around issues such as corrupted naming and matchmaking services, insecure communication channels, insecure delegation and lack of accountability [160]. Interaction between agents would also involve higher level security considerations such as trust and reputation ([144], [67]).

As application areas such as e-commerce and online auctions become more widespread on the Internet, there is a likelihood that mobile agents will begin to be employed within a multi-agent framework because of the previously identified advantages (reduction of network latency, asynchronous execution on mobile devices, etc) that they confer when operating in a distributed system. An early example is found in [68], where mobile agents are used to migrate agent code to remote auction platforms where they can participate in the bidding process asynchronously and independently of their owners. A description of the different market structures in electronic markets that would benefit from the use of mobile agents is provided in [93], while [153] examines the advantages of deploying intelligent mobile agents to work with real life e-commerce sites such as eBay and addresses some of the security issues that might arise, particularly with regards to information sharing and ownership.

The possible convergence between mobile agent and multi agent systems may lead to a new security concern: the integrity of inter agent communication. The general assumption in the e-commerce code security techniques that were discussed earlier is that the mobile agent only interacted with the local services resident at a remote host platform. Therefore an attack on the agent would only compromise that agent and the interests of the user it represented. In multi-agent systems however, agents interact primarily through communication in order to perform activities such as negotiation or bidding. The inclusion of the issue of mobility would now mean that two agents communicating with each other could both be executing on two distinct, untrusted host platforms. Since the host platform execution environment will mediate all external communications of an agent, there is a distinct possibility that a malicious host could interfere by terminating the communication, routing the message to an incorrect recipient or altering the contents of the message. To compound matters further, invalid information sent by a corrupted agent on a malicious platform to a normal agent could indirectly result in the corruption of the state of that normal agent. In heavily interactive environments, invalid information resulting from such corruption would quickly propagate to many other agents as well resulting in serious consequences if economic concerns are involved.

The issue of inter agent communication is tackled directly in the replicating agents approach which relies on constant communication between the different agent replicas to ensure code security is achieved. These group of replicas, however, essentially represents a version of a single agent meant to accomplish a specific task. If there are two or more agent versions in the system required to accomplish different tasks, then a communication attempt between these different versions would necessitate complicated communication scenarios involving all of their constituent replicas. Obfuscation, tamper-resistant hardware and computing with encrypted functions are incapable of preventing a malicious platform from tampering with outgoing communications; even if a platform is unable to interpret the data in the outgoing communication (as is the case for encrypted functions), it is still able to intercept and misroute it. Execution tracing provides a record of the executed statements of the agent which would include any input / output statements as well. If this trace can be verified immediately at the end of agent execution on an untrusted host platform (rather than when the mobile agent completes its itinerary and returns to its owner), any misrouting or tampering of information can be detected by comparing traces of the execution of two agents that were communicating with each other.

3.6.2 Agent replay

One of the possible actions a malicious host platform could undertake on a mobile agent would be to execute it repeatedly with the same or different set of inputs each time. If each replay of the execution of this agent involved external communication to other agents, then these other agents would inadvertently be affected as they should have only received a single set of inputs from the tampered agent under normal execution conditions. Replay of an agent cannot be prevented in any of the code security techniques discussed earlier, but again execution tracing can help to detect it in a similar way. The trace of an agent effectively becomes the 'official' version of agent execution that a particular host platform cannot repudiate; i.e. regardless of how many times the host replays an agent, the trace represents a particular version of execution that the host is held accountable for. Comparing traces of other agents that were involved in communication with the affected agent in question would determine whether there was any malicious activity involved.

Another area of concern for agent replay is in the generation of digital signatures to conclude an electronic transaction. In the various code security techniques for e-commerce, different approaches have been presented to allow generation of a signature without exposure of the private key or some other equivalent secret. However, replaying an agent repeatedly under the right circumstances could allow a malicious host platform to trick an agent into generating many digital signatures to conclude bogus transactions. None of the techniques presented so far adequately addresses this problem. Again execution tracing presents a possible solution in that the trace would also include the statements that actually triggered the signature generation in an official version of agent execution. Verification of digital signatures produced on various contracts by an agent could be contrasted with the number of times a signature is shown to have been triggered in the relevant traces, and an inconsistency would suggest the possibility of a replay occurring.

3.6.3 Resource provision

The host platform could deny requests for resources from the mobile agent. Such denial might extend from system resources that the agent should be entitled to (resulting in substantial delay in the agent completing its required tasks) to the complete termination of the agent itself. Since most code security approaches assume a single agent operating in isolation without communication with other entities, there is nothing that can be done to recover properly from the termination of the agent other than to launch another agent into the network. In addition, it may not even be possible to detect the hostile platform responsible for this termination act. There has been some recent attention in this area, for example [156] looks at designing a parallel dispatch model for mobile agents that incorporates secure route structures and which also permits the detection of termination attack by hostile hosts, while [35] studies the size and cost of an optimal solution involving mobile agents in the task of locating a hostile host that terminates agents randomly. However, neither of these two approaches provides any code security technique to deal with less virulent attacks by a hostile host that do not necessarily result in the complete termination of the agent.

The co-operating agents approach does provide a way for detecting this type of attack, since the agent replicas are in constant communication with each other while migrating in parallel through the network. A co-operating agent is not a replica of the actual agent to be protected however, rather it is an agent that is specifically designed to support the actual agent in specific scenarios via constant communication so that it is immediately aware whenever the actual agent is compromised (e.g. through termination). In such an event, it is only capable of noting or reporting the compromise but is incapable of continuing the compromised agent's agenda on its own.

Execution tracing might be useful in this situation as well if the original protocol is changed so that traces are required to be submitted for verification at the end of every agent execution on a given host platform. Failure to submit this trace within a predefined time period could then be used as an indication of a possible compromise by the host concerned, where the party responsible for receiving and verifying the trace can identity the host it is due to obtain a trace from for which future punitive action can be taken.

3.6.4 Advantages over other approaches

In addition to its suitability for the concerns described above, execution tracing also presents some important advantages over the other code security techniques. Techniques such as obfuscation and computing with encrypted functions could ostensibly cause conflicts with host security solutions because the functionality of the agent code in these two approaches is opaque, increasing the risk that a potentially dangerous action in the code could be unintentionally allowed by the execution environment of the host platform. Execution tracing does not suffer from any such drawback as the code statements to be executed are in clear text and their associated functionality is unambiguous.

Cryptographic protocols that are based on the principle of forward integrity are only capable of safeguarding partially accumulated data in an agent; they are unable to ensure the execution integrity of agent code. This is not particularly important if an agent only functions primarily as a passive data accumulator in its migration path; however, if an agent needs to perform some computation on its accumulated data on a remote, untrusted host then some guarantee has to be provided that this computation is carried out in exact accordance with the logic and semantics of the executed agent code. Execution tracing provides this important guarantee through the production of a non-repudiable trace which associates the host with, and makes it liable, for a specific execution sequence of the agent concerned.

Approaches such as co-operating and replicated agents require the agent replicas to be structured in an application-specific manner; i.e. the design and behaviour of the co-operating agent that supports the actual agent would change in accordance with the scenario in which the agents were deployed. Since execution tracing operates on the internal code statements of an agent rather than on any specific external activity that an agent may engage in, it is application-independent and should be deployable in any scenario involving mobile agents.

3.7 Conclusion

An overview is provided of the security threats in a mobile agent system and the two most important aspects, host security and code security, is discussed. A short description of the code security techniques in e-commerce systems as well as the security mechanisms in current mobile agent systems is provided. We argue that the execution tracing approach represents the most promising way forward in code security research based on the following points:

1. It provides a way to detect attacks against communication between agents executing on potentially hostile hosts.
2. It can be used to detect replay of the agent by a hostile host.
3. It provides a possible way to safeguard against resource provision attacks.
4. It confers some distinct advantages over existing code security techniques.

However, as noted earlier in section 3.3.6, the execution tracing approach suffers from several drawbacks which need to be addressed to make it a more viable code security technique in line with the motivations for its usage that have just been discussed. The core work of this thesis, which is covered in the remaining chapters, is concerned with refining and improving the various ideas behind the execution tracing approach in a way that represents a contribution to the area of code security research. In the next two chapters (Chapter 4 and 5), we present and formalize a new approach that captures the essence of execution tracing, but minimizes the size of the trace and eliminates the need for a higher level language, thus simplifying a possible implementation.

Chapter 4

Interaction Tracing

In the previous chapter, we argued that the core idea underlying the execution tracing methodology presented a promising way forward in addressing code security in mobile agent systems, provided that some of its current limitations could be overcome. Here, we present the concept of *interaction tracing* which improves on execution tracing by overcoming two of its main limitations: the size of the traces produced and the complexity in producing them. The primary contributions of this chapter towards the thesis are:

1. Outlining the evolution from execution tracing to interaction tracing and the advantages provided by the interaction tracing approach.
2. Providing a shopping agent example that illustrates the difference between these two approaches.

In section 4.1, we detail the approach employed in the execution tracing approach in creating and verifying traces and suggest a possible way of improving this approach. We then present our model of an agent as a black box in section 4.2 and show how the concept of interaction tracing follows logically from this model. In section 4.3, we detail an example of mobile agent deployment in a shopping scenario to contrast the execution tracing and interaction tracing approach to creating and verifying traces. In section 4.4, we conclude and summarize the work developed.

4.1 Execution tracing

The primary goal of execution tracing developed by Vigna [148] is to detect any illicit manipulation of agent state by a potentially hostile host platform. Such illicit tampering can happen in two ways: the hostile host may directly manipulate state information in the agent, or it may alter the flow of execution in the agent so that program code is executed incorrectly. In the latter case, state information is affected indirectly since it will be changed erroneously as a result of incorrect program execution. As we have previously discussed in section 3.6.4, data encapsulation protocols can be used to protect information that is passively appended to the agent. However, if state information accumulated on a previous host is going to be actively used in computation by an agent on the current host, then this state information will be inevitably exposed to manipulation by a host with malicious intentions.

The execution tracing approach can be classified into two separate parts: the first is concerned with the process of creating traces during agent execution and verifying these traces, while the other details an interaction protocol that ensures the secure and non-repudiable delivery of traces and agent code between the various hosts in the system. In this section, we focus on examining how the idea behind the first part is evolved to produce the concept of interaction tracing.

4.1.1 Creating and verifying a trace

In the execution tracing approach, the code segment of an agent is composed of a sequence of statements that Vigna classifies as either *white* or *black*. A white statement is an operation that modifies the agent's execution state on the basis of the value of the agent's internal variables only, while a black statement modifies the state of the program using information received from the external execution environment. A trace of the execution of a program would be composed of a sequence of pairs $< n, s >$, where n represents a unique identifier of a statement, and s is a signature (not to be confused with a cryptographic digital signature). When associated with a black statement, the signature contains the new values assumed by internal variables as a consequence of the statement execution. For example, if *read(x)* gets the input 3 from its environment, the associated signature will be $x := 3$. The signature is empty for white statements. A trace is created for each agent executed on a remote host, who will retain the trace to be submitted to the agent owner for verification once an agent completes its migration and returns to the owners site.

The verification process is initiated by the agent owner on the suspicion that some tampering had been inflicted on it while the agent was traversing the various remote hosts in its itinerary. Under what conditions such suspicion might arise in the first instance is not clearly defined, but suggested possibilities could be an uncharacteristically high price for an item purchased or selected by an agent or high charges on the owner for resource consumption by the agent on a remote host. In cases such as these, the agent owner would request the trace of agent execution and its final state from the first remote host that the agent migrated to (the final state refers to the agent state prior to its migrating to the next remote host from the first remote host).

A copy of the agent code and its initial state prior to migrating to the first remote host is now executed (we shall call this a *replay* of the agent) using the submitted trace to drive the execution. The replay involves executing the code step by step, and at each step the identifier of the current statement is compared with the one contained in the trace of the corresponding step. Every time a statement that involves some input from the external computation environment is executed in the replayed agent, the input value is extracted from the corresponding signature. The complexity of this validation process is therefore linear with the size of the trace. If at some point, a discrepancy between the statements in the replay and the trace is found or if the final state of the replayed agent does not match the submitted final state, this would then indicate that the specified remote host had tampered in the execution of the agent. A similar process is then repeated for all the remaining remote hosts in the agent's itinerary in order to detect any possible malicious hosts.

The second part of the execution tracing approach, the interaction protocol for submission of traces, agent code and state between the various remote hosts and the agent owner site, is used to provide non-repudiable evidence that certain actions were undertaken by specific parties. This is used to provide further support to the core activity of trace verification, by providing proof of various other activities such as the migration of an agent from one host to another host or the acceptance of migrated agent code from another host. Such evidence can also be used to establish the actual perpetrator of a malicious tampering incident and is submitted to a recognized trusted third party in the system that intermediates in claims or disputes regarding agent tampering. It should be noted that since execution tracing is a detection mechanism, there is a need for such a trusted third party to have the capability of imposing some form of punitive sanctions on the remote hosts whose guilt has been established. We will study this protocol for submission of traces in more detail in Chapter 6 of this thesis, but an example of how it is used in supporting execution tracing is shown in section 4.3.

Several assumptions are necessary in order to implement the execution tracing mechanism, which we note are valid for a large majority of mobile agent systems.

- agent code remains immutable during an agent's migration lifetime;
- the execution environments for agent code (for example, the JVM for Java-based mobile agent systems) that are provided on host platforms exhibit identical behaviour (i.e. they execute agent code in an identical manner);
- execution of agent code results in deterministic change of agent state; code should not implement functions such as random number generation;
- execution of agent code should not be time dependent;

Comparing a submitted trace with a replayed agent's execution sequence in execution tracing has been generalized as a form of protection mechanism, termed the reference states approach in a later study [65]. Here, the nature of an attack on a mobile agent is defined specifically as a difference in behaviour between an attacking host and a non-attacking or reference host, i.e. one that acts as expected ('reference behaviour') given the same execution environment. Other code security mechanisms that have been classified as falling under the reference states approach include state appraisal [37] and agent replication [103], but the execution tracing approach has been argued to be the most advanced and reliable in relative comparison.

4.1.2 Improving on execution tracing

We outline two main ways in which the execution tracing approach can be improved upon. On closer examination, it can be seen that the final state of the agent prior to migration is the primary state of interest, since it will become the initial state of the agent when it commences execution on the next platform. The agent owner is always aware of the valid initial state of the agent: this would either be the original state of the agent prior to migrating to the first remote host or the final state of the agent at the end of a verified execution sequence on any given remote host. All that is required then to obtain a valid final agent state is the actual agent code whose execution is responsible for transforming agent state and the external inputs required by the execution process. Therefore, as long as it can be ascertained that the final state recorded in a submitted trace matches that of the replayed agent, it is no longer necessary to record or compare the particular sequence of instructions followed in order to reach this final state.

In addition, one of the characteristics of the execution tracing approach is that it implicitly assumes that problems only arise from tampering with state of the current agent. This can be seen from the fact that black statement identifiers are only concerned with input from the external environment. Outputs to the external environment are white statements and do not contain a signature indicating the value that is being output. However, as we have already noted in section 3.6.1, while outputs themselves do not change the internal state of an agent, such outputs (which are susceptible to malicious tampering by the current host) may be directed to other agents on remote hosts and hence capable of corrupting them indirectly. If we need to provide a way to detect tampering of such a nature, then the values associated with outputs to the external environment must be recorded in the trace as well (i.e. output statements should be made black statements as well).

We therefore arrive at the observation that the three items really required in a trace in an execution tracing approach are black input statements, black output statements and the final state of the agent. In the next section, we argue that even state information is no longer required in the new approach of interaction tracing that we develop.

4.2 Interaction tracing

We introduce the concept of *interaction tracing* here as an approach that renders the process of creating and verifying traces more tractable and efficient in comparison to the execution tracing approach. This is achieved by adopting a model of an agent that focuses on its behaviour, rather than on its internal state, as a measure of correctness. The creation and verification of traces then follows logically from this model.

4.2.1 Observing agent interactions

At a more conceptual level, an agent could be considered to be a system whose behaviour can be compared with other agents (or systems) using a black-box approach. This approach, which is widely used in many scientific disciplines, essentially involves the observation of the events through which a system interacts with its environment. Systems producing identical (respectively distinct) sequences of events are regarded as equal (respectively different). For example, the Turing test for defining intelligence is based on the inability to distinguish a program from a human being through specified interactions. In programming language semantics, Felleisen [39] studies the expressiveness of programming language constructs using a notion of observational equivalence, which consists of analysing the final result produced by expression evaluation, irrespectively of the transitions that led to this result — the only observation being the termination of programs and the result they produce. π-calculus [100] defines bisimulation as an equivalence class of processes in terms of their capability to follow the same sequences of observable transitions. Automata theory [66] relies on trace semantics, defined in terms of sequence of observable transitions, to prove safety and liveness properties of systems [8].

A common feature of all these approaches is that the inner workings of two systems being compared do not need to be modelled or analysed: whatever happens within these systems does not matter as long as their interactions with their environments remain the same. For our case, if we define a trace as a log of all inputs into and outputs from the agent, then we effectively have an *interaction trace* of an agent. Two agents exhibiting identical behaviour (i.e. having identical interaction traces) in a particular itinerary run may not necessarily have identical state; ultimately this is of no concern to us, as only their behaviour in that itinerary run is of concern. If at some stage in its migration path, an agent's state is corrupted by a host, then this tampering will eventually show up in a trace of its interactions (although not necessarily at the same host that tampered with it). If sanctions are to be enforced against the malicious host in situations such as this, there may then be a need to 'backtrack' through the itinerary of hosts previously traversed by an agent in order to determine the actual host responsible for introducing the initial state corruption. The cryptographic evidence provided and retained through the trace exchange interaction protocol (which we introduce in Chapter 6) permits the determination of the host identities necessary for this backtracking process.

If the tampering does not show up however, then the state corruption resulting from the tampering did not affect the agent's behaviour in a way that would be significant to its external interactions. The key point to note here is that we shift the focus away from protecting agent state against malicious tampering (in the execution tracing approach) to protecting the agents external interactions from malicious tampering. This is the essential novelty of the interaction tracing approach.

4.2.2 Trace verification in interaction tracing

Based on the black box approach that we have just detailed, an agent's behaviour can be determined solely from its interaction with its environment via its inputs and outputs. This provides a means of measuring the equivalence of behaviour of two different agents by supplying them with identical inputs and comparing the outputs produced. For two agents with identical functionality executing in a deterministic manner in identical environments, we can safely assume that their outputs would be identical to each other if they are supplied with identical inputs. We would therefore consider the behaviour of the agent on its owner's site as a reference behaviour [65] against which behaviour of the agent on remote hosts (which are assumed to be capable of malicious tampering) can be contrasted. If the comparison of the outputs in these two situations shows differences, it can then be concluded that the remote host in question is guilty of malicious tampering. Alternatively, this approach can be refined to establish the level to which tampering occurred, for example by tabulating the number of discrepancies between the outputs recorded in the two different environments. A higher number of discrepancies could then indicate a qualitatively higher level of malicious intent on part of an implicated host.

A remote host executing an agent would provide inputs to the agent in the form of an ordered input stream, from which the agent could extract values or messages as dictated by its execution flow. Any outputs produced by the agent would again be accumulated by the same environment and dispersed to their appropriate destinations. This input stream is recorded and becomes the *input trace* of the remote host, while the outputs are accumulated as the *output trace*. Upon completion of execution of a given program, the host environment would submit to the agent owner the input trace it used to drive the execution of the agent as well as the output trace produced from the execution. This input trace is subsequently used to drive the execution of an identical agent in the owner's environment, resulting in the production of an output trace (we shall call this the *replay trace* to distinguish it from the output trace of the host). Trace verification would then consist of comparing the replay and output trace, and detecting the differences that arise.

From a practical viewpoint, interaction tracing confers several important advantages over the execution tracing approach:

1. The size of the trace is reduced since there is no need to record every executed statement. This is particularly relevant in situations where the agent is largely computation bound.
2. For the large majority of mobile agent systems which are Java-based, it obviates the need for a modified JVM, which may be necessary to extract state information.
3. It eliminates the need for a special purpose mobile agent programming language capable of recording statement identifiers (such as SALTA, used in execution tracing).
4. It records the values associated with output values, which is necessary if safeguarding of inter agent communication is desired.

4.3 Shopping agent example

To illustrate the difference between execution tracing and interaction tracing, we employ the shopping agent example provided in Vigna's paper on execution tracing [148]. The program statements that constitute the body of this agent are given in Fig. 4.1. The **go** command is used to migrate the agent to a remote site specified as the command argument. Upon arrival, agent execution commences with the next statement following the **go** command. The **request** statement is used to access services provided by a site, takes as arguments the service name and service parameters and returns a result from the subsequent invocation of the service on the host site.

In the shopping agent scenario, the mobile agent specified in Fig. 4.1 leaves its user's site (*home*) and migrates to the *ShoppingMall* site, which maintains a

```
 1 : go(ShoppingMall)
 2 : shopList = request(list, homevideo)
 3 : for each shop in shopList {
 4 :     go(shop)
 5 :     price[shop] = request(directory, Matrix)
   : }
 6 : bestPrice = 20
 7 : bestShop = none
 8 : for each shop in shopList
 9 :     if (price[shop] < bestPrice ) {
10 :         bestPrice = price[shop]
11 :         bestShop = shop
   :     }
   : }
12 : if (bestPrice > 20) {
13 :     result = "no offers"
   : } else{
14 :     go(bestShop)
15 :     result = request(purchase, Matrix)
   : }
16 : go(home)
17 : request(check, result)
```

Figure 4.1: Shopping mobile agent

directory of electronic shops. Once there, the agent performs a directory query for sites offering home videos, after which it proceeds to visit these sites in turn. At each site, the agent invokes the local catalog service to determine the current price of the *Matrix* home video. When all prices have been collected, the agent identifies the best offer and if the best price is less than a specified amount, the agent migrates to the selected site and purchases the home video through an appropriate invocation of a service there. The transaction identifier is stored in the variable *result*. Upon completion of its task, the agent then migrates back to its user's site and returns the result via a *request* call to a service on the user's site which will then verify this result by initiating the replay of the agent.

We assume that two sites, *Blockbuster* and *EmiRecords*, are provided in the list by the directory service at *ShoppingMall*, with the prices for the *Matrix* video offered at 15 and 17 dollars respectively at these sites. The traces for correct execution of the agent at the various sites are shown in Fig. 4.2. The + sign is used before a statement identifier to indicate that it constitutes the second trace produced at that specific site. For example, the BlockBuster site has two traces; the first consists of statement identifiers 5, 3 and 4 corresponding to the

home	ShoppingMall	BlockBuster	EmiRecords
1	2	5	5
+17	shopList=	price(BlockBuster) = 15	price(EmiRecords) = 17
	BlockBuster,emiRecords	3	6
	3	4	7
	4	+15	8
		result = BlockBuster	9
		+16	10
			11
			8
			9
			12
			14

Figure 4.2: Traces for correct execution

agent's first visit to that site, and the second consists of statements 15 and 16 corresponding to the agent's return to the site in order to complete the purchase.

Consider a possible scenario where *EmiRecords* behaves in a malicious manner in order to sabotage the cheaper offer from *Blockbuster*. A simple attack approach would involve altering the price variable provided by *Blockbuster* so that the agent would select it (*EmiRecords*) instead as the cheapest site. The traces produced as a result of such an attack are shown in Fig. 4.3, with the invalid state alteration highlighted in italic.

For either scenario (honest or tampered execution), the owner may decide to initiate verification of execution once the agent returns back to its owner's site. It commences this activity by retrieving the trace produced by the directory site, (*ShoppingMall*), and executes a copy of the original agent. The statement identifiers executed during this replay process match those recorded in this trace, and identify *BlockBuster* as the next destination that the agent migrates to. The agent owner site then proceeds to obtain the traces (there are two) produced by *BlockBuster*, and commences to replay the agent copy and compares the sequence of identifiers in the replay process with the first trace in a similar manner. The process is repeated once *EmiRecords* is identified as the next destination. In the replay of the agent copy corresponding to the actual agent execution on *EmiRecords*, the sequence of statement identifiers produced that correspond to correct execution is shown in Fig. 4.2. For the situation where *EmiRecords* maliciously tampers with the value of *price(BlockBuster)*, the trace submitted by *EmiRecords* (5,6,7,8,9,10,11,8,9,10, ...) would be inconsistent with the sequence of correct execution (5,6,7,8,9,10,11,8,9, 12, ...).

home	ShoppingMall	BlockBuster	EmiRecords
1	2	5	5
+17	shopList=	price(BlockBuster) = 15	price(EmiRecords) = 17
	BlockBuster,EmiRecords	3	*price(BlockBuster) = 19*
	3	4	6
	4		7
			8
			9
			10
			11
			8
			9
			10
			11
			12
			14
			+15
			+result = EmiRecord
			+16

Figure 4.3: Traces with malicious tampering by EmiRecords

To simplify the description of this example, we have omitted description of the submission of the hashes of the traces and signed checksums of the agent's state, which form part of the protocol for securely migrating agent code between various hosts as well as obtaining traces from the various sites by the agent owner. As noted earlier in section 4.1.1, this protocol forms the second component of the execution tracing technique and is important because it provides a non-repudiable property (through hashes and digital signatures) of the traces produced by participating sites. Such a property can be used by the agent owner as proof of intentional malicious manipulation to a third party that will arbitrate in the event of disputes arising. In addition, it also provides non-repudiable evidence as to whether a particular site received a migrating agent from another site.

As an example, consider again the malicious manipulation just described. *EmiRecords* could attempt to circumvent the trace verification process by submitting a valid trace corresponding to a correct execution of the agent to the agent owner, while actually executing the tampered version of the agent. However, as can be seen in Fig. 4.2, a correct execution would result in the agent migrating to *BlockBuster* as its next destination. The agent owner, on reception of a valid trace corresponding to agent execution on *EmiRecords*, would now turn to validating the second trace from *BlockBuster*. Of course, if the agent had in truth being tampered with, then

<table>
<tr><th>home</th><th>ShoppingMall</th><th>BlockBuster</th><th>EmiRecords</th></tr>
<tr><td>Input Trace</td><td>Input Trace
1. ShopList=
BlockBuster,EmiRecords</td><td>Input Trace
1. price(BlockBuster)
=15
+2. result
=BlockBuster</td><td>Input Trace
1. price(EmiRecords)
=17</td></tr>
<tr><td>Output Trace
+1. result
=BlockBuster</td><td>Output Trace
1. (list,homevideo)</td><td>Output Trace
1. (directory,matrix)
+2. (purchase,matrix)</td><td>Output Trace
1. (directory,matrix)</td></tr>
</table>

Figure 4.4: Interaction traces for correct execution

it would have migrated back to *EmiRecords* and *BlockBuster* would only have one trace (Fig. 4.3). However, *EmiRecords* would still need to produce cryptographic evidence via the provided protocol to indicate that *BlockBuster* had accepted the migration of the agent from *EmiRecords* (which should occur if the agent execution on *EmiRecords* had been valid). Since *EmiRecords* is unable to produce such evidence (simply because it did not execute or migrate the agent correctly), it would therefore be considered liable for possible malicious action by the agent owner.

In interaction tracing, a trace of the input and output statements issued by the agent in the order that they are executed is created. There is no need to maintain identifiers on every statement in the program code in this situation. In this context, the **request** command in agent code can serve as only an output or both an output or input statement, depending on whether results are returned from its execution and assigned to an internal state variable (such as the case in statements 2, 5 and 15). An example of interaction traces (input trace and output trace) on all the various sites for the case of valid agent execution is show in Fig. 4.4. The + sign is used to indicate that the input or output action is part of the second trace produced at that specific site. The corresponding traces in the case of tampering by EmiRecords in shown in Fig. 4.5. The illegal alteration of *price(BlockBuster)* is shown in italic.

Verification of interaction traces is similar to verifying traces produced in execution tracing. The main difference here arises from the fact that malicious tampering of the agent cannot be detected solely from the first trace submitted by *EmiRecords* as would be the case for execution tracing. Since interaction traces do not capture the flow of execution logic in the program (which would be different as the result of malicious alteration of the *price(BlockBuster)* state variable), this form of tampering would not be detectable on the site it occurred at. The situation

home	ShoppingMall	BlockBuster	EmiRecords
Input Trace	**Input Trace** 1. ShopList= BlockBuster,EmiRecords	**Input Trace** 1. price(BlockBuster) =15	**Input Trace** 1. price(EmiRecords) =17 *price(BlockBuster)* *=19* + 2. result =EmiRecords
Output Trace +1. result =EmiRecords	**Output Trace** 1. (list,homevideo)	**Output Trace** 1. (directory,matrix)	**Output Trace** 1. (directory,matrix) +2. (purchase,matrix)

Figure 4.5: Interaction traces with malicious tampering by EmiRecords

in this instance is analogous to the case in the execution tracing approach where the malicious site submits a correct trace but executes a tampered agent instead. As we have seen, the replay of the agent copy on the owner site would indicate migration to *BlockBuster* had the execution being valid. Once again, *EmiRecords* would still need to produce cryptographic evidence via the provided protocol to indicate that *BlockBuster* had accepted the migration of the agent from *EmiRecords*. The failure of *EmiRecords* to do this would subsequently pinpoint it as the culprit responsible for a tampering attack on the agent state.

In Chapter 6, we will develop a new trace and agent exchange protocol for interaction tracing that provides the properties of the existing protocol for execution tracing, as well as including an additional ability to detect possible resource provision attacks (section 3.6.3) on mobile agents.

4.4 Conclusion

In this chapter we describe the execution tracing approach to creating and verifying a trace of an agent program's execution. We then demonstrate how this approach can be improved upon by leaving out all statement identifiers other than those that involve input statements, as well as recording the values associated with output statements. The black box model of an agent is proposed, and we show how the interaction tracing approach follows from this model. The use and advantages of interaction tracing is provided. An example to illustrate the difference between these two approaches to tracing an agent's execution is provided in the context of the shopping agent scenario.

There has been an attempt to re-examine the execution tracing approach prior to our work. In [65], the approach has been implemented using the idea of checking the fulfilment of certain post conditions in program code based on first order logic. This implementation however still suffers from the need to examine agent state as well as a new requirement for program code to be instrumented manually by the programmer via the insertion of appropriate code to support the replay process in the host. It also fails to address the issue of recording outputs in order to protect against tampering of inter agent communication.

It is important to note that interaction tracing (like execution tracing) does not offer complete protection against other types of attacks such as extraction of sensitive data from within the body agent. To avoid this, sensitive data should not be in a form that can be directly accessible by the agent in a clear text form. It can be protected using a form of the data encapsulation protocols (section 3.3.1) while sensitive items that need to be used for a predefined purpose (such as private keys for signing) can be protected using blinded signatures or undetachable signature schemes (section 3.4). In this sense, interaction tracing can be combined with other code security techniques to produce a more complete solution.

Chapter 5

Formalizing Interaction Tracing

There has been some work in formalizing the notions of mobility and security in the context of a mobile agent system. The Seal calculus [151] is a distributed process calculus that extends the π-calculus [100] to include abstractions of hierarchical locations, mobility and resource access control. The Dπ-calculus [119] is another calculus that extends the π-calculus and uses a typing system that provides resource access control on the basis of typing, proving that access violations cannot occur in well typed systems. The formalization of access control mechanisms and type safety (both host security considerations) in both these calculi provides a rigorous understanding of the nature of security achievable in mobile agent systems that were designed using them.

However, the nature of security achieved by the various code security techniques, particularly those using a detection approach, is still described in vague, qualitative terms. In this chapter, we address this shortcoming in existing work by providing a formalization for the interaction tracing methodology that we described intuitively in the previous chapter. In particular, we are interested in understanding formally the meaning of creating and verifying a trace in our approach. The primary contributions of this chapter towards the thesis are:

1. Providing operational semantics to illustrate the process of creating traces and replaying agents to verify traces based on two possible agent types: single threaded and multithreaded with shared state of execution.
2. Providing definitions for the evaluation of programs and verification of traces, and showing formally the security property that is achieved in the verification process.

1. Syntactic categories

Exp	in	$Expression$
$\mathcal{C}$	in	$Command$
Seq	in	$Sequence$
op	in	$Operator$
i	in	$Identifier$
v	in	$Values$
n	in	$Numeral$

2. Definitions

$$\begin{array}{rcl} Seq & ::= & Seq; \mathcal{C} \mid \mathcal{C} \\ \mathcal{C} & ::= & i := Exp \mid input\ i \mid output\ i \\ Exp & ::= & v \mid Exp\ \ op\ \ Exp \\ v & ::= & i \mid n \\ op & ::= & + \mid - \mid * \mid div \end{array}$$

Figure 5.1: Abstract syntax for agent language

In order to demonstrate the dynamics of interaction tracing, we consider an idealized agent programming language that is used to construct mobile agents since the activity of creating traces is intimately linked to the mechanics of program execution. Such a programming language would contain the three fundamental features necessary for studying the interactions of an agent with its environment (its inputs, outputs and a changeable internal state). In order to account for increasing complexity, we consider two possible cases: a single thread of deterministic execution (section 5.1) and multiple independent threads of deterministic execution that interact on the basis of shared state (section 5.2). For both cases, we develop and prove a lemma and theorem for the verification process and discuss the security associated implications.

5.1 Single thread of deterministic execution

In this section, we develop a formal model for the idealized agent programming language that can be used to construct a program that executes as a single deterministic thread. Fig. 5.1 provides an overview of the abstract syntax of this language. The syntactical definitions employed are abstract in that they are only designed to specify the structure of expressions allowable in the language. In particular, they are not concerned with acceptable strings of symbols nor with assigning to each such sequence of symbols a unique structure or parse tree. Instead, they describe the abstract syntax trees for the language.

5.1.1 Abstract syntax for agent language

The syntax we provide revolves around inputs into and outputs from the agent as this provides the basis for the interaction traces that are going to be created. In addition, it is also consistent with the message based paradigm underlying agent communication languages such as KQML [42] and FIPA [44]. A program will consist of a sequence of commands; each command is in one of three categories. The first assigns the value of an expression to an identifier. The second is used to obtain a value from the external environment and store it into an identifier, while the third outputs into the external environment the value of an identifier. The expression is either a value or two expressions separated by an operator; the operators being one of the 4 common arithmetic operators associated with standard programming langauges. A value can either be an identifier or a numeral n.

Before we move on to develop the operational semantics for this simple language, we note that we have not included other constructs generally associated with programming languages, such as conditional operations ($If..then...else$) or iteration ($While$). While these constructs are necessary to express the complete functionality of an agent programming language, they are (from an operational semantic viewpoint) only capable of altering the sequence in which the control unit executes the commands passed to it. Since we are primarily interested in observing interactions with the external environment, we only focus on interaction capable constructs and those that change internal state. However, we note that loops with a guard known at compile time can be unfolded in our idealized language.

5.1.2 State space of operational semantics

From this syntax, we can now derive a simple operational semantics that is essentially an abstract machine which functions as a formal interpreter for the language (Fig. 5.2). This machine represents the execution environment of the host executing agent code and has an arithmetic unit for evaluating arithmetic expressions as well as a control unit holding the program or expression that is to be executed or evaluated. It is capable of executing an agent program that can:

1. Receive inputs from its execution environment.
2. Transform its own internal state by operating on the inputs it receives.
3. Send parts of its internal state as outputs to the execution environment.

In Fig 5.2, st refers to tuples that describe the states of configuration of the machine. We assume that the machine is deterministic in that for every state st, there will be at most one state st' such that a state transition $\longrightarrow$ that occurs as

1. Syntactic categories			**2. Definitions**		
st	in	*States*	env	$::=$	$\epsilon \mid a.env$
env	in	*Env*	a	$::=$	(i, n)
a	in	*Assoc*	st	$::=$	$\langle env, Seq, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT} \rangle$
$\mathcal{O}$	in	*Output*	$\mathcal{I}$	$::=$	$\epsilon \mid n.\mathcal{I}$
$\mathcal{I}$	in	*Input*	$\mathcal{O}$	$::=$	$\epsilon \mid n.\mathcal{O}$
$\mathcal{OT}$	in	*OutputTrace*	$\mathcal{IT}$	$::=$	$\epsilon \mid n.\mathcal{IT}$
$\mathcal{IT}$	in	*InputTrace*	$\mathcal{OT}$	$::=$	$\epsilon \mid n.\mathcal{OT}$
n	in	*Numeral*			

Figure 5.2: State space for operational semantics

a result of a command being executed would result in a transition from *st* to *st*' (*st* $\longrightarrow$ *st*').

The environment of the model, *env*, is used as a store which associates a numeral with each identifier in order to evaluate any expression containing identifiers, i. In general, *env*, is an object that maps every identifier, from a possibly infinite set of identifiers, to a numeral. We start evaluation with an empty environment. At any moment during evaluation, a finite number of transitions will have been performed, which result in a necessarily finite number of changes to the environment, which therefore remains finite. Thus we can represent the environment as a finite list, given by the abstract syntax $env ::= \epsilon | (i, n).env$. A binding between i and n is denoted by a.

In Fig 5.2, $\mathcal{I}$ is an ordered input stream from which all programs will obtain their inputs through the *input i* command. Similarly, $\mathcal{O}$ represents the output stream to which all programs will write values via the *output i* command. The semantics of the input and output streams do not specify whether the inputs or outputs from a program have to be synchronous or otherwise. The input stream $\mathcal{I}$ into this operational semantics is derived from the inputs that may originate from other agents executing in the same software environment or from agents executing on remote hosts. The output stream $\mathcal{O}$ will eventually be directed to agents in the local environment or remote hosts. The input trace $\mathcal{IT}$ and output trace $\mathcal{OT}$ are utilized for recording the inputs and outputs respectively for a single agent program, and are the main components to be used in the trace verification process.

The execution environment provided by the current host intermediates in the relaying of inputs from the input stream into the agent program or assembling the program outputs into the output stream and transmitting them onwards to their final destinations. We note that the operational semantics do not indicate how the execution environment will perform such an intermediary operation. This is not relevant for consideration however, as neither of these streams is involved in the trace verification process and are merely included to demonstrate the creation of $\mathcal{IT}$ and $\mathcal{OT}$ respectively. We also note that $\mathcal{I}$ refers to a globally accessible structure which is available for reading by any program, whereas the input trace $\mathcal{IT}$ represents the actual inputs read by a specific agent program. Similar comments apply to $\mathcal{O}$ and $\mathcal{OT}$.

5.1.3 Operational semantics

Fig. 5.3 shows the rules and state transitions resulting from the execution of the different commands in the language. The constant rule states that a constant always evaluates to itself in a given environment. The environment lookup rule expresses the association between an identifier i and a numeral n in the environment, as discussed earlier. The second environment lookup rule states that if the environment is traversed from left to right in order to locate an association pertaining to a given identifier i, it is the case that the latest associated numeral n will be located at the front. The binary expression rule indicates that if two values of expressions Exp_1 and Exp_2 are used in a expression involving op, their associated values in that environment will be utilized in an operation involving op to obtain the final result n. The actual mechanics of evaluating $n_1 \ op \ n_2$ by the arithmetic unit is of no significance here, and so we abstract away from it by assuming that each expression involving a combination of two numerals and an operator reduces to a unique numeral n. The expression assignment rule expresses a state transition resulting from the evaluation of an expression, and the association of the result with an identifier in the state. This transition is indicated with the h subscript to indicate that it occurs in the execution environment hosting the original agent. The input rule simply reads in a value from the input stream $\mathcal{I}$ and associates it with an identifier in the environment. The input trace $\mathcal{IT}$ is also updated to record this value as well. In an execution environment with only one single-threaded agent program running that consumes all the values from the input stream, $\mathcal{IT}$ at the end of program execution would be identical to the initial value of $\mathcal{I}$ at the commencement of program execution. Finally, the output rule appends to the output stream $\mathcal{O}$ the numeral associated with i in the environment. The output trace $\mathcal{OT}$ is also updated to record this value as well. Again, we note that $\mathcal{OT}$ will be identical to $\mathcal{O}$ for the case of one single-threaded agent program executing in the environment.

1. Constant
$env, n \vdash n$

2. Environment lookup
$(i,n).env, i \vdash n$

3. Environment lookup2
$$\frac{env, n \vdash n \quad i \neq i'}{(i',n').env, i \vdash n}$$

4. Binary expression
$$\frac{env, Exp_1 \vdash n_1 \quad env, Exp_2 \vdash n_2 \quad n_1\ op\ n_2 = n}{env, Exp_1\ op\ Exp_2 \vdash n}$$

5. Expression assignment
$$\frac{env, Exp \vdash n}{\langle env, i := Exp; Seq, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle \longrightarrow_h \langle (i,n).env, Seq, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle}$$

6. Input
$$\langle env, input\ i; Seq, n.\mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle \longrightarrow_h \langle (i,n).env, Seq, \mathcal{I}, \mathcal{O}, \mathcal{IT}.n, \mathcal{OT}\rangle$$

7. Output
$$\frac{env, i \vdash n}{\langle env, output\ i; Seq, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle \longrightarrow_h \langle env, Seq, \mathcal{I}, \mathcal{O}.n, \mathcal{IT}, \mathcal{OT}.n\rangle}$$

Figure 5.3: Operational semantics - Rules and state transitions

5.1.4 Evaluation of a single threaded program

A program will initially start off as a tuple of $\langle \epsilon, Seq, \mathcal{I}, \epsilon, \epsilon, \epsilon\rangle$ and undergo a series of transitions in prescription with the rules just described. The termination of these state transitions occurs when the sequence of commands in Seq is exhausted, in a tuple of $\langle env, \epsilon, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle$. Note that $\mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}$ may all be empty, depending on the nature of the command sequences. The symbol $\longrightarrow^*$ is used to indicate a transitive closure of a single state transition $\longrightarrow$. Thus $st \longrightarrow^* st'$ if for some $k \geq 0$, there exists tuples $st_0, st_1, \ldots, st_k$ such that $st = st_0 \longrightarrow st_1, \ldots, st_{k-1} \longrightarrow st' = st_k$. We provide a more formal definition for the evaluation of a program in this abstract machine:

Definition 1 (Evaluation of a program) *The evaluation of a program is defined as* $Eval(Seq, \mathcal{I}) = \langle \mathcal{IT}, \mathcal{OT}\rangle$, *if the machine commences in the state* $\langle \epsilon, Seq, \mathcal{I}, \epsilon, \epsilon, \epsilon\rangle$ *and eventually terminates in a state* $\langle env, \epsilon, \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle$ *for some given* $env, \mathcal{IT}$ *and* $\mathcal{OT}$*, where* $\langle \epsilon, Seq, \mathcal{I}, \epsilon, \epsilon, \epsilon\rangle \longrightarrow^* \langle env, \epsilon, \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT}\rangle$.

The function $Eval$ assumes that the state tuples $st_0, st_1, \ldots, st_k$ within the transitive closure of $\langle \epsilon, Seq, \mathcal{I}, \epsilon, \epsilon, \epsilon\rangle$ and $\langle env, \epsilon, \epsilon, \mathcal{O}, \mathcal{IT}, OT\rangle$ are reached through state transitions that correspond to those shown in Fig. 5.3. This is important to note since the execution environment provided by the host is ultimately responsible for triggering the state transitions in accordance with the operational semantics of the command sequence Seq. The execution environment could potentially cause a state transition that does not correspond with the semantics of the particular command to be executed (for example, creating a new association pair in Env when an *output* i command occurs). This in effect constitutes tampering with the flow of execution of the agent. Thus $Eval$ is only defined with respect to valid state transitions that occur in accordance with the predefined operational semantics.

5.1.5 Determining trace equivalence

The input trace and output trace obtained at the end of execution will be submitted by the host to a party which will undertake the process of verifying the output trace. This party would typically be the owner's site, although more generally it can be any trusted third party. In this context, we define a trusted third party as a site that the owner trusts as being capable of undertaking the process of verification on its behalf. As discussed in section 4.2.2, an identical agent copy is executed on the trusted party's site using the submitted input trace; this guarantees that it consumes exactly the same sequence and values as the original agent on the submitting host. The replay trace produced by this agent copy can now be verified against the original submitted output trace. Since there is only one thread of execution, then there is only one possible sequence for inputs to be consumed and outputs to be produced during the flow of execution of either the original agent code or the agent copy. Since both the trusted party's site and the host should provide identical execution environments for the agent code, both agent threads can be considered to exhibit identical behaviour if their traces are equivalent. We consider either an input trace or a replay trace, $\mathcal{T}$, as an ordered sequence of numerals. The notion of equivalence can be defined as below.

Definition 2 (Equivalence of two traces) *Two traces, $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be equivalent ($\mathcal{T}_1 \;\backsimeq_S\; \mathcal{T}_2$) if they satisfy the following conditions:*

1. $\phi \;\backsimeq_S\; \phi$
2. $\dfrac{n_1 = n_2 \qquad \mathcal{T}_1 \;\backsimeq_S\; \mathcal{T}_2}{n_1.\mathcal{T}_1 \;\backsimeq_S\; n_2.\mathcal{T}_2}$

We will use this definition of equivalence in a later subsection when we consider the operational semantics for verification.

1. Syntactic categories			**2. Definitions**		
st	in	*States*	env	$::=$	$\epsilon \mid a.env$
env	in	*Env*	a	$::=$	(i, n)
a	in	*Assoc*	st	$::=$	$\langle env, Seq, \mathcal{IT}, \mathcal{RT}\rangle$
$\mathcal{RT}$	in	*ReplayTrace*	$\mathcal{RT}$	$::=$	$\epsilon \mid n.\mathcal{RT}$
$\mathcal{OT}$	in	*OutputTrace*	$\mathcal{OT}$	$::=$	$\epsilon \mid n.\mathcal{OT}$
$\mathcal{IT}$	in	*InputTrace*	$\mathcal{IT}$	$::=$	$\epsilon \mid n.\mathcal{IT}$
n	in	*Numeral*			

Figure 5.4: State space for operational semantics in verification

5.1.6 State space of operational semantics for verification

We define the operational semantics (Fig. 5.4) for the environment of the trusted third party which will replay an agent copy. This is virtually identical to that of the host (Fig. 5.2) with the exception of the replay trace $\mathcal{RT}$, which now performs the same function as $\mathcal{OT}$ of Fig. 5.2, and $\mathcal{IT}$ here will perform the same function as $\mathcal{I}$ in the original model. Note that there is no equivalent of an output stream $\mathcal{O}$ here; any outputs from a replayed agent are only accumulated in $\mathcal{RT}$ since there is no need to relay these outputs on to their intended destinations (this process having already been accomplished during the execution of the original agent on the untrusted host).

5.1.7 Operational semantics for verification

The equivalent transitions are shown in Fig. 5.5. Note that we use the subscript t to distinguish the transitions that occur here from the transitions that take place on the host platform executing the original version of the agent. This is similar to the operational semantics of section 5.1.3, with the main difference being that the input rule reads in a value from the input trace $\mathcal{IT}$, while the output rule appends to the replay trace $\mathcal{RT}$. A replayed agent will commence execution in a state $\langle \epsilon, Seq, \mathcal{IT}, \epsilon\rangle$ and terminate in a state $\langle env, \epsilon, \epsilon, \mathcal{RT}\rangle$. We can now provide a formal definition for the verification of an output trace from a host:

Definition 3 (Verification of an output trace) *A program is verified with respect to some input ($\mathcal{IT}$) and output ($\mathcal{OT}$) traces, $Verify(Seq, \mathcal{IT}, \mathcal{OT})$, if the machine commences in the state $\langle \epsilon, Seq, \mathcal{IT}, \epsilon\rangle$ and eventually terminates in a state $\langle env, \epsilon, \epsilon, \mathcal{RT}\rangle$ for some given store, env, and $\mathcal{RT} \backsimeq_S \mathcal{OT}$.*

The determination of trace equivalence $\backsimeq_S$ is computationally linear in relation to the size of the trace .

1. Constant

$env, n \vdash n$

2. Environment lookup

$(i, n).env, i \vdash n$

3. Environment lookup2

$$\frac{env, n \vdash n \quad i \neq i'}{(i', n').env, i \vdash n}$$

4. Binary expression

$$\frac{env, Exp_1 \vdash n_1 \quad env, Exp_2 \vdash n_2 \quad n_1 \ op \ n_2 = n}{env, Exp_1 \ op \ Exp_2 \vdash n}$$

5. Expression assignment

$$\frac{env, Exp \vdash n}{\langle env, i := Exp; Seq, \mathcal{IT}, \mathcal{RT}\rangle \longrightarrow_t \langle (i, n).env, Seq, \mathcal{IT}, \mathcal{RT}\rangle}$$

6. Input

$\langle env, input\ i; Seq, n.\mathcal{IT}, \mathcal{RT}\rangle \longrightarrow_t \langle (i, n).env, Seq, \mathcal{IT}, \mathcal{RT}\rangle$

7. Output

$$\frac{env, i \vdash n}{\langle env, output\ i; Seq, \mathcal{IT}, \mathcal{RT}\rangle \longrightarrow_t \langle env, Seq, \mathcal{IT}, \mathcal{RT}.n\rangle}$$

Figure 5.5: Operational semantics for verification - Rules and state transitions

5.1.8 Theorem for the verification process

We seek to understand formally the security properties achieved by the successful verification of an output trace (Definition 3) by formulating a theorem that encapsulates the nature of the verification process. In the previous section, we have seen that the operational semantics for the replayed agent is identical to that of the original agent on the host. Intuitively, it can be seen that if agent program execution follows the predefined semantics on the host (Definition 1), then its output trace will be verified correctly on the trusted third party. A first step towards the formulation of this theorem involves taking into account a possible unique mapping between every state configuration in the evaluation of an agent program on the host to a state configuration in the replay of the agent copy during the verification of an output trace. Proceeding from that, we wish to show (Fig. 5.6) that for any two uniquely mapped state configurations ($\mathcal{CF}_h$ and $\mathcal{CF}_t$), a transition from both these states will lead to another pair of mapped state configurations ($\mathcal{CF}_h$' and $\mathcal{CF}_t$'). This idea can be demonstrated by formulating it in terms of the diamond property [98] as applied in the context of our scenario. The h and t transitions correspond to the state transitions described in Fig. 5.3 and Fig. 5.5.

$$\begin{array}{lcr} \mathcal{CF}_h & \longrightarrow_h & \mathcal{CF'}_h \\ \downarrow_T & & \downarrow_T \\ \mathcal{CF}_t & \longrightarrow_t & \mathcal{CF'}_t \end{array}$$

Figure 5.6: Diamond property for state configurations

We therefore need to define a transformation relationship, T, that maps any state configuration from the program evaluation process on the host to a unique state configuration from the agent replay process on the trusted third party. In defining this relationship, we will need to consider the final terminating state $\mathcal{CF}'_{hf}$ on the host in a program evaluation. This is due to the fact that the input trace $\mathcal{IT}_t$ of any state configuration on the trusted third party can only be derived using the combination of both the input trace $\mathcal{IT}_h$ of a given state configuration on the host and the input trace $\mathcal{IT}_{hf}$ of the final state configuration of program evaluation. We first define an auxiliary function to aid in the definition of the transformation relationship and other proofs that we subsequently develop.

Given an ordered finite sequence of numerals, $\mathcal{T}$, the function $rem(x, \mathcal{T})$ removes the first numeral x from $\mathcal{T}$ going from left to right, and can be defined inductively as:

1. Base case : $rem(x, \phi) = \phi$;
2. Inductive case : $rem(x, x.L) = L$;
   $rem(x, a.L) = \mathrm{a}.rem(x, L)$ when $x \neq a$.

Given two ordered finite sequence of numerals, $\mathcal{T}_1$ and $\mathcal{T}_2$, the function $remseq(\mathcal{T}_1, \mathcal{T}_2)$ removes from $\mathcal{T}_2$ all the numerals in $\mathcal{T}_1$. We can define $remseq(\mathcal{T}_1, \mathcal{T}_2)$ inductively as:

1. Base case : $remseq(\phi, \mathcal{T}_2) = \mathcal{T}_2$;
2. Base case : $remseq(\mathcal{T}_1, \phi) = \phi$;
3. Inductive case : $remseq(x.L_1, a.L_2) = remseq(L_1, L_2)$ if $x = a$
   $remseq(x.L_1, a.L_2) = a.remseq(x.L_1, L_2)$ if $x \neq a$.

The transformation relationship is defined thus:

Definition 4 (Definition of a transformation relationship) *An initial state on the host, $\mathcal{CF}_{h0}$, will undergo a series of transitions eventually leading to a final terminating state, $\mathcal{CF}_{hf} = \langle env_{hf}, \epsilon, \epsilon, \mathcal{O}_{hf}, \mathcal{IT}_{hf}, \mathcal{OT}_{hf}\rangle$. A transformation relationship that maps a given state in this series of transitions on the host to an equivalent state on the trusted third party is given as follows:*

$$T[\![\langle env_h, Seq_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h\rangle, \mathcal{CF}_{hf}]\!] = \langle env_t, Seq_t, \mathcal{IT}_t, \mathcal{RT}_t\rangle$$

with $env_h = env_t, Seq_h = Seq_t, \mathcal{IT}_t \backsimeq_S remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}), \mathcal{OT}_h \backsimeq_S \mathcal{RT}_t$

We now state the diamond property as a lemma and proceed to prove it. This proof will aid towards our original goal of demonstrating that if agent program execution follows the predefined semantics on the host then its output trace will be verified correctly on the trusted third party. The statement of the lemma includes the final state of program evaluation on the host, $\mathcal{CF}_{hf}$, as this is required in the transformation relationship.

Lemma 1 (Diamond property) *Let $\mathcal{CF}_h$ refer to a particular state in program evaluation on the host, $\mathcal{CF}_{hf}$ to the final state and $\mathcal{CF}_t$ to a state in the trusted third party. Then, if $\mathcal{CF}_h \longrightarrow_h \mathcal{CF}'_h$, $\mathcal{CF}'_h \longrightarrow^*_h \mathcal{CF}_{hf}$ and $T[\![\mathcal{CF}_h, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_t$, there exists a $\mathcal{CF}'_t$ where $\mathcal{CF}_t \longrightarrow_t \mathcal{CF}'_t$, and $T[\![\mathcal{CF}'_h, \mathcal{CF}_{hf}]\!] = \mathcal{CF}'_t$.*

Proof of Lemma 1 In order to prove Lemma 1, we consider all configurations on both the host and the trusted server which have a non-empty command sequence since a transition can only occur from these places. We consider three main categories based on the three commands that can affect a transition:

1. Expression assignment, $\mathcal{C} = i := Exp$, and
   $\mathcal{CF}_h = \langle env_h, \mathcal{C}; Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle$
   First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:
   $\langle env_h, i := Exp; Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle \longrightarrow_h \langle (i, n).env_h, Seq', \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h \rangle$
   The values for $\mathcal{I}'_h$ and $\mathcal{O}'_h$ remain unchanged from the transition, with $\mathcal{I}'_h \simeq_S \mathcal{I}_h$ and $\mathcal{O}'_h \simeq_S \mathcal{O}_h$.

   Using the transformation relation to obtain the equivalent state configuration on the trusted server, we have:
   $T[\![\langle env_h, i := Exp; Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle, \mathcal{CF}_{hf}]\!] =$
   $\langle env_t, i := Exp; Seq', \mathcal{IT}_t, \mathcal{RT}_t \rangle$

   Applying the same state transition rule to the equivalent configuration of the trusted server, we find:
   $\langle env_t, i := Exp; Seq', \mathcal{IT}_t, \mathcal{RT}_t \rangle \longrightarrow_t \langle (i, n).env_t, Seq', \mathcal{IT}'_t, \mathcal{RT}'_t \rangle$

   Finally, applying the transformation relation on the new state configuration of the host, $\mathcal{CF}'_h$, to obtain the new state configuration of the trusted server, we derive:

$T[\![\langle(i,n).env_h, Seq', \mathcal{I}_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h\rangle, \mathcal{CF}_{hf}]\!] =$
$\langle(i,n).env_t, Seq', \mathcal{IT}'_{2t}, \mathcal{RT}'_{2t}\rangle$

We consider the various transitions involving the input trace $\mathcal{IT}$ in establishing the diamond property:

(a) by $\longrightarrow_h, \mathcal{IT}'_h \simeq_S \mathcal{IT}_h$

(b) by $\longrightarrow_t, \mathcal{IT}'_t \simeq_S \mathcal{IT}_t$

(c) by $T[\![]\!], \mathcal{IT}_t \simeq_S remseq(\mathcal{IT}_h, \mathcal{IT}_{hf})$

(d) by $T[\![]\!], \mathcal{IT}'_{2t} \simeq_S remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})$

In a similar manner, we look at the transitions for the output trace $\mathcal{OT}$ and replay trace $\mathcal{RT}$.

(a) by $\longrightarrow_h, \mathcal{OT}'_h \simeq_S \mathcal{OT}_h$

(b) by $\longrightarrow_t, \mathcal{RT}'_t \simeq_S \mathcal{RT}_t$

(c) by $T[\![]\!], \mathcal{RT}_t \simeq_S \mathcal{OT}_h$

(d) by $T[\![]\!], \mathcal{RT}'_{2t} \simeq_S \mathcal{OT}'_h$

We note that $\mathcal{IT}'_{2t} \simeq_S \mathcal{IT}'_t$ and $\mathcal{RT}'_{2t} \simeq_S \mathcal{RT}'_t$ by transitivity of $\simeq_S$, and thus the diamond property is satisfied here.

$$\begin{array}{ccccc}
 & \mathcal{IT}_h & \simeq_S & \mathcal{IT}'_h & \\
 & \mathcal{IT}_t & \simeq_S & \mathcal{IT}'_t & \mathcal{IT}'_{2t} \\
 & \simeq_S & & & \simeq_S \\
remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}) & & \simeq_S & & remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})
\end{array}$$

$$\begin{array}{ccccc}
\mathcal{OT}_h & \simeq_S & \mathcal{OT}'_h & \simeq_S & \mathcal{RT}'_{2t} \\
\simeq_S & & & & \\
\mathcal{RT}_t & \simeq_S & \mathcal{RT}'_t & &
\end{array}$$

2. Input command, $C = Input\ i$. Again, we wish to show that the diamond property holds for this command for the case of an empty and non-empty input stream.

   - For an empty input stream, $\mathcal{I} = \epsilon$, the state transition rule cannot fire and the premises of the lemma do not hold, which implies that the lemma is trivially true.

- Non-empty input stream $\mathcal{I} = n.\mathcal{I}'$, and
  $\mathcal{CF}_h = \langle env_h, \mathcal{C}; Seq', n.\mathcal{I}'_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle$
  First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:
  $\langle env_h, input\ i.Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle \longrightarrow_h$
  $\langle (i,n).env_h, Seq', \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h \rangle$
  The values for $\mathcal{I}'_h$ and $\mathcal{O}'_h$ after the transition are $\mathcal{I}'_h \;\simeq_S\; rem(n, \mathcal{I}_h)$ and $\mathcal{O}'_h \;\simeq_S\; \mathcal{O}_h$.

  The transformation relation to obtain the equivalent state configuration on the trusted server is similar to that in the case of the expression assignment command, which we have already outlined. This is equally applicable for the state transition rule to obtain the equivalent configuration on the trusted server, and the final transformation relation on the new state configuration on the host to obtain the new state configuration of the trusted server.

  We consider the various transitions involving the input trace $\mathcal{IT}$ in establishing the diamond property.

  (a) by $\longrightarrow_h, \mathcal{IT}'_h \simeq_S \mathcal{IT}_h.n$
  (b) by $\longrightarrow_t, n.\mathcal{IT}'_t \simeq_S \mathcal{IT}_t$
  (c) by $T[\![]\!], \mathcal{IT}_t \simeq_S remseq(\mathcal{IT}_h, \mathcal{IT}_{hf})$
  (d) by $T[\![]\!], \mathcal{IT}'_{2t} \simeq_S remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})$

  In a similar manner, we look at the transitions for the output trace $\mathcal{OT}$ and replay trace $\mathcal{RT}$.

  (a) by $\longrightarrow_h, \mathcal{OT}'_h \simeq_S \mathcal{OT}_h$
  (b) by $\longrightarrow_t, \mathcal{RT}'_t \simeq_S \mathcal{RT}_t$
  (c) by $T[\![]\!], \mathcal{RT}_t \simeq_S \mathcal{OT}_h$
  (d) by $T[\![]\!], \mathcal{RT}'_{2t} \simeq_S \mathcal{OT}'_h$

  We see that $\mathcal{IT}'_{2t} \simeq_S remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf}) \simeq_S remseq(\mathcal{IT}_h.n, \mathcal{IT}_{hf})$. From the transitivity of $\simeq_S$, we note that $n.\mathcal{IT}'_t \simeq_S remseq(\mathcal{IT}_h, \mathcal{IT}_{hf})$, or $\mathcal{IT}'_t \simeq_S rem(n, remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}))$. This can also be expressed as $\mathcal{IT}'_t \simeq_S remseq(\mathcal{IT}_h.n, \mathcal{IT}_{hf}))$, from which we can conclude that $\mathcal{IT}'_t \simeq_S \mathcal{IT}'_{2t}$. We also note that $\mathcal{RT}'_{2t} \simeq_S \mathcal{RT}'_t$ by transitivity of $\simeq_S$, and thus the diamond property is satisfied here

$$\begin{array}{rcccl}
\mathcal{IT}_h.n & \simeq_S & \mathcal{IT}'_h & & \\
\mathcal{IT}_t & \simeq_S & n.\mathcal{IT}'_t & & \mathcal{IT}'_{2t} \\
\simeq_S & & & & \simeq_S \\
remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}) & & & & remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})
\end{array}$$

$$\begin{array}{rcccc}
\mathcal{OT}_h & \simeq_S & \mathcal{OT}'_h & \simeq_S & \mathcal{RT}'_{2t} \\
\simeq_S & & & & \\
\mathcal{RT}_t & \simeq_S & \mathcal{RT}'_t & &
\end{array}$$

3. Output command, $C = output\ i$, and
$\mathcal{CF}_h = \langle env_h, \mathcal{C}; Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle$
First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}_h$', on the host:
$\langle env_h, output\ i.Seq', \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h \rangle \longrightarrow_h$
$\langle env_h, Seq', \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h \rangle$
The values for $\mathcal{I}'_h$ and $\mathcal{O}'_h$ after the transition are $\mathcal{I}'_h \simeq_S \mathcal{I}_h$ and $\mathcal{O}'_h \simeq_S n.\mathcal{O}_h$.

The remaining transformations and transitions are similar to the case of the expression assignment and input command, and we will not repeat them here. As usual, we consider the various transitions involving the input trace $\mathcal{IT}$ in establishing the diamond property.

(a) by $\longrightarrow_h, \mathcal{IT}'_h \simeq_S \mathcal{IT}_h$

(b) by $\longrightarrow_t, \mathcal{IT}'_t \simeq_S \mathcal{IT}_t$

(c) by $T[\![]\!], \mathcal{IT}_t \simeq_S remseq(\mathcal{IT}_h, \mathcal{IT}_{hf})$

(d) by $T[\![]\!], \mathcal{IT}'_{2t} \simeq_S remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})$

In a similar manner, we look at the transitions for the output trace $\mathcal{OT}$ and replay trace $\mathcal{RT}$.

(a) by $\longrightarrow_h, \mathcal{OT}'_h \simeq_S \mathcal{OT}_h.n$

(b) by $\longrightarrow_t, \mathcal{RT}'_t \simeq_S \mathcal{RT}_t.n$

(c) by $T[\![]\!], \mathcal{RT}_t \simeq_S \mathcal{OT}_h$

(d) by $T[\![]\!], \mathcal{RT}'_{2t} \simeq_S \mathcal{OT}'_h$

We note that $\mathcal{IT}'_{2t} \simeq_S \mathcal{IT}'_t$ and $\mathcal{RT}'_{2t} \simeq_S \mathcal{RT}'_t$ by transitivity of $\simeq_S$, and thus the diamond property is satisfied here. With this, we conclude the proof of Lemma 2.

$$\begin{array}{ccccc}
 & \mathcal{IT}_h & \simeq_S & \mathcal{IT}'_h & \\
 & \mathcal{IT}_t & \simeq_S & \mathcal{IT}'_t & \mathcal{IT}'_{2t} \\
 & \simeq_S & & & \simeq_S \\
remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}) & & \simeq_S & & remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf})
\end{array}$$

$$\begin{array}{ccccccc}
\mathcal{OT}_h & \mathcal{OT}_h.n & \simeq_S & \mathcal{OT}'_h & \simeq_S & \mathcal{RT}'_{2t} \\
\simeq_S & \simeq_S & & & & \\
\mathcal{RT}_t & \mathcal{RT}_t.n & \simeq_S & \mathcal{RT}'_t & &
\end{array}$$

We now present a theorem regarding the verification process which we had intuitively described at the start of this section. Proving this theorem allows us to deduce some specific security properties that the verification process confers.

Theorem 1 *For any program Seq, and for any input stream I,* $Eval(Seq, \mathcal{I}) = \langle \mathcal{IT}, \mathcal{OT} \rangle$ *if and only if* $Verify(Seq, \mathcal{IT}, \mathcal{OT})$*, for some traces* $\mathcal{IT}$ *and* $\mathcal{OT}$*.*

Proof of Theorem 1

From the initial configuration on the host $\mathcal{CF}_{h0} = \langle \epsilon, Seq, I, \epsilon, \epsilon, \epsilon \rangle$, there is a series of transitions that eventually leads to a final terminating configuration $\mathcal{CF}_{hf} = \langle env, \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT} \rangle$, by Definition 1. There is also the initial configuration on the trusted server $\mathcal{CF}_{t0} = \langle \epsilon, Seq, \mathcal{IT}, \epsilon \rangle$ such that $T[\![\mathcal{CF}_{h0}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{t0}$.

For each transitition $\mathcal{CF}_{hn} \longrightarrow_h \mathcal{CF}_{hn+1}$, Lemma 1 can be applied. We obtain the following diagram:

$$\begin{array}{ccccccc}
\mathcal{CF}_{h0} & \longrightarrow_h^* & \mathcal{CF}_{hn} & \longrightarrow_h & \mathcal{CF}_{hn+1} & \cdots\cdots\cdots & \mathcal{CF}_{hf} \\
\downarrow_r & & \downarrow_r & & \downarrow_r & & \downarrow_r \\
\mathcal{CF}_{t0} & \longrightarrow_t^* & \mathcal{CF}_{tn} & \longrightarrow_t & \mathcal{CF}_{tn+1} & \cdots\cdots\cdots & \mathcal{CF}_{tf}
\end{array}$$

We can prove that: $\mathcal{CF}_{h0} \longrightarrow_h^* \mathcal{CF}_{hn}$ if and only if $\mathcal{CF}_{t0} \longrightarrow_t^* \mathcal{CF}_{tn}$, with $T[\![\mathcal{CF}_{hi}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{ti}$, for $i = 0, \ldots, n$, and $\mathcal{CF}_{hn} \longrightarrow_h^* \mathcal{CF}_{hf}$. We proceed by induction on the length of the sequence $\mathcal{CF}_{h0} \longrightarrow_h^* \mathcal{CF}_{hn}$.

First let us prove the forward implication. Let us consider the base case when the sequence length is zero; then $\mathcal{CF}_{h0} = \mathcal{CF}_{hf}$ and $\mathcal{CF}_{t0} = \mathcal{CF}_{tf}$, with $T[\![\mathcal{CF}_{h0}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{t0}$, which proves the property.

In the inductive case, we consider the situation where we have a series of transitions of length $n+1$: $\mathcal{CF}_{h0} \longrightarrow_h^* \mathcal{CF}_{hn} \longrightarrow \mathcal{CF}_{hn+1}$, with $\mathcal{CF}_{hn+1} \longrightarrow_h^* \mathcal{CF}_{hf}$. Let us prove that $\mathcal{CF}_{t0} \longrightarrow_t^* \mathcal{CF}_{tn+1}$ with $T[\![\mathcal{CF}_{hi}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{ti}$, $i = 0, \ldots, n+1$.

By application of the inductive hypothesis (with $n < n+1$), we derive that: $\mathcal{CF}_{t0} \longrightarrow_t^* \mathcal{CF}_{tn}$ with $T[\![\mathcal{CF}_{hi}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{ti}$, $i = 0, \ldots, n$. Then, by application of Lemma 1, we find $\mathcal{CF}_{tn+1}$, such that $\mathcal{CF}_{tn} \longrightarrow_t^* \mathcal{CF}_{tn+1}$ and $T[\![\mathcal{CF}_{hn+1}, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_{tn+1}$.

The backward implication also holds. In addition, for every initial configuration on the trusted server $\mathcal{CF}_{t0}$ for a program Seq, there is a corresponding configuration $\mathcal{CF}_{h0}$ on the host. Transitions on both the trusted server and the host are deterministic. So, we can conclude that: $\mathcal{CF}_{h0} \longrightarrow_h^* \mathcal{CF}_{hn}$ <u>*if and only if*</u> $\mathcal{CF}_{t0} \longrightarrow_t^* \mathcal{CF}_{tn}$.

Since $\mathcal{CF}_{hn} \longrightarrow_h^* \mathcal{CF}_{hf}$ and $\mathcal{CF}_{hn} \longrightarrow_h^* \mathcal{CF}_{tf}$ and Lemma 1 can be applied at all stages of these transitions in the way that we have done for the inductive case earlier, we further conclude that: $\mathcal{CF}_{h0} \longrightarrow_h^* \mathcal{CF}_{hf}$ <u>*if and only if*</u> $\mathcal{CF}_{t0} \longrightarrow_t^* \mathcal{CF}_{tf}$.

This concludes the proof of the theorem.

5.1.9 Discussion of theorem

We first note that Lemma 1 (and hence Theorem 1) is applicable for the case of configurations on the host that satisfy the diamond property; thus when there is an input command occurring in the command sequence when the input stream is empty, the evaluation of the program is suspended and cannot continue. If the input stream remains empty indefinitely, it either implies that the inputs an agent expects during the course of its execution are not forthcoming from the external network environment of the host platform or that the host platform is actively sabotaging the input stream. In both cases, the evaluation of the program cannot complete successfully and hence any traces (input, output, fork, etc) produced up to this point in time will subsequently be incomplete as well and cannot be used in the verification process on the trusted server.

In Definition 3 and Theorem 1, $Eval(Seq, \mathcal{I})$ denotes a successful evaluation in accordance with the rules of the respective operational semantics. The violation of these rules would indicate that either:

- program functionality has been altered (i.e. the code or state of the agent has been altered prior to execution), or;
- program execution flow has been tampered with (i.e. the state of the agent does not change correctly in accordance to the execution of the code).

Such violations would render invalid the process of agent program execution by the host environment and the subsequent trace verification (Definition 3) on the trusted server will fail as a consequence.

The proof of Theorem 1 indicates two important properties about the verification process. Firstly, it indicates that proper completion of the verification process on the trusted party would signify that the original execution of the agent on the host would have been valid. In other words, there is no possibility of a successful verification process on the trusted party in the event of malicious tampering of agent execution by the host (i.e. the evaluation of a program does not proceed according to the predefined semantics and Definition 1 does not hold). This is important because it demonstrates that the interaction tracing methodology is capable of detecting any tampering that results in deviation from the predefined semantics, and thus fulfils the original goals of execution tracing. We note that there was no attempt to formally prove in any way such capability on the part of execution tracing in its original proposition; it was only argued for in an intuitive manner as we had described in the preceding chapter.

Secondly, it shows that if the original agent was executed in a valid manner on the host, then it is guaranteed that the verification process will be successful as well. In other words, there is no possibility of the verification process failing and flagging an otherwise honest host as being responsible for tampering. This eliminates the problem of faulty identification of malicious hosts. Again this possibility was not addressed in the original proposition of execution tracing.

Both of these properties depend on the important assumption that the traces required in the verification process are securely delivered with their integrity intact via the network medium connecting the host platform and the trusted server. Thus, discrepancies between the replay trace and output trace can be rightly attributed to the host platform. In practice, information delivered through a public network will always be subject to the risk of tampering, replay and interception. Secure submission of traces from the host to the trusted server is assured through the protocol discussed in section 4.1.1. for the case of execution tracing, and we will develop our own protocol that ensures this assumption holds in Chapter 6. It is also possible for an error (intentional or otherwise) to occur in the creation of the input or output trace during program execution on the host; such an incidence will also result in a failed verification process. Although this technically does not constitute an actual tampering of the agent per se during the execution process, we still consider it as a malicious attack on the agent.

5.2 Multiple thread of deterministic execution with shared state

One of the shortcomings of the execution tracing approach was that its ability to verify correct traces was limited to those created from the execution of single threaded agents only. However, consideration of multithreaded agents is important as virtually all agent programming languages have the capability of expressing multithreading constructs. In this section, we examine how the activity of creating and verifying traces in the interaction tracing approach can be extended for the case of an agent program consisting of multiple threads, but with a single shared state among the threads. Since the interleaving between threads is random for each execution run of the agent, the order in which state-affecting commands are executed in specific threads would influence the resulting state changes and, subsequently, the interaction behaviour of the agent. The outcome of this is that even if an agent is executed correctly according to the operational semantics that we defined for the previous case (single threaded agent), the output trace produced from such an execution may not be equivalent to the replay trace produced from the verification process (i.e. Definition 3 may no longer hold). We show a possible way of handling this complication through externalizing some aspects of the multithreaded execution of the original agent. We first define the functions that are to be used in the operational semantics that we develop.

5.2.1 Auxiliary functions in operational semantics

Consider $\mathcal{T}$ to be an ordered finite sequence of pairs of numerals $(x_1, y_1) \ldots (x_n, y_n)$. We use the notation $(x_1, y_1).L$ to refer to $\mathcal{T}$ where (x_1, y_1) is the first pair in the ordered sequence and L is the remaining sequence of pairs.

$findr(x, \mathcal{T})$

The function $findr(x, \mathcal{T})$ returns the second element of the first pair which has x as its first element, in the sequence $\mathcal{T}$, and can be inductively defined as:

1. Base case: $findr(x, \phi) = \phi$;
2. Inductive case: $findr(x, (x, y).L) = y$;
   $findr(x, (a, b).L) = findr(x, L)$ when $x \neq a$.

$findl(y, \mathcal{T})$

The function $findl(y, \mathcal{T})$ returns the first element of the first pair which has y as its second element, in the sequence $\mathcal{T}$, and can be inductively defined as :

1. Base case : $findl(y, \phi) = \phi$;
2. Inductive case : $findl(y, (x, y).L) = x$;
   $findl(y, (a, b).L) = findl(y, L)$ when $y \neq b$.

$rem(x, \mathcal{T})$

The function $rem(x, \mathcal{T})$ removes the first pair from $\mathcal{T}$ that has x as its first element, and can be defined inductively as:

1. Base case : $rem(x, \phi) = \phi$;
2. Inductive case : $rem(x, (x, y).L) = L$;
   $rem(x, (a, b).L) = (\mathrm{a.b}).rem(x, L)$ when $x \neq a$.

$proj(x, \mathcal{T})$

The function $proj(x, \mathcal{T})$ builds a sequence of pairs whose first elements are x from $\mathcal{T}$, and can be defined inductively as:

1. Base case : $proj(x, \phi) = \phi$;
2. Inductive case : $proj(x, (x, y).L) = (x, y).proj(x, L)$;
   $proj(x, (a, b).L) = proj(x, L)$ when $x \neq a$.

$remseq(\mathcal{T}_1, \mathcal{T}_2)$
The function $remseq(\mathcal{T}_1, \mathcal{T}_2)$ removes from $\mathcal{T}_2$ the sequence of pairs in $\mathcal{T}_1$. We can define $remseq(\mathcal{T}_1, \mathcal{T}_2)$ inductively as :

1. Base case : $remseq(\phi, \mathcal{T}_2) = \mathcal{T}_2$;
2. Base case : $remseq(\mathcal{T}_1, \phi) = \phi$;
3. Inductive case : $remseq((x, y).L_1, (a, b).L_2) = remseq(L_1, L_2)$ if $x = a$ and $y = b$;
   $remseq((x, y).L_1, (a, b).L_2) = (a, b).remseq((x, y).L_1, L_2)$ if either $x \neq a$ or $y \neq b$.

$map(\mathcal{T}_1, \mathcal{T}_2)$
The function $map(\mathcal{T}_1, \mathcal{T}_2)$ uses the $findr$ function to create a new sequence of pairs from $\mathcal{T}_1$ using a search on the exisiting pairs in $\mathcal{T}_2$:

1. Base case : $map(\phi, \mathcal{T}_2) = \phi$;
2. Inductive case : $map((x, y).L_1, \mathcal{T}_2) = (a, y).map(L_1, \mathcal{T}_2)$
   where $a = findr(x, \mathcal{T}_2)$.

$join(\mathcal{T}_1, \mathcal{T}_2)$
If $\mathcal{T}_1$ and $\mathcal{T}_2$ both contain the same number of pairs, the function $join(\mathcal{T}_1, \mathcal{T}_2)$ creates a new sequence of pairs composed from the left elements of each matching pair in $\mathcal{T}_1$ and $\mathcal{T}_2$, and can be defined as:

1. Base case : $join(\phi, \phi) = \phi$;
2. Inductive case : $join((x, y).L_1, (a, b).L_2) = (x, a).join(L_1, L_2)$.

5.2.2 State space of operational semantics

The abstract syntax for operational semantics that we develop here (Fig. 5.7) is extended from the one that we developed in section 5.1.2, to include the need to consider multiple threads of execution and the fact that the environment is now shared between these threads. Each thread is given an identifier *tid*, which we assume is uniquely assigned by the execution environment. When input or output occurs, values are read into, or generated from, specific threads; this association is

1. Syntactic categories

$$
\begin{array}{rll}
tl & \text{in} & ThreadStateList \\
ts & \text{in} & ThreadStates \\
ss & \text{in} & SystemStates \\
env & \text{in} & Env \\
enva & \text{in} & EnvAssoc \\
ta & \text{in} & TraceAssoc \\
fa & \text{in} & ForkAssoc \\
tid & \text{in} & ThreadID \\
\mathcal{O} & \text{in} & Output \\
\mathcal{I} & \text{in} & Input \\
\mathcal{OT} & \text{in} & OutputTrace \\
\mathcal{IT} & \text{in} & InputTrace \\
\mathcal{ST} & \text{in} & StoreTrace \\
\mathcal{FT} & \text{in} & ForkTrace
\end{array}
$$

2. Definitions

$$
\begin{array}{rlll}
env & ::= & \epsilon \mid enva.env & \text{(Environment for all shared threads)} \\
enva & ::= & (i, n) & \text{(Association of identifier/numeral)} \\
ta & ::= & (tid, n) & \text{(Association of thread identifier/numeral)} \\
fa & ::= & (tid, tid') & \text{(Association of parent thread id/child thread id)} \\
ts & ::= & \langle tid, Seq \rangle & \text{(Description of a single thread)} \\
tl & ::= & \epsilon \mid ts.tl & \text{(List of thread states)} \\
ss & ::= & \langle tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST} \rangle & \text{(State of entire system)} \\
\mathcal{I} & ::= & \epsilon \mid n.\mathcal{I} & \text{(Input stream)} \\
\mathcal{O} & ::= & \epsilon \mid n.\mathcal{O} & \text{(Output stream)} \\
\mathcal{IT} & ::= & \epsilon \mid ta.\mathcal{IT} & \text{(Input trace)} \\
\mathcal{OT} & ::= & \epsilon \mid ta.\mathcal{OT} & \text{(Output trace)} \\
\mathcal{ST} & ::= & \epsilon \mid tid.\mathcal{ST} & \text{(Store trace)} \\
\mathcal{FT} & ::= & \epsilon \mid fa.\mathcal{FT} & \text{(Fork trace)} \\
tid & ::= & n &
\end{array}
$$

Figure 5.7: State space for operational semantics

established through the use of ta. Existing threads have the capability of creating new threads with a given sequence of commands. fa is used to establish the association between the child and parent thread identifiers, and these associations are accumulated as pairs in the fork trace $\mathcal{FT}$. This trace is used later during the replay process in the trusted server environment.

For a shared state among threads, the sequence in which commands are executed between threads will influence the final state of the agent and its subsequent interaction with its environment. Thus replaying execution of the program in the trusted server environment requires a way of capturing the interleaving sequence of commands in the original host environment. This means we need to externalize the interleaving sequence aspect of multithreaded execution in order to support the replay process, and this externalization is achieved through the store trace, $\mathcal{ST}$.

1. Constant

$$env, n \vdash n$$

2. Environment lookup

$$(i,n).env, i \vdash n$$

3. Environment lookup2

$$\frac{env, n \vdash n \quad i \neq i'}{(i',n').env, i \vdash n}$$

4. Binary expression

$$\frac{env, Exp_1 \vdash n_1 \quad env, Exp_2 \vdash n_2 \quad n_1 \ op \ n_2 = n}{env, Exp_1 \ op \ Exp_2 \vdash n}$$

5. Expression assignment

$$\frac{env, Exp \vdash n}{\langle\langle tid, i := Exp; Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle\langle tid, Seq\rangle.tl, (i,n).env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}.tid\rangle}$$

6. Input

$$\langle\langle tid, input\ i; Seq\rangle.tl, env, n.\mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle\langle tid, Seq\rangle.tl, (i,n).env, \mathcal{I}, \mathcal{O}, \mathcal{IT}.(tid,n), \mathcal{OT}, \mathcal{FT}, \mathcal{ST}.tid\rangle$$

7. Output

$$\frac{env, i \vdash n}{\langle\langle tid, output\ i; Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle\langle tid, Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}.n, \mathcal{IT}, \mathcal{OT}.(tid,n), \mathcal{FT}, \mathcal{ST}.tid\rangle}$$

8. Fork

$$\frac{tid' \text{ - fresh thread identifier}}{\langle\langle tid, fork(Seq'); Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle\langle tid, Seq\rangle.\langle tid', Seq'\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}.(tid,tid'), \mathcal{ST}.tid\rangle}$$

Figure 5.8: Operational semantics - Rules and state transitions

5.2.3 Operational Semantics

The rules for the operational semantics are shown in Fig. 5.8. The first four rules are identical to those in Fig. 5.3 and have already been discussed in section 5.1.3. The last four rules are similar to those in Fig. 5.3, with the additional difference of the thread identifier being accumulated in the store trace $\mathcal{ST}$. Each time a command is executed by a thread in the host environment which affects (input, expression assignment) or accesses (output) a variable in the shared store, the store trace allows the sequence in which the commands are executed to be captured. The store trace can then be used in the replay of the agent copy on the trusted server in a way that mirrors exactly the interleaving sequence between command execution in different threads on the original host. The last rule (fork) involves the creation of a new thread based on a predefined command sequence (Seq), with the association between the identifiers of the parent and child thread stored in the fork trace. This will also be submitted to the trusted server, where it will be used in the creation of a replay trace. Although the fork command does not affect or access the shared state, we still accumulate its thread identifier in $\mathcal{ST}$. This is necessary to simplify the transformation relationship that we define later on (Definition 9). The inclusion of thread identifiers for the fork command is negligible on the overall size of the trace since we can safely assume that the number of input / output commands will significantly outnumber the fork commands.

5.2.4 Evaluation of a multi threaded program

A program will initially start off as a single thread in the state $\langle\langle tid, Seq\rangle, env, \mathcal{I}, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon\rangle$ and undergoes a series of transitions in prescription with the rules just described. The termination of these state transitions occurs when the sequence of commands in Seq in all the threads in the thread list tl is exhausted, in a state $\langle\langle tid_1, \epsilon\rangle \ldots \langle tid_x, \epsilon\rangle, env, \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle$. A more formal definition would be as follows:

Definition 5 (Evaluation of a multithreaded program) *Given a list containing a single thread* $tl = \langle tid, Seq\rangle$*, the evaluation of a program is defined as* $Eval(tl, env, \mathcal{I}) = \langle\mathcal{IT}, \mathcal{OT}\rangle$*, if the machine commences in the state* $\langle tl, env, \mathcal{I}, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon\rangle$ *and eventually terminates in the state* $\langle tl', env', \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle$ *where* $tl' = \langle tid_1, \epsilon\rangle \ldots \langle tid_y, \epsilon\rangle$*, for some given* env' *and* $\langle tl, env, \mathcal{I}, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon\rangle \longrightarrow^* \langle tl', env', \epsilon, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle$

Similar comments apply to $Eval$ as in the case of section 5.1.4.

5.2.5 State space of operational semantics in verification

The state space of the operational semantics for verification is shown in Fig. 5.9. The verification process requires the input, output, fork and store traces which must be submitted by the host which executed the original agent. The thread identifiers assigned by the trusted server environment in the replay of the agent copy will be likely to be different from those assigned by the host environment. This is due to the fact that identifiers assigned usually depend on factors that may not be identical for both these environments, such as other processes currently executing in the environments. Since the verification of the traces involves $\mathcal{OT}$ and $\mathcal{RT}$, both of which contain thread identifiers from the host and trusted server environments respectively, an association needs to be established between the assigned identifiers in these two different environments. This association is achieved through the pair ma which is accumulated in the map trace, $\mathcal{MT}$. As is for the case of a single threaded agent program, the input trace $\mathcal{IT}$ is used to drive the replay of the agent copy on the trusted server, whose outputs are subsequently accumulated in $\mathcal{RT}$.

1. Syntactic categories

$$\begin{array}{rll}
tl & \text{in} & ThreadStateList \\
ts & \text{in} & ThreadStates \\
ss & \text{in} & SystemStates \\
env & \text{in} & Env \\
enva & \text{in} & EnvAssoc \\
ma & \text{in} & MapAssoc \\
ta & \text{in} & TraceAssoc \\
fa & \text{in} & ForkAssoc \\
tid & \text{in} & ThreadID \\
\mathcal{OT} & \text{in} & OutputTrace \\
\mathcal{IT} & \text{in} & InputTrace \\
\mathcal{ST} & \text{in} & StoreTrace \\
\mathcal{FT} & \text{in} & ForkTrace \\
\mathcal{RT} & \text{in} & ReplayTrace \\
\mathcal{MT} & \text{in} & MapTrace
\end{array}$$

2. Definitions

$$\begin{array}{rlll}
env & ::= & \epsilon \mid a.env & \text{(Environment for all shared threads)} \\
enva & ::= & (i, n) & \text{(Association of identifier/numeral)} \\
ta & ::= & (tid, n) & \text{(Association of thread id/numeral)} \\
fa & ::= & (tid, tid') & \text{(Association of parent thread id/child thread id)} \\
ma & ::= & (tid, tid') & \text{(Association of new thread id/original thread id)} \\
ts & ::= & \langle tid, Seq \rangle & \text{(Description of a single thread)} \\
tl & ::= & \epsilon \mid ts.tl & \text{(List of thread states)} \\
ss & ::= & \langle tl, env, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, \mathcal{ST} \rangle & \text{(State of entire system)} \\
\mathcal{RT} & ::= & \epsilon \mid ta.\mathcal{RT} & \text{(Replay trace)} \\
\mathcal{MT} & ::= & \epsilon \mid ma.\mathcal{MT} & \text{(Map trace)} \\
\mathcal{IT} & ::= & \epsilon \mid ta.\mathcal{IT} & \text{(Input trace from host platform)} \\
\mathcal{OT} & ::= & \epsilon \mid ta.\mathcal{OT} & \text{(Output trace from host platform)} \\
\mathcal{FT} & ::= & \epsilon \mid fa.\mathcal{FT} & \text{(Fork trace from host platform)} \\
\mathcal{ST} & ::= & \epsilon \mid tid.\mathcal{ST} & \text{(Store trace from host platform)} \\
tid & ::= & n &
\end{array}$$

Figure 5.9: State space for operational semantics in verification

5.2.6 Determining trace and trace state list equivalence

The input, output, store and fork traces obtained at the end of execution on the host are submitted to the trusted server which will undertake the process of verifying the output trace. The agent copy is executed using the input, fork and store traces and the replay trace produced at the end is verified against the original output trace. The interleaving of command executions between the different threads in the replay process is dictated by the store trace $\mathcal{ST}$ so that the sequence is identical to that which occurred on the original host. The identical sequence of execution means that the situation effectively reduces to the case of a single threaded agent, where there is only one possible sequence for inputs to be consumed and outputs to be produced during the flow of execution of either the original agent code or the agent copy. The notion of equivalence of the output and replay trace can be defined below:

Definition 6 (Equivalence of two traces) *For two traces in a multi-threaded environment, we can define equivalence as*

$\mathcal{T}_1 \ \simeq_P \ \mathcal{T}_2$, *if for all* $tid \in ThreadID$

$proj(tid, \mathcal{T}_1) \ \simeq_S \ proj(tid, \mathcal{T}_2)$

In addition, we also need to define the equivalence of thread state lists, tl, which we will be using later in the lemma and theorem that we need to prove. This notion of equivalence is based on the fact that every thread spawned from the execution of the agent on the original host has a matching thread during the replay on the trusted server, and the map trace $\mathcal{MT}$ provides the mapping between the different thread identifiers in these two environments.

Definition 7 (Equivalence of two thread state lists) *A thread state list on the host environment* tl_h *is equivalent to a thread state list in the trusted server environment* tl_t, $tl_h \simeq_T tl_t$, *if given a map trace* $\mathcal{MT}$, $map(tl_t, \mathcal{MT}) = tl_h$

5.2.7 Operational semantics for verification

The rules for the operational semantics for verification are shown in Fig. 5.10. The first four rules are identical to those for the operational semantics on the original host. For the next three rules (expression assignment, input and output), $\mathcal{ST}$ is used to dictate the sequence of command execution in the interleaving between threads. The trusted server environment selects the thread whose identifier is in the pair at the front of $\mathcal{ST}$ in order to conduct the replay process. As mentioned earlier, the intent is that the command execution sequence between the interleaved threads is identical to the sequence which occurred on the original host. Since $\mathcal{ST}$ is based on thread identifiers from the host, $\mathcal{MT}$ is used to obtain the corresponding thread identifiers on the trusted server environment.

In the input rule, the value to be read by the designated thread is obtained from $\mathcal{IT}$. Since the pairs in $\mathcal{IT}$ use thread identifiers assigned from the original host environment, it is necessary to locate the original thread identifiers first using $\mathcal{MT}$. The original thread identifier value and the corresponding value read from $\mathcal{IT}$ can then be associated as a pair in the environment. For the output rule, the thread identifier that is to be associated with the numeral in the pair to be appended to $\mathcal{RT}$ should be the original thread identifier from the host environment to ensure that the comparison of $\mathcal{RT}$ with $\mathcal{OT}$ during the trace verification process is accurate. The original thread identifier is again obtained from $\mathcal{MT}$.

The fork command creates a new thread, whose assigned identifier from the trusted server environment is tid'. The identifier tid' on the trusted server has to be associated with its matching thread identifier previously assigned by the original host environment. The value of this original thread identifier is obtained from $\mathcal{FT}$, and the pair of the new thread identifiers (from trusted server and host) is then stored in $\mathcal{MT}$. The association between these two new thread identifiers is necessary to ensure that the correct pair is recorded in $\mathcal{RT}$ in the event of the newly spawned thread performing its own inputs and outputs. Replay of the agent program is started with a single thread which will eventually spawn all other new threads through the fork command.

We can define the verification of output traces on the trusted server as follows:

Definition 8 (Verification of an output trace for multithreaded programs) *A thread list tl containing a single thread $\langle tid, Seq\rangle$ is verified with respect to some input trace $\mathcal{IT}$, output trace $\mathcal{OT}$, store trace $\mathcal{ST}$, and fork trace $\mathcal{FT}$, $Verify(tl, \mathcal{IT}, \mathcal{OT}, \mathcal{ST}, \mathcal{FT})$, if the machine commences in the state $\langle\ tl, \epsilon, \mathcal{IT}, \epsilon, \mathcal{FT}, \mathcal{MT}, \mathcal{ST}\rangle$ and eventually terminates in the state $\langle tl', env, \epsilon, \mathcal{RT}, \epsilon, \mathcal{MT}, \epsilon\rangle$, for a given $\mathcal{MT}$ and $\mathcal{RT}$, with $\mathcal{RT}\ \simeq_P\ \mathcal{OT}$, where $tl' =\ \langle tid_1, \epsilon\rangle \ldots \langle tid_y, \epsilon\rangle$*

5.2.8 Theorem for the verification process

The theorem and lemma that we develop here will be similar in nature to the one that we presented in section 5.1.8. The diamond property (Fig. 5.6) is applicable here as well. We need to develop a transformation relationship, T, that maps any state configuration from the program evaluation process on the host to a unique state configuration from the agent replay process on the trusted third party. In defining this relationship, we will again need to consider the final terminating state $\mathcal{CF}'_{hf}$ on the host in a program evaluation as we have for the case of the single threaded agent. This is due to the fact that the input trace $\mathcal{IT}_t$, fork trace $\mathcal{FT}_t$ and store trace $\mathcal{ST}_t$ of any state configuration on the trusted third party can only

1. Constant

$env, n \vdash n$

2. Environment lookup

$(i,n).env, i \vdash n$

3. Environment lookup2

$$\frac{env, n \vdash n \quad i \neq i'}{(i',n').env, i \vdash n}$$

4. Binary expression

$$\frac{env, Exp_1 \vdash n_1 \quad env, Exp_2 \vdash n_2 \quad n_1\ op\ n_2 = n}{env, Exp_1\ op\ Exp_2 \vdash n}$$

5. Expression assignment

$$\frac{pid = findr(tid, \mathcal{MT}), \qquad env, Exp \vdash n}{\begin{array}{l}\langle\langle tid, i := Exp; Seq\rangle.tl, env, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, (pid,i).\mathcal{ST}\rangle \longrightarrow \\ \langle\langle tid, Seq\rangle.tl, (i,n).env, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, \mathcal{ST}\rangle\end{array}}$$

6. Input

$$\frac{pid = findr(tid, \mathcal{MT})}{\begin{array}{l}\langle\langle tid, input\ i; Seq\rangle.tl, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, (pid,i).\mathcal{ST}\rangle \longrightarrow \\ \langle\langle tid, Seq\rangle.tl, (i, findr(pid, \mathcal{IT})).env, rem(pid, \mathcal{IT}, \epsilon), \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, \mathcal{ST}\rangle\end{array}}$$

7. Output

$$\frac{pid = findr(tid, \mathcal{MT}), \qquad env, i \vdash n}{\begin{array}{l}\langle\langle tid, output\ i; Seq\rangle.tl, env, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, (pid,i).\mathcal{ST}\rangle \longrightarrow \\ \langle\langle tid, Seq\rangle.tl, env, \mathcal{IT}, \mathcal{RT}.(findr(pid, \mathcal{MT}), n), \mathcal{FT}, \mathcal{MT}, \mathcal{ST}\rangle\end{array}}$$

8. Fork

$$\frac{pid = findr(tid, \mathcal{MT}), \quad tid' \text{ is a fresh thread identifer}}{\begin{array}{l}\langle\langle tid, fork(Seq'); Seq\rangle.tl, env, \mathcal{IT}, \mathcal{RT}, \mathcal{FT}, \mathcal{MT}, (pid,i).\mathcal{ST}\rangle \longrightarrow \\ \langle\langle tid, Seq\rangle.\langle tid', Seq'\rangle.tl, env, \mathcal{IT}, \mathcal{RT}, rem(pid, \mathcal{FT}), \mathcal{MT}.(tid', findr(pid, \mathcal{FT})), \mathcal{ST}\rangle\end{array}}$$

Figure 5.10: Operational semantics for verification - Rules and state transitions

be derived using the combination of the input trace $\mathcal{IT}_h$, fork trace $\mathcal{FT}_h$ and store trace $\mathcal{ST}_h$ of a given state configuration on the host and $\mathcal{IT}_{hf}$, $\mathcal{FT}_{hf}$ and $\mathcal{ST}_{hf}$ of the final state configuration of program evaluation.

Definition 9 (Definition of a transformation relationship) *Consider an initial state configuration on the host, $\mathcal{CF}_{h0}$, that will undergo a series of transitions eventually leading to a final terminating state,*

$\mathcal{CF}_{hf} = \langle tl_{hf}, env_{hf}, \mathcal{I}_{hf}, \mathcal{O}_{hf}, \mathcal{IT}_{hf}, \mathcal{OT}_{hf}, \mathcal{FT}_{hf}, \mathcal{ST}_{hf}\rangle$. *A transformation relationship that maps a given state configuration in this series of transitions on the host environment to an equivalent state configuration on the trusted server environment is given as follows:*

$$T[\![\langle tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle, \mathcal{CF}_{hf}]\!] =$$

$$\langle tl_t, env_t, \mathcal{IT}_t, \mathcal{RT}_t, \mathcal{FT}_t, \mathcal{MT}_t, \mathcal{ST}_t\rangle$$

with $tl_h \simeq_T tl_t, env_h = env_t, \mathcal{IT}_t \simeq_P remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}), \mathcal{OT}_h \simeq_P \mathcal{RT}_t, \mathcal{FT}_t \simeq_P remseq(\mathcal{FT}_h, \mathcal{FT}_{hf}), \mathcal{ST}_t \simeq_P remseq(\mathcal{ST}_h, \mathcal{ST}_{hf}) and \mathcal{MT}_t \simeq_P join(tl_t, tl_h)$

The formulation of the diamond property as a lemma is identical to what had been done in section 5.1.8, and we repeat it here again before we demonstrate its proof for the multi threaded agent scenario.

Lemma 2 (Diamond property for multiple thread, shared state) *Let* $\mathcal{CF}_h$ *refer to a particular state configuration in the host environment,* $\mathcal{CF}_{hf}$ *to the final state configuration in the host environment and* $\mathcal{CF}_t$ *to a state configuration in the trusted server environment. Then, if* $\mathcal{CF}_h \longrightarrow_h \mathcal{CF}'_h$*,* $\mathcal{CF}'_h \longrightarrow^*_h \mathcal{CF}_{hf}$ *and* $T[\![\mathcal{CF}_h, \mathcal{CF}_{hf}]\!] = \mathcal{CF}_t$*, there exists a* $\mathcal{CF}'_t$ *where* $\mathcal{CF}_t \longrightarrow_t \mathcal{CF}'_t$*, and* $T[\![\mathcal{CF}'_h, \mathcal{CF}_{hf}]\!] = \mathcal{CF}'_t$*.*

Proof of Lemma 2 In order to prove Lemma 2, we consider all configurations on both the host and the trusted server which have a non-emnpty command sequence since a transition can only occur from these places. We consider four main categories based on the four commands that can affect a transition:

1. Fork assignment, $\mathcal{C} = fork(Seq')$. We consider this command occuring in the command sequence of a thread with an identifier tid within tl_h, where $tl_h = tl.\langle tid, fork(Seq'); Seq_{tid}\rangle$. First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:
   $\langle tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle \longrightarrow_h$
   $\langle tl'_h, env_h, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle$
   The other contents of the new state configuration are as follows:

   - $tl'_h = tl_h.\langle tid', Seq_{tid'}\rangle$, where tid' is a fresh thread identifier.
   - $\mathcal{I}'_h \simeq_S \mathcal{I}_h$
   - $\mathcal{O}'_h \simeq_S \mathcal{O}_h$
   - $\mathcal{IT}'_h \simeq_P \mathcal{IT}_h$
   - $\mathcal{OT}'_h \simeq_P \mathcal{OT}_h$
   - $\mathcal{FT}'_h \simeq_P \mathcal{FT}_h.(tid, tid')$
   - $\mathcal{ST}'_h \simeq_S \mathcal{ST}_h.tid$

   Using the transformation relation to obtain the equivalent state configuration on the trusted server, we have:
   $T[\![\langle tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle, \mathcal{CF}_{hf}]\!] =$
   $\langle tl_t, env_t, \mathcal{IT}_t, \mathcal{RT}_t, \mathcal{FT}_t, \mathcal{MT}_t, \mathcal{ST}_t\rangle$

where $tl_t = tl.\langle nid, fork(Seq'); Seq_{nid}\rangle$ and $(nid, tid) \in \mathcal{MT}_t$, with nid a unique identifier assigned by the trusted server environment.

For the trusted server, $\mathcal{ST}_t$ is used to dictate the interleaved sequence in which commands of the various threads are executed. Thus only the thread whose identifier is at the front of the sequence $\mathcal{ST}_t$ will be able to trigger a state transition. Since $\mathcal{ST}_t = remseq(\mathcal{ST}_h, \mathcal{ST}_{hf})$, and $\mathcal{ST}'_h \simeq_S \mathcal{ST}_h.tid$, the current identifier at the front of the sequence $\mathcal{ST}_t$ is tid. As $(nid, tid) \in \mathcal{MT}_t$, the next thread whose command is to be executed is the one with identifier nid:

$\langle tl_t, env_t, \mathcal{IT}_t, \mathcal{RT}_t, \mathcal{FT}_t, \mathcal{MT}_t, \mathcal{ST}_t\rangle \longrightarrow_t$
$\langle tl'_t, env_t, \mathcal{IT}'_t, \mathcal{RT}'_t, \mathcal{FT}'_t, \mathcal{MT}'_t, \mathcal{ST}'_t\rangle$

The other contents of the new state configuration are as follows:

- $tl'_t = tl_t.(nid', Seq_{nid'})$ where nid' is a new thread identifier
- $\mathcal{IT}'_t \simeq_P \mathcal{IT}_t$
- $\mathcal{RT}'_t \simeq_P \mathcal{RT}_t$
- $\mathcal{FT}'_t \simeq_P rem(tid, \mathcal{FT}_t)$
- $\mathcal{MT}'_t \simeq_P \mathcal{MT}_t.(nid', tid')$
- $\mathcal{ST}'_t \simeq_S rem(nid, \mathcal{ST}_t)$

Finally, applying the transformation relation on the new state configuration of the host, $\mathcal{CF}'_h$ to obtain the new state configuration of the trusted server, we derive:

$T[\![\langle tl'_h, env'_h, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle, \mathcal{CF}_{hf}]\!] =$
$\langle\langle tl'_{2t}, env'_t, \mathcal{IT}'_{2t}, \mathcal{RT}'_{2t}, \mathcal{FT}'_{2t}, \mathcal{MT}'_{2t}, \mathcal{ST}'_{2t}\rangle$

The considerations for the input trace $\mathcal{IT}$, output trace $\mathcal{OT}$ and replay trace $\mathcal{RT}$ are similar to that for the case of the command expression in the single threaded agent while proving Lemma 1. In a similar fashion, $\mathcal{IT}'_{2t} \simeq_P \mathcal{IT}'_t$ and $\mathcal{RT}'_{2t} \simeq_P \mathcal{RT}'_t$. To see whether the diamond property holds, we now need in addition to consider $\mathcal{ST}'_{2t}, \mathcal{FT}'_{2t}$ and $\mathcal{MT}'_{2t}$. For the case of $\mathcal{ST}'_{2t}$, we note from Fig. 5.11 that:

- $\mathcal{ST}'_{2t} = remseq(\mathcal{ST}_h.tid, \mathcal{ST}_{hf})$
- $\mathcal{ST}'_t = rem(tid, remseq(\mathcal{ST}_h, \mathcal{ST}_{hf}))$

$$\begin{array}{rccccl}
& \mathcal{ST}_h & \longrightarrow_h & \mathcal{ST}'_h & \simeq_S & \mathcal{ST}_h.tid \\
& \downarrow_{T[\![]\!]} & & \downarrow_{T[\![]\!]} & & \\
& & & \mathcal{ST}'_{2t} & \simeq_S & remseq(\mathcal{ST}'_h, \mathcal{ST}_{hf}) \\
& \mathcal{ST}_t & \longrightarrow_t & \mathcal{ST}'_t & \simeq_S & rem(tid, \mathcal{ST}_t) \\
& \simeq_S & & & & \\
remseq(\mathcal{ST}_h, \mathcal{ST}_{hf}) & & & & &
\end{array}$$

Figure 5.11: Changes for $\mathcal{ST}$ in fork command

$$\begin{array}{rccccl}
& \mathcal{FT}_h & \longrightarrow_h & \mathcal{FT}'_h & \simeq_P & \mathcal{FT}_h.(tid, tid') \\
& \downarrow_{T[\![]\!]} & & \downarrow_{T[\![]\!]} & & \\
& & & \mathcal{FT}'_{2t} & \simeq_P & remseq(\mathcal{FT}'_h, \mathcal{FT}_{hf}) \\
& \mathcal{FT}_t & \longrightarrow_t & \mathcal{FT}'_t & \simeq_P & rem(tid, \mathcal{FT}_t) \\
& \simeq_P & & & & \\
remseq(\mathcal{FT}_h, \mathcal{FT}_{hf}) & & & & &
\end{array}$$

Figure 5.12: Changes for $\mathcal{FT}$ in fork command

Now, $\mathcal{ST}'_{2t}$ can also be expressed as $remseq(tid, remseq(\mathcal{ST}_h, \mathcal{FT}_{hf}))$, which reduces to $rem(tid, remseq(\mathcal{ST}_h, \mathcal{ST}_{hf}))$. Thus we can conclude $\mathcal{ST}'_t \simeq_S \mathcal{ST}'_{2t}$ by transitivity of $\simeq_S$, and the diamond property is satisfied here.

For the case of $\mathcal{FT}'_{2t}$, we note from Fig. 5.12 that:

- $\mathcal{FT}'_{2t} \simeq_P remseq(\mathcal{FT}_h.(tid, tid'), \mathcal{FT}_{hf})$, since $\mathcal{FT}'_h \simeq_P \mathcal{FT}_h.(tid, tid')$
- $\mathcal{FT}'_t \simeq_P rem(tid, remseq(\mathcal{FT}_h, \mathcal{FT}_{hf}))$, since $\mathcal{FT}_t \simeq_P remseq(\mathcal{FT}_h, \mathcal{FT}_{hf})$

Now, $\mathcal{FT}'_{2t}$ can also be expressed as $remseq((tid, tid'), remseq(\mathcal{FT}_h, \mathcal{FT}_{hf}))$, which reduces to $rem(tid, remseq(\mathcal{FT}_h, \mathcal{FT}_{hf}))$. Thus we can conclude $\mathcal{FT}'_t \simeq_P \mathcal{FT}'_{2t}$ by transitivity of $\simeq_P$.

For the case of $\mathcal{MT}'_{2t}$, we note from Fig. 5.13 that $\mathcal{MT}'_{2t} \simeq_P join(tl'_t, tl'_h) \simeq_P join(tl_t.(nid', Seq_{nid'}), tl_h.(tid', Seq_{tid'})) \simeq_P join(tl_t, tl_h).(nid', tid') \simeq_P \mathcal{MT}_t.(nid', tid') \simeq_P \mathcal{MT}'_t$. Therefore through the transivity of $\simeq_P$, $\mathcal{MT}'_{2t} \simeq_P \mathcal{MT}'_t$. The diamond property is thus established for the case of the fork command.

2. Expression assignment, $\mathcal{C} = i := Exp$. We consider this command occuring in the command sequence of a thread with an identifier tid within tl_h. First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:

   $\langle\langle tid, i := Exp; Seq_{tid}\rangle.tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle \longrightarrow_h$

   $\langle\langle tid, Seq_{tid}\rangle.tl_h, (i, n).env_t, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle$

$$
\begin{array}{rcrcl}
tl_h & \longrightarrow_h & tl'_h & = & tl_h.\langle tid', Seq_{tid'}\rangle \\
\downarrow_{T[\![]\!]} & & \downarrow_{T[\![]\!]} & & \\
 & & \mathcal{MT}'_{2t} & \simeq_P & join(tl'_t, tl'_h) \\
tl_t, \mathcal{MT}_t & \longrightarrow_t & tl'_t & = & tl_t.(nid', Seq_{nid'}) \\
 & & \mathcal{MT}'_t & \simeq_P & \mathcal{MT}_t.(nid', tid')
\end{array}
$$

Figure 5.13: Changes for $\mathcal{MT}$ in fork command

The contents of the new state configuration are as follows:

- $\mathcal{I}'_h \simeq_S \mathcal{I}_h$
- $\mathcal{O}'_h \simeq_S \mathcal{O}_h$
- $\mathcal{IT}'_h \simeq_P \mathcal{IT}_h$
- $\mathcal{OT}'_h \simeq_P \mathcal{OT}_h$
- $\mathcal{FT}'_h \simeq_P \mathcal{FT}_h$
- $\mathcal{ST}'_h \simeq_S \mathcal{ST}_h.tid$

Using the transformation relation to obtain the equivalent state configuration on the trusted server, we have:
$T[\![\langle\langle tid, i := Exp; Seq_{tid}\rangle.tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle, \mathcal{CF}_{hf}]\!] =$
$\langle\langle nid, i := Exp; Seq_{nid}\rangle.tl_t, env_t, \mathcal{IT}_t, \mathcal{RT}_t, \mathcal{FT}_t, \mathcal{MT}_t, \mathcal{ST}_t\rangle$
where nid is an existing thread identifier previously assigned by the trusted server environment, with $(nid, tid) \in \mathcal{MT}$.

The comments for the transition on the trusted server using $\mathcal{ST}_t$ is the same as for the case of the fork comamnd. The transition is as follows:
$\langle\langle nid, i := Exp; Seq_{nid}\rangle.tl_t, env_t, \mathcal{IT}_t, \mathcal{RT}_t, \mathcal{FT}_t, \mathcal{MT}_t, \mathcal{ST}_t\rangle \longrightarrow_t$
$\langle\langle nid, Seq_{nid}\rangle.tl_t, (i, n).env_t, \mathcal{IT}'_t, \mathcal{RT}'_t, \mathcal{FT}'_t, \mathcal{MT}'_t, \mathcal{ST}'_t\rangle$
The contents of the new state configuration are as follows:

- $\mathcal{IT}'_t \simeq_P \mathcal{IT}_t$
- $\mathcal{RT}'_t \simeq_P \mathcal{RT}_t$
- $\mathcal{FT}_t \simeq_P \mathcal{FT}'_t$
- $\mathcal{MT}'_t \simeq_P \mathcal{MT}_t$

- $\mathcal{ST}'_t \ \simeq_S \ rem(nid, \mathcal{ST}_t)$

Finally, applying the transformation relation on the new state configuration of the host to obtain the new state configuration of the trusted server, we derive:
$T[\![\langle\langle tid, Seq_{tid}\rangle.tl_h, (i,n).env_h, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle, \mathcal{CF}_{hf}]\!] =$
$\langle\langle nid, Seq_{nid}\rangle.tl_t, (i,n).env_t, \mathcal{IT}'_{2t}, \mathcal{RT}'_{2t}, \mathcal{FT}'_{2t}, \mathcal{MT}'_{2t}, \mathcal{ST}'_{2t}\rangle$
In all these transitions, $\mathcal{IT}$, $\mathcal{OT}$, $\mathcal{RT}$ and $\mathcal{MT}$ do not change and hence $\mathcal{IT}_{2t} \simeq_P \mathcal{IT}_t$, $\mathcal{RT}_{2t} \simeq_P \mathcal{RT}_t$ and $\mathcal{MT}_{2t} \simeq_P \mathcal{MT}_t$ in a manner similar to that already demonstrated. For the case of $\mathcal{ST}'_{2t}$, we have already seen from Fig. 5.11 that $\mathcal{ST}'_t \simeq_S \mathcal{ST}'_{2t}$ by transitivity of $\simeq_S$, and thus the diamond property is satisfied here.

3. Input command, $C = Input\ i$. Again, we wish to show that the diamond property holds for this command for the case of both an empty and non-empty input stream. We consider this command occuring in the command sequence of a thread with an identifier tid within tl_h.

   - For an empty input stream, $\mathcal{I} = \epsilon$ and the premises of the lemma do not hold, which implies that the lemma is trivially true.
   - For a non-empty input stream $\mathcal{I} = n.\mathcal{I}'$, and $\mathcal{CF}_h =$
     $\langle\langle tid, input\ i; Seq_{tid}\rangle.env_h, tl_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle$
     First we apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:
     $\langle\langle tid, input\ i; Seq_{tid}\rangle.tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle \longrightarrow_h$
     $\langle\langle tid, Seq_{tid}\rangle.tl_h, (i,n).env_h, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle$
     The contents of the new state configuration are as follows:
     - $n.\mathcal{I}'_h \ \simeq_S \ \mathcal{I}_h$
     - $\mathcal{O}'_h \ \simeq_S \ \mathcal{O}_h$
     - $\mathcal{IT}'_h \ \simeq_P \ \mathcal{IT}_h.(tid, n)$
     - $\mathcal{OT}'_h \ \simeq_P \ \mathcal{OT}_h$
     - $\mathcal{FT}'_h \ \simeq_P \ \mathcal{FT}_h$
     - $\mathcal{ST}'_h \ \simeq_S \ \mathcal{ST}_h.tid$

The transformation relation to obtain the equivalent state configuration on the trusted server is the same as in the case for the expression assignment command. The transition on the trusted server using $\mathcal{ST}_t$ is also the same as well, and the contents of the new state configuration are as follows:

- $\mathcal{IT}'_t \ \backsimeq_P \ rem(nid, \mathcal{IT}_t)$
- $\mathcal{RT}'_t \ \backsimeq_P \ \mathcal{RT}_t$
- $\mathcal{FT}'_t \ \backsimeq_P \ \mathcal{FT}_t$
- $\mathcal{MT}'_t \ \backsimeq_P \ \mathcal{MT}_t$
- $\mathcal{ST}'_t \ \backsimeq_S \ rem(nid, \mathcal{ST}_t)$

The final transformation relation on the new state configuration of the host, $\mathcal{CF}'_h$, to obtain the new state configuration of the trusted server is the same as the expression assignment command. In all these transitions, $\mathcal{OT}$, $\mathcal{RT}$ and $\mathcal{MT}$ do not change and hence $\mathcal{RT}_{2t} \backsimeq_P \mathcal{RT}_t$ and $\mathcal{MT}_{2t} \backsimeq_P \mathcal{MT}_t$ in a manner similar to that already demonstrated. The values of $\mathcal{ST}$ in these transitions are identical to those for the case of the expression command and $\mathcal{ST}'_{2t} \backsimeq_S \mathcal{ST}'_t$ in the same manner. For the case of $\mathcal{IT}$, we note from Fig. 5.14:

- $\mathcal{IT}'_{2t} \backsimeq_P remseq(\mathcal{IT}_h.(tid, n), \mathcal{IT}_{hf})$
- $\mathcal{IT}'_t \backsimeq_P rem(tid, remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}))$

Now, $\mathcal{IT}'_{2t}$ can also be expressed as $remseq((tid, n), remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}))$, which reduces to $rem(tid, remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}))$. Thus we can conclude $\mathcal{IT}'_{2t} \backsimeq_P \mathcal{IT}'_t$ by transitivity of $\backsimeq_P$, and the diamond property is satisfied here.

$$\begin{array}{ccccc}
\mathcal{IT}_h & \longrightarrow_h & \mathcal{IT}'_h & \backsimeq_P & \mathcal{IT}_h.(tid, n) \\
\downarrow_{T[\![]\!]} & & \downarrow_{T[\![]\!]} & & \\
 & & \mathcal{IT}'_{2t} & \backsimeq_P & remseq(\mathcal{IT}'_h, \mathcal{IT}_{hf}) \\
\mathcal{IT}_t & \longrightarrow_t & \mathcal{IT}'_t & \backsimeq_P & rem(tid, \mathcal{IT}_t) \\
\backsimeq_P & & & & \\
remseq(\mathcal{IT}_h, \mathcal{IT}_{hf}) & & & &
\end{array}$$

Figure 5.14: Changes for $\mathcal{IT}$ in input command

4. Output command, $C = output\ i$. We consider this command occuring in the command sequence of a thread with an identifier tid within tl_h. First we

apply the state transition rule to obtain the new state configuration, $\mathcal{CF}'_h$, on the host:
$\langle\langle tid, output\ i; Seq_{tid}\rangle.tl_h, env_h, \mathcal{I}_h, \mathcal{O}_h, \mathcal{IT}_h, \mathcal{OT}_h, \mathcal{FT}_h, \mathcal{ST}_h\rangle \longrightarrow_h$
$\langle\langle tid, Seq_{tid}\rangle.tl_h, env_h, \mathcal{I}'_h, \mathcal{O}'_h, \mathcal{IT}'_h, \mathcal{OT}'_h, \mathcal{FT}'_h, \mathcal{ST}'_h\rangle$
The contents of the new state configuration are as follows:

- $\mathcal{I}'_h \ \simeq_S \ \mathcal{I}_h$
- $\mathcal{O}'_h \ \simeq_S \ \mathcal{O}_h.n$
- $\mathcal{IT}'_h \ \simeq_P \ \mathcal{IT}_h$
- $\mathcal{OT}'_h \ \simeq_P \ \mathcal{OT}_h.(tid, n)$
- $\mathcal{FT}'_h \ \simeq_P \ \mathcal{FT}_h$
- $\mathcal{ST}'_h \ \simeq_S \ \mathcal{ST}_h.tid$

The transformation relation to obtain the equivalent state configuration on the trusted server is the same as in the case for the expression assignment command. The transition on the trusted server using $\mathcal{ST}_t$ is also the same as well, and the contents of the new state configuration are as follows:

- $\mathcal{IT}'_t \ \simeq_P \ \mathcal{IT}_t$
- $\mathcal{RT}'_t \ \simeq_P \ \mathcal{RT}_t.(nid, n)$
- $\mathcal{FT}'_t \ \simeq_P \ \mathcal{FT}_t$
- $\mathcal{MT}'_t \ \simeq_P \ \mathcal{MT}_t$
- $\mathcal{ST}'_t \ \simeq_S \ rem(nid, \mathcal{ST}_t)$

The final transformation relation on the new state configuration of the host, $\mathcal{CF}'_h$, to obtain the new state configuration of the trusted server is the same as the expression assignment command. In all these transitions, $\mathcal{MT}$ and $\mathcal{IT}$ do not change and hence $\mathcal{MT}'_{2t} \ \simeq_S \ \mathcal{MT}'_t$ and $\mathcal{IT}'_{2t} \ \simeq_S \ \mathcal{IT}'_t$. The values of $\mathcal{ST}$ in these transitions are identical to those for the case of the expression command and $\mathcal{ST}'_{2t} \simeq_S \mathcal{ST}'_t$ in the same manner. For the case of $\mathcal{OT}$ and $\mathcal{RT}$, we note as well that $\mathcal{RT}'_{2t} \ \simeq_S \ \mathcal{RT}'_t$ (Fig. 5.15), and the diamond property is satisfied here as well.

$$\begin{array}{llllll} \mathcal{OT}_h & \mathcal{OT}_h.(tid,n) & \simeq_P & \mathcal{OT}'_h & \simeq_P & \mathcal{RT}'_{2t} \\ \simeq_P & \simeq_P & & & & \\ \mathcal{RT}_t & \mathcal{RT}_t.(tid,n) & \simeq_P & \mathcal{RT}'_t & & \end{array}$$

Figure 5.15: Changes for $\mathcal{RT}$ in output command

The proof of Lemma 2 is concluded here. In the previous section on a single threaded agent, we presented Theorem 1 and we restate it here formally for the case of a multi threaded agent.

Theorem 2 *For any program consisting of a collection of threads tl, each thread with its individual control sequence Seq, and any input stream I and shared environment env, $Eval(tl, env, \mathcal{I}) = \langle \mathcal{IT}, \mathcal{OT} \rangle$ if and only if $Verify(tl, \mathcal{IT}, \mathcal{OT}, \mathcal{ST}, \mathcal{FT})$, for some traces $\mathcal{IT}$, $\mathcal{OT}$, $\mathcal{ST}$ and $\mathcal{FT}$*

The proof for this is identical to that for Theorem 1, the exception being that the contents of the state configurations on both the host and the trusted server are different for the multi threaded as compared to the single threaded case. As this does not impact on the induction that we use to complete the proof, we will not repeat it here.

5.3 Conclusion and discussion

In this chapter, we formalize the process of creating and verifying a trace in the interaction tracing approach through the use of operational semantics. We consider two main cases: the single threaded and multi threaded agent program with shared state. A theorem is developed and proved for both cases, and the security implications of this theorem is discussed. In particular, we note that proper completion of the verification process on the trusted party would signify that the original execution of the agent on the host was valid and conversely, a valid agent execution on the host would guarantee a successful verification process as well.

Existing effort in code security research that involves formal analysis of the techniques is primarily in the area where cryptographic methods are employed such as with encrypted functions (section 3.3.5) or detached / blinded digital signatures (section 3.4). In both cases, the focus of formalization is on demonstrating that the cryptographic techniques employed accomplish the desired result in their specialized application at a level of security comparable to that achievable in more conventional cryptographic uses. For example, existing cryptographic literature demonstrates that it is computationally intractable to derive knowledge of a private key solely from the analysis of digital signatures produced with that private key [136]. Thus, work such as [41] demonstrates that while a blinded key-pair embedded in an agent can be used by a potentially hostile host to produce signatures in the same way as a conventional private key, determining the owners private key by analyzing the blinded key-pair (a potential attack in this scenario) is at least as difficult as determining the private key by analyzing the digital signatures produced from it.

Cryptographic-based code security techniques are preventive in nature. The formalization we present in this chapter is the first attempt at understanding the security achievable in a code security approach (interaction tracing) that is based on detection. The nature of the security property in such approaches would in general define what constitutes a successful detection of an attack for a given approach and the nature of attack undertaken by a hostile host in the event of such a detection.

In addition, it is worthwhile to note that the submission of an output trace that verifies properly merely indicates that the host had undertaken at least a single valid execution of the agent program to produce that trace. This however does not provide any guarantee that the host will not tamper with the agent state before it migrates the agent onwards to its next destination. In the case of the original execution tracing protocol, such a tampering of agent state is detected when the trusted party (the agent owner site in the original protocol) contacts the next host in the agent's itinerary in order to obtain its trace. The initial state of the agent on the new host will be obtained as well, and any discrepancy in comparison with the state of the replayed agent copy on the agent owner's site will indicate potential tampering activity on the part of the previous host.

However, regardless of whether the verification process in the execution tracing protocol detects a hostile host through discrepancies in the submitted traces or through comparison of agent states, the fact remains that the verification only occurs after the agent completes its itinerary and returns to the agent owner's site. This essentially implies that the activity of the agent on all honest hosts after leaving a particular malicious host is suspect because the agent's tampered state may result in unpredictable behaviour on these honest hosts. In the next chapter, we see how this shortcoming can be addressed in the trace exchange protocol that we develop.

Chapter 6

Security Protocol for Trace Exchange

As we discussed in section 5.1.9, the security implications of interaction tracing depend on the important assumption that the traces used in the verification process are securely transmitted between the host and the trusted server. In this chapter, we develop an interaction protocol that provides for such a secure transmission and formally verify and model our protocol using the FDR finite-state model checker. Formal verification and modelling of the protocol permit specific security properties to be established and affirm that existing protocol flaws as documented in distributed systems security literature do not exist in the proposed protocol. In addition, we noted previously in section 4.1.2 that a trace of an agent program execution should contain the outputs in order to provide the possibility for detecting attacks conducted by a hostile host on the external communications of the agent (i.e. communication with other agents resident on remote hosts). In this chapter, we will develop the concept of trace reconciliation that allows the detection of attacks on inter agent communications. Trace reconciliation is implemented by extending the secure interaction protocol that we develop, and this extension is also verified and modelled in a similar manner. The primary contributions of this chapter towards the thesis are:

1. Developing a secure interaction protocol for exchange of traces that supports the interaction tracing approach.
2. Formally modelling this secure protocol using the FDR model checker based upon CSP process algebra.

3. Introducing the idea of reconciliation of traces and an extension on the secure protocol for achieving this, formally modelled and verified based on CSP as well.

In section 6.1, we demonstrate the flow of messages, agent code and traces between various entities in this interaction protocol during agent migration. The actual contents of this messages are described, modelled and verified using the FDR checker and the Casper modelling tool in section 6.2. In section 6.3, we explain how inter agent communication is protected through trace reconciliation and model and verify another interaction protocol to implement trace reconciliation. The limitations of our modelling approach with respect to existing attacks that have already been detected are discussed in section 6.4, followed by a conclusion in section 6.5. The material in this chapter has been previously presented as two different publications ([141], [142]).

6.1 Interaction protocol for interaction tracing

In section 4.1.1, the significance of the interaction protocol for the submission of agent traces, agent code and state in supporting trace verification in Vigna's execution tracing approach [148] is explained with the shopping agent example in section 4.3. This interaction protocol however suffers from a major disadvantage in that it is voluntarily initiated (in conjunction with the trace verification process) on the basis of suspicion on part of the agent owner only when the agent completes its itinerary and returns to the agent owner's site. As discussed in the conclusion of the previous chapter, this implies that in the event of a malicious attack occurring partway through an agent's itinerary, the activity of the agent on all honest hosts after leaving the malicious host responsible is suspect because the agent's tampered state may result in unpredictable behaviour on the remaining honest hosts. This may necessitate the launching of another mobile agent to revisit the honest hosts in the previously compromised agent's itinerary in order to ensure that its original task is completed successfully. In addition, a compromised mobile agent leaving a malicious host may engage in damaging behaviour (as a result of illegal state tampering) on the remaining hosts in its itinerary in the event that these hosts do not implement sufficient host security mechanisms.

Just as interaction tracing reduces some of the problems associated with the execution tracing approach (section 4.2.2), the interaction protocol that we develop here to support interaction tracing aims to overcome the disadvantages of the protocol used in execution tracing. The main features of our interaction protocol towards this end are as follows:

- A trusted third party undertakes the activity of trace verification on behalf of the agent owner. We shall term this entity a *verification server*, and this effectively now becomes the trusted server that we discussed in the formalization of the previous chapter.
- Trace verification is undertaken every time an agent completes execution on a host, and is completed before the agent migrates onwards to the next host in its itinerary.
- There may be more than one verification server in the system that undertakes the process of trace verification for the various hosts that a mobile agent visits, and a verification server can handle trace verification for more than one host.

The fact that trace verification is initiated at the end of an agent's execution on a given host means that a hostile attack is detected as soon as it is perpetuated, provided that the attack manifests a visible attack on the agent's inputs or outputs. In particular, it also allows the timely detection of attacks on external agent communication through the process of trace reconciliation, which we shall cover in section 6.4.

Since trace verification is mandatory and completed (with a successful outcome or otherwise) before the agent migrates, there is no longer a need for the host to retain the traces it produced for the purposes of that verification process. This overcomes the disadvantage with the original execution tracing approach (section 3.3.6) where the traces had to be retained by the host (since verification was voluntary on part of the agent owner) and the size of traces might incur problems with storage over a prolonged period of time. In addition, if a verification server possesses knowledge of the time at which an agent is migrated to a new host, there is the possibility for some form of safeguard against provision attacks (section 3.6.3). The verification server could assume possible malicious intent on the part of a host if it fails to submit the required traces for verification within a certain time period after receiving a newly migrated agent.

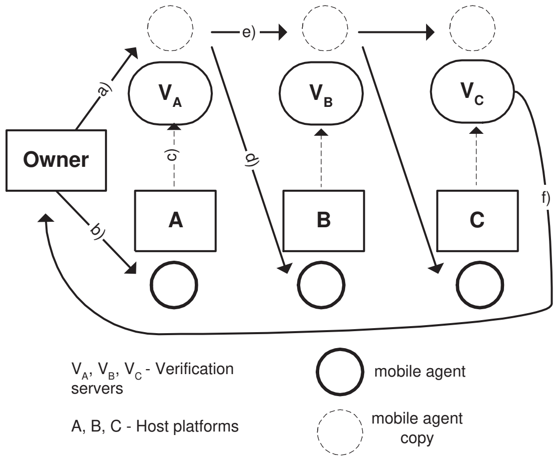

Figure 6.1: Verification servers in interaction tracing

6.1.1 Interaction protocol sequence

The new interaction protocol for exchanging traces and agent code is detailed in Fig. 6.1. When the agent owner platform launches a mobile agent to a host platform (b), it also creates a copy of the agent's code and state and forwards it to a verification server (a) designated by the host platform. As the host platform executes the agent, it creates the necessary traces that we have examined in the previous chapter. Upon request of migration from the agent, the host then forwards these traces (c) to a designated verification server, which undertakes the verification process using the agent copy it had received from the agent owner.

A successful verification of the traces would indicate that the agent copy on the verification server would be valid and identical in state to the actual mobile agent on the host. This agent copy is then forwarded from the verification server to its next destination host (d) and a copy (e) is sent to the corresponding verification server. Verification and migration then proceed in this fashion until the agent completes its itinerary and returns to its owner (f). A complete traversal of an agent along its itinerary, starting from where it departs from its owner (b) to the point when it returns again (e) constitutes a single *protocol run.* This may include any possible loops in its path (i.e. when an agent returns to a previously visited host).

Using the agent copy rather than the actual agent itself for the next hop in the itinerary confers two important advantages:

- There is no possibility of a host receiving a corrupted agent which may inadvertently attack it (as discussed earlier at the start of this section). It also avoids certain types of attacks that we discuss in greater detail in section 6.5.

- An unsuccessful verification process (signifying a possible attack by the host on the actual agent) could be used to signal an exception in the agent copy on the verification server. This agent copy could then undertake appropriate actions, such as recording the identity of the hostile host or migrating itself to the next host in its itinerary. The recovery from a malicious host attack in this manner preserves the intentions of the original mobile agent. An agent is thus guaranteed to complete its itinerary and return to the agent owner site in a valid form, regardless of the number of malicious hosts it may encounter on the way. This negates the need to relaunch a new agent to complete the tasks of a corrupted agent, as would be necessary in the case of the original interaction protocol for execution tracing.

Host platforms do not need to retain traces once they are submitted to the verification server; verification servers only retain submitted traces which do not verify properly. These faulty traces can later be submitted as evidence to the agent owner or a suitable arbitrator for appropriate sanctions to be undertaken if so required. Only verification servers are allowed to migrate agents to host platforms; correspondingly, honest platforms will only accept agents from authenticated verification servers.

Prior to the commencement of agent migration in the system, host platforms will need to interact with verification servers to determine the servers that are willing (or capable) of verifying traces of agents executing in their environment. A host platform could thus have a choice of several verification servers to use in verifying any trace from its environment; conversely, a verification server could be responsible for verifying traces from several different hosts. Verification servers may delegate verification activities to other verification servers in the system if they are overloaded; this allows the formation of trust relationships between servers. This aspect is discussed in greater detail in Chapter 8.

6.1.2 Shopping agent example revisited

We consider again the shopping agent example that we initially presented in section 4.3, taking into account our formalization of interaction tracing and the interaction

home	ShoppingMall	BlockBuster	EmiRecords
Input Trace	**Input Trace** 1. BlockBuster, EmiRecords	**Input Trace** 1. 15 +2. BlockBuster	**Input Trace** 1. 17
Output Trace +1. BlockBuster	**Output Trace** 1. (list,homevideo)	**Output Trace** 1. (directory,matrix) +2. (purchase,matrix)	**Output Trace** 1. (directory,matrix)

Figure 6.2: Interaction traces for correct execution

protocol that we have just presented. The list of input and output traces for the four different entities is shown in Fig. 6.2. Just as is the case in the original example, the agent will return to specific sites to complete its transaction. The + sign is again used to indicate that the input or output values are part of the second trace produced at the site in question corresponding to the second instance of agent execution there.

The important difference with the original example is the situation when the *EmiRecords* site attempts malicious tampering of the agent through the illegal manipulation of the internal state variable *price(BlockBuster)*. As we have just seen, the new interaction protocol sequence mandates that destination hosts only receive migrating agent copies from a valid verification server. The only way that *EmiRecords* might be able to affect the state of the agent copy is through the submission of its input trace, which is used in the replay of the agent copy on the verification server.

However, the input trace must also be accompanied by a corresponding digital signature (which we shall examine in the next section), which effectively establishes liability for the trace to the host that submitted it. In effect, the single value in the input trace becomes a non-repudiable offer by the site producing the trace for the *Matrix* item sought by the agent. In this example, *EmiRecords* could certainly choose to submit a value lower than 15 in its input trace in order to ensure that the agent migrates to its site instead, just as is the case in the original attack (Fig. 4.5). Of course, *EmiRecords* will now have to honour this lower offer, which does not in any way impact the agent's original task of purchasing the item at the cheapest price from the sites available. This essentially means there is no possibility for *EmiRecords* to submit a more expensive offer than *BlockBuster* and still manipulate the agent to conduct a transaction with it instead.

A different avenue of attack that *EmiRecords* could attempt (which we did not

home	ShoppingMall	BlockBuster	EmiRecords
Input Trace	**Input Trace** 1. BlockBuster, EmiRecords	**Input Trace** 1. 15 +2. BlockBuster	**Input Trace** 1. 12
Output Trace +1. BlockBuster	**Output Trace** 1. (list,homevideo)	**Output Trace** 1. (directory,matrix) +2. (purchase,matrix)	**Output Trace** *1. (directory,aliens)*

Figure 6.3: Interaction traces with malicious tampering by EmiRecords

discuss in the original example) is to manipulate the agent into requesting for a different home video title, which *EmiRecords* could offer at a cheaper price. For example, the title *Aliens* might be available for 12 dollars at its site; *EmiRecords* could manipulate the agent code execution so that it requests for *Aliens* instead of the correct title, *Matrix*. The interaction traces would then appear as shown in Fig. 6.3. The verification process on the server would produce a replay trace ($\mathcal{RT}$: (directory,matrix)) that would contradict the single value recorded in the output trace submitted by *EmiRecords*. At this point of time, an exception would be signalled to the agent copy on the verification server, which would then undertake the appropriate remedial measures as dictated by its relevant exception handling code.

6.2 Message sequence in interaction protocol

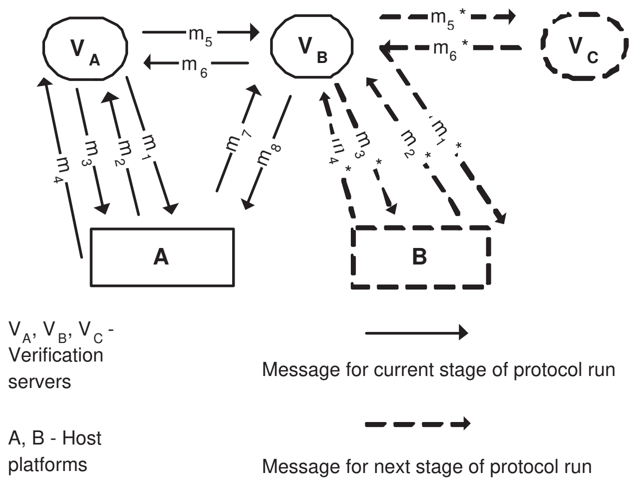

Figure 6.4: Message sequence in extended protocol

In this section, we detail the messages involved in the interaction protocol for one stage of a single protocol run (Fig. 6.4). We assume that a Public Key Infrastructure (PKI) is operating in the background, from which appropriate certificates and corresponding public keys can be obtained to perform encryption of data or verification of digital signatures. The contents and sequence of messages exchanged in the protocol are shown in Fig. 6.5; these follow the format in Vigna's paper [148].

m_1	:	$\{\{V_A, A, n_{V_A}, Ic, \{c\}_{SPr(Owner)}\}_{SPr(V_A)}\}_{Pb(A)}$	(Request agent)
m_2	:	$\{\{A, V_A, t_A^1, n_{V_A}, Ic, Acc, V_B, \{A, V_B\}_{SPr(V_B)}\}_{SPr(A)}\}_{Pb(V_A)}$	(Request response)
m_3	:	$\{\{V_A, A, t_A^1, n_{V_A}, Ic, S(c, V_A, T(c, V_A))\}_{SPr(V_A)}\}_{Pb(A)}$	(Decision)
m_4	:	$\{\{A, V_A, t_A^2, n_{V_A}, Ic, S(c, V_A, T(c, V_A))\}_{SPr(A)}\}_{Pb(V_A)}$	(Ack receipt)
m_5	:	$\{\{V_A, V_B, \{c\}_{SPr(Owner)}\}_{SPr(V_A)}, m_4\}_{Pb(V_B)}$	(Send agent)
m_6	:	$\{V_B, V_A, n_{V_A}, Ic\}_{Pb(V_A)}$	(Ack agent)
m_7	:	$\{\{A, V_B, Ic, n_{V_A}, T(c, A), S(c, A, T(c, A))\}_{SPr(A)}\}_{Pb(V_B)}$	(Send trace)
m_8	:	$\{V_B, A, n_{V_A}, Ic\}_{Pb(A)}$	
(Ack trace)		(Ack trace)	

Notation

- $\{X\}_{SPr(Y)}$ - indicates a message X signed with the private key of entity Y
- $\{X\}_{Pb(Y)}$ - indicates a message X encrypted with the public key of entity Y
- t_Y^X - refers to the X-th time stamp created at entity Y
- n_Y - indicates a nonce created at entity Y
- Ic - refers to a unique agent identifier
- c - refers to mobile agent code
- $T(c, Y)$ - refers to the traces created by entity Y after undertaking execution of agent code c
- $S(c, Y, T(c, X))$ - refers to the state of the agent code c after execution at entity Y, using the traces of the agent execution specified by $T(c, X)$

Figure 6.5: Protocol messages

m_1: This message sent by V_A is in reaction to the mobile agent's request to migrate to A. It is essentially a request to A to accept a mobile agent with the unique identifier Ic. A nonce n_{V_A} is included here to keep track of a protocol run, and will be subsequently included in the following messages as a reference to that protocol run. n_{V_A} is generated by V_A when m_1 is to be sent.

m_2: Upon receipt of m_1, A can decide whether or not to accept the mobile agent, and indicate accordingly in Acc. This decision may involve performing security checks on the code itself (i.e. host security) using techniques such as byte-code verification or proof carrying code. The last part of this message indicates the verification server (in this case V_B) that will verify the execution of any dispatched agent to A. This is accompanied by a certification signed by the verification server in question ($\{A, V_B\}_{SPr(V_B)}$). Message m_2 must be dispatched regardless of A's willingness to accept the mobile agent to allow V_A to distinguish between communication/server failure or agent rejection. A time stamp t_A^1 is included so that a record can be kept of m_2 in the event of a rejection.

m_3: An affirmative decision (with $Acc = Accept$) results in the state of the agent prior to migration being sent to A. In the event of rejection of platform A, the protocol run will terminate at this stage (with an appropriate exception flagged to the mobile agent), and recommence again at m_1 if so required by the mobile agent.

m_4: An acknowledgment message from A confirming the receipt of the agent. This message is vital to provide non-repudiation in the event that A attempts a denial of service attack once it has received the agent. The time stamp t_A^2 can be used to provide a reference value to implement a time-out mechanism in V_B to safeguard against denial of service.

m_5: A copy of the agent code and the entire contents of m_4 (which includes the agent's state) along with the appended signature created by $SPr(A)$ is then dispatched to V_B, the server that will be responsible for verifying correct execution on A. The identity of this verification server is obtained from the second portion of message m_2. Upon receipt of this message, a time-out mechanism can be effected at V_B using t_A^2 to ensure that m_6 arrives within a reasonable period of time, the failure of which is an indication of either a denial of service attack or a possible failure at A.

m_6: An acknowledgement of receipt of m_5 by V_B with inclusion of n_{V_A} and Ic to keep track of the current protocol run and agent instance.

m_7: Upon completion of agent execution, the traces created during the process of execution ($T(c, A)$) are submitted along with the new agent state $S(c, A, T(c, A))$ to the appropriate verification server.

m_8: An acknowledgement of receipt of m_7 by V_B with inclusion of n_{V_A} and Ic to keep track of the current protocol run and agent instance.

Upon receipt of m_7, V_B will commence the verification process using the submitted traces $T(c, A)$, proceeding in the manner dictated by the operational semantics of the previous chapter. If the verification is successful and the next stage of the protocol run can be initiated, V_B will dispatch m_1* (the equivalent of m_1 in the next stage of the protocol) to B and the traces used in the verification can then be discarded. Otherwise, an appropriate exception is raised in the agent copy by the execution environment of the verification server and the faulty trace and state is retained as evidence for further action by the agent owner platform or an arbitrator (if so required).

The pseudocode for the operation of the host platform and verification server in this protocol for exchanging traces is given in Fig. 6.6 and Fig. 6.7. Verification servers should be capable of undertaking the verification process for traces submitted from different host platforms simultaneously, while an individual host platform may have traces of agent execution in their environments verified by more than one verification server.

6.3 Formally modelling and verifying the protocol

At the start of this chapter, we noted that the security implications of interaction tracing depend on an interaction protocol that provides for secure transmission of traces, agent code and state between host platforms and verification servers. In the previous section, we outlined the various cryptographic primitives that we used in the messages of the protocol sequence to provide such security. The original protocol for submitting execution traces in execution proceeds in a similar manner as well but fails to formally verify the satisfaction of any security property. As it has been noted in the literature that developing good security protocols is notoriously difficult [2], we believe that some form of modelling and verification is necessary in order to provide a basic assurance that certain specific security properties are achievable. In the case of our interaction protocol, we are primarily interested in two security properties that provide guarantees of correct and secure dispatch of agents and traces:

- Mutual authentication of verification servers and host platforms. It is important to make a distinction between these two entities as host platforms should only accept mobile agents that are dispatched from verification servers. This ensures that honest host platforms will never accept agents with potentially corrupted states directly from other platforms. The possibility of a hostile host platform spawning multiple copies of an agent and dispatching it ran-

Operation of host platform

Accept m_1;
Verify signature $SPr(V_A)$ and identity of verification server (V_A) through a certificate;
Verify signature on agent code $(\{c\}_{SPr(Owner)})$ to detect possible tampering;
Perform security check on agent code $(\{c\})$
If decision is made to commit to running agent code *then*
  Select verification server V_B to verify traces;
  Submit in m_2 suitable certification $\{A, V_B\}_{SPr(V_B)}$;
  If verification server responds with m_3 *then*
    Verify signature on $S(m, V_A, T(m, V_A))$ to safeguard against possible tampering;
    Respond with acknowledgment m_4;
    Instanstiate agent code with verified state;
    Commence execution of mobile agent and create traces of its execution sequence $T(m, A)$;
    Upon completion of an agent run, sign trace and submit in m_7 to V_B;
  else
    Terminate protocol run and await commencement from another verification server;
else
  Indicate refusal in reply m_2;
  Terminate protocol run and await commencement from another verification server;

Figure 6.6: Pseudocode for operation of host platform

Operation of verification server V_B

Receive initial state of agent $S(c, V_A, T(c, V_A))$ in m_5;
Implement time-out mechanism using t_A^2 from m_4;
If trace $T(c, A)$ in m_7 arrives in specified time period *then*
  Verify identity of host submitting trace;
  Replay agent execution from initial state and submitted trace to obtain final state $S(c, V_B, T(c, A))$;
  If final state $S(c, V_B, T(c, A))$ is equivalent to submitted state $S(c, A, T(c, A))$ *then*
    Identify destination platform contained in $S(c, V_B, T(c, A))$;
    Submit m_1* to destination;
    Receive m_2*;
    If Acc in m_2* is positive *then*
      Verify identity of the verification server associated with submitted certification;
      Submit final state of agent $S(c, V_B, T(c, A))$ in m_3*;
      Receive acknowledgment of reception m_4* from the host platform;
      Forward $S(c, V_B, T(c, A))$ and $\{c\}_{SPr_{Owner}}$ on to the next verification server in m_5*;
    else
      Signal exception to agent;
      Record identity of host that refused to accept the agent;
  else
    Retain traces $T(c, A)$ as evidence;
    Signal exception to agent;
    Record occurrence of verification failure in agent;
else
  Signal exception to agent;
  Record occurrence of time-out in agent;

Figure 6.7: Pseudocode for operation of verification server

domly to other platforms in the system is also circumvented. In a similar context, a verification server has to ensure that it receives a copy of an agent from an authentic verification server to ensure that it is verifying the correct agent instance for a particular protocol run.

- Non-repudiation of commitment to executing agents. It is important to retain evidence of the fact that a host platform has committed to executing a particular mobile agent instance in a given protocol run to prevent a denial of service attack (i.e. terminating a mobile agent or delaying its execution for an inordinate period of time). This is primarily achieved by a digital signature appended to m_4 which is retained by V_A. The traces of agent execution as encapsulated in m_7 are also retained by V_B in the event that the verification process fails. They can be later submitted to a third party arbitrator or the agent owner for sanctions to be taken against the hostile platform if the need arises.

As we had previously noted in section 4.2.1, there may be situations where tampering detected at a particular host are caused by state corruption initiated at an earlier host in an agent's itinerary. The properties of mutual authentication and non-repudiation are useful this context by establishing clearly the sequence of hosts previously traversed by an agent, thus permitting the detection of the identity of the actual malicious host.

In considering the security properties of the protocol, it should be mentioned that the basic underlying assumption is that the verification server is treated as a trusted third party. Thus we assume that verification servers will not engage in any action that will directly or indirectly lead to the corruption of an agent's state. We also make the usual assumption that the basic cryptographic primitives used are resistant to standard cryptanalysis and that private keys are not compromised. There are many approaches available for formally modelling and verifying properties of a security protocol. Some of the more commonly utilised ones include:

1. BAN logic of authentication [23], which reasons about the states and beliefs of agents involved in a protocol run and how these beliefs evolve with the reception of new information.
2. Spi-calculus [1], which is an extension of π-calculus designed to deal with cryptographic primitives

3. Strand-spaces approach [36] uses the concept of a strand to represent the sequence of actions in which a particular protocol principal may participate and then reasons about how the strands interact or intertwine as participants interact by the exchange of messages.

4. CSP-based approach [87] models the protocol interactions as a system described by CSP process algebra [63], for which violation of given specifications can be detected through the use of a finite-state model checker such as FDR [91].

We have chosen the last approach as it has been used successfully in discovering attacks upon cryptographic protocols ([87], [88], [90]). In addition, modelling protocol runs as interactions between entities using a process algebra like CSP is a reasonably straight forward approach. As a complement to this approach, tools such as Casper [89] have been developed that are capable of converting a high-level description of a security protocol to a CSP specification of the model that can be fed as input into the FDR model checker for subsequent verification. Such tools greatly simplify the process of CSP specification, which can be tedious and error-prone for complicated protocols. We employ Casper in describing our protocol in this section, further details on this protocol specification language can be found at [89].

6.3.1 Modelling the protocol in Casper

For the free variables section, we have declared the following variable types:

```
#Free variables

A, BadHost : Agent
VA, VB : Server
nva, nt1, nt2, nb : Nonce
ca, cb : AgentCode
asva, asa, asb : AgentState
ata, atvb, atb : AgentTrace
PK : Agent -> PublicKey
SK : Agent -> SecretKey
SSK : Server -> ServerSecretKey
SPK : Server -> ServerPublicKey
hash : HashFunction
InverseKeys = (PK,SK), (SSK, SPK)
```

The agent code c, trace $T(c, Y)$ and state of the agent $S(c, Y, T(c, X))$ (Fig. 6.5) are represented by the types `AgentCode`, `AgentState` and `AgentTrace` respectively. Although the state of an executed agent is a one way function of its code, initial state and trace, the code, state and trace are presented here as independent and separate variables. During the checking of a model, the three variables can therefore assume a combination of values that would not be possible if the relationship between them were to be taken into account. This effectively models the possibility of a malicious host making invalid changes to either one of these variables during the a protocol run, and indirectly, this enhances the strength of the security property that we intend to establish. In accordance with the convention of modelling protocols in Casper, a distinction is made between the public and private keys of host platforms and verification servers. This is necessary since public and private keys do not share identical functionality in cryptographic primitives such as signatures. A hash function is used to model the unique agent identifier, Ic, which we treat simply as a hash of the agent code.

There are three processes representing V_A (SERVERINITIATOR), V_B (SERVER-RESPONDER) and A (HOSTRESPONDER) respectively of Fig. 6.4. The values in parentheses indicate the values that the entity in question possesses knowledge of at the start of a protocol run, with the first value being the identity of the entity. Thus SERVERINITIATOR is a process that corresponds to V_A, who will possess knowledge of its nonce, agent code and agent state (`nva, ca, asva`). The keyword `knows` is used to indicate knowledge of the respective public and private key functions, for example V_B is able to access its own secret key (`SSK(VB)`) and the public keys of all the other verification servers and hosts.

```
#Processes

SERVERRESPONDER(VB) knows SSK(VB), SPK, PK
SERVERINITIATOR(VA, nva, ca, asva) knows SSK(VA), SPK, PK
HOSTRESPONDER(A, nt1, nt2, asa, ata) knows SK(A), SPK, PK
```

The protocol is modelled below, where steps 1 – 8 correspond to $m_1 - m_8$ of the protocol (Fig. 6.5). Casper assumes that time stamps found in protocol messages are used to enforce the notion of freshness, i.e. to distinguish messages from a previous protocol run with messages from the current one. Since the time stamps t_A^1 and t_A^2 in our protocol are used for other reasons (as a means to implement a time-out mechanism), we model them as nonces instead. They are represented as `nt1` and `nt2` (of type `Nonce`). In all the steps listed below, the notation `PK{X}` or `SPK{Y}` is used to indicate that anything to the left of it will be encrypted with the public key of host X or verification server Y respectively. The same comments apply to `SK{X}` and `SSK{Y}`, which indicate signature with the private key of host X or verification server Y.

We briefly go through the steps shown below. The protocol run is preceded with step 0c, which establishes the result of an earlier interaction where A obtains a certification from a verification server V_B certifying V_B's capability of verifying agent traces from A. The `%` symbol is used in Casper to assign the preceding expression on its left to a temporary variable on its right, where it is reused at a later point in the protocol run. In this case, the certification $\{A, V_B\}_{SPr(V_B)}$ is stored in the temporary variable, `storecert`. Step 1 is self explanatory. In Step 2, `storecert` is used to send the certification obtained previously. In Step 3, `asva` is used to indicate the agent state $S(c, V_A, T(c, V_A))$. In Step 4, the entire contents of m_4 are stored in a temporary variable `enc`, where it is then used in Step 5 to embed m_4 in m_5. Steps 6 and 8 are self explanatory, while in Step 7 `asa` refers to the agent state $S(c, A, T(c, A))$.

```
#Protocol description

0.    -> VA : A
0a.   -> A : VA, VB
0b.  -> VB : A
0c.  VB -> A : {{A, VB}{SSK(VB)} % storecert}{PK(A)}
1.  VA -> A : {{VA, A, nva, ca, hash(ca)}{SSK(VA)}}{PK(A)}
2.  A -> VA : {{A, VA, nva, nt1, hash(ca), VB,
               storecert % {A, VB}{SSK(VB)}}{SK(A)}}{SPK(VA)}
3.  VA -> A : {{VA, A, nt1, hash(ca), asva}{SSK(VA)}}{PK(A)}
4. A -> VA : {{A, VA, nt2, nva, hash(ca), asva}{SK(A)},
             {A, VA, nt2, nva, hash(ca), asva}
             {SK(A)} % enc}{SPK(VA)}
5. VA -> VB : {{VA, VB, ca}{SSK(VA)},
               enc % {A, VA, nt2, nva, hash(ca), asva}
               {SK(A)}}{SPK(VB)}
6. VB -> VA : {VB, VA, nva, nt2}{SPK(VA)}
7. A -> VB : {{A, VB, hash(ca), nva, ata, asa}{SK(A)}}{SPK(VB)}
8. VB -> A : {VB, A, nva, hash(ca)}{PK(A)}
```

In order to test the integrity of the protocol against possible man-in-the-middle attacks, Casper models an intruder that is capable of interacting with the other specified entities during a protocol run. Such an intruder may be capable of intercepting messages destined for other entities or inserting its own messages in a given protocol run. At the start, it is necessary to specify the knowledge available to an intruder. In our model, the intruder is aware of its identity (`BadHost`) and the identity of the other entities in the protocol run (`A, VA, VB`), can create its own nonce, agent code, state and trace (`nb, cb, asb, atb`) and is aware of the public keys of both the host and verification servers (`PK, SPK`) as well as possessing a private key of its own (`SK(BadHost)`).

```
#Intruder Information

Intruder = BadHost
IntruderKnowledge = {A, VA, VB, Badhost, nb, cb, asb, atb, PK, SPK, SK(BadHost)}
```

6.3.2 Specifying security properties

As mentioned earlier, the two important security properties to be established are mutual authentication and non-repudiation. The concept of authentication in Casper is briefly expressed in the form of the statement `Agreement(A, B, [x])`, which states that `A` is authenticated to `B` on the basis of the fact that both `A` and `B` agree on the value of `x`. A more formal statement of this [89] is that if `B` (taking the role of responder) completes a protocol run, apparently with `A`, using the data value `x`, then the same entity `A` (taking the role of initiator) has previously been running the protocol, apparently with `B`, using the same value `x`. In addition, each such run of `B` corresponds to a unique run of `A`, with `x` being some unique data item (such as nonce or time-stamp) known only to `A` or `B`. Mutual authentication will therefore require the additional statement `Agreement(B, A, [x])` to be verified as well.

For the case of V_A and A, we can claim that these two entities are properly authenticated to each other after the exchange of m_1 - m_4, if only these two entities agree on the values $Ic, t_A^2, n_{V_A}, S(c, V_A, T(c, V_A))$. The identifier Ic is necessary to provide reference to the unique agent instance, n_{V_A} provides reference to the current protocol run, t_A^2 and $S(c, V_A, T(c, V_A))$ provides reference to A's response in m_4.

```
Agreement(VA, A, [nva, nt2, hash(ca), asva])
Agreement(A, VA, [nva, nt2, hash(ca), asva])
```

Similarly mutual authentication between V_A and V_B and between V_B and A can be expressed as:

```
Agreement(VA,VB, [nva, nt2])
Agreement(VB,VA, [nva, nt2])
Agreement(VB, A, [nva, nt2])
Agreement(A, VB, [nva, nt2])
```

Non-repudiation can be simplified to the more general case of maintaining secrecy of specific data items whose non-repudiation is to be established. If we know that only entity A issues item x in a protocol exchange between itself and another entity B, and if we can establish that item x remains secret in such a protocol exchange, then we can conclude that A is indeed responsible for issuing x. In our protocol, we are not interested whether x can later be duplicated by another entity (such as B) in another protocol run as non-repudiation does not seek to prevent duplication, only to indicate that a specific entity was responsible for issuing the item x. Appending a digital signature to x lets us identify the entity issuing x, which guarantees the property of non-repudiation. In our protocol, non-repudiation is necessary for messages:

1. m_2 - to provide evidence a host accepts or denies an agent for a particular protocol run indicated by n_{V_A}, and an agent instance indicated by Ic;
2. m_4 - to provide evidence a host has received the state $S(c, V_A, T(c, V_A))$ necessary to begin execution of the agent instance Ic in the protocol run indicated by n_{V_A};
3. m_7 - to provide evidence the agent instance was Ic executed with a trace $T(c, A)$ to provide a state $S(c, A, T(c, A))$.

In Casper, the statement `Secret(A, x, [B])` is used to express the property that `A` believes `x` remains secret in an interaction between itself and an entity that appears to be `B`. If this entity is not `B`, then `x` will remain hidden to the entity. For the interaction between V_A and A, we seek to determine that V_A has transmitted the state, agent code and the nonce n_{V_A} in secret to A, which provides non-repudiation for messages m_2 and m_4 (Ic is based on a hash of ca). In the other direction, the secrecy requirement is applied to the time stamp t_A^1 generated by A in m_2 and which should only be available to V_A in its response in m_3.

```
Secret(VA, nva, asva, ca, [A])
Secret(A, nt1, [VA])
```

For m_7, the secrecy requirement would be to ensure that the new agent state and trace are only known between A and V_B. The nonce `nva` is not included as it would have become known to V_B through m_5 prior to the start of interaction between A and V_B.

```
Secret(A, ata, asa,  [VB])
```

Note that these security specifications are provided in an abbreviated form; specifications for secrecy should be in the form Secret(VA, x, [A]) for each item x. Both specifications for mutual authentication and secrecy were satisfied in the resulting CSP model that was checked using FDR.

This indicates that the proposed interaction protocol does not suffer from any existing flaws of the type that Casper would be capable of detecting [87]. In addition, we have also demonstrated formally that the two important security properties of mutual authentication and non-repudiation are achieved in the protocol, which provides the necessary assumption for interaction tracing that we discussed in section 5.1.9.

6.4 Protecting inter agent communication

In section 3.6.1, we noted that agents executing on untrusted hosts are vulnerable to attacks on their communications with each other and one way of detecting such attacks in interaction tracing is through the inclusion of agent outputs in the traces describing an agent's execution (section 4.1.2). In the operational semantics developed in the previous chapter, a hostile host platform could record the actual value to be output correctly to the output trace $\mathcal{OT}$ but refrain from relaying this value correctly to its intended destination. This could take the form of:

1. Refraining completely from relaying the value onwards to its designated agent recipient;
2. Relaying a correct value more than once onwards to the designated agent recipient;
3. Relaying an incorrect value once or more than once onwards to the valid destination;
4. Relaying the correct/wrong value once or more than once onwards to an invalid agent destination.

In a similar manner, a host platform receiving messages relayed from an agent on another host may also seek to disrupt the correct conveying of the messages to agents executing in its environment. This would be achieved in a similar manner to that detailed above. To overcome the possibility of the compromise of agent communications in this way, there is therefore a need for the output trace corresponding to the execution of a single agent to be contrasted against the input traces of the other agents that this agent communicated with during the lifetime of its execution. Such input traces may already reside on the verification server of the agent in question, or they may be found on other verification servers. In the event of the latter case, the current verification server may need to interact with other verification servers to obtain the input traces in question to perform a necessary comparison. We explain in the next section the modifications that need to be made to the operational semantics that we have developed in the previous chapter in order to accommodate the comparison of input and output traces, a process that we shall term as *trace reconciliation*.

6.4.1 Operational semantics for trace reconciliation

The primary modification that needs to be made to the original operational semantics is the concept of a unique agent identifier in the system. Each thread within an agent program will have an agent identifier associated with it, and the information appended to the output stream, $\mathcal{O}$, output trace, $\mathcal{OT}$, and replay trace, $\mathcal{RT}$, will now be a triplet consisting of the agent identifier, thread identifier and numeral. Taking into account the agent identifier, the abstract syntax for the operational semantic model as well as the rules and transitions for the case of a multithreaded agent with shared state is now shown in Fig. 6.8 and Fig. 6.9. The identifier of the remote agent whose input is being consumed by the current agent is given in the input command $input(aid, i)$. In $\mathcal{IT}$, it is the identifier of this remote agent that is kept in ta. Simiar comments apply for the output command $output(aid, i)$, and the identifier of the targeted remote agent is accumulated in $\mathcal{OT}$ as well.

Assume the agent identifier of the current agent being verified on a given verification server is aid_c. From the output trace for this agent, we can obtain the input traces for all the other agents whose identifiers are listed in its output trace. These traces may or may not reside on the current verification server; it is the task of the given verification server to locate the verification servers responsible for the verification of these agents in order to obtain their input traces. We define a function that we will use in the algorithm to demonstrate trace reconciliation.

Given that $\mathcal{T}$ is a sequence of tuples (aid, tid, n), the function $proj(x, \mathcal{T})$ builds a sequence of numerals consisting of numerals n, and can be defined inductively as:

1. Base case : $proj(x, \phi) = \phi$

1. Syntactic categories

tl	in	$ThreadStateList$
ts	in	$ThreadStates$
ss	in	$SystemStates$
env	in	Env
aga	in	$AgentAssoc$
$enva$	in	$EnvAssoc$
ta	in	$TraceAssoc$
fa	in	$ForkAssoc$
tid	in	$ThreadID$
aid	in	$AgentID$
$\mathcal{O}$	in	$Output$
$\mathcal{I}$	in	$Input$
$\mathcal{OT}$	in	$OutputTrace$
$\mathcal{IT}$	in	$InputTrace$
$\mathcal{ST}$	in	$StoreTrace$
$\mathcal{FT}$	in	$ForkTrace$

2. Definitions

env	::=	$\epsilon \mid enva.env$	(Environment of a single thread)
$enva$	::=	(i, n)	(Association of variable id/numeral)
aga	::=	(aid, aid', n)	(Association of sending agent id/target agent id/numeral)
ta	::=	(aid, tid, n)	(Association of agent id/thread id/numeral)
fa	::=	(tid, tid')	(Association of parent thread id/child thread id)
ts	::=	$\langle tid, Seq\rangle$	(State of a single thread)
tl	::=	$\epsilon \mid aid.ts.tl$	(List of thread states)
ss	::=	$\langle tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle$	(State of entire system)
$\mathcal{I}$	::=	$\epsilon \mid aga.\mathcal{I}$	(Input stream)
$\mathcal{O}$	::=	$\epsilon \mid aga.\mathcal{O}$	(Output stream)
$\mathcal{IT}$	::=	$\epsilon \mid ta.\mathcal{IT}$	(Input trace)
$\mathcal{OT}$	::=	$\epsilon \mid ta.\mathcal{OT}$	(Output trace)
$\mathcal{ST}$	::=	$\epsilon \mid tid.\mathcal{ST}$	(Store trace)
$\mathcal{FT}$	::=	$\epsilon \mid fa.\mathcal{FT}$	(Fork trace)
tid	::=	n	

Figure 6.8: Abstract syntax for operational semantic model

2. Inductive case : $proj(x, (aid, tid, n).L) = n.proj(x, L)$ when $aid = x$
   $proj(x, (aid, tid, n).L) = proj(x, L)$ when $aid \neq x$

Assume that a verification server is about to verify the execution of an agent with identifier aid_c. In order to perform reconciliation, it needs to obtain the input traces corresponding to all the remote agents that the agent it is currently verifying has sent messages to. The algorithm below illustrates how the required input traces are obtained:

For an agent aid_c with an output trace $\mathcal{OT}$
$idlist = \phi$;
For every tuple (aid_i, tid_i, n_i) in $\mathcal{OT}$
    If $aid_i \notin idlist$ then

1. Constant $env, n \vdash n$	**2. Environment lookup** $(i,n).env, i \vdash n$
3. Environment lookup2 $\dfrac{env, n \vdash n \quad i \neq i'}{(i',n').env, i \vdash n}$	**4. Binary expression** $\dfrac{env, Exp_1 \vdash n_1 \quad env, Exp_2 \vdash n_2 \quad n_1\ op\ n_2 = n}{env, Exp_1\ op\ Exp_2 \vdash n}$

5. Expression assignment

$$\frac{env, Exp \vdash n}{\langle aid.\langle tid, i := Exp; Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle aid.\langle tid, Seq\rangle.tl, (i,n).env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}.tid\rangle}$$

6. Input

$$\langle aid.\langle tid, input\ (aid', i); Seq\rangle.tl, env, (aid', aid, n).\mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle aid.\langle tid, Seq\rangle.tl, (i,n).env, \mathcal{I}, \mathcal{O}, \mathcal{IT}.(aid', tid, n), \mathcal{OT}, \mathcal{FT}, \mathcal{ST}.tid\rangle$$

7. Output

$$\frac{env, i \vdash n}{\langle aid.\langle tid, output\ (aid', i); Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle aid.\langle tid, Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}.(aid, aid', n), \mathcal{IT}, \mathcal{OT}.(aid', tid, n), \mathcal{FT}, \mathcal{ST}.tid\rangle}$$

8. Fork

$$\frac{tid' \text{ - fresh thread identifier}}{\langle aid.\langle tid, fork(Seq'); Seq\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}, \mathcal{ST}\rangle \longrightarrow \langle aid.\langle tid, Seq\rangle.\langle tid', Seq'\rangle.tl, env, \mathcal{I}, \mathcal{O}, \mathcal{IT}, \mathcal{OT}, \mathcal{FT}.(tid, tid'), \mathcal{ST}.tid\rangle}$$

Figure 6.9: Operational semantics - Rules and state transitions

Obtain identity of verification server that verifies agent aid_i;
Obtain $\mathcal{IT}_i$ for agent aid_i from verification server;
$idlist = idlist.aid_i$;

Once all the input traces have been gathered, the reconciliation of them against the output trace $\mathcal{OT}$ for agent aid_c proceeds as follows:

For every input trace $\mathcal{IT}_i$ that originated from an agent aid_i
  $\mathcal{PIT}_i = proj(aid_c, \mathcal{IT}_i)$;
  $\mathcal{POT}_i = prof(aid_i, \mathcal{OT})$;
  If $\mathcal{PIT}_i \simeq_S \mathcal{POT}_i$
    Successful reconciliation;
  else
    Failure;
    Record value of aid_i and take appropriate action;

Unsuccessful reconciliation of an input trace with the given output trace would signify one of two possibilities: either the host platform executing the agent associated with the output trace corrupted the outputs or the host platform executing the agent associated with the input trace corrupted the incoming inputs. It is however impossible to determine which one of the two possibilities is the case. Such a decision would have to take into account additional factors such as the incidence of verification failure of an agent (indicating possible compromise) on one of these two suspect host platforms.

We note that it is possible for two malicious hosts to collude in an inter agent communication attack. For example, host A executing agent a_1 could submit a valid output trace to its verification server, but relay incorrect values towards another agent b_1 executing on another host B. At the same time, B colludes with A to submit an input trace that matches the output trace that A submitted, while supplying the incorrect values to b_1. In such a situation, the reconciliation process will not detect the violation of the communication between a_1 and b_1. However, the verification of the output traces from b_1 would reveal the discrepancies resulting from its subverted inputs. If B chooses to fake the output traces as well so that they verify correctly, there is effectively no attack since it is the agent copy (with the correct state) on the verification server that migrates on to the next host. Thus, collusion attacks are guaranteed to be detected on the second host involved in the collaboration, and if they are not, then the attacks are of no consequence anyway.

With regards to the previous example, agent a_1 may, during a valid execution sequence, communicate with a large number of agents residing on remote hosts distributed throughout the system. The verification server for host A would then need to engage in interactions with the verification servers responsible for verifying all these remote hosts. This will not significantly affect the complexity of the trace reconciliation process itself (which is linear in relation to the number of remote verification servers that need to be contacted), but the issue of scalability might need to be taken into consideration. Scalability considerations are however outside the scope of this thesis, but will be relevant for future work that we discuss in Chapter 9.

6.4.2 Message sequence for trace reconciliation

As was the case in section 6.2, trace reconciliation will also involve an interaction protocol which allows a given verification server to obtain the input traces it requires from other verification servers in the system. The message protocol sequence is shown in Fig. 6.10 and can take place at any point in time after the reception of the output traces from the host in question (m_7 in the original proto-

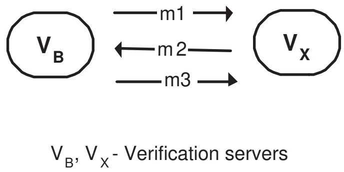

Figure 6.10: Message sequence for obtaining input traces

col - Fig. 6.5). Note that V_X here refers to any verification server in the system, rather than the verification server that is responsible for the host in the next hop of the agent's itinerary (V_C in Fig. 6.5).

m_1	:	$\{\{V_B, V_X, n_{V_B}, Ic,\}_{SPr(V_B)}\}_{Pb(V_B)}$	(Request trace)
m_2	:	$\{\{V_X, V_B, n_{V_B}, n_{V_X}, Ic, \mathcal{IT}\}_{SPr(V_X)}\}_{Pb(V_X)}$	(Send trace)
m_3	:	$\{V_B, V_X, n_{V_X}, Ic\}_{Pb(V_B)}$	(Ack trace)

Notation

- $\{X\}_{SPr(Y)}$ - indicates a message sequence X signed with the private key of entity Y
- $\{X\}_{Pb(Y)}$ - indicates a message sequence X encrypted with the public key of entity Y
- n_Y - indicates a nonce created at entity Y
- Ic - refers to a unique agent identifier
- $\mathcal{IT}$ - refers to a collection of input traces

Figure 6.11: Protocol messages

m_1: This message is sent to V_X in order to request all input traces currently residing on V_X which record messages originating from an agent with identifier Ic. This is the agent whose output trace V_B wishes to perform a reconciliation process on. The nonce n_{V_B} is included here to keep track of a protocol run. The message is signed as a form of non-repudiation to ensure that V_B cannot deny request of some input traces.

m_2: Upon receipt of m_2, V_X will retrieve the required input traces and return them to V_B, along with a new nonce n_{V_X} and the received nonce n_{V_B}. Again this is signed to ensure non-repudiation with regards to the input traces $\mathcal{IT}$ that V_X elects to send.

m_3: V_B responds with the received nonce n_{V_X} to acknowledge receipt of the input traces and concludes the protocol run.

It is pertinent to note here that during the replay of the agent as part of the verification process, any outputs directed from the replayed agent to external agents on remote hosts are merely recorded in the replay trace. The verification server conducting the replay does not relay these messages onwards to their intended destinations, as this should have been accomplished earlier by the host executing the original agent. It is the responsibility of trace reconciliation to ascertain whether agent communications had actually been undertaken correctly by the mediating host platforms. In this sense, trace reconciliation and trace verification can be considered as independent, orthogonal activities that address two different security aspects with respect to secure agent execution.

In order to prevent errors from propagating to other agents as a result of tampered communications, we can impose an additional restriction of requiring that an agent's input trace be reconciled in its entirety before the agent is permitted to migrate onwards to its next destination. As an example, consider verification server V_A responsible for verifying the execution of agent a. Assume that the replay trace for a is verified and the trace reconciliation using the output trace for a is successful. This success would indicate that the outputs from a to remote agents have not been tampered with by the mediating hosts. It does not, however, indicate anything about the validity of the inputs into a. To take that into account, V_A could elect to wait for the remote verification servers responsible for all the various external agents that have provided inputs into a, to engage in the trace reconciliation protocol with it. Only when these interactions are complete, will a be permitted to migrate to its next destination host. In this way, errors arising from tampered communications can be guaranteed to be limited to the pair of agents directly involved as well as the hosts that were responsible for executing them.

6.4.3 Trace reconciliation protocol scenario

We consider a situation in Fig. 6.12 involving two verification servers, V_A and V_B, which are responsible for verifying the executions of two agents on hosts A and B respectively. If both these agents communicate with each other during the course of their execution, then the process of trace reconciliation will require interaction between both V_A and V_B. Arrows a and b represent the submission of the input and output traces from the hosts to their respective verification servers, which as we

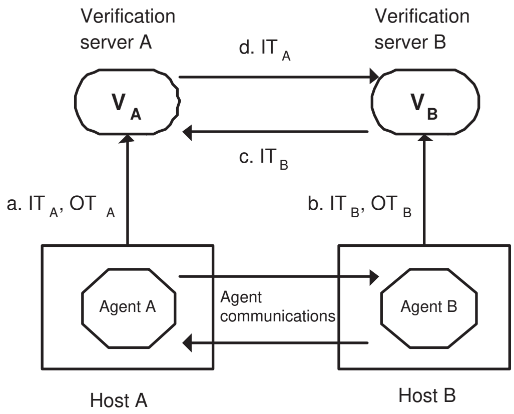

Figure 6.12: Interaction protocol in trace reconciliation

have seen (Fig. 6.4), occurs at m_7 during an interaction protocol run involving the host and its associated verification server. When V_A performs trace reconciliation using the output trace just submitted, it will receive the corresponding input trace in m_2 of the protocol that we have just detailed. This is represented by c in Fig. 6.12, while d represents the equivalent input trace submission from the viewpoint of V_B.

In a sequencing of events, c is dependent on b, while d requires a. However, the temporal order of c or d (i.e. which occurs first) during an interleaving of the trace reconciliation protocol runs of V_A and V_B is of no consequence as they are independent of each other. This means that a deadlock cannot occur as there are no cyclical dependencies in the submission of input traces between both verification servers. However, there may be a possibility of deadlock occurring in the agent executions on A and B due to the way that communication mechanisms have been structured in these agents. In such a situation, the agents will not be able to complete execution and hence no submission of traces (a or b) occurs. The timeout mechanism that exists in the original interaction protocol (m_5 in Fig. 6.4) will flag this event as a possible denial of service attack, resulting in the verification process being terminated on both servers. Correspondingly, there is no longer any need for trace reconciliation to be performed.

6.4.4 Modelling the protocol in Casper

We proceed to model the protocol described previously in Casper for purposes of verification in the same manner that we have done in section 6.3 for the original interaction protocol. For the free variables section, we have declared the following variable types:

```
#Free variables
BadHost : Agent
VB, VX, BadHost : Server
nvb, nvx, nb : Nonce
aid, aidb : AgentIdentifier
it, itb : InputTrace
PK : Agent -> PublicKey
SK : Agent -> SecretKey
SSK : Server -> ServerSecretKey
SPK : Server -> ServerPublicKey
hash : HashFunction
InverseKeys = (PK,SK), (SSK, SPK)
```

Once again, we distinguish between the public and private keys of host platforms and verification servers as we regard them as two distinct classes of entities in our protocol. *aid* represents the agent identifier, and is similar to Ic in the original protocol. There are two processes representing V_B (SERVERINITIATOR) and V_X (SERVERRESPONDER) respectively of Fig. 6.4. Both entities will have knowledge of their respective nonces and secret keys and will be able to access the public keys of each other.

```
#Processes

SERVERINITIATOR(VB, nva) knows SSK(VB), SPK, PK
SERVERRESPONDER(VX, nvb) knows SSK(VX), SPK, PK
```

The protocol is modelled below, where lines 1 – 3 correspond to m_1 – m_3 of the protocol. No time stamps or certification is required (as was in the case of the original interaction protocol) as the current interaction is between two trusted verification servers. All three steps are self explanatory with regards to m_1 – m_3 respectively.

```
#Protocol description

0.    -> VB : VX
0b.  -> VX : VB
1.  VB -> VX : {{VB, VX, nvb, aid}{SSK(VB)}}{PK(B)}
2.  VX -> VB : {{VX, VB, nvb, nvx,  aid, it}{SSK(VX)}}{PK(A)}
3.  VB -> VX : {{VB, VX, nvx, aid}{SSK(VB)}}{PK(A)}
```

As was the case for the Casper model for the original interaction protocol, we assume that the potential intruder here will be capable of creating its own input traces, nonces as well as accessing the public keys and identities of all entities in the system.

```
#Intruder Information

Intruder = BadHost
IntruderKnowledge = {VB, VX, BadHost, nb, aidb, itb, PK, SPK, SK(BadHost)}
```

6.4.5 Specifying security properties

The most important property here will be mutual authentication. Non-repudiation is not essential as we can assume that the verification servers are trusted entities that would not undertake an illegal activity intentionally and then deny liability afterwards. However, we take into account this property so that both interaction protocols we develop share identical security properties. Mutual authentication between V_B and V_X can be said to occur if they agree on the values nva, nvb, aid, where nva and nvb provide reference to the current protocol run.

```
Agreement(VB,VX, [nva, nvb, aid])
Agreement(VX,VB, [nva, nvb,aid])
```

We use the secrecy property to show that non-repudiation for IT and aid is achieved between V_B and V_X.

```
Secret(VB, aid, it [VX])
Secret(VX, nvb, [VB])
```

Both specifications for mutual authentication and secrecy were again satisfied in the resulting CSP model that was checked using FDR. We note that the additional three messages that consitute this additional protocol could have been integrated into the original protocol after m_7 (Fig. 6.5). It is developed separately because the interaction to obtain input traces for reconciliation can occur in conjunction with or after the process of trace verification, and may overlap the protocol run for a new, incoming agent. For example, V_B may accept a new agent from V_A (m_5, m_6) while using the additional interaction protocol to obtain the required input traces to reconciliate the outputs of the current agent that it is verifying.

6.5 Limitations of modelling in Casper and FDR

The use of Casper to model security protocols for mobile agent systems has been attempted previously by Hannotin in [60]. In this work, there is an attempt to verify the property of data integrity in a data encapsulation protocol [30] that intends to safeguard data accumulated by a mobile agent during its itinerary from invalid tampering. The protocol functions by making tampering of this data by a malicious host (for example: modification or truncation of previous offers) detectable by either the home platform of the agent or the next honest host that the agent migrates to. Although this property is verified in the CSP model that Hannotin [60] develops, the same protocol (as well as other protocols with a similar motivation of protecting accumulated data) was shown to be vulnerable to a certain type of attack described by Roth in [121], which has a general two-step approach:

1. Protocol data from an honest mobile agent is cut and pasted on to a 'dummy' mobile agent generated by a hostile host. This mobile agent is then launched to another honest host with which it interacts in a certain manner to acquire critical information about the current protocol run (which it would not normally be able to acquire without the use of the 'stolen' protocol data).
2. The 'dummy' agent migrates back to the hostile host with this information, which is then used by the host in some way to change the accumulated data of the honest agent. This change will subsequently be undetectable when the honest agent is migrated on to the next host or back to its home.

The strategy of this attack is not new and is similiar in motivation to an earlier well-known attack on the Needham-Schroeder public key protocol described by Lowe [88]. In his attack, a replayed message from a previous protocol run is used by an intruder to initiate a new protocol run in which an unsuspecting participant is then abused as an unwitting oracle to reveal confidential information. This information can then be used to compromise the integrity of a communication channel. Roth's technique is essentially the same with the primary difference being that a new mobile agent (instead of a replayed message) is used by a malicious host in a new protocol run to initiate the oracle attack. In order to nullify the attack, it is necessary to prevent one or both of these steps from occurring. Roth presents a method to prevent the first step by using authentication to uniquely associate the identity of an agent instance along with the protocol data transported by it.

This would allow an honest host to discern whether an incoming agent is carrying protocol data that belongs to it or that was 'stolen' from another agent. The host could then refuse to accept or execute agents carrying 'stolen' data, thus preventing itself from being abused as an oracle. This security measure is in actual fact a form of host security, and we have here an interesting illustration of how the two different aspects of code and host security (which often appear to be orthogonal to each other) can be actually closely interlinked.

The main reason underlying the failure of Hannotin's Casper model to detect such attacks is the inability to model a mobile agent accurately using traditional cryptographic protocol analysis methods. In those methods, a fundamental assumption in analysis is that the format of messages exchanged between static entities and the sequence in which they occur within a single protocol run are predefined and remain fixed throughout the duration of the protocol run. Thus, attacks can only occur through judicious interleaving, reflection or replay of messages from different protocol runs.

In Hannotin's approach (and our approach as well), the mobile agent is implicitly treated as a unique, static portion of a message which permits reasonably straightforward modelling. Although messages are exchanged between hosts rather than agents in both approaches, there exists a possibility for the mobile agent (whose code and state would constitute part of these messages) to indirectly alter the sequence and contents of these messages through its interaction with the various hosts. For example, in the second step of the type of attack used by Roth, the data carried back by the 'dummy' agent to the hostile host has to be part of the specification of the protocol run, since this is the data that actually allows the hostile host to successfully carry out its attack. Obviously, the format and contents of this data cannot be predefined and will depend on the interaction of the agent with the honest host.

Since we also treat the mobile agent as a static message in our approach, our model is equally susceptible to the same vulnerabilities as Hannotin's. However, our trace exchange protocol for interaction tracing differs from the approach employed by [30] (as well as most other data encapsulation protocols) and the original execution tracing protocol in an important way: agent state (and code) is replicated. In our approach, the agent that is actually migrated on to the next host platform is the mobile agent copy on the verification server (the trusted platform), and not the actual agent on the current host. As such, the current host will not be able to directly manipulate the state of the mobile agent, in effect nullifying step two of Roth's attack. The only way a hostile host can affect the state of the agent copy is through the trace it supplies; this however will also contain the signature of the host to act as a measure of non-repudiation. Supplying faulty traces as a form of attack is thus meaningless as liability can eventually be established for the resulting problems that arise (this, of course, is based on the assumption that the economic cost of being sanctioned for an attack is greater than the economic cost resulting from the attack). Therefore, our only concern is ensuring that traces, agent code and agent state are securely propagated in our system, and that traces are correctly associated with the corresponding agents. Mobile agent behaviour is subsequently of no concern to us any longer. We are therefore justified in using Casper to model our protocol as the nature of our protocol is now analogous to those modelled in standard cryptographic protocols.

6.6 Conclusion

In this chapter, we introduce the concept of a trusted third party, the verification server, that undertakes the activity of trace verification on behalf of the agent owner platform. A protocol for exchanging traces involving this verification server is presented and contrasted against the interaction protocol of the execution tracing approach. The sequence of messages for the new protocol is described in detail, and is then modelled in CSP using the high-level security protocol description language, Casper. The model is then analysed in FDR to determine whether specific security specifications are valid. Reconciliation of input and output traces in order to protect inter agent communication in a mobile multi agent system is then explained. Modified operational semantic models that incorporate the concept of an agent identifier are presented, and an algorithm that performs the reconciliation of the traces is presented. The message sequence protocol that allows the reconciliation of traces is described, modelled and verified in a similar manner to the trace exchange protocol. Finally, we discuss the limitations of modelling the protocol using Casper and finite state model checkers such as FDR.

As we have noted, one of the advantages of having verification performed after execution on every single host is to provide some form of defence against the denial of service attacks that was highlighted in section 3.6.3. There are few code security techniques that address this problem directly. The more prominent ones which do are the replicated agents approach [103] or co-operating agents [120]. In [103], replicated agents are executed on different hosts and simple voting is used to determine the outcome of computational results. This approach is extended in [120], where other strategies such as secret sharing, remote authorization or remote storage of commitments can be used as part of protocols involving two co-operating (but not necessarily identical) agents that communicate with each other while migrating in different host platform domains. Some of the criticisms regarding these approaches are that they require replication of services on all host platforms and may fail if the number of malicious hosts outnumbers the honest ones (for the case of replicated agents). Co-operating agents appear to be more feasible but require that a specific co-operating agent and associated protocol be created for each application scenario, thus making it difficult to use for generic mobile agents.

Our interaction protocol demonstrates some similarities to both of these approaches. Since it employs replication of agent code and state, it is similar in motivation to the replicated agents approach. However, by imposing the trusted third party concept (i.e. the verification server is assumed to be a trustworthy entity that would not willingly collaborate with other hostile parties), we eliminate the need for replication of hosts as well as the possibility of failure that may arise when the number of malicious hosts outnumber honest ones in a voting scheme.

With regards to the co-operating agents approach, the agent copy forwarded to verification servers for purposes of verifying the actual agent that migrates along a separate itinerary of host platforms can be regarded as a 'co-operating' agent that helps to detect tampering of the actual agent. However, our interaction protocol offers the additional advantage of fault tolerance. A co-operating agent is not a replica of the actual agent to be protected, rather it is an agent that is specifically designed to support the actual agent in specific scenarios via constant communication so that it is immediately aware whenever the actual agent is compromised. In such an event however, it can only note or report the compromise but is incapable of continuing the compromised agent's agenda on its own. In our protocol however, if a verification server detects tampering in a trace, it can signal an exception to the agent copy executing in its environment so that suitable action can be taken.

Chapter 7

Evaluation of Protocol Implementation

In the previous chapter, we outlined an interaction protocol for the secure exchange of traces and its subsequent verification using Casper and FDR. In this chapter, we describe a practical implementation of the protocol in a mobile agent system and conduct an evaluation of it. To date, virtually all quantitative performance evaluations of mobile agent systems have been targeted towards measuring how well they perform or scale in comparison to other existing methodologies such as client-server setups in specific application scenarios ([55], [34], [59]). However, as we have already seen in section 3.4, security is often an important consideration in some applications and security generally implies additional performance costs (cryptographic primitives and protocols must be used). These costs may possibly offset the performance gains achieved through the use of a non-secure mobile agent system. In this respect, a quantitative performance evaluation of a secure mobile agent system in comparison to a non-secure one would be useful, and this chapter details an effort towards this direction in the field of code security. The focus is however on developing a proof of concept agent system which incorporates the secure trace exchange protocol, and does not incorporate all the facilities that may be found in more complete mobile agent systems.

This primary contributions of this chapter towards the thesis are:

1. A practical implementation of the protocol for secure exchange of traces and their verification is constructed;
2. A quantitative evaluation of the implementation with regards to the cost

(in time) incurred for a sample mobile agent system utilizing the security protocol against one that does not is performed.

In section 7.1, we examine the details of the implementation in a mobile agent framework. The evaluation is discussed in section 7.2, and we conclude in section 7.3.

7.1 Implementation of protocol

We have implemented the security protocol in SoFAR [106], a mobile agent system developed at the University of Southampton. In this section, we examine the key features of SoFAR and the reason for its use as a mobile agent platform to implement the protocol. This is followed by a discussion of some of the relevant features of the implementation itself.

7.1.1 SoFAR mobile agent system

SoFAR was initially developed with the motivation of providing a framework for coordinating the activities of agents involved in distributed information management [33]. Communications are declarative and possess a context, which includes information such as the sender, receiver, send time, message identifier and conversation thread. It also incorporates the notion of an 'agent communication language', derived from standards such as KQML and FIPA. This is implemented through structured agent communications that are classified according to a predefined and limited set of message categorizations referred to as performatives.

An abstract communication model is used that is based on the concept of a startpoint and endpoint to implement communications between agents. An endpoint extracts messages from a communication link and passes them onto the receiving agent. A startpoint is the other end of the communication link, from which messages get sent to an endpoint. Communications are initiated by invoking one of the performatives on a startpoint; this results in the transfer of the data to the remote endpoint. There may be several startpoints for a given agent, each acting as a representative of the agent at remote locations. Each startpoint is associated with an endpoint by a virtual link, and encapsulates a specific communication protocol. There are two main methods of communicating. In the location-aware mode, messages are sent to a location and will fail if the agent at the location has migrated; whereas in the location-transparent mode, messages are routed transparently to the agents based on a distributed algorithm [105] that does not assume the existence of any fixed or centralized control. Ontologies have also been developed for mobility related concepts, particularly to deal with ad-hoc meetings of users.

Host security is provided by allocating permissions via the policy manager in Java to code sources which are composed of the codebase (the URLs where mobile code is downloaded from) and signature of classes; when a platform receives a request for an incoming migrating agent, it will only restart the agent if it has the permission to be executed for a specific user. Access control is achieved via user identification in combination with the stack inspection mechanism in Java.

In order to implement interaction tracing effectively, a mobile agent system that provides a well defined interface through which an agent can interact (or communicate) with its external environment is necessary. Many of the more well known mobile agent systems (e.g. NOMADS [137], D'Agents [54]) require that the programmer construct from scratch the communication mechanisms to allow an agent to communicate with another agent on a remote host. SoFAR in this respect has the following advantages:

- It employs a structured and limited set of messages (performatives) through which communication is achieved. These performatives can be easily mapped to the input and output commands of the operational semantic models that were developed in Chapter 5.
- The use of a startpoint/endpoint paradigm simplifies communication between agents located on different host platforms. In the location transparent mode, messages are routed by the system without agents being aware of the actual host location of the agent that they are sending a message to. Such a communication mechanism fits in very well with the inter host communication model that we develop in Chapter 6 that assumes only the need for a unique agent identifier in order to route a message to an agent.

7.1.2 Details of implementation

From the viewpoint of interaction tracing, an agent in SoFAR will only be able to interact with its environment through these predefined performatives. Therefore, a trace of an agent's execution would simply consist of an ordered sequence of all performatives invoked by or on an agent. As far as mobility is concerned, SoFAR is Java-based and supports weak mobility. Code migration is achieved using RMI's ability to load classes dynamically when receiving the serialised state of the agent.

Tracing and replay of the mobile agent is achieved through the use of startpoints. Every time a performative is invoked on a startpoint object in the agent code, a record is made of the parameters passed, the results returned, the type of performative invoked, as well as other relevant information (we shall term this record as a performative identifier). A trace of an agent's activity is thus the ordered sequence of all performative identifiers in all startpoint objects utilized by that agent when communicating with its external environment.

When an agent requests to migrate, the agent's traces are automatically submitted to the verification server, using the *inform* performative. On the verification server, new copies of startpoint objects are instantiated using the information obtained from the performative identifiers in the trace. Once this is complete, the mobile agent copy is restarted with the same initial state as the original agent on the host platform. Each time a performative is invoked on a startpoint object in the agent code, a check is made to ensure that the sequence, type and parameters passed are correct. A mismatch indicates a problem with the trace and an exception will be thrown. Otherwise, the results are passed back immediately from the startpoint object, without the need for the framework to relay any message onwards to the corresponding endpoint (as would be the case for the normal version of the framework installed on the host and client platforms).

In effect, the framework on the verification platform is engineered so as to simulate the same external environment interaction (i.e. providing the same set of startpoints) that the agent would acquire on the host platform. As far the agent is concerned, it is unable to distinguish whether it is currently executing on the host or verification platform. This imparts the important advantage of transparency in the process of agent composition, as the code composer need not 'reinstrument' the original agent code to produce a usable version for verification. The need for identical agent code to be executed on both the verification server and host can be seen from section 6.1.1, where the interaction protocol requires that the agent copy be forwarded to both a verification server and host platform for the next stage in the agents itinerary.

If the agent run completes successfully without any exception occurring, the request to migrate this time will now be relayed onwards to the runtime classes of the framework. Agent state is then serialized and passed onwards to the new host platform specified in the migrate request as well as to the next verification server responsible for verifying this host platform. As far as agent behaviour is concerned, incorrect performatives that arise from invalid tampering of its execution flow or state by a malicious host would manifest as an exception occurring during the invocation of a performative during the replay process at the verification server. This exception provides the ability for the agent itself to decide (through appropriate exception handling code) on the appropriate manner of responding to the host platform's attempt to corrupt it, instead of delegating this ability to the verification server. Possible corrective action could include dynamically altering its itinerary or logging the identity of the malicious host. Since exception handling code will be executed on a trusted domain (the verification server), the integrity of the agent in responding to such situations is guaranteed.

The various cryptographic primitives (encryption, digital signatures and hashes) used in the protocol are implemented using the standard security packages in Java [111]. The host and verification servers are provided with the certificates and keys that they require in the interaction protocol, and are aware of their relationship to each other prior to the start of evaluation (i.e. each host is aware of the verification server that is responsible for verifying its traces and vice versa). In order to make the process of implementation more tractable, we do not implement any additional host security mechanisms other than those existing in SoFAR, which rely on the Java security policy manager. In addition, the evaluation will only examine the costs of valid execution on the host platforms, i.e. the hosts are set up so that illegal tampering (and hence, failure of the verification process) does not occur.

7.2 Evaluating the implementation

In this section, we evaluate the relative costs of using three different approaches quantitatively: client-server, mobile agent and mobile agent with the interaction protocol. The evaluation is conducted in the context of an information retrieval task, and is divided into two main parts: the first deals with all three different approaches without the need to consider reconciliation of traces, and the second part takes into account this requirement. The task to be implemented involves retrieving a set number of strings from a database (which may be located locally or remotely, depending on the approach employed) and then searching on these strings to obtain words that match specific criteria. This task is a very basic version of the type of computational task that might be used in an agent-based information retrieval application, and was used (albeit in a more complicated form) to evaluate the performance of mobile-agent against client-server systems in a recent study [55].

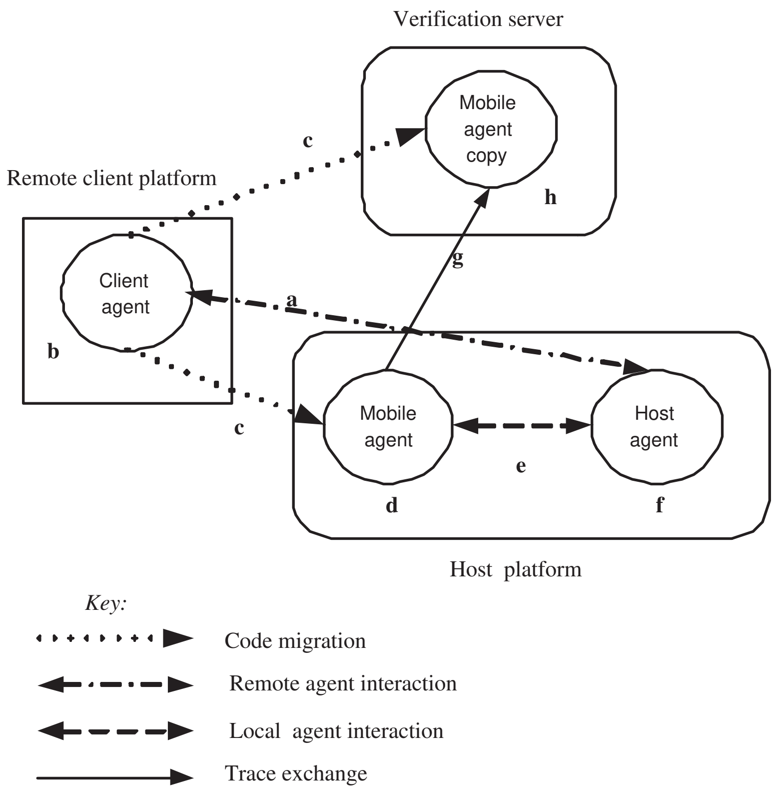

Figure 7.1: Cost components without trace reconciliation

7.2.1 Cost components of secure protocol

In implementing the task, an agent-oriented engineering approach is used which allows the task to be effectively described as a series of interactions between different agent entities in the system (Fig. 7.1). We consider a single hop agent, i.e. an agent that only visits one host before returning to its home platform, and therefore only consider a single host platform and verification server. This can be extended to the case of an n-hop agent with n host platforms and verification servers. The host agent is effectively a proxy for the host platform to which the mobile agent channels requests for resources in its execution environment; in our evaluation, these resources are strings from text documents in the database. The search functionality scans the strings to locate words that meet specific criteria and is implemented in its entirety by the client agent. The mobile agent is simply a mobile version of the client agent and implements the same search functionality. All mobile agents in the evaluation are structured so that their execution is single threaded.

In the case of a client-server deployment, the client agent directly interacts with the host agent in order to obtain the records it requires. A mobile agent deployment would involve migrating the client agent to the host platform where it would interact locally with the host platform. In the event that the interaction protocol is employed to support interaction tracing, an additional step of submitting traces to the verification server is undertaken prior to migration to the next host. We consider an evaluation for the interaction protocol that does not involve the reconciliation of traces in the first instance. The different cost components involved in these three approaches (Fig. 7.1) are:

1. a - Network traffic involving remote messages and strings exchanged between host and client agent.
2. b - Execution of client agent code to implement the search functionality on the retrieved strings.
3. c - Computational overhead of a migration (serialization and de-serialization, network protocol and reinstating an agent) as well as network transfer time. This includes migration in both directions, as the agent needs to return to the client platform upon completion of processing at the host platform.
4. d - Execution of mobile agent code to implement the search functionality on the retrieved strings. This also includes the cost of accumulating trace information pertaining to the requests sent by the mobile agent.
5. e - Local messages and strings exchanged between host agent and mobile agent.
6. f - Database access by host agent code in response to requests from the mobile agent or remote client agent.
7. g - Transmission of trace information to the verification server.
8. h - Replaying the mobile agent copy with the trace information as part of the verification process.

For a basic client-server approach, the cost involved in completing a single agent run would be from components a, b and f. A mobile agent approach would involve the cost components c, d, e and f. The cost savings traditionally associated with the use of mobile agents in such a scenario would then rely on a considerably outweighing c, e and the difference between b and d. Employing verification to support secure agent execution requires the cost of g and h in addition to those incurred by a normal mobile agent.

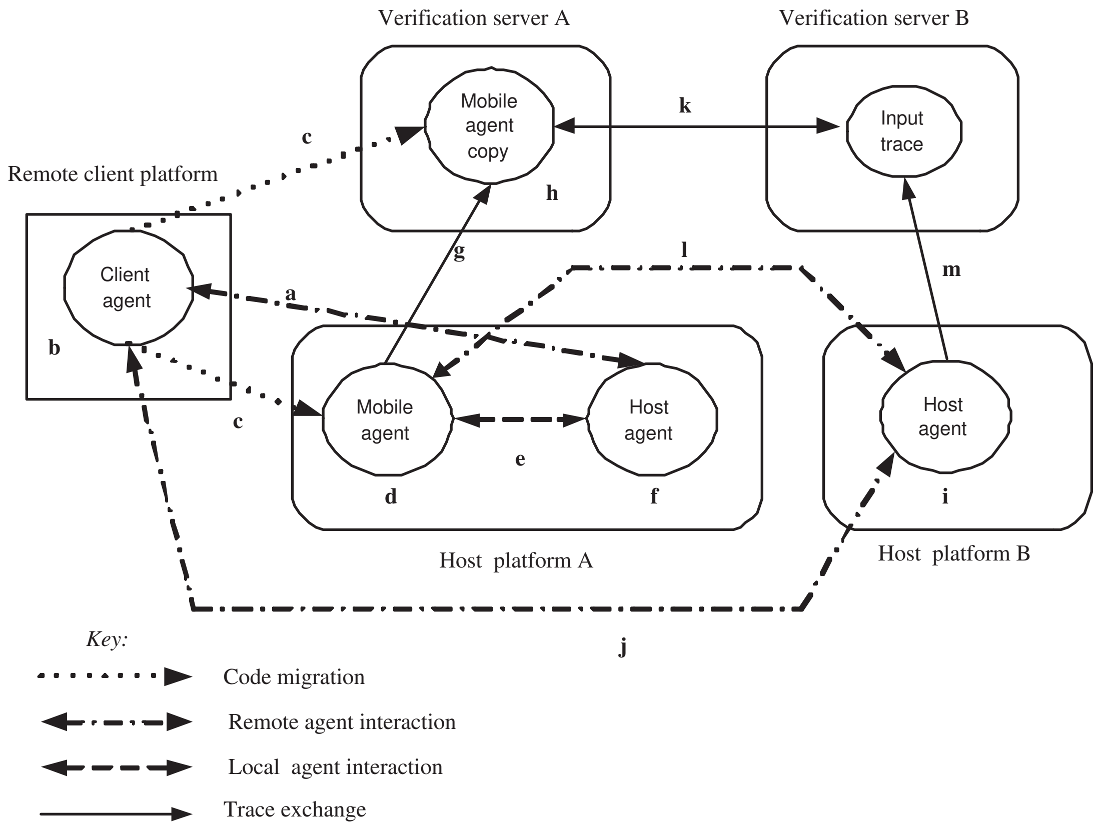

Figure 7.2: Cost components with trace reconciliation

For the second part of evaluation, we study the effect of the additional cost of trace reconciliation (Fig. 7.2). A second host platform B is introduced with its own host agent that processes the results from the search function performed earlier either by the client or mobile agent. For the client-server setting, the client agent will pass these results to the host agent at the completion of its search function. The same applies for the mobile agent setting as well, with the addition that platform B will generate an input trace for its host agent during this interaction and submit it to verification server B. The trace reconciliation will then involve verification server A obtaining the input trace for the host agent from verification server B and comparing it against the output trace from the mobile agent. The additional cost components are:

1. h - This component will now not only include the cost of replaying and verifying traces, it also encompasses the cost of trace reconciliation.
2. i - Execution of host agent code to process results sent by either the mobile agent or the client agent.
3. j - Network traffic involving results sent to the host agent from the client agent.
4. k - Network traffic for interaction between verification server A and B in order to obtain the input traces for the trace reconciliation process.
5. l - Network traffic involving results sent to the host agent from the mobile agent.
6. m - Transmission of trace information to the verification server.

Here, the extra costs involved in the client server approach would be component j which is analogous to component l involved in the normal mobile agent approach. Including verification incurs the additional cost of k for interaction between verification servers and h for the reconciliation of traces. The cost savings, if any, that would result from the use of a secure mobile agent system with trace reconciliation then rely on j outweighing the combination of h, k and l.

7.2.2 Results and discussion

The performance evaluation being conducted is a comparative one (i.e. communications between agents in all three approaches are implemented using performatives); as such, we have not considered the individual costs of communication (achieved through performative invocation) and migration in SoFAR. We measure the overall cost of completing the search task (Fig. 7.1) for the three cases: client-server (components a, b, f), mobile agent alone (components c, d, e, f), and mobile agent with verification (components c, d, e, f, g, h) against the number of performatives used to obtain strings from the host agent. For the case of the interaction protocol without trace reconciliation, the performative used to communicate and obtain strings is component a for the client server approach and component e for the mobile agent approach with and without verification. There are two ranges

$$\begin{aligned}
Total\ cost\ with\ client-server\ approach &= b\ +\ a\ +\ f \\
Total\ cost\ with\ mobile\ agent\ approach &= c\ +\ d\ +\ e\ +f \\
Cost\ savings\ from\ local\ agent\ interactions &= a-e \\
Total\ cost\ savings\ without\ verification &= (b+a+f)\ -\ (c+d+e+f) \\
&= (a-e)-(d-b)-c \\
&\simeq (a-e)-c \\
Total\ cost\ savings\ with\ verification &\simeq (a-e)\ -\ c\ -\ (g+h)
\end{aligned}$$

Figure 7.3: Total costs of the information retrieval task

for the number of performatives involved in the evaluation: the first measures the overall cost to complete the search task in the three approaches for runs involving between 5 to 55 performatives (Fig. 7.4), the second measures the overall cost for runs involving between 100 to 1000 performatives (Fig. 7.5). The average of 10 repetitions was used for each cost measurement, and a standard deviation of 1.34 and 2.36 was obtained for the first and second plots respectively. In all cases, we only deploy a single mobile/client agent to interact with a single host agent. All machines used in the evaluation have the same specifications (Pentium III, 256 MB, Windows NT, 100 Mbps LAN) and SoFAR was deployed using the Java 1.2 JVM.

The main hypothesis that we wish to consider from the results obtained is the proposition that the additional cost of security still justifies the use of mobile agents in an information retrieval task. However, we believe that the point at which this cost justification becomes valid is subsequently higher for the case of secure mobile agent deployment in comparison to a non-secured one. The cost savings calculations is shown in Fig. 7.3. The functionality implemented by the static agent is a simple linear search on n strings to extract words that meet a given criteria. The computational cost of the search (measured either by b or d) for an increasing n is negligible in comparison to the cost of network communication (a), even for small values of n. Thus $(a-e) \gg d$, $(a-e) \gg b$ and consequently, $(a-e) \gg (d-b)$, and the cost savings reduces to $(a-e)-c$ (Fig. 7.3). The cost of c remains relatively constant regardless of the size of n, since the mobile agent only incorporates the extracted words from its search in its internal state when it migrates back to the client platform, while $(a-e)$ increases steadily with increasing n. As such, there will be a break-even point when $(a-e) \simeq c$, after which the use of the mobile approach becomes justifiable in terms of cost compared to a static agent. From Fig. 7.4, this occurs around 22 performatives.

For the case of mobile agent with verification, the additional cost from verifica-

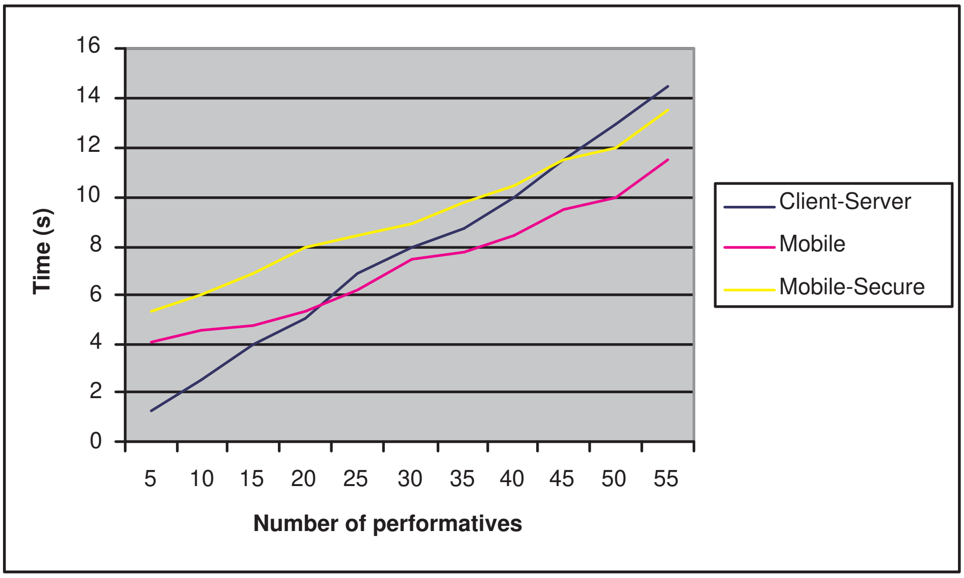

Figure 7.4: Cost between 5 and 55 performatives

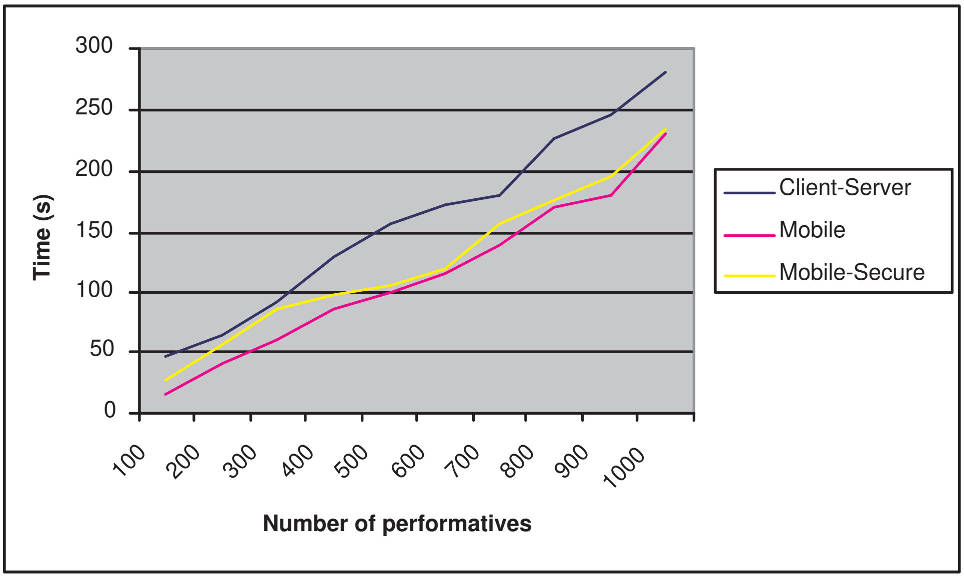

Figure 7.5: Cost between 100 and 1000 performatives

tion comes from components g and h, and hence any cost savings achievable now reduces to become $(a - e) - c - g - h$ (Fig. 7.3). Since performative invocations in the replayed agent now only cost as much local method invocations (the results are returned to the agent copy using the trace information instantiated in the start-point objects) and the functionality of the agent copy is identical to the original, h is again negligible in comparison to the cost of performatives. Unlike c, g increases linearly with n as the size of the submitted input trace is also linear with respect to n. However, its increase with an increasing n is not as rapid as that of $(a-e)$. The difference between the increase rates is due to the communication mechanisms in SoFAR, where the cost of performing n remote communications containing a single item (as is the case for a) is significantly greater than performing a single remote communication containing n items (for the case of g, these items would be the values on the input trace being submitted). Therefore, eventually another break even point is achieved when $(a-e) \simeq (c+g)$, which occurs at around 45 performatives. As expected, for a high number of performatives, the cost savings from local interactions, $(a-e)$, becomes so large that the difference between mobility with and without tracing becomes minimal (Fig. 7.5).

Our evaluation presents a scenario where agents are largely I/O bound, thus the cost savings are very evident due to the high cost of performatives. For situations where agents impose a high computation load, the use of mobile agents might still be justifiable but the same is unlikely to apply for the verification and replay process. Consider a situation where the computation of the client agent, b, consumes as much time as the savings achieved from local interactions, $(a-e)$, and that b, d and h are roughly equivalent. In that instance, the cost savings associated with basic mobility, $(a-e)-(d-b)-c$, might still be justifiable because $(d-b)$ would be nearly negligible and the cost still reduces down to $(a-e)-c$. With verification included, the cost savings becomes $(a-e)-c-g-h$, as we noted earlier. Since $(a-e)$ is roughly equivalent to h, there is no savings achieved and in fact the situation is clearly more expensive than the client-server setting.

For the case of interaction tracing with trace reconciliation, the average of 10 repetitions were plotted and the standard deviation for the plots in Fig. 7.6 and Fig. 7.7 are 3.4 and 5.6 respectively. The performatives used to communicate and obtain strings are components a and j for the client-server approach, and e and l (for the mobile approach with and without verification). The overall cost of completing the search task (Fig. 7.2) for the three cases are: client-server (components a, b, f, i, j), mobile agent alone (components c, d, e, f, i, l), and mobile agent with verification (components c, d, e, f, g, h, i, k, l). We assume that component m will proceed nearly in parallel with g, and therefore do not take it into account in our serial composition of costs. This assumption is valid for our evaluation scenario, since host platform B (with only a single host agent) is less loaded than host platform A, and consequently will submit its required traces as at least as fast as A can. In a practical mobile multi-agent situation however, it is very likely that m will be variable due to differing loads on host platform B as

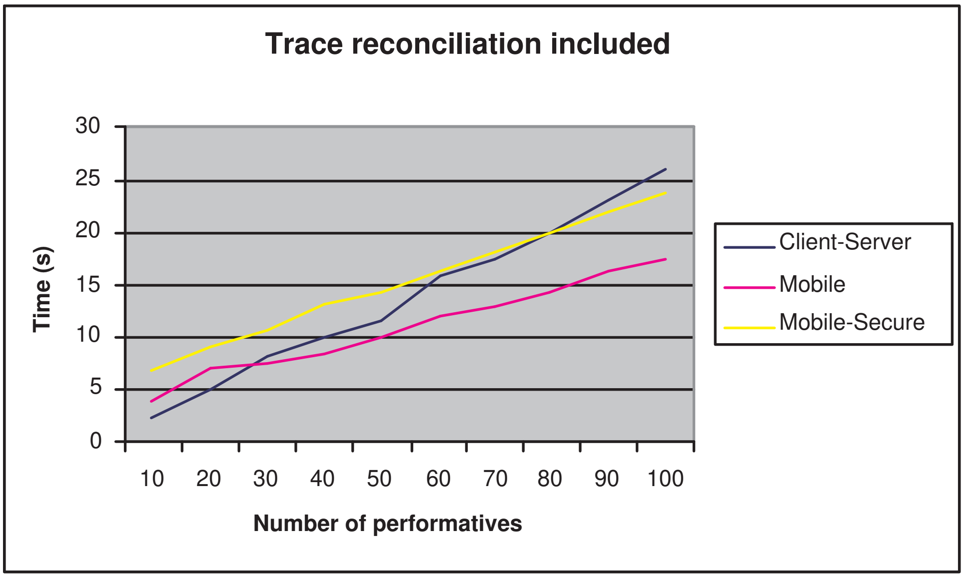

Figure 7.6: Cost between 10 and 100 performatives

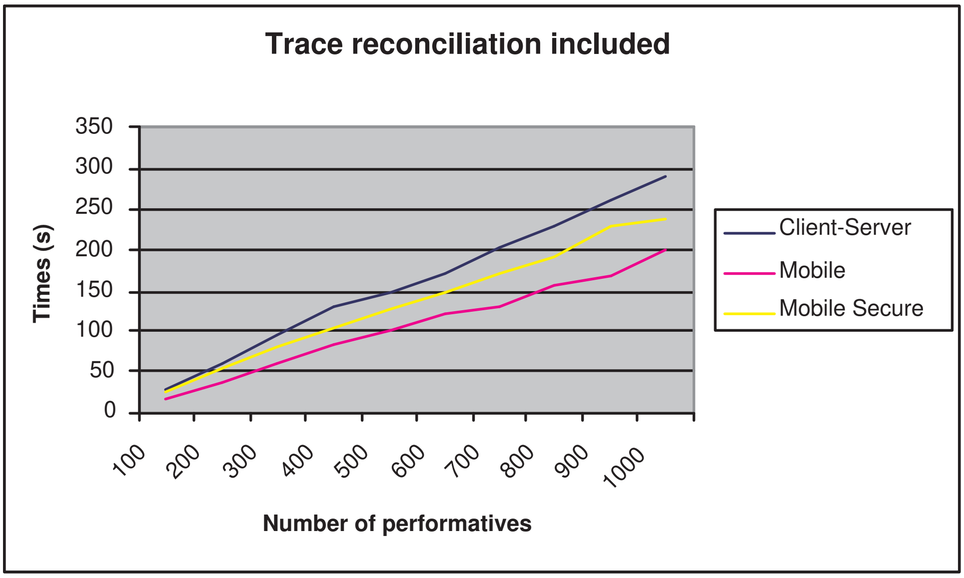

Figure 7.7: Cost between 100 and 1000 performatives

well as potential network latencies.

We have seen earlier in the case of the evaluation without trace reconciliation that the computational cost of search for an increasing n is negligible in comparison to the cost of communication, and the cost savings from the use of a mobile agent reduces to $(a - e) - c$. With trace reconciliation, the additional interaction with the host agent on host platform B requires j for the client-server case and l for the mobile agent case. If we assume that these two components are roughly equal (since they are both network interactions involving essentially the same data), then the cost savings remains the same. The break-even point (when the use of the mobile approach becomes justifiable in terms of cost compared to a static agent) is thus the same as in the previous case (25 performatives).

For the case of mobile agent with verification, the additional cost from verification and reconciliation comes from components g, h and k, and hence any cost savings achievable now reduces to become $(a - e) - c - g - h - k$. The cost of h now includes not only the replay process of verification, but the comparison of input and output traces as part of trace reconciliation. The latter activity merely involves a sequential comparison of values from both traces and is virtually negligible in comparison to the time required for the replay process. As is the case for evaluation without trace reconciliation, we can consider h again to be negligible in comparison to the cost of performatives. Like g, k is achieved through a single performative act which returns the required input traces. We now have a roughly constant cost for c, and both g and k increasing linearly with n (since the size of the input traces are also linear with respect to n). The increase of the combined cost of g and k is not as rapid as that of $(a - e)$, and so eventually another break even point is achieved when the cost of verification is also justified (this occurs at around 80 performatives). For a higher number of performatives, $(a - e)$ becomes so large that the difference between mobility with and without tracing becomes insignificant (Fig. 7.7).

7.3 Conclusion

In this chapter, we outline a practical implementation of the protocol for secure exchange of traces and their verification. A quantitative evaluation of this implementation is then conducted to gauge the cost effectiveness of using a mobile agent system with this approach in contrast to a standard mobile agent system and the traditional client-server approach. The results indicate tentatively that the additional cost of the security protocol is justifiable in that the additional overheads that it adds to a non-mobile agent system are eventually outweighed by the cost-savings in network traffic that the mobile agent paradigm is traditionally associated with.

There has been some work in studying the cost advantages of using mobile agent systems as contrasted to client-server architectures. In [59], RMI and Aglets [75] are used in a comparative evaluation of mobile agent system against a client-server model, while [55] examines the scalability of the mobile agent and client-server approaches in an information retrieval scenario, using the DAgents [54], EMAA [97], KAoS [19] and NOMADS [137] mobile agent platforms. A similarly motivated evaluation (client-server vs. mobile agent) was also undertaken in the network management application arena [10]. In all these evaluations however, the issues of host/code security and their associated costs were not taken into account.

More recent work [122] investigated different techniques of enhancing encapsulation protocols to protect accumulated data on mobile agents (section 3.3.1) as well as measured the migration time of an agent with a varying payload. The evaluation examined the additional costs to migration resulting from the need to execute cryptographic primitives (such as encryption and signatures) that were relevant to the encapsulation protocols used to secure the mobile agent. However, there was no attempt to contrast the combined costs of mobility and security against a traditional client-server approach within the context of a specific application. In this respect, our work presents an initial effort towards the goal of combining the comparison of three relevant scenarios: client-server, mobile agent without and mobile agent with supporting security mechanisms.

It would be interesting to consider the possibilities for performing verification in other ways which could further reduce the cost of incorporating the security protocol. The approaches described below merit further investigation but are beyond the scope of this thesis.

Dynamic verification. In this approach, agent replay is started as soon as the mobile agent copy arrives at the verification platform, and its execution proceeds simultaneously with the original agent at the host platform. When a performative is invoked in the original agent, a copy of the results is simultaneously returned to the verification platform and relayed to the copy when it invokes the same performative. Appropriate synchronization mechanisms would be required to keep the execution of both copies on both platforms in step with each other. Although replay may not complete at the same time as the original agent on the other platform, h is significantly reduced. On the other hand, g is now correspondingly increased, since every local performative call at the host platform must now be accompanied by a remote performative call to the verification platform. However, if we can assume a more restricted bandwidth between the client and host platform than between the host and verification platform (i.e. $a \gg g$), the cost savings with verification is still justified. This assumption would certainly be valid for cases where the client platform represents a low-bandwidth mobile device, or when both host and verification platforms are within the same administrative domain through which external access (by the client platform) is through a bandwidth limited connection.

Speculative trace verification. In this approach, the mobile agent will submit its final state (prior to its migration request) as well as its traces to the verification server. The server then makes another copy of the current agent copy in its environment and forwards the new copy onwards to the next host platform in its itinerary (and corresponding verification server). Replay and verification of the agent copy is then started by the current server. Thus, we speculate that verification would indeed be successful and initiate the next hop in the migration path on the basis of this speculation. Such a speculative migration could occur right at the start of the replay process (which results in the largest cost savings), or once a critical section of agent code has been replayed successfully. In the best case situation, for an agent itinerary spanning a total of K host and verification platforms, overlapping migration and trace verification in this manner results in an overall reduction in the cost of verification from $h.K$ to h (verification on the last platform prior to return to the client platform). Of course, the problem with this approach is the difficulty of 'catching' up with and aborting a corrupted migrating agent in the event a malicious host is detected. The mobile agent run can however still be continued from the most recent valid agent copy on a verification platform that was visited by the agent.

Chapter 8

Trust in the Trace Exchange Protocol

In the last three chapters, we formalized the interaction tracing approach to code security, developed and tested a secure trace exchange protocol for supporting this approach and then performed a quantitative evaluation of it. In this chapter, we seek to investigate a way of augmenting the utility of this secure protocol by merging concepts from other areas of agent and security research. As we noted in section 3.6.1, there may be a future convergence of some areas in the multi-agent and mobile agent research communities resulting from the wider application of agents in e-commerce scenarios. Mobile agent research generally addresses lower level issues that are associated with aspects of mobility such as weak/strong migration, communication mechanisms, transport mechanisms and so on. Multi-agent research takes a higher-level approach by examining fundamentally useful ways through which agents can interact: for example, through commitments, negotiations and so on. One of the higher level concepts that influences the interaction of agents from the viewpoint of security is the concept of trust.

In this chapter, we show how trust can be incorporated into a mobile agent framework that uses the secure trace exchange protocol. This is achieved by introducing the concept of trust relationships that are derived from existing work in security for distributed systems. An algorithm is then developed to show how these relationships can be combined to form new relationships which extend the flexibility of the trace exchange protocol. The primary contributions of this chapter towards the thesis are:

1. Arguing for the use of trust to enhance code security for mobile agents and providing an analogy to the way the concept of trust is utilized in a conventional distributed security architecture such as a Public Key Infrastructure (PKI);

2. Adapting an existing PKI trust model for use with the trace exchange protocol;

3. Demonstrating how trust relationships are formed in this new model and how they can be combined to form new relationships, as well as the significance of these relationships in extending the flexibility of the trace exchange protocol.

In section 8.1, we provide a justification for incorporating trust in a mobile agent framework and demonstrate how a trust model for mobile agents might be developed using an analogous trust model for a PKI. We describe the various trust relationships between hosts, verification servers and agent owners in section 8.2, as well as show how new relationships can be derived from existing ones. section 8.3 presents the trust derivation algorithm and describes how it might be used. The work presented in this chapter has been previously published in [140].

8.1 Incorporating trust

There are many definitions for the concept of trust in multi-agent research ([104], [3]). An example of a widely used definition for trust is the one provided in [50] which defines trust as a particular level of the subjective probability with which an entity will perform a particular action before such action can be ascertained to have taken place. However, in the context of our work, we will employ a definition of trust that is binary. This fits in well with the nature of the verification process discussed in Chapter 5, which essentially is capable of only a binary outcome (i.e. either success or failure) for the equivalence of traces. It also provides a simple and direct way to understand trust interactions in a mobile agent system without delving too deeply into statistical formulations.

We note however that a more refined and developed trust model would probably include the notion of subjective probability in the formulation of trust relationships. There are many ways to achieve this, for example, we could propose the concept of differing levels of equivalence. A pair of traces that reveal only a single discrepancy in the verification process could be considered to be more strongly equivalent than another pair for which ten discrepancies have been flagged. This of course raises questions on correlating the different equivalence levels to the actual nature of the operational semantics on the verification and host environments, and is beyond the scope of our work in this thesis.

With that in mind, we note that there are several ways in which trust can be used in conjunction with a code security technique in a mobile agent framework.

- *Reducing the need for code security.* As we have already seen, the secure protocol for trace exchange necessitates that an agent copy resides on the trusted verification server which is to be used in the verification process and which is subsequently migrated to the next host / verification server in the event of a success. If we can establish an initial notion of trust in the ability of a host to execute an agent correctly, then this level of trust can be upgraded or downgraded over a period of time in accordance with the results of the verification of agents that execute on it. Once a specific level is attained, the requirement for trace verification could be eliminated (although periodic checks may still be required), resulting in a drastic reduction in deployment cost. In this way, the detection ability of interaction tracing is employed not to guarantee constant code security, but rather to bring about a situation where the likelihood of attacks is greatly decreased.
- *Flexibility in applying sanctions.* One of the issues that has not been considered in depth yet is the action of censuring a hostile host platform. The general assumption is that uniform punitive action is taken towards all perpetrators. On reflection, we can see that this inflexibility might not be desirable in every situation. For example, in e-commerce scenarios, it is possible to acquire additional economic interests or benefits that result from constant interaction with a trusted host, which may not be readily available from untrusted hosts. Thus, when we discover that a trusted host has violated an agent's integrity, we do not immediately bar our agents from visiting that platform (as might have been the case with an untrusted host). Instead, we could permit migration, but with possibly a more comprehensive code security technique applied. Trust thus provides us with the basis for deciding on a suitable course of action to be taken in dealing with the violation of the agent in the event that such flexibility is advantageous in a given situation.
- *Scalability through the delegation of trace verification.* In the trace exchange protocol, a verification server could easily become overwhelmed if it is re-

sponsible for performing an excessively large number of verifications of agent executions. In order for such a system to scale, the verifying platform must be able to delegate some of its verification activities to other entities in the system. Again, some notion of trust between entities is required for the deploying platform to delegate its activities in this manner.

- *Combining different code security techniques.* A security infrastructure that addresses the code security issue comprehensively may need to incorporate a combination of techniques, rather than a ubiquitous 'one-size-fits-all' solution. Trust levels can be used as a mechanism for selecting the appropriate technique or combination of techniques to use, depending on the execution environment and targeted application. For example, in the discussion of the tamper-proof hardware approach [163], there is a suggestion for the use of trust to negate the requirement for hardware to be installed on all execution environments. It could be permissible to run a mobile agent in a software-only environment (using other purely software based code security techniques), if the deployers of the agent have a certain amount of trust in that environment. Tamper-proof hardware would only need to be installed in environments whose behaviour or reputation is unknown to the deployers.

The benefits of incorporating trust and using trust models in a distributed system in general [138] and a mobile agent system in particular [132] have already been identified. We continue this work by developing a trust model explicitly in the context of a mobile agent system through the definition of the trust relationships possible in such a system.

8.1.1 Trust in a public key infrastructure (PKI)

A PKI [4] is essentially a system that provides all the necessary maintenance activities associated with the complete life cycle of certificates, which is one of the main elements of a distributed authentication service [84]. The main issue in such a service is ensuring that a public key is correctly associated with the identity of an entity that owns the corresponding private portion. PKI systems involve a trusted third party termed a *certificate authority* (CA) that is responsible for verifying name-key bindings through the issuance of certificates. On a large scale basis, a single CA would be incapable of handling the name-key binding activity for all users; thus the need for several CAs arises. In such a situation, an end-user may not be able to immediately identify a certificate received, and may require that the certificate be verified in turn by a CA that he or she is familiar with. This in effect creates a *certification path* through which CAs verify the certificates of other CAs all the way up to a *root authority*, which a user would be acquainted with. The certification path thus reflects the propagation of trust between different CAs and users in the system.

In a PKI, the task of associating the public key correctly with an identity (a security requirement for any entity before it can commence utilizing the key) has essentially been delegated from the entity to the CA. In the trace exchange protocol, the verification server functions as an intermediary between an agent owner platform (the platform from which the mobile agent is initially launched) and a host platform. The task of verifying the correct execution of the mobile agent on the host platform has now been delegated from the agent owner platform to the verification server. In addition, verification servers may delegate execution verification activities to other verification servers in the system, analogous to the manner in which CAs verify certificates of other CAs in a PKI.

Trust is generally established with regards to a specific activity, rather than as an unconstrained notion. For example, it is too general to simply state that an entity A trusts another entity B; it would be more accurate to say that A trusts B with respect to a certain activity. The context of trust used in a PKI is generally with respect to the name-key binding activity (other activities may include secure key pair generation, in the event the CA is responsible for key generation as well).

For the general case of code security, we provide a possible classification for the various aspects of an agent that a code security technique might seek to safeguard:

- *Execution integrity.* The correct transformation of the current state of the mobile agent to a new state in accordance with the semantics of agent code is the goal here; this is safeguarded using techniques such as interaction tracing.
- *State/code integrity.* The state and code of the mobile agent need to be protected from invalid manipulation.
- *State/code visibility.* It may be necessary to permit only certain parts of the state and code to be made visible to the host platform since other parts may contain sensitive information.
- *Resource provision.* It is also important to guarantee the provision of nec-

essary system resources (within the constraints of the resource and security policy of the host platform) to the mobile agent in order for it to execute successfully.

These aspects can then be used as a context for establishing trust. They are all applicable for the case of interaction tracing with the exception of state/code visibility.

8.1.2 A trust model for mobile agents

One of the seminal papers to discuss the idea of trust relationships in a distributed authentication system is [162]. There are two types of trust relationships introduced in this paper: direct trust and recommended trust. Direct trust is analogous to the trust obtained between a CA and an entity which generates a public key pair. In this instance, the CA can directly verify the identity of this entity and issue a certificate binding this identity to the entity's public key. Recommended trust is analogous to the trust obtained between an entity that has just received a certificate and the CA that issued it. In this instance, the entity has no way of determining directly that the public key in the certificate is bound correctly to the identity contained within, and trusts the CA to perform this activity for it. A trust derivation algorithm (also presented in [162]) can be used to generate new derived trust relationships from an existing set of direct trust and recommended trust relationships existing in the system.

To achieve a fine grained trust model, we introduce the idea of partitioning a complete mobile agent into several smaller, self-contained state and code components. Mobile agents can be considered as complex pieces of code composed by their deployers from individual components, with the idea that the components can be combined to fit application domain-specific requirements ([94], [95]). In this instance, we can selectively apply different code security techniques to different state and code components, and thus establish trust relationships of different contexts with respect to different state and code components.

8.2 Trust and belief relationships in a mobile agent system

The nature of trust that we employ in the context of trust relationships is binary in nature (as discussed in section 8.1) and is established on the basis of direct interaction between two entities (i.e. an entity A trusts another entity B on the basis of some action undertaken by B that can be evaluated directly by A). For example, a verification server could establish some trust in a given host upon successful verification of a trace submitted by that host. On the other hand, another entity which is unable to evaluate the actions of a host directly itself (e.g., agent owners do not verify traces in our protocol) will be unable to establish any measure of trust in that host. As we will see later in section 8.2.1, we provide a way for a derived trust relationship to be established between an agent owner and a host.

In addition to trust, the multi-agent community also relies on the use of the concept of reputation [3] to enhance the use of trust. Within the context of well known commercially deployed reputation systems such as eBay [117], reputation can be considered as a function of the cumulative positive and non-positive ratings given to a specific entity by other entities in the system over a period of time. For an existing user of the system who has yet to interact with a given entity in order to establish some measure of trust (or otherwise) in it, the reputation of the entity serves as a useful guide as to whether an initial interaction is advisable in the first instance.

In our interaction protocol, if a form of reputation can be established with respect to a host, then an agent owner can use this reputation to determine whether to include the host in the potential itinerary of the mobile agent. The reputation should therefore be able to influence the development of any derived trust relationship formed between the agent owner and a host. We use the concept of a *belief* relationship between two entities to encapsulate this notion of reputation that we wish to employ in the context of our trust model.

The state space encompassing all the relevant entities in our system is as follows:

$$
\begin{array}{rcll}
\mathcal{OP} & = & \{\ A,\ B,\ \ldots\ \} & \text{(Set of agent owner platforms)} \\
\mathcal{VSP} & = & \{\ VS_0,\ VS_1,\ \ldots\ \} & \text{(Set of verification servers)} \\
\mathcal{HP} & = & \{\ H_0,\ H_1,\ \ldots\ \} & \text{(Set of host platforms)} \\
\mathcal{SC} & = & \{\ s_0, s_1,\ \ldots\ \} & \text{(Set of code/state components)} \\
\mathcal{SO} & = & \{\ x_0, x_1,\ \ldots\ \} & \text{(Set of security aspects)}
\end{array}
$$

We assume that all mobile agents in the system can be composed from a combination of predefined sets of code and state components. Agent owner platforms are platforms where mobile agents are launched from; these agents will migrate through an itinerary of host platforms before terminating or returning to their respective agent owner platforms. Trust and belief relationships between entities in the system are established with respect to a set of security aspects; for the case of interaction tracing this would be execution integrity, state/code integrity and resource provision.

$$
\begin{array}{lll}
(1) & VS_0\ \mathit{trusts.exe}\ H_0\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Server-host execution trust)} \\
(2) & A\ \mathit{trusts.exe}\ H_0\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Owner-host execution trust)} \\
(3) & A\ \mathit{trusts.ver}\ VS_0\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Owner-server verification trust)} \\
(4) & VS_0\ \mathit{trusts.ver}\ VS_1\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Server-server verification trust)} \\
(5) & A\ \mathit{believes.exe}\ H_0\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Owner-host belief)} \\
(6) & A\ \mathit{believes.ver}\ VS_0\ \mathit{with}\ (\mathcal{X}, \mathcal{S}) & \text{(Owner-server belief)}
\end{array}
$$

where $A \in \mathcal{OP},\ \mathcal{X} \in \mathcal{SO},\ \mathcal{S} \subseteq \mathcal{SC},\ VS_0, VS_1 \in \mathcal{VSP},\ H_0 \in \mathcal{HP}$

Figure 8.1: Basic trust and belief relationships

We now describe the basic trust and belief relationships (Figure 8.1) that are possible between the three different types of entities in the system (host platforms, agent owner platforms and verification servers). The first relationship, server-host trust, represents the trust that a verification server, VS_0, has in a host platform, H_0, to undertake the transformation of the state components specified in $\mathcal{S}$ correctly with respect to a security aspect $\mathcal{X}$. Therefore, $\mathcal{S}$ and $\mathcal{X}$ are constraints on the context for which this relationship is applicable. We give this type of trust relationship the term *execution trust*. A server-host trust relationship is initially established when a verification server successfully validates the interaction trace submitted by a host platform. This trace pertains to a mobile agent composed of the components $\mathcal{S}$, undertaken by the verification server to test the reliability of the host platform in question.

The owner-host relationship represents the trust an agent owner platform, A, has in a host platform, H_0, to undertake the transformation of the code/state components specified in $\mathcal{S}$ correctly with respect to a security aspect $\mathcal{X}$. Since verification servers are the only entities in our system possessing the functionality necessary to ascertain execution correctness, this type of relationship cannot be established directly by an agent owner platform. It can only be derived from other existing trust relationships, as will be demonstrated later.

The owner-server relationship refers to the situation when an agent owner platform, A, trusts a verification server VS_0 to undertake correct verification of the transformation of state components specified in $\mathcal{S}$, with respect to a security aspect $\mathcal{X}$. This trust relationship is analogous to the idea of recommendation or recommended trust as described in [162]. In our context, we shall give it the term *verification trust*. Verification trust is established directly when an agent owner platform decides to delegate the task of verifying correct execution of its mobile agents to a verification server. Once such a relationship is in place, the agent owner platform will supply the verification server with the necessary information and resources (for example, a copy of all mobile agents launched by the agent owner) to undertake the verification successfully.

The server-server trust relationship is interpreted to mean that a verification server, VS_0, trusts another verification server, VS_1 to undertake verification of the correct transformation of state components, specified in $\mathcal{S}$, with respect to a security aspect $\mathcal{X}$. This would result from a verification server delegating the responsibility of verifying certain host platforms to other verification servers in the system, and is thus a relationship that can be established directly.

All of these trust relationships, with the exception of owner-host trust, express trust on the basis of explicit actions (or the results of those actions) undertaken by entities involved in the relationship. Section 8.3.2 elaborates further on how these relationships are initially established and how they might evolve over a period of time. As we have previously discussed, we use the belief relationship to express the analogy of reputation in our trust model. For example, an agent owner platform A may choose to believe (based on the knowledge acquired from other agent owners or external sources) that host platform $\mathcal{H}_0$ is capable of executing $\mathcal{S}$ correctly with respect to $\mathcal{X}$. This can be expressed as an owner-host belief relationship between A and $\mathcal{H}_0$ (statement 5), instead of a trust relationship. From the viewpoint of reputation, the owner-host belief relationship indicates the willingness of an agent owner to launch a mobile agent to the host in question based on the reputation that the host has acquired from its previous interactions with other agent owners.

We do not, however, consider how the reputations of the various hosts are created and managed in our trust model. The idea of a server-host and a server-server belief relationship is equally valid in this context; they are not included in order to simplify the trust derivation algorithm that we develop in the next section.

It is important to note that trust and belief relationships are not symmetric in our system (i.e VS_0 *trusts.ver* VS_1 does not necessarily imply that VS_1 *trusts.ver* VS_0). Also, we did not introduce the idea of trust originating from a host platform (i.e. the host platform is the terminating point for an execution trust relationship). This could be useful if we wish to extend our model to encompass the issue of host security (i.e. the host needs to be able to trust that agent owner platforms do not dispatch malicious agents to its environment), but we do not address this point here.

8.2.1 Deriving new relationships

The different ways in which new trust and belief relationships can be formed from existing ones is illustrated in Figure 8.2. For the server-host and server-server trust derivations, there is an intersection of the code/state component constraint sets $\mathcal{S}_1$ and $\mathcal{S}_2$ in the derived relationship. This intersection indicates that the relationship should be in the context of the code/state components common to both relationships that it is derived from. Owner-host and owner-server trust derivations will, in addition, involve owner-host and owner-server belief relationships as well. In this case, we only derive a new trust relationship if there already exists a belief relationship between the agent owner platform and the host platform or verification server concerned. Again, this new relationship will be in context of the code/state components common to both the two initial relationships as well as the belief relationship ($\mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3$).

Note that we do not include the idea of inferring new trust or belief relationships by using an existing belief relationship as a starting point. This is due to the fact that while trusting behaviour is transitive (resulting for example, from delegation of the verification activity in the case of server-server or server-host trust derivations), trusting belief is not. However, the context of an existing belief relationship can be altered independently by an agent owner platform depending on the results of the new trust relationships derived. For example, if in deriving a new owner-host trust relationship, $\mathcal{S}_1$ and $\mathcal{S}_2$ are both supersets of $\mathcal{S}_3$, then the agent owner platform could choose to expand the constraints of its current belief relationship with the host platform from $\mathcal{S}_3$ to $\mathcal{S}_1 \cup \mathcal{S}_2$ instead. This ensures that the next time a trust relationship is derived from the existing owner-server and server-host relationships, a wider constraint can be achieved.

Deriving server-host trust

If there exists relationships of the form

a) VS_1 *trusts.exe* H_0 *with* $(\mathcal{X}, \mathcal{S}_1)$ b) VS_0 *trusts.ver* VS_1 *with* $(\mathcal{X}, \mathcal{S}_2)$

then we can infer a new relationship of the form

VS_0 *trusts.exe* H_0 *with* $(\mathcal{X}, \mathcal{S}_1 \cap \mathcal{S}_2)$

Deriving server-server trust

If there exists relationships of the form

a) VS_0 *trusts.ver* VS_1 *with* $(\mathcal{X}, \mathcal{S}_1)$ b) VS_1 *trusts.ver* VS_2 *with* $(\mathcal{X}, \mathcal{S}_2)$

then we can infer a new relationship of the form

VS_0 *trusts.ver* VS_2 *with* $(\mathcal{X}, \mathcal{S}_1 \cap \mathcal{S}_2)$

Deriving owner-host trust

If there exists relationships of the form

a) A *trusts.ver* VS_0 *with* $(\mathcal{X}, \mathcal{S}_2)$ b) VS_0 *trusts.exe* H_0 *with* $(\mathcal{X}, \mathcal{S}_1)$

and if there exists a belief relationship of the form

c) A *believes.exe* H_0 *with* $(\mathcal{X}, \mathcal{S}_3)$

then we can infer a new trust relationship of the form

A *trusts.exe* H_0 *with* $(\mathcal{X}, \mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3)$

Deriving owner-server trust

If there exists relationships of the form

a) A *trusts.ver* VS_0 *with* $(\mathcal{X}, \mathcal{S}_1)$ b) VS_0 *trusts.ver* VS_1 *with* $(\mathcal{X}, \mathcal{S}_2)$

and if there exists a belief relationship of the form

c) A *believes.ver* VS_1 *with* $(\mathcal{X}, \mathcal{S}_3)$

then we can infer a new trust relationship of the form

A *trusts.ver* VS_1 *with* $(\mathcal{X}, \mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3)$

Figure 8.2: Deriving new trust relationships

8.2.2 Verification path

It is important to note the exact sequence in which the existing relationships are combined to provide a context for the new trust relationships obtained. To achieve this, the idea of a verification path is introduced. A verification path refers to a sequence of entities (verification servers, agent owner platforms or host platforms) involved in the derivation of new trust relationships. For example, consider the following trust relationship statements:

(a) VS_0 *trusts.ver* VS_1 *with* ...

(b) VS_1 *trusts.ver* VS_2 *with* ...

(c) VS_2 *trusts.exe* H_0 *with* ...

(a) and (b) can be combined to obtain a new trust relationship :

(d) VS_0 *trusts.ver* VS_2 *with* ...

The verification path at this stage involves the entities VS_0, VS_1 and VS_2 in that given sequence. Thus, c) and d) can subsequently be combined to obtain a new trust relationship :

(e) VS_0 *trusts.exe* H_0 *with* ...

The verification path sequence now involves the entities VS_0, VS_1, VS_2 and H_0. Further derivation of new trust relationships from (e) will involve expanding the verification path in a similiar manner. The idea of a verification path will be used in the algorithm that we develop next.

8.3 Trust derivation algorithm

We can now detail a trust derivation algorithm (Figure 8.3), which we extend from the one presented in [162] by incorporating the idea of using beliefs in the process of deriving new trust relationships. The algorithm demonstrates how the derivations just explained can be systematically applied in a system described by an initial set of trust relationships. The goal of this algorithm is to generate a set of tuples, $\mathcal{HS}$, from this initial set of trust relationship expressions. This set of tuples will describe the derived owner-host trust relationships that exist between any given agent owner platform A and all other host platforms in the system. The algorithm works on the elements within two sets, $\mathcal{HS}$ and $\mathcal{N}$.

$\mathcal{HS}$ is a set of tuples, with each tuple consisting of a host platform H as well as a set of state components, C. Initially, $\mathcal{HS}$ is empty and the algorithm will append elements to it during its execution. At the termination of the algorithm, each tuple in $\mathcal{HS}$ refers to a new host platform H in which the agent owner platform A can establish a new derived owner-host trust relationship with the form $A\ trusts.exe\ H\ with\ (\mathcal{X}, C)$.

Signature
$C = \mathbb{P}(\mathcal{SC})$
$HostTuple = H \times C$
$\mathcal{HS} = \mathbb{P}(HostTuple)$
$\mathcal{HS} ::= \{\langle H_1,\ C_1\rangle,\ \langle H_2,\ C_2\rangle, \ldots\}$
$seq = Ownerplatform \times VS \times \ldots \times VS$
$PathTuple = VS \times seq \times \mathcal{S}c$
$\mathcal{N} = \mathbb{P}(PathTuple)$
$\mathcal{N} ::= \{\langle VS_1,\ seq_1, \mathcal{S}c_1\rangle,\ \langle VS_2,\ seq_2, \mathcal{S}c_2\rangle, \ldots\}$

Initialisation
$\mathcal{TBS} = \{$ Set of initial trust and belief relationships $\}$
$\mathcal{HS} = \{\}$
$\mathcal{N} = \{\langle VS_1, [A, VS_1], \mathcal{S}c_1\rangle, \langle VS_2, [A, VS_2], \mathcal{S}c_2\rangle, \ldots, \langle VS_j, [A, VS_j], \mathcal{S}c_j\rangle\}$
where A is the current agent owner platform to which the algorithm is being applied and $VS_1, VS_2, \ldots, VS_j$ are all the verification servers for all trust relationships $A\ trusts.ver\ VS_i\ with\ (\mathcal{X}, \mathcal{S}c_i) \in \mathcal{TBS}$

boolean $foundtuple$;
Do until $\mathcal{N} = \emptyset$:
  Select a $PathTuple\ \langle VS_i, seq_i, \mathcal{S}c_i\rangle$ from $\mathcal{N}$
  for every $VS_i\ trusts.exe\ H_k\ with\ (\mathcal{X}, \mathcal{S}_l) \in \mathcal{TBS}$
    if $A\ believes.exe\ H_k\ with\ (\mathcal{X}, \mathcal{S}_m) \in \mathcal{TBS}$
      begin
        $foundtuple := false$
        for every $HostTuple\ \langle H_j, C_j\rangle$ in $\mathcal{HS}$
          if $H_k = H_j$
            $C_j := C_j \cup (\mathcal{S}_l \cap \mathcal{S}_m \cap \mathcal{S}c_i)$, $foundtuple = true$
        if $not\ foundtuple$
          $\mathcal{HS} := \mathcal{HS} \cup \langle H_k, \{\mathcal{S}_l \cap \mathcal{S}_m \cap \mathcal{S}c_i\}\rangle$
      end
  end
  for every $VS_i\ trusts.exe\ VS_n\ with\ (\mathcal{X}, \mathcal{S}_p) \in \mathcal{TBS}$
    if $A\ believes.ver\ VS_n\ with\ (\mathcal{X}, \mathcal{S}_q) \in \mathcal{TBS}$
      if $VS_n \notin seq_i$
        $\mathcal{N} := \mathcal{N} \cup \langle VS_n,\ seq_i \bullet VS_n, \{\mathcal{S}_p \cap \mathcal{S}_q \cap \mathcal{S}c_i\}\rangle$
  end
  $\mathcal{N} := \mathcal{N} \setminus \langle VS_i, seq_i, \mathcal{S}c_i\rangle$

Figure 8.3: Trust derivation algorithm

$\mathcal{N}$ is also a set of tuples, each tuple representing a possible next step in a verification path. The constituent components of each tuple are:

- a verification server, VS_i, which is the next possible entity in a verification trust path;
- a sequence $seq = [A, VS_1, VS_2 \ldots, VS_i]$ which represents the sequence of the verification path traversed so far;
- a set of code/state components $\mathcal{S}c$ which represents the code/state components for which trust is still applicable on the given verification path.

The expression $seq \bullet VS_i$ is used to indicate that a new entity VS_i is being appended to a sequence seq of a verification path. At the start, $\mathcal{N}$ is initialized with the tuples that correspond to all initial trust relationships of the form $A\ trusts.ver\ VS_i\ with\ (\mathcal{X}, \mathcal{S}c_i)$, where A is the current agent owner platform to which the algorithm is being applied to.

8.3.1 Example

To illustrate the operation of the algorithm, we provide a simple example. Consider a system consisting of a set of host platforms, **HE**, verification servers, **VE**, code/state components **SE** and a single agent owner platform, A (Fig. 8.4).

$$\mathbf{VE} = \{VS_a, VS_b, VS_c, VS_d\}$$

$$\mathbf{HE} = \{H_p, H_q\}$$

$$\mathbf{SE} = \{s_0, s_1, s_2, s_3, s_4\}$$

At the start, we assume that the system already has the following initial trust and belief relationships :

1. $A\ trusts.ver\ VS_a\ with\ (\mathcal{X}, \mathbf{SE})$
2. $A\ trusts.ver\ VS_c\ with\ (\mathcal{X}, \mathbf{SE})$
3. $A\ trusts.ver\ VS_e\ with\ (\mathcal{X}, \mathbf{SE})$
4. $A\ believes.ver\ VS_b\ with\ (\mathcal{X}, \{s_1, s_2\})$
5. $A\ believes.exe\ H_p\ with\ (\mathcal{X}, \{s_2, s_3\})$
6. $VS_a\ trusts.ver\ VS_b\ with\ (\mathcal{X}, \{s_0, s_1, s_2\})$
7. $VS_b\ trusts.exe\ H_p\ with\ (\mathcal{X}, \{s_1, s_2\})$
8. $VS_c\ trusts.ver\ VS_d\ with\ (\mathcal{X}, \{s_1, s_4\})$
9. $VS_d\ trusts.exe\ H_q\ with\ (\mathcal{X}, \mathbf{SE})$
10. $VS_e\ trusts.exe\ H_p\ with\ (\mathcal{X}, \{s_3, s_4\})$

We now proceed to apply the algorithm to A to determine all the new trust relationships that can be derived with the host platforms in the system. We start

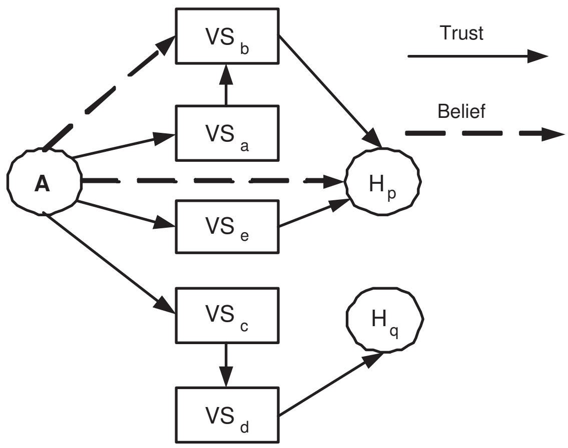

Figure 8.4: Trust derivation example

by initializing $\mathcal{N}$ with all the trust relationships that originate from the agent owner platform A (1, 2 and 3).

$\mathcal{N}\ = \{\langle VS_a,\ [A, VS_a],\ \mathbf{SE}\rangle,\ \langle VS_c,\ [A, VS_c],\ \mathbf{SE}\rangle,\ \langle VS_e,\ [A, VS_e],\ \mathbf{SE}\rangle\}$

$\mathcal{HS}\ = \{\}$

After the first pass of the algorithm, we have:

$\mathcal{N}\ = \{\langle VS_b,\ [A, VS_a, VS_b],\ \{s_1, s_2\}\rangle,\ \langle VS_c,\ [A, VS_c],\ \mathbf{SE}\rangle,\ \langle VS_e,\ [A, VS_e],\ \mathbf{SE}\rangle\}$

$\mathcal{HS}\ = \{\}$

After the second pass of the algorithm, we have:

$\mathcal{N}\ = \{\langle VS_c,\ [A, VS_c],\ \mathbf{SE}\rangle,\ \langle VS_e,\ [A, VS_e],\ \mathbf{SE}\rangle\}$

$\mathcal{HS}\ = \{\langle H_p, \{s_2\}\rangle\}$

After the third pass of the algorithm, we have:

$\mathcal{N}\ = \{\langle VS_e,\ [A, VS_e],\ \mathbf{SE}\rangle\}$

$\mathcal{HS}\ = \{\langle H_p, \{s_2\}\rangle\}$

After the fourth pass of the algorithm, we have

$\mathcal{N}\ = \{\}$

$\mathcal{HS}\ = \{\langle H_p, \{s_2, s_3\}\rangle\}$

Thus we can form a new trust relationship of the form

$A\ trusts.exe\ H_p\ with\ (\mathcal{X}, \{s_2, s_3\})$

On the basis of this trust relationship, A would therefore be willing to launch an agent for execution on H_p as part of its itinerary, with the agent being composed of components s_2 and s_3.

8.3.2 Using the trust model and derivation algorithm

We discuss intuitively how the trust model can now be used in conjunction with the trace exchange protocol. Consider a community of host platforms, verification servers and agent owner platforms with trust relationships already established among themselves. A new agent owner platform that wishes to participate in the community will need to establish trust relationships with one or more verification servers. Through a short interaction with a selected verification server, the agent owner platform could determine the host platforms that the server in question has a trust relationship with. The agent owner platform can initially ascertain the reliability of that verification server by composing a mobile agent and launching it to a host platform, with the interaction trace being submitted back to both the agent owner platform and the verification server. The results reported back by the verification server are checked for consistency with the validation of the trace by the agent owner platform.

Once the agent owner platform is satisfied, it establishes a trust relationship with the verification server (with respect to specific components) and executes the trust derivation algorithm to obtain new trust relationships with other host platforms. This provides it with a potential itinerary for future mobile agents that it wishes to launch (in the event that the agent owner platform supplies a predefined itinerary), or useful information that can be embedded in the agent itself (should the agent be capable of dynamically deciding its itinerary while it migrates). New verification servers that join the community can establish trust relationships with existing verification servers in a similar manner. New host platforms on the other hand, could advertise their presence through a registry service after which they can be tested for reliability in hosting mobile agents by verification servers who express interest in establishing trust relationships with them.

The key to the evolution of the trust relationships in this framework is the verification of an interaction trace submitted by a host platform to a verification server. Trust relationships are initially established as described above, and remain constant as long as all traces checked are valid. The moment a verification server detects an invalid trace, the nature of its current trust relationships with the offending host platform will be altered. This could range over several possible alternatives : severing all existing trust relationships, severing some trust relationships or degrading existing relationship(s) by reducing the number of components that the relationship(s) is valid for.

Returning to the analogy that we introduce at the start of section 8.1.1 (i.e. the verification server being roughly equivalent to a CA in a PKI), we note that the CA is trusted to maintain the integrity of the key-to-name binding within a certificate. Certificate revocation is employed when such integrity becomes suspect (for example, due to a suspected key compromise before the expiry of the certificate). Revocation is typically implemented using a periodic publication mechanism such as certificate revocation lists (CRLs), which can be accessed by other entities that need to validate certificates. A verification server, on the other hand, is trusted to maintain the integrity of mobile agent execution, and alters its trust relationship (degradation or destruction) with the offending platform when a violation of this integrity is detected. Information about this relationship change (*trust information*) is then propagated to all other verification servers or agent owner platforms (the trustors) that have established trust relationships with the server that detected the violation (the trustee). This will in turn result in a corresponding change in trust relationships on those servers and platforms as well. In effect, trust information is equivalent to a CRL in a PKI, with the difference that trust information is propagated instead of being published. Agent owner platforms could use the event of reception of trust information as a trigger to execute the trust derivation algorithm again in order to recalculate their new trust relationships with existing host platforms.

The actual mapping between the detection of a violation in an interaction trace and the subsequent change affected in a trust relationship (destruction or degradation) is a function of a security policy which can be either administered locally or globally administered. The discussion of such policies and their implication on the trust dynamics of the system as a whole is beyond the scope of this thesis, but remains important work to be accomplished in studying the effects of a trust model in a mobile agent security framework.

8.4 Conclusion

We argue that the notion of trust can aid in a more flexible and scalable deployment of existing code security techniques. We also show that the manner in which trust would be employed in a security infrastructure for mobile agents has many parallels to the way it is used currently in a distributed authentication system such as a PKI. Based on this motivation, we propose a trust model for the secure trace exchange protocol that resembles the trust model of a PKI. Drawing from existing work on trust relationships in such an infrastructure, we define several trust relationships for the entities in such a system. We then demonstrate how new trust relationships could be derived from existing ones, and present an algorithm to demonstrate our approach.

An initial attempt to introduce a trust model in the context of a code security technique was undertaken in [157]. The approach there attempts to show how trust can be used in conjunction with tamper proof hardware to protect the execution integrity of mobile agents dispatched by a user or a trusted third party. The notion of trust is not as fine grained as our approach, and the idea of a chain of trust relationships is not discussed. We believe that the material developed here is representative of the initial work required in the construction of a complete trust model for a mobile agent system (regardless of whether interaction tracing is used or not). Such a model would permit a detailed insight into the complex interactions that involve trust in a mobile agent system.

Chapter 9

Conclusions and Future Work

In this chapter, we summarize the work of the thesis, provide indications for future work in relevant areas and then conclude with some comments on the general direction of mobile agent security research.

9.1 Summary

Code security is identified as the most relevant area in mobile agent security research, and the existing techniques to address it are examined. Arguments are presented to support the proposition that the execution tracing approach represents the best way forward in contrast with other existing code security techniques. Interaction tracing is then developed as a new improvement over execution tracing that overcomes some of its original disadvantages. An operational semantics is used to formalize the process of code execution, creation and verification of a trace. This allows a formal establishment of the security properties achievable utilizing the interaction tracing approach. An interaction protocol that allows secure exchange of traces and code between verification servers and host platforms is developed, modelled formally and verified using the FDR model checker. In addition, a protocol to allow the reconciliation of traces in order to detect potential attacks on inter agent communication is modelled and verified in a similar manner as well. We then perform a quantitative evaluation of the cost of using the interaction protocol to secure a mobile agent in an information retrieval task. This cost is contrasted against the costs of using an unsecured mobile agent and a conventional client-server arrangement to accomplish the same task. Finally, we show how trust relationships can be formed between the verification servers and hosts in the system in order to permit the activity of trace verification to be delegated between verification servers. A trust derivation algorithm to develop new relationships between these entities is then presented. This provides for a more flexible deployment of the interaction protocol as well as presenting a possible way towards integrating multi-agent security concepts into code security research.

9.2 Future work

In this section, we discuss possibilities for future work in the following relevant areas of the thesis.

9.2.1 Combining security techniques

We saw in section 4.4 that interaction tracing needs to be combined with some other code security technique (such as a data encapsulation protocol) if protection against extraction of data from within the agent body was required. The idea of combining different security techniques was also put forward as one of the reasons for incorporating trust into the trace exchange protocol (section 8.1), and is also an emerging theme in recent research into code security solutions. For example, [5] uses encrypted functions in conjunction with tamper proof hardware while [166] involves encrypted functions along with the use of replicated agents and servers. Investigating the ways in which these techniques can be used in conjunction with each other presents a possible avenue for future work.

There is also the need to examine how a specific code security solution might impact existing host security mechanisms that most mobile agent systems already possess. A real-time deployment of a mobile agent system in an application area where security is paramount must obviously employ both code and host security techniques. Interaction tracing appears to be amenable to joint deployment with most host security techniques. Other techniques such as encrypted functions may possibly interfere with access control mechanisms (for example, the security manager in Java) because the functionality of the agent code is obscured in these approaches. It would be interesting to examine the relationship between host and code security, and any effects (adverse or otherwise) that they may have on each other in a general or specific application scenario.

9.2.2 Sanctions against hostile hosts

Interaction tracing is derived from execution tracing, which is a detection based approach to code security. As we have already noted, detection based approaches rely on a trusted authority or intermediary to effect some form of sanction against the hostile hosts. The nature of such sanctions and how they may affect the future viability of the perpetrator in question to execute agents is a subject yet to be examined properly in current code security research. In section 9.2.7, we discuss how the concept of trust relationships can be used to provide a more flexible means of undertaking such sanctions towards malicious hosts.

9.2.3 Time dependency and scalability

One of the assumptions made with regards to the activity of creating and verifying a trace (section 4.1.1) was that execution of agent code would not be time dependent. However, there may be situations during code execution where time dependency may be required. For example, a specific action may need to be undertaken by an agent if a specific external condition does not occur within a certain time period. The operational semantics that we develop in Chapter 5 for formalizing interaction tracing does not support the notion of time and cannot support the replay of an agent with such a time dependent requirement. The introduction of time dependency may affect the determination of trace equivalence and the trace reconciliation process, both of which rely upon the examination and comparison of input and output traces. In light of this, additional work needs to be undertaken to fully examine the effects of time dependency on the sequence of inputs received and outputs produced during agent execution.

The process of trace reconciliation may require that a verification server interact with a large number of other verification servers in order to obtain the required input traces for reconciliation with a given output trace. The issue of scalability of trace reconciliation arises with the need to consider real-time distributed systems, where network transmission delays may further lengthen the interaction time with other verification servers in the execution of the trace reconciliation protocol. Performing trace verification and reconciliation at every step of an agent's path might then result in an unacceptable time duration required for a mobile agent to complete migration through its itinerary. For future work, a quantitative assessment of scalability could be conducted (for example, repeating the evaluation presented in Chapter 7 for an increasing number of agents) and methods of simplifying the communication overhead in trace reconciliation could be investigated.

We note here that time dependency and scalability are important issues in any multi-agent system in general [114], independent of any specific aspects of mobility or security. In particular, scalability has also been addressed in previous quantitative evaluations of non-secure mobile agent systems [161] and the observations made in these evaluations would be useful in further refining interaction tracing and trace reconciliation.

9.2.4 Guarding against resource provision attacks

In the interaction protocol for trace exchange, the possibility of using time-outs to provide some form of protection against a resource provision attack (section 3.6.3) is presented. In order to achieve a reasonable estimate of the amount of time required for a mobile agent to complete execution successfully on a given host, advance knowledge of the amount of computational resources provided by the host for the agent in question is required. Knowledge of this nature could be achieved, for example, through a market-based model for resource allocation [20] where mobile agents interact with potential hosts through bidding mechanisms in order to obtain shares of the net computational resources for offer among the hosts. If this interaction is accomplished prior to the commencement of agent migration, an estimate could be calculated for the execution time on each host. The hosts for these agents could in turn, provide resource control [137] or accounting schemes for monitoring CPU and memory usage [15] in order to ensure that the resource-consumption by the agents do not exceed some earlier pre-negotiated limits.

However, agent code may contain a large amount of external communication activities as well, which would complicate the process of estimating execution time on different hosts. Investigating an approach of this nature (based on time-outs) would provide an interesting avenue for future work, and safeguarding resource provision in general remains an area to be addressed properly in code security approaches.

9.2.5 Securing inter agent communications

The issue of protecting inter agent communication on potentially hostile hosts was addressed in our work through trace reconciliation; however this issue in general is inadequately addressed in the code security research area. We do not believe that it is possible to detect such attacks without the use of a trusted third party to intermediate in some form during or after the communication acts themselves. Other techniques that are incapable of detecting attacks (such as encrypted functions) must examine precautionary strategies that agents should adopt when communicating with remote agents in order to reduce the impact of dubious communication content originating from a possibly corrupted agent.

9.2.6 Modelling and verifying security protocols

We had noted in section 6.5 that the Casper model is not able to model a mobile agent accurately if we adopt the assumption that the format of messages exchanged between static entities and the sequence in which they occur within a single protocol run are predefined and remain fixed. Thus, in order for a model to be able to detect such attacks, two additional requirements are necessary:

1. It must be able to encapsulate all possible behaviours of a mobile agent (as a function of its code, internal state and state of its execution environment) that have the ability to alter the format or sequence of messages exchanged within a single protocol run.
2. The model must be able to take into account all these different possibilities of message contents and sequences when it is used to simulate a protocol run.

With regards to the first requirement, the identification of the specific state or code of an agent that is capable of altering the format or sequence of messages is clearly not a trivial matter. Even if this could be accomplished, the additional possibilities for protocol runs with different message contents and sequences will greatly increase the number of possible interleavings of protocol runs, consequently creating a potential explosion in the state space to be explored. This may make it less suitable for use on a finite state space model checker such as FDR. In that case, checking the viability of the model may require techniques to reduce the state space explosion (for example, the automation of the Strand Spaces Model [135] results in a dramatic reduction of state space exploration).

As a matter of interest, we note that the attacks described by Roth [123] were discovered in an ad hoc, intuitive manner without resort to any formal methods of verification. More recent work in the area of encapsulation protocols to protect accumulated data [92] does not attempt to model or verify their approach in any way, and may thus be susceptible to more subtle attacks. If any of these protocols are to be deployed in a commercial setting, it is imperative that some formal security properties are established about them. Adapting existing models for modelling cryptographic protocols to the specific arena of code security remains an interesting and important area for future research.

9.2.7 Evaluation of individual costs and reimplementation

To gain a better insight into the evaluation between the three approaches, the costs of the individual mechanisms (such as serialization, network transfer, class loading, encryption and so on) need to be ascertained within the agent framework being used. There may also be a need to take into account additional costs such as retrieval of certificates from a central server or preliminary interaction between verification servers and hosts in order to establish which servers are responsible for verifying which hosts. Implementation of interaction tracing and the trace exchange protocol could be done in different mobile agent systems such as Aglets, D'Agents and NOMADS, all of which have been used in past quantitative comparative studies. On a more general note, more quantitative studies that evaluate the additional cost burden of code security in a mobile agent system (such as in [122]) need to be undertaken to better understand the tradeoffs involved in the use of mobile agent technology in the first instance.

9.2.8 Using trust and reputation concepts

In order to implement the trust derivation algorithm in a distributed manner, a global knowledge of the various relationships in the system is required. Future work towards this direction could examine a way of coordinating the propagation of trust-related information through the different hosts and verification servers in the system. It would be interesting to consider the use of reputation from the multi-agent research community to enhance the model of trust relationships that we developed. There are many definitions of reputation; one of the more common ones [3] views reputation as a form of social control in the context of trust propagation that is used to influence agents to cooperate in order to avoid gaining bad reputation. Reputation reporting systems have been used successfully in e-commerce systems, such as on eBay [117]. This could be used to a similar effect in our system, where the reputation of a host is the function of the cumulative positive and non-positive ratings given by agent owners over a period of time. Empirical analysis of this approach [118] has demonstrated that the system does encourage transactions; an equivalent analogy for our case would be an increased willingness among agent owners to dispatch their agents to untrusted hosts, which in turn leads to wider-scale deployment of the mobile agent system. There will of course be a need to address the possibility of a malicious agent owner maligning a host platform by giving it undeservedly negative ratings. Such a situation could be tackled using detection techniques that take into account overall ratings from other agent owners in a similar manner to that undertaken for deception detection in distributed reputation management [164].

Another possibility for consideration is the nature of the trust relationship in our work; this is essentially bipolar in that either a host or verification server is trusted completely or it is not trusted at all. If we can quantify trust instead as a numerical range between these two extremes [96], it is possible to achieve greater flexibility in the imposition of sanctions on the hostile hosts. For example, an owner may commence with a predefined trust value in all the hosts; repeated deployment of agents over time to these various hosts will result in their associated trust values being degraded or upgraded in accordance with the results of trace verifications. Once the trust degrades below a predefined minimum, agents would no longer be dispatched to the errant host. Other approaches might include exposing different parts of agent code to different hosts; completely trusted hosts would be able to gain access to the entire body of code, while more sensitive portions would be obscured when the agents execute on hosts with degraded levels of trust. This could be achieved by encrypting different portions of the agent code with different keys, and providing a mechanism for the agent owner to distribute these keys to the appropriate hosts. We briefly explored work of this nature in [143].

9.3 Conclusion

The thrust of the work in this thesis has been towards developing a code security technique that reflects the security considerations of possible future applications of mobile agents. As such we can identify two main themes that motivate the work in this respect:

- *Interaction between agents in a multi-agent scenario.* As multi-agent systems become more wide spread in e-commerce scenarios such as online shopping and auctions [62], utilizing mobility to enhance their application (for example, by reducing network traffic) becomes an interesting possibility. As noted before, security however still remains an issue of primary concern [153]. Most of the work to date in the area of code security does not take into account the need to protect communication between agents executing on potentially hostile hosts. Trace reconciliation in the interaction tracing approach is an attempt to address such a need. However, the current direction in code security research will most likely continue to focus on development and refining existing approaches in the area of encapsulation protocols and encrypted functions. The issue of inter agent communication will only become relevant in research once mobile agent systems begin to be widely deployed in real time e-commerce scenarios. Recent work investigating the usage of mobile agents in conjunction with Web Services [29] suggest that such a deployment might be possible in the future, taking into account the expansion in e-commerce Web Services-based applications such as Yahoo!Shopping and uBid.com [115].
- *Establishing formal security properties.* Cryptographic and security protocol research in general utilize some formalization or model verification to indicate that a proposed cryptographic technique or protocol satisfies some predefined property or requirement. The formalization that we undertake for interaction tracing and the verification of the interaction protocol in Casper reflects this intention. Most approaches to code security (with the exception of encrypted functions) only provide an informal argument for the security properties they wish to achieve. Again, we believe that formalization will become more prominent once mobile agent systems begin to be deployed in e-commerce scenarios and a higher standard of security needs to be demonstrated.

The future direction for mobile agent security research in general will depend to a large extent on the viability of current application areas for mobile agents as well as possible new areas of application. The majority of code security techniques (including the one proposed in this thesis) have been traditionally geared toward the shopping scenario. Research on mobile agent deployment in other well-known areas such as network management still remains active [131] and it is possible that work on security concerns that are specific to this context will appear soon. At the moment, the general assumption within the mobile agent community is that an ideal solution for code security is likely to be either intractable or not cost-effective. Mobile agents are thus only considered for potential deployment in areas where host security is the only issue of consideration.

In light of the work we present in this thesis, we believe that the assumptions of intractability or unjustifiable costs are unwarranted. Such assumptions may arguably hold for current research in code security focusing on encrypted functions, as it suffers from a lack of concrete implementations. In areas where working implementations are shown to be possible (such as interaction tracing or data encapsulation protocols), there is a good chance that a viable and comprehensive code security solution will be presented in the near future. Even then, however, there is still no guarantee that the usage of mobile agents will gain wider acceptance in the distributed computing or multi-agent communities. As was pointed out in a key discussion over the future of mobile agent research and application [102], many researchers and system developers still feel that the cost of restructuring distributed applications in order to employ the mobile agent paradigm remains unreasonably high and not completely justified.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[3] A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *33rd Hawaii International Conference on System Sciences, Maui, Hawaii*, page 6007. IEEE Computer Society Press, 2000.

[4] C. Adams and S. Lyold. *Understanding Public Key Infrastructure: Concepts, standards and deployment considerations*. Que, 1999.

[5] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. In *Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California*, pages 2–11. IEEE Computer Society Press, 2001.

[6] P. Anthony, V. Dang, and N. R. Jennings. Autonomous agents for participating in multiple online auctions. In *Proceedings of the IJCAI Workshop on E-Business and Intelligent Web, New Jersey*, pages 54–64. IJCAI, 2001.

[7] P. Anthony and N. R. Jennings. Developing a bidding agent for multiple heterogeneous auctions. *ACM Transactions on Internet Technology*, 3(3):185–217, 2003.

[8] R. J. R. Back and H. Mannila. On the suitability of trace semantics for modular proofs of communicating processes. *Theoretical Computer Science*, 39(1):47–68, 1985.

[9] M. Baldi, S. Gai, and G. P. Picco. Exploiting code mobility in decentralised and flexible network management. In K. Rothermeland R. Popescu-Zeletin, editor, *Proceedings of the 1st International Workshop on Mobile Agents*, number 1219 in LNCS, pages 13–26. Springer-Verlag, 1997.

[10] M. Baldi and G. P. Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In *Proceedings of the 1998 International Conference on Software Engineering, Kyoto, Japan*, pages 146–155. ACM Press, 1998.

[11] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *Advances in Cryptology (CRYPTO2001), Santa Barbara, California*, number 2139 in LNCS, pages 1–18. Springer-Verlag, 2001.

[12] C. Baumer, M. Breugst, S. Choy, and T. Magedanz. GrassHopper: A universal agent platform based on OMG MASIF and FIPA standards. Technical report, IKV++ Technologies, 2000.

[13] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. Wiener, editor, *Advances in Cryptology (CRYPTO '99), Santa Barbara, California*, number 1666 in LNCS, pages 431–448. Springer-Verlag, 1999.

[14] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. *IEEE Communications Surveys*, 1(1):2–9, 1998.

[15] W. Binder, J. Hulaas, A. Villazon, and A. Vilaz. Portable resource control in the J-SEAL2 mobile agent system. In J. P. Muller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Quebec*, pages 222–223. ACM Press, 2001.

[16] W. Binder and B. Lichtl. Using a secure mobile object kernel as operating system on embedded devices. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 154–170. Springer-Verlag, 2002.

[17] W. Binder and V. Roth. Secure mobile agent systems using Java: Where are we heading ? In *Proceedings of the 17th ACM Symposium on Applied Computing, Madrid, Spain*, pages 115–119. ACM Press, 2002.

[18] W. Binder, G. D. Serugendo, and J. Hulaas. Towards a secure and efficient model for Grid computing using mobile code. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP) Workshop on Mobile Agent Systems: Agent Applications and New Frontiers, Malaga, Spain*, 2002.

[19] J. M. Bradshaw, S. Dutfield, P. Benoit, and J. D. Wooley. KAoS: Towards an industrial-strength open agent architecture. In J. Bradshaw, editor, *Software Agents*. MIT Press/AAAI Press, 1997.

[20] J. Bredin, R. T. Maheswaran, C. Imer, T. Başar, D. Kotz, and D. Rus. Computational markets to regulate mobile agent systems. *Autonomous Agents and Multi-Agent Systems*, 6(3):235–263, 2003.

[21] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile agents for distributed information retrieval. In M. Klusch, editor, *Intelligent Information Agents*. Springer-Verlag, 1999.

[22] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande, Palo Alto, California*, pages 97–105. ACM Press, 2001.

[23] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[24] C. Busch, V. Roth, and R. Meister. Perspectives on electronic commerce with mobile agents. In S. Rodinov and M. Vinogradov, editors, *Proceedings of the 11th Amaldi Conference on Problems of Global Security, Moscow*, pages 89–101. Nauka Publishers, 1998.

[25] C. Cachin, J. Camenisch, J. Kilian, and J. Muller. One-round secure computation and secure autonomous mobile agents. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of the 27th Colloquium on Automata, Languages and Programming (ICALP), Geneva, Switzerland*, number 1853 in LNCS, pages 512–523. Springer-Verlag, 2000.

[26] M. Carvalho and M. Breedy. Supporting flexible data feeds in dynamic sensor grids. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 171–185. Springer-Verlag, 2002.

[27] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. Simple Network Management Protocol. In *RFC 1157*, 1990.

[28] J. Claessens, B. Preneel, and J. Vandewalle. Can mobile agents do secure electronic transactions on untrusted hosts ? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, 2003.

[29] D. Cooney and P. Roe. Mobile agents make for flexible web services. In *Proceedings of the 9th Australian World Wide Web Conference*, 2003. http://ausweb.scu.edu.au/aw03/papers/cooney/paper.html.

[30] A. Corradi, R. Montanari, and C. Stefanelli. Mobile agents integrity in e-commerce applications. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop, Austin, Texas*, page 59. IEEE Computer Society Press, 1999.

[31] A. Das and Y. Gongxuan. A secure payment protocol using mobile agents in an untrusted host environment. In W. Kou, Y. Yesha, and C. J. Tan, editors, *Proceedings of the Second International Symposium on Electronic Commerce Technologies: ISEC 2001, Hong Kong*, number 2040 in LNCS, pages 33–41. Springer-Verlag, 2001.

[32] M. Delamaro and G. P. Picco. Mobile code in .NET: A porting experience. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 16–31. Springer-Verlag, 2002.

[33] D. DeRoure, W. Hall, H. Davis, and J. Dale. Agents for distributed multimedia information management. In *Practical Application for Intelligent Agents and Multi-Agent Systems, London, UK*, pages 91–102, 1996.

[34] M. D. Dikaiakos and G. Samaras. A performance analysis framework for mobile agent systems. In T. Wagner and O. F. Rana, editors, *International Workshop on Infrastructure for Agents, Multi-Agents and Scalable Multi-Agent Systems, Barcelona, Spain*, number 1887 in LNCS, pages 180–187. Springer-Verlag, 2001.

[35] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks: Optimal mobile agent protocols. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, Monterey, California*, pages 153–162. ACM Press, 2002.

[36] J. T. Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct ? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, California*, pages 160–171. IEEE Computer Society Press, 1998.

[37] W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Authentication and state appraisal. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the European Symposium on Research in Computer Security, Berlin*, number 1146 in LNCS, pages 118–130. Springer-Verlag, 1996.

[38] W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th NIST-NCSC National Information Systems Security Conference, Baltimore, MD*, pages 591–597, 1996.

[39] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75, 1991.

[40] S. Fenet and S. Hassas. A distributed intrusion detection and response system based on mobile autonomous agents using social insects communication paradigm. In K. Fischer and D. Hutter, editors, *Proceedings of the 1st Workshop on Security of Mobile Multiagent Systems (SEMAS 2001): Electronic Notes in Theoretical Computer Science*, volume 63. Elsevier, 2002.

[41] L. C. Ferreira and R. Dahab. Blinded-key signatures: Securing private keys embedded in mobile agents. In *Proceedings of the 17th ACM Symposium on Applied Computing, Madrid, Spain*, pages 82–86. ACM Press, 2002.

[42] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management, Gaithersburg, MD*, pages 456–463. ACM Press, 1994.

[43] FIPA Architecture Board. Agent management support for mobility specification. Technical report, Foundation for Intelligent Physical Agents, Geneva, 2000.

[44] Foundation for Intelligent and Physical Agents. FIPA ACL message representation in string specification. 2000. http://www.fipa.org/specs/fipa00070/.

[45] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the Grid: An open Grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure Work Group, Global Grid Forum*, 2002. http://www.globus.org/research/papers.html.

[46] J. Fou. Web Services and mobile intelligent agents. Technical report, Web Services Architect, 2001. http://www.webservicesarchitect.com/content/articles/fou02.asp.

[47] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. 1996. http://www.netscape.com/eng/ssl3/ssl-toc.html.

[48] M. Fukuda, Y. Tanaka, N. Suzuki, L. F. Bic, and S. Kobayashi. A mobile agent based PC Grid. In *Autonomic Computing Workshop: Fifth Annual International Workshop on Active Middleware Services*, page 142. IEEE Computer Society Press, 2003.

[49] S. Funfrocken and F. Mattern. Mobile agents as an architectural concept for internet-based distributed applications: The WASP project approach. In K. Steinmetz, editor, *Proceedings of Kommunikation in Verteilten Systemen (KiVS)*, pages 32–43. Springer-Verlag, 1999.

[50] D. Gambetta. Can we trust trust ? In D. Gambetta, editor, *Trust: Making and Breaking Cooperative Relations*. Basil Blackwell, 1990.

[51] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 2003.

[52] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, and Y. Nakamura. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. SAM Publishing, 2001.

[53] R. S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the 4th annual Tcl/Tk workshop, Monterey, California*, pages 9–23. IEEE Computer Society Press, 1996.

[54] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple language, mobile agent system. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 154–187. Springer-Verlag, 1998.

[55] R. S. Gray, D. Kotz, R. A. Peterson, and J. Barton. Mobile agent versus client/server performance: Scalability in an information retrieval task. In G. P. Picco, editor, *Proceedings of the 5th IEEE International Conference on Mobile Agents, Altanta, Georgia*, number 2240 in LNCS, pages 229–243. Springer-Verlag, 2001.

[56] A. Grimstrup, R. Gray, D. Kotz, M. Breedy, and M. Carvalho. Toward interoperability of mobile agent systems. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 106–120. Springer-Verlag, 2002.

[57] The Object Management Group. Mobile agent facility. Technical report, The Object Management Group, 2000. http://www.omg.org/technology/documents/formal/mobile-agent-facility.htm.

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

[59] D. Hagimont and L. Ismail. A performance evaluation of the mobile agent paradigm. In *Proceedings of the ACM conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado*, pages 306–313. ACM Press, 1999.

[60] X. Hannotin, P. Maggi, and R. Sisto. Formal specification and verification of mobile agent data integrity properties: A case study. In G. P. Picco, editor, *Proceedings of the 5th International Conference on Mobile Agents, Altanta, Georgia*, number 2240 in LNCS, pages 42–53. Springer-Verlag, 2001.

[61] A. Harter, A. Hooper, P. Steggeles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *Proceedings of the Conference on Mobile Computing and Networking (MOBICOM), Seattle, Washington*, pages 59–68. ACM Press, 1999.

[62] M. He, Nicholas R. Jennings, and H. F. Leung. On agent-mediated electronic commerce. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):985–1003, 2003.

[63] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[64] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 92–113. Springer-Verlag, 1998.

[65] F. Hohl. A framework to protect mobile agents by using reference states. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan*, pages 410–416. IEEE Computer Society Press, 2000.

[66] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, 2000.

[67] Y. Hu. Some thoughts on agent trust and delegation. In J. P. Muller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the 5th International Conference on Autonomous Agents, Montreal, Quebec*, pages 489–496. ACM Press, 2001.

[68] Q. Huai and T. Sandholm. Mobile agents in an electronic auction house. In O. Etzioni, J. P. Muller, and J. M. Bradshaw, editors, *Workshop on Mobile Agents in the Context of Competition and Cooperation: Proceedings of the 3rd International Conference on Autonomous Agents, Seattle, Washington*, pages 24–33. ACM Press, 1999.

[69] K. Jacobsen and D. Johansen. Ubiquitous devices united: Enabling distributed computing through mobile code. In *Proceedings of the 14th ACM Symposium on Applied Computing, San Antonio, Texas*, pages 399–404. ACM Press, 1999.

[70] W. Jansen. Countermeasures for mobile agent security. In *Computer Communications, Special Issue on Advances in Research and Application of Network Security.* Elsevier Science BV, 2000.

[71] W. Jansen, P. Mell, T. Karyginannis, and D. Marks. Applying mobile agents to intrusion detection and response. TR-6416. Technical report, National Institute of Standards and Technology, 1999.

[72] M. Kahn. CoABS Grid User's Manual. Technical report, Global InfoTEK Incorporated, 2000.

[73] G. Karjoth. Secure mobile agent-based merchant brokering in distributed marketplaces. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and 4th International Symposium on Mobile Agents (ASA/MA)*, number 1882 in LNCS, pages 44–56. Springer-Verlag, 2000.

[74] G. Karjoth and N. Asokan. Protecting the computation results of free-roaming agents. In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents, Stuttgart, Germany*, number 1477 in LNCS, pages 195–207. Springer-Verlag, 1998.

[75] G. Karjoth, D. B. Lange, and M. Oshima. A security model for Aglets. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 188–205. Springer-Verlag, 1998.

[76] N. Karnik. *Security in Mobile Agent Systems.* PhD thesis, Department of Computer Science and Engineering, University of Minnesota, 1998.

[77] S. Kent and R. Atkinson. Security architecture for the Internet protocol. In *RFC 2401 IETF*, 1998.

[78] F. Kon, B. Gill, M. Anand, R. Campbell, and M. D. Mickunas. Secure dynamic reconfiguration of scalable CORBA systems with mobile agents. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and 4th International Symposium on Mobile Agents (ASA/MA)*, number 1882 in LNCS, pages 86–98. Springer-Verlag, 2000.

[79] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–68, 1997.

[80] D. Kotz and R. S. Gray. Mobile agents and the future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, 1999.

[81] P. Kotzanikolaou, M. Burmester, and V. Chrissikopoulos. Secure transactions with mobile agents in hostile environments. In E. Dawson, A. Clark, and C. Boyd, editors, *Proceedings of the 5th Australasian Conference on Information Security and Privacy*, number 1841 in LNCS, pages 289–297. Springer-Verlag, 2000.

[82] P. Kotzanikolaou, G. Katsirelos, and V. Chrissikopoulos. Mobile agents for secure electronic transactions. In N.E. Mastorakis, editor, *Recent Advances in Signal Processing and Communications*, pages 363–368. World Scientic Engineering Society, 1999.

[83] H. Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece*, pages 108–115. ACM Press, 2000.

[84] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[85] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[86] A. Lomuscio, M. Wooldridge, and N. R. Jennings. A classification scheme for negotiation in electronic commerce. In F. Dignum and C. Sierra, editors, *Agent-Mediated Electronic Commerce: A European Agentlink Perspective*, number 1191 in LNCS, pages 19–33. Springer-Verlag, 2001.

[87] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of Tools*

and Algorithms for Construction and Analysis of Systems, number 1055 in LNCS, pages 147–166. Springer-Verlag, 1996.

[88] G. Lowe. Some new attacks on security protocols. In *9th IEEE Computer Security Foundations Workshop, County Kerry, Ireland*, pages 162–169. IEEE Computer Society Press, 1996.

[89] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[90] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

[91] F.S.E. Ltd. Failures-Divergence Refinement: FDR2 User Manual. Technical report, Formal Systems Europe, 1999. http://www.formal.demon.co.uk/fdr2manual.

[92] P. Maggi and R. Sisto. A configurable mobile agent data protection protocol. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia*, pages 851–858. ACM Press, 2003.

[93] T. Mandry, G. Pernul, and A. Rohm. Mobile agents on electronic markets: Opportunities, risks, agent protection. *International Journal of Electronic Commerce*, 5(2):47–60, 2000.

[94] P. Marques, R. Fonseca, P. Simoes, L. M. Silva, and J. G. Silva. A component-based approach for integrating mobile agents into the existing web infrastructure. In *Proceedings of the IEEE International Symposium on Applications and the Internet, Nara, Japan*, pages 100–108. IEEE Computer Society Press, 2002.

[95] P. Marques, L. M. Silva, and J. G. Silva. Going beyond mobile agent platforms: Component-based development of mobile agent systems. In *Proceedings of the 4th International Conference on Software Engineering and Applications, Las Vegas*, pages 119–125. IASTED/ACTA Press, 2000.

[96] S. Marsh. *Formalising Trust as a Computational Concept.* PhD thesis, Department of Mathematics and Computer Science, University of Stirling, 1994.

[97] S. McGrath, D. Chacon, and K. Whitebread. Intelligent mobile agents in the military domain. In *Proceedings of the Autonomous Agents Workshop on Agents in Industry*, 2000.

[98] B. J. McLennan. *Functional Programming: Practice and Theory.* Addison-Wesley, 1990.

[99] P. Mell, D. Marks, and M. McLarnon. A denial of service resistant intrusion detection architecture. *Computer Networks Journal*, 34:641–658, 2000.

[100] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100(1):1–77, 1992.

[101] D. Milojicic, M. Breugst, I. Busse, and J. Campbell. MASIF: The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Conference on Mobile Agents, Stuttgart, Germany*, number 1477 in LNCS, pages 50–67. Springer-Verlag, September 1998.

[102] Dejan Milojicic. Trend Wars: Mobile agent applications. *IEEE Concurrency*, 7(3):80–90, 1999.

[103] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland*, pages 109–114. ACM Press, 1996.

[104] B. Misztal. *Trust in Modern Societies.* Polity Press, Cambridge MA, 1996.

[105] L. Moreau. Distributed directory service and message routing for mobile agents. *Science of Computer Programming*, 39(2-3):249–272, 2001.

[106] L. Moreau, N. Gibbins, D. DeRoure, S. El-Beltagy, W. Hall, G. Hughes, D. Joyce, S. Kim, and D. Michaelides. SoFAR with DIM agents: An agent framework for distributed information management. In *Fifth International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents, Manchester, UK*, pages 369–388, 2000.

[107] L. Moreau and C. Queinnec. Distributed computations driven by resource consumption. In *Proceedings of IEEE International Conference on Computer Languages (ICCL'98), Chicago, Illinois*, pages 68–77. IEEE Computer Society Press, 1998.

[108] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI '96), Seattle, Washington*, pages 229–243. ACM Press, 1996.

[109] S. K. Ng and K. W. Cheung. Intention spreading: An extensible theme to protect mobile agents from read attack hoisted by malicious hosts. In J. Liu and N. Zhong, editors, *Intelligent Agent Technology: Systems, Methodologies, and Tools: Proceedings of the first Asia-Pacific Conference on Intelligent Agent Technology*, pages 406–415. World Scientific, 1999.

[110] H. S. Nwana, J. Rosenschein, T. Sandholm, and C. Sierra. Agent-mediated electronic commerce: Issues, challenges and some viewpoints. In K. P. Sycara and M. Wooldridge, editors, *Proceedings of the 2rd International Conference on Autonomous Agents, Minneapolis, Minnesota*, pages 189–196. ACM Press, 1998.

[111] S. Oaks. *Java Security (2nd ed)*. O'Reilly, 2001.

[112] L.A.G. Oliveira, P.C. Oliveira, and E. Cardozo. An agent-based approach for quality of service negotiation and management in distributed multi-media systems. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1st International Workshop on Mobile Agents, Berlin*, number 1219 in LNCS, pages 1–12. Springer-Verlag, 1997.

[113] T. Papaioannou. *On the structuring of distributed systems: The argument for mobility*. PhD thesis, Department of Computer Science, Loughborough University, 2000.

[114] O. F. Rana and K. Stout. What is scalability in multi-agent systems ? In C. Sierra, M. Gini, and J. S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents, Barcelona, Spain*, pages 56–63. ACM Press, 2000.

[115] Keith Regan. Web services get practical. In *E-Commerce Times*, 2003. http://www.ecommercetimes.com/perl/story/21094.html.

[116] K. Reinholtz. Java will be faster than C++. *ACM SIGPLAN*, 35(2):25–28, 2000.

[117] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[118] P. Resnick and R. Zeckhauser. Trust among strangers in Internet transactions: Empirical analysis of eBay's reputation system. In M. R. Baye, editor, *The Economics of the Internet and E-Commerce*, volume 11 of *Advances in Applied Microeconomics*. Elsevier Science, 2002.

[119] J. Riely and M. Hennessy. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.

[120] V. Roth. Mutual protection of co-operating agents. In J. Vitek and C. Jensen, editors, *Secure internet programming: Security issues for mobile and distributed objects*, number 1603 in LNCS, pages 275–285. Springer-Verlag, 1999.

[121] V. Roth. On the robustness of some cryptographic protocols for mobile agent protection. In G. P. Picco, editor, *Proceedings of the 5th IEEE International Conference on Mobile Agents, Atlanta, Georgia*, number 2240 in LNCS, pages 1–14. Springer-Verlag, 2001.

[122] V. Roth. Empowering mobile software agents. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 47–63. Springer-Verlag, 2002.

[123] V. Roth. Programming Satan's agents. In K. Fischer and D. Hutter, editors, *Proceedings of the 1st Workshop on Security of Mobile Multiagent Systems (SEMAS 2001): Electronic Notes in Theoretical Computer Science*, volume 63. Elsevier, 2002.

[124] V. Roth and M. Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems, Dallas, Texas*, pages 435–442. IEEE Computer Society Press, 2001.

[125] V. Roth and J. Peters. A scalable and secure global tracking service for mobile agents. In G. P. Picco, editor, *Proceedings of the 5th IEEE International Conference on Mobile Agents, Atlanta, Georgia*, number 2240 in LNCS, pages 169–181. Springer-Verlag, 2001.

[126] A. Sahai and C. Morin. Enabling a mobile network manager through mobile agents. In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents, Stuttgart, Germany*, number 1477 in LNCS, pages 249–260. Springer-Verlag, 1998.

[127] A. Sahai and C. Morin. Mobile agents for location independent computing. In *Proceedings of the 14th ACM Symposium on Applied Computing, San Antonio, Texas*, pages 408–409. ACM Press, 1999.

[128] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 44–60. Springer-Verlag, 1998.

[129] I. Satoh. MobiDoc: A framework for building mobile compound documents from hierarchical mobile agents. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and 4th International Symposium on Mobile Agents (ASA/MA)*, number 1882 in LNCS, pages 113–125. Springer-Verlag, 2000.

[130] I. Satoh. Physical mobility and logical mobility in ubiquitous computing environments. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 186–202. Springer-Verlag, 2002.

[131] I. Satoh. Configurable network processing for mobile agents on the Internet. *Cluster Computing (The Journal of Networks, Software Tools and Applications)*, 7(1):73–83, 2004.

[132] K. Schelderup and J. Ølnes. Mobile agent security: Issues and directions. In H. Zuidweg, editor, *Proceedings of the 6th International Conference on Intelligence and Services in Networks, Barcelona, Spain*, number 1597 in LNCS, pages 155–167. Springer-Verlag, 1999.

[133] F. B. Schneider. Towards fault-tolerant and secure agentry. In M. Mavronicolas and P. Tsigas, editors, *11th International Workshop on Distributed Algorithms, Saarbrucken, Germany*, number 1320 in LNCS, pages 1–14. Springer-Verlag, 1997.

[134] V. Shoup. Practical threshold signatures. In B. Preneel, editor, *Advances in Cryptology (EUROCRYPT 2000)*, number 1807 in LNCS, pages 207–220. Springer-Verlag, 2000.

[135] D. Song, S. Berezin, and A. Perrig. Athena, a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1):47–74, 2001.

[136] W. Stallings. *Cryptography and Network Security: Principles and Practice (3rd ed.)*. Prentice Hall, 2002.

[137] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T.Groth, G. A. Hill, and R. Jeffers. Strong mobility and fine grained resource control in NOMADS. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and 4th International Symposium on Mobile Agents (ASA/MA)*, number 1882 in LNCS, pages 2–15. Springer-Verlag, 2000.

[138] V. Swarup and J. T. Fabrega. Trust: Benefits, models and mechanisms. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 3–18. Springer-Verlag, 1999.

[139] H. Tai and K. Kosaka. The Aglets project. *Communications of the ACM*, 42(3):100–101, 1999.

[140] H. K. Tan and L. Moreau. Trust relationships in a mobile agent system. In G. P. Picco, editor, *Proceedings of the 5th IEEE International Conference on Mobile Agents, Atlanta, Georgia*, number 2240 in LNCS, pages 15–30. Springer-Verlag, 2001.

[141] H. K. Tan and L. Moreau. Certificates for mobile code security. In *Proceedings of the 17th ACM Symposium on Applied Computing, Madrid, Spain*, pages 76–81. ACM Press, 2002.

[142] H. K. Tan and L. Moreau. Extending execution tracing for mobile code security. In K. Fischer and D. Hutter, editors, *Proceedings of the 2nd International Workshop on Security of Mobile MultiAgent Systems (SEMAS 2002): The 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy*, pages 51–59. ACM Press, 2002.

[143] H. K. Tan and L. Moreau. Mobile Code for Key Propagation. In K. Fischer and D. Hutter, editors, *Proceedings of the 1st Workshop on Security of Mobile Multiagent Systems (SEMAS 2001): Electronic Notes in Theoretical Computer Science*, volume 63. Elsevier, 2002.

[144] Y. H. Tan and W. Thoen. Formal aspects of a generic model of trust for electronic commerce. *Source Decision Support Systems*, 33(3):233–246, 2002.

[145] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, 1997.

[146] P. Thati, Po-Hao Chang, and Gul Agha. Crawlets: Agents for high performance Web search engines. In G. P. Picco, editor, *Proceedings of the 5th IEEE International Conference on Mobile Agents, Atlanta, Georgia*, number 2240 in LNCS, pages 119–134. Springer-Verlag, 2001.

[147] A. Tripathi, T. Ahmed, V. Kakani, and S. Jaman. Distributed collaborations using network mobile agents. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and*

4th International Symposium on Mobile Agents (ASA/MA), number 1882 in LNCS, pages 126–137. Springer-Verlag, 2000.

[148] G. Vigna. Cryptographic traces for mobile agents. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 137–153. Springer-Verlag, 1998.

[149] G. Vigna, B. Cassel, and D. Fayram. An instrusion detection system for Aglets. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 64–77. Springer-Verlag, 2002.

[150] A. Villazon and W. Binder. Portable resource reification in Java-based mobile agent systems. In G. P. Picco, editor, *Proceedings of the 5th International Conference on Mobile Agents, Altanta, Georgia*, number 2240 in LNCS, pages 213–228. Springer-Verlag, 2001.

[151] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *Proceedings of the Workshop on Internet Programming Languages*, number 1686 in LNCS, pages 47–77. Springer-Verlag, 1999.

[152] D. Volpano and G. Smith. Language issues in mobile program security. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in LNCS, pages 25–43. Springer-Verlag, 1998.

[153] C. Wagner and E. Turban. Are intelligent e-commerce agents partners or predators ? *Communications of the ACM*, 45(5):84–90, 2002.

[154] R. Wahbe, S. Lucco, and T. Anderson. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM Press, 1993.

[155] W. Walker, P. Lamere, and P. Kwok. FreeTTS - A performance case study. TR-2002-114. Technical report, Sun Microsystems, 2002.

[156] Y. Wang and X. Pang. Security and robustness enhanced route structures for mobile agents. *Mobile Networks and Applications*, 8(4):413–423, 2003.

[157] U. G. Wilhelm, S. Staamann, and L. Buttyan. Introducing trusted third parties to the mobile agent paradigm. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 469–489. Springer-Verlag, 1999.

[158] S. Willmot and B. Faltings. Using adaptation and organisational knowledge to coordinate mobile agents. In D. Kotz and F. Mattern, editors, *Proceedings of the 2nd International Symposium on Agent Systems amd Applications and 4th International Symposium on Mobile Agents (ASA/MA)*, number 1882 in LNCS, pages 138–150. Springer-Verlag, 2000.

[159] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M.Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1st International Workshop on Mobile Agents, Berlin*, number 1219 in LNCS, pages 86–97. Springer-Verlag, 1997.

[160] H. C. Wong and K. Sycara. Adding security and trust to multi-agent systems. In R. Falcone, C. Castelfranchi, Y. H. Tan, and B. Firozabadi, editors, *Workshop on Deception, Fraud and Trust in Agent Societies: Proceedings of the 3rd International Conference on Autonomous Agents, Seattle, Washington*, pages 149–161. ACM Press, 1999.

[161] M. Woodside. Scalability metrics and analysis of mobile agent systems. In T. Wagner and O. F. Rana, editors, *Infrastructure for Agents, Multi-Agents, and Scalable Multi-Agent Systems*, number 1887 in LNCS, pages 234–245. Springer-Verlag, 2001.

[162] R. Yahalom, B. Klein, and T. Beth. Trust relationships in secure systems: A distributed authentication perspective. In *Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California*, pages 150–164. IEEE Computer Society Press, 1993.

[163] B. S. Yee. A sanctuary for mobile agents. In J. Vitek and C. Jensen, editors, *Secure Internet Programming : Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 261–274. Springer-Verlag, 1999.

[164] B. Yu and M. P. Singh. Detecting deception in reputation management. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia*, pages 73–80. ACM Press, 2003.

[165] N. Zaini and L. Moreau. Mobile intermediaries supporting information sharing between mobile users. In N. Suri, editor, *Proceedings of the 6th IEEE International Conference on Mobile Agents, Barcelona, Spain*, number 2535 in LNCS, pages 121–137. Springer-Verlag, 2002.

[166] S. Zhong and Y. R. Yang. Verifiable distributed oblivious transfer and mobile agent security. In *Joint Workshop on Foundations of Mobile Computing: Proceedings of the International Conference on Mobile Computing and Networking, San Diego, California*, pages 12–21. ACM Press, 2003.