TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

PROPOSAL

Introduction

The client-server paradigm introduced the concept of remote procedure calls as the basis for distributed computing (Comer & Stevens, 1999). In traditional approaches such as these, distributed information access brought the data to the point of computation. This however, required crucial conditions such as a continuous link between the client and the server when requesting information.

The next step from the client-server technology saw the introduction of mobile agents. The concept of a mobile agent brings the computation to the data and the mobility and autonomy attributes make permanent connections unnecessary. The notion of a mobile agent arose from that of a software agent.

Software agent can be defined as software components that communicate by exchanging messages in an agent communication language (Genesereth & Ketchpel, 1994). The concept of software agents is based on objects as recognized in the object-oriented environment and they can adopt several different forms such as stationary agents, intelligent agents and mobile agents.

According to an IBM white paper (1998) intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy and in doing so, employ some knowledge or representation of the user's goals or desires

A mobile agent can be defined as an autonomous program that moves between networks to

take advantage of the services supplied by stationary agents (Karnik, 2000). The stationary agent resides on a specific host with the inability to move about but with the ability to offer services or perform tasks on behalf of its owner. A mobile agent, on the other hand carries along its complete implementation and interacts with a host system as well as other mobile and stationary agents. Mobile agents have a distinct computational advantage by moving computation close to the resources they need to access, hence reducing network communication, bandwidth and latency.
Mobile agents are deployed for various purposes. Some of the most common purposes include information searching, filtering and retrieving applications, low level network maintenance, testing, fault diagnosis and the dynamic upgrading of existing services, concluding certain e-commerce deals or negotiating with other mobile or stationary agents (Karnik, 2000; Tripathi *et al*., 2001).

An agent is generally regarded as *mobile* when its execution can be interrupted (usually briefly), before it migrates to a new host and is then resumed after the transportation to the new runtime environment. Mobile agents do not transport themselves, but depend on the mobile agent system to move their binary images between execution layers over a variety of media (Ylitalo, 2000).

Mobile agents open several new possibilities for conducting business in a network and especially the Internet environment, but they also introduce a new dimension of security issues. In fact, full-scale adoption of mobile agent technology in untrustworthy network environments, such as the Internet, has been hampered and delayed by several security complexities (Montanari, 2001). Jansen (2000) categorises threats in the mobile agent environment into four distinct classes, namely threats imposed by the mobile agent to the host, threats imposed by a mobile agent host to a mobile agent, threats imposed by a mobile agent to another mobile agent and threats imposed from other entities to mobile agents.

A malicious hosting node can launch several types of security attacks on the mobile agent and divert its intended execution towards a malicious goal or alter its data or other information in order to benefit from the agent's mission (Sander & Tschudin, 1998). On the other hand, agents can also have malicious intentions against hosts or against other agents. For example, a virus or a trojan horse can masquerade as a mobile agent and then attack the hosting node’s resources. An agent can also interfere with, or hijack other agents so that they cannot carry out their tasks or become corrupted (Zeltser, 2000). Thirdly, network entities outside the hosting node can launch attacks against a mobile agent in transit, or against the mobile agent system and steal its secrets (such as an encryption key) or corrupt its integrity (Jansen, 1999).

Ylitalo (2000) categorised threats imposed by the mobile agent on the host as damage, denial of service, breach of privacy, harassment and social engineering. Typical examples of agents with these kinds of behaviours are found in computer viruses. Reliable security measures to counter these threats have been (and still are) extensively researched and proposed and include, for example, solutions like *software-based fault isolation* (Whabe *et al.*, 1993) and *state appraisal* (Farmer *et al.*, 1996). In general, research on host protection is based on security techniques in the arena of *Computer Security*. The focus of our research is on the malicious host problem, where the agent is vulnerable to an attack of a malicious hosting node.

Problem Statement

A complete security solution for mobile agent systems would typically insist that the execution environment itself be secured and also that the executing code (mobile agent itself) be deployed in the execution environment and be designed and implemented according to specific security requirements. As mentioned in 1.1, security measures for protecting hosts against malicious agents are well researched and we consider it to fall outside the scope of this thesis.

The *mobility* attribute of a mobile agent, which implies that these agents are executed in an open environment, introduces new security threats to mobile agent applications. Countermeasures for these threats are in the initial phases of research. In fact, the type of threats that are imposed by a malicious host on a mobile agent cannot simply be resolved by *Computer Security* solutions. This is due to the fact that the mobile agent carries along its code, data, attributes and state and as a result presents a different challenge to current security solutions.

To overcome the drawbacks associated with mobile agent technology, it is essential that mobile agent systems must have an integrated security framework, which offers different security techniques to provide an overall secure system. Current security solutions from the perspective of a malicious host are largely based on the security provided by the underlying operating system and/or the programming language. The research question that is addressed in this study deals with how a security framework can be constructed to resolve the malicious host problem without introducing high costs or restraining the mobile agent's mobility, autonomy or performance.

1.3 Solution Approach

The purpose of this study is to propose a mobile agent security framework that could provide a basis for the secure implementation of mobile agents. A combination of research techniques is used in this undertaking namely, literature reviews, constructing arguments and propositions, prototyping, experimentation and testing, as well as contextual evaluation.

In the context of this research a technical survey will be conducted on the existing literature on mobile agent systems and mobile agent security with specific reference to the malicious host problem. Through this survey, essential information will then be extracted to identify the most salient characteristics in existing security frameworks and mobile agent systems and also isolate the drawbacks, which up to this point still leave a mobile agent vulnerable for malicious host attacks.

From the studied information, classifications will be proposed to categorise different classes of threats and the possible countermeasures will be evaluated. Further arguments and propositions will then be constructed to propose the security framework from the perspective of protecting the mobile agent as an entity when executing on a node in a mobile agent system. A prototype that has been constructed, as a proof-of-concept of the proposed framework, will then be described. The undertaking also involves evaluation of a suitable implementation infrastructure, prototype design, test case construction and evaluation, which will also be discussed.

A prototype will be constructed that serves as a vehicle for experimentation, during which the practical implementation of the framework will be tested and evaluated. Thereby, different current security solutions will be combined to find the best emulsion for different scenarios. These will then be evaluated within the context of the depicted scenario.

1.4 Research Context

As suggested earlier, the security threats associated with mobile agent technology hamper its wide acceptance to such an extent that there has been a notable decline in mobile agent publications in the late nineties. However, literature shows that in the past two or three years, there has been a considerable rise in the number of research publications involving mobile agent technology. The renewed interest is largely due to the recent materialisation of the semantic web (Berners-Lee *et al.*, 2001, Kagal *et al.*, 2003), as well as the continued exponential growth of Internet applications and the establishment of open standards for these applications. Yet, many of the specific problems associated with mobile agent technologies have not been resolved and now it is almost a matter of urgency to address the remaining problems.

The current demand for secure mobile agent applications can be seen in the increase of e-commerce transactions and Internet information retrieval requirements over the past few years. It is within this context of relevance that our research resides.

1.5 Scope of Study

The following issues do not fall within the scope of this research undertaking:

The study does not consider malicious agent or malicious entity security problems.

The purpose of the study is not to suggest new countermeasures to the malicious host problem, but rather to propose a complete security framework where various countermeasures reside. Therefore, the study uses available countermeasure techniques for its implementation and experimentation, instead of proving the feasibility of new or un-implemented countermeasures.

1.6 Synopsis

Chapter 2 provides state-of-the art information on mobile agent systems. The types of threats imposed by malicious hosts on mobile agents are investigated and categorised. Chapter 3 covers the different countermeasures that have been proposed by researchers throughout the field, as well as the analysis thereof in relation to the identified threats discussed in Chapter 2. Chapter 4 provides information on the existing implemented and proposed mobile agent models, frameworks and architectures. Current mobile agent systems and developed applications are also detailed in this chapter. In Chapter 5, the criteria for an integrated security framework is investigated by considering current agent standards as well as the general challenges to security solutions for mobile agent technologies. The various requirements for a secure mobile agent framework are considered in Chapter 6, which also particularizes the design of the proposed security framework. A prototype and implementation details of, as well as experimentation efforts on the framework are described Chapter 7. In conclusion, Chapter 8 presents an analysis of implementation results and lessons learnt. Furthermore, recommendations are also made regarding avenues for future work based on this analysis.

Addendums are added to the thesis in order to provide additional information with regards to the implementation specifics of the framework. The detailed security policy created for the implementation and testing of the framework (as implemented on every remote host to be visited by the agent) is presented in Addendum A. Addendum B provides a listing of a mobile agent created for information retrieval purposes, while the source code for a mobile agent that not only retrieves information on specified hosts but also conducts certain degrees of computation, is provided in Addendum C. The creation of audit information to detect possible malicious modifications is achieved by including a history of the hosts visited by the mobile agent (Addendum D) as well as the authentication of results retrieved at the different hosts (Addendum E).

CHAPTER 2

MOBILE AGENT PARADIGM

2.1 Introduction

The mobile agent paradigm stems from two distinct approaches, namely the distributed systems environment and the distributed artificial intelligence environment. Due to these environments being a recent research area, several different descriptions and architectures exist to describe this paradigm. The purpose of this chapter is twofold, namely to describe the essence of a mobile agent as well as a mobile agent system and secondly, to introduce the specific security threats relating to this paradigm, as this forms the basis of our research.

2.2 Background

In order to provide a description of mobile agents, it is important to review their history. As mentioned, mobile agent research has been influenced by at least two important directions of study, namely *distributed artificial intelligence* and *distributed systems.* Each of these has its own interest in mobile agent research and therefore brings a unique understanding and corresponding influence in the field. On the one hand agent-based systems is a niche area of interest in artificial intelligence (D'Inverno & Luck, 2001), while on the other hand the efforts of Picco (1998) and Papaioannou (2000) show that the mobile agent system paradigm developed independently from distributed artificial intelligent agent research as a result of shortcomings in the client-server paradigm. This was due to demands placed by technologies such as Remote Procedure Call (RPC).

Fugetta *et al.* (1998) summarise this position and proposes the mobile agent as a refinement of distributed applications in that it utilises network connections uniquely.

They furthermore identify the mobile code as an *executing unit* that is composed of execution state, code segment and data space. Three types of mobile code systems are identified: *remote evaluation, code on demand* and *mobile agent system.* Of these three the mobile agent system is described as unique due to its capability to transport code and data space to a different location on a network.

2.3 Advantages and Uses

The mobile agent paradigm introduces several advantages. Some of the most salient include:

- The mobile agent is not bound to the system where it begins execution. A mobile agent has the unique ability to transport itself from one system in a network to another. The ability to travel, allows a mobile agent to move to a system that contains an object with which the agent wants to interact and then to take advantage of being in the same host or network as the object (Lange, 1998).
- Both bandwidth limitations as well as the support for disconnected operation capabilities are eminent problems experienced in the wireless and mobile environments. By moving the computation to the host and as a result decreasing the amount of packets on the network, mobile agents can assist in alleviating these problems (Suri *et al.*, 2000).
- Mobile agents provide the ability to conduct intelligent information retrieval, such as retrieving appropriate information from a number of hosts as well as performing some computations (Aerts *et al.*, 2002).
- Mobile agents overcome network latency in that real-time systems need to respond to changes in their environment. Controlling such systems through a large network involves significant latencies for which mobile agents can offer a solution (Lange, 1998).

These advantages open up several applications that will benefit from the use of mobile agent systems. Typical applications range from information searching, filtering and retrieval to electronic commerce on the Web where they act as personal assistants for their owners. Mobile agents can also be used in network management maintenance, testing, fault diagnosis, and for dynamically upgrading the capabilities of existing services (Tripathi *et al.*, 1999). Other uses include workflow management, air traffic control, information retrieval management and education (Grimley & Monroe, 1999).

2.4 Software Agents

Mobile agents form a special subset of software agents. Software agents and their specific attributes therefore deserve a brief review. There are many definitions in literature to define a software agent (for example see Genesereth & Ketchpel (1994); Smith *et al*. (1994); Franklin & Graesser (1997)) For the purpose of this research we consider the definition of the Object Management Group (OMG).

According the Object Management Group (2000) a software agent is defined as an autonomous software entity that can interact with its environment. Some of the properties that agents may possess in various combinations are as follows:

Autonomy: It can act without direct external intervention and is able to initiate activities.

Interactivity: Agents can communicate with the environment as well as other agents.

Adaptivity: A software agent can respond to other software agents and/or its environment.

Sociability: A software agent can act sociably by being companionable or friendly.

Mobility: Contains the ability to transport itself from one environment to another.

Proactively: It is goal oriented and does not simply react to the environment.

Intelligence: According to Wooldridge (2002) intelligence implies the inclusion of at least three distinct properties, namely reactivity, pro-activeness and social-ability.

Rationality: It is able to choose an action based on its internal goals.

Coordinative-ness: Able to complete a task in a certain environment together with other agents.

Cooperativeness: Able to coordinate with other agents to reach a common goal.

Some other properties (that are not included in the scope of this research) include unpredictability, accountability, ruggedness, competitiveness, etc. For more information on these properties see Etziani & Weld (1995).

According to these properties, different types of software agents can be specified such as Autonomous agents, Interactive agents, Adaptive agents, Mobile agents, Coordinative agents, Intelligent agents and Wrapper agents. See Object Management Group (2000) and Bradshaw (1997) for detailed information on the different types of software agents. Two of these types namely *stationary agents* and *mobile agents* need to be described in more detail for the purpose of this research.

2.4.1 Stationary Agent

A stationary agent executes only on the system where it begins execution. If it needs information that is not on that system, or needs to interact with an agent on a different system, it typically uses a communication mechanism such as remote procedure calls (Lange, 1998).

The purpose of a stationary agent is to provide support and services to other agents such as mobile agents and assist them in keeping their objectives. Stationary agents remain resident at a single platform, while mobile agents are capable of suspending activity on one platform and move to another, where they resume execution (Jansen, 2000).

2.5 Mobile Agent Context

Being a type of software agent and essential to the existence of mobile agent systems, the description and features of a mobile agent are set out below.

2.5.1 Description

A mobile agent can be defined as a program that represents a user in a network and that is capable of migrating autonomously from node to node, performing computations or tasks on behalf of that user (Tripathi *et al*, 1999).

2.5.2 Components

A mobile agent as defined consists of a number of required components, which are identified as the following:

- Code: the program that defines the behaviour or required tasks of the agent (Fugetta *et al*., 1998).
- State: This is data relating to the technical execution of the mobile agent such as stack and program pointer, which enable the mobile agent to resume its activities after migrating to another host (Fugetta *et al*., 1998).
- Data: This data relates to the results of the mobile agent's purpose or tasks. The mobile agent migrates the network, executing instructions as it goes along. These instructions in certain cases may produce results, which are carried with the mobile agent. Data can be divided into two groups, namely *initial data* that the mobile agent takes along for reaching its goal and data as *generated or received* on the different nodes (which can be used in further computations or saved until arrival at initiator or home node) (Lange, 1998).
- Itinerary: The path that defines the agent's journey between the different hosts is called the itinerary of the agent. This can be determined during creation by the mobile agent's creator or it can be determined at run-time according to specific input variables as received during computation (Jansen, 2001).
- Unique Identifier: A unique identity is dependant on an algorithm that will during the creation of the mobile agent; give it a unique identification for address or navigation purposes. This identification will be carried with the mobile agent in the form of data (Jansen, 2001).

2.5.3 Features

The following features form part of a mobile agent and is regarded as essential to its definition and objectives:

- *Mobility*: Mobility allows an agent to move between different mobile agent platforms (Jansen, 2000). Two types of mobility are defined, namely strong mobility and weak mobility. After being dispatched, the mobile agents become independent from the creating process and can operate asynchronously and autonomously (Lange, 1998).
- *Autonomy*: Mobile agents have control over the actions they initiate (Sundsted, 1998). They also adapt dynamically and have the ability to monitor their execution environment and react autonomously to changes (Lange, 1998).

- *Security*: A mobile agent has to protect its code, state and data from malicious entities such as hosts and other mobile agents (Jansen, 2000).
- *Reactivity*: A mobile agent reacts on environmental changes, such as determining a new host to migrate to, if for example, the predefined next host specified on the itinerary does not exist or is unavailable (Hoffmann *et al.*, 2002).
- *Pro-activeness*: Mobile agents do not just react on changes of the environment, but are able to act before changes in the environment occur (Hoffmann *et al.*, 2002).
- *Persistence*: A mobile agent persists in obtaining its objectives without interference such as denial of services from the environment (Hoffmann *et al.*, 2002).
- *Goal oriented*: A mobile agent is designed with certain objectives in mind. A common example is where a mobile agent is designed to retrieve the cheapest airline fares between specified destinations (Sundsted, 1998).
- *Communicativeness*: Mobile agents need to communicate with a number of entities in order to reach their design goals. These entities may for example include the mobile agent platform, different hosts and stationary agents (Sundsted, 1998).

2.5.4 Mobile Agent Lifecycle

The lifecycle of a mobile agent consists of a number of phases that are depicted in Figure 2.1 as well as in the description below:

- Creation: The mobile agent is created and contains its objectives, goals, code and initial data.
- Initiate: The mobile agent is initiated at the home mobile agent platform. For the purpose of this research, the home mobile agent platform is called the *local host*. The local host is seen as the first address on the mobile agent's itinerary as well as the last address, being that the mobile agent always returns home in order to convey its results.
- Repeat for a number of *n-2* mobile agent platforms on the itinerary:
  - Request for migration: The mobile agent issues a request to its host to be packed and sent to the next specified host. A host in this instance may refer to the local host or a *remote host*, which is a platform where the mobile agent is executed in order to pursue its design goals.

  - Migration: The mobile agent is sent from the current host to the next host specified in its itinerary.
  - Unpacked and acceptance: The mobile agent is presented in a form suitable for execution on the remote host. This also includes a number of processes to which the mobile agent is subjected before acceptance on the remote host, such as authentication and authorisation procedures.
  - Execution: As soon as the mobile agent arrives on the next remote host it starts execution according to its design goals. During this phase the mobile agent makes use of resources provided on the remote host in order to aid in its goals, such as the retrieving of data and the computation of results. The execution of the mobile agent also includes making use of the services provided by the remote host such as communication mechanisms.
- Return to local host: The mobile agent requests migration to the local host, where it conveys the results of its journey. It is not mandatory that the mobile agent returns to its local host, but can be self-exterminating after its specific task or tasks have been completed.

Figure 2.1: The mobile agent lifecycle

2.6 Mobile Agent System

The most important entity in the mobile agent paradigm is the mobile agent itself. However, it needs a mobile agent system as a basis of its existence. A mobile agent system is seen as a distributed system consisting of several components that are outlined below.

2.6.1 Mobile Agent System Components

A mobile agent system consists of a number of elements in order to be identified as such a system.

Host

The host is the physical machine connected to other hosts through a network. The host is responsible for providing resources such as processing power and information through a protected mobile agent execution environment to the visiting mobile agent (Karnik, 1998).

Mobile agent platform

An *agent platform* acts as the interface between the mobile agents and the services provided by the host (Tripathi *et al*., 1999). It also provides the computational environment in whichan agent operates. The agent platform is responsible for hosting and executing any mobile agent that arrives over the network and for providing primitive operations to agent programmers such as migration, communication and the accessing of host resources. A mobile agent platform can also be specialised to provide application-specific services and keeps track of status information regarding the mobile agents, such as active/inactive status, error conditions and resource consumption (Karnik, 1998).

Besides furnishing the engine on which an agent executes its code, other services offered by a mobile agent platform include the capability for an agent to clone itself, to spawn or create new agents, to terminate any spawned agents, to locate other agents at the platform or a platform elsewhere, to send messages to other agents, and to relocate the agent to another platform (Jansen, 1999). One or more hosts may comprise an agent platform and an agent platform may support multiple computational environments or meeting places, where agents can interact (Jansen & Karygiannis, 1999).

For the purpose of this research we use the term host to refer to both the host itself as well as the mobile agent platform.

2.6.2 Mobile Agent Platform Features / Tasks

An agent platform is responsible for the following:

Migration

The primary identifying characteristic of a mobile agent is its ability to autonomously migrate from host to host. Migration involves the transfer of different components of a mobile agent between different hosts. The support for agent mobility is thus a fundamental requirement of the agent's infrastructure. If a mobile agent requests to migrate, then the host must deactivate the mobile agent, capture its state and transmit it to the next host as specified by the itinerary of the mobile agent. The destination host is responsible for restoring the state of the mobile agent, as well as reactivating it (Karnik, 1998).

Migration can either result from a hard coded itinerary designed by a programmer, or from the reactiveness property of the mobile agent in response to its environment. Braun *et al.* (2000) describe migration in terms of a *mobility model*. A mobility model defines abstractions as well as the behaviour of the host when mobile agents migrate to a new host. The mobility model views migration capabilities from three different perspectives, namely how the programmer implements migration; which migration strategy the mobile agent chooses; and the influence of the transmission strategy on the underlying network.

Programmers generally address mobility in two ways: *weak mobility* in which the agent's state is represented in program-defined data structures, allowing migration only at specific points in the agent code; and *strong mobility* which captures the agent's state at the underlying thread or process and allows migration at any point in the agent's execution (Tripathi *et al*., 2001). With weak mobility the agent restarts on the new host from the beginning of its data, while strong mobility allows the agent to continue execution from the point in its instructions when it was transferred (Horvat *et al*., 2000). If the thread of control needs to be retained in an agent system supporting weak mobility, additional programming is required to save the execution state manually. In a system with strong mobility, migration is completely transparent to the migrated program, reducing programming effort as well as the size of the transported code (Picco, 2001).

The mobile agent is able to choose a strategy for migration, such as *pull-code* where the code is downloaded by the executing host from a specified source and *push-code* where the code is sent in advance to the executing host. In the pull-code strategy, the code of the mobile agent is not sent together with the mobile agent's data, but is loaded dynamically by the new host after migration. The class files can be loaded individually or as a package once a specific class file is requested. In the push-code strategy, the code can be sent to all hosts as specified on the itinerary (if the itinerary is known), or just to the next remote host (Braun *et al*., 2000).

The transmission strategy defines the way the mobile agent is physically transmitted to the destination host. It states how the data and code is transmitted on a protocol level, such as TCP/IP or UDP (Braun *et al*., 2000).

Communication

Sustaining the idea of a mobile agent system, communication primitives are (or should be) embedded in each mobile agent. These primitives are necessary to enable inter-agent and agent-to-host communication. Different mechanisms can be used in order to establish a communication service, such as *message passing* or *method invocation* (Karnik, 1998). According to Ford & Karmouch (1997), the mobile agent must also be able to consult with its owner in the case of the mobile agent requesting additional information, before migrating to a next host or making a decision. Stationary agents on the local host should also be able to communicate with remote agents.

Agent monitoring and control

The owner of a mobile agent (thus the local host) must be able to monitor the agent's status while executing on a remote host. The local host must also be able to perform remote control tasks such as terminating the agent, or recalling it to transfer back to the local host (Karnik, 1998).

Resource management

Agents on a remote host will request resources available on the remote host such as memory, CPU time and information contained in databases. The remote host therefore has to manage these requests by for example checking authorisations and quotas as contained in the security policy of the host as well as the agent (Fünfrocken & Mattern, 1999).

Execution support

Certain classes and libraries are not necessarily built into agent code in order to enhance a lightweight agent for transportability. When executed at a remote host, the agent may therefore request access to classes or libraries on the remote host, or require class transfers from a designated code-base server that are not available on the remote host (Gray *et al.*, 2002; Milojicic *et al*., 1998). Access to these classes and libraries are not essential during execution, but are also required for remote creation of new mobile agents (Fünfrocken & Mattern, 1999). Execution support is therefore an important requirement for the mobile agent.

Naming and name resolution

Tripathi *et al*. (2001) describe the necessity for a global naming scheme and name service to locate resources, specify agent servers for migration and to establish inter-agent communication, while Milojicic *et al*. (1998) argue the need for suitable agent names to identify, control and locate mobile agents. Traditional name services such as DNS or NIS are not designed to keep track of mobile agents that move very dynamically. One example of a scheme is where the mobile agent programmer enables the mobile agent to leave a proxy object on the local host and connects regularly to the proxy to notify it of its new host address (Fünfrocken & Mattern, 1999).

Security manager

The remote host is responsible for maintaining security policies to protect the host against a malicious or uncontrolled agent. The remote host is also responsible for run-time activities, such as transport-level security, communications and audit trails. These activities are normally managed by a security manager component, which forms part of the remote host. A method to enforce security in mobile agent systems is by employing security policies for the different entities such as the mobile agent and the host. The security manager is responsible for enforcing these security policies and can, at its discretion, increase the level of security requested by an agent. It cannot decrease the level of services requested by an agent but must inform the agent that the requested service level cannot be provided (OMG Document, 2000).

2.7 Security Issues

Two main categories of threats can be identified in mobile agent systems, namely threats against the host and attacks against the mobile agent. The categories are defined as follows:

2.7.1 Host Threats

The mobile agent platform is responsible for the acceptance and execution of mobile agents. Jansen (2000) defines two categories of threats against the remote host. They are possible threats caused by a malicious mobile agent during execution and threats from other entities such as another remote host attacking the host.

Attacks stemming from malicious mobile agents can be divided into attacks where the mobile agent can firstly gain unauthorised access to the host (and information on the host) and secondly where this gained access can be used to conduct malicious behaviour. Examples of attacks that a malicious mobile agent can perform on a remote host include the unauthorised modification of resources, the unauthorised use of system resources located on the host and the leaking of sensitive data. A malicious mobile agent can for example be a virus that causes damage to the remote host, or it can launch denial of service attacks against hosts and prevent other agents from executing (Karnik, 1998).

Attacks from other entities such as other remote host involve denial of service attacks as well as attacking the platform through masquerading (Jansen, 2000).

A large number of solutions to protect the remote host against attacks from a mobile agent have been proposed and some have been implemented. The reason for this being that traditionally, conventional prevention techniques used in trusted systems and communications security can be used to provide adequate protection for the remote host (Jansen, 2000). Methods for countering attacks on a host include *software-based isolation* (Whabe *et al.*, 1993), *code signing* (Karjoth *et al.*, 1997), *path histories* (Chess *et al.*, 1995) and *state appraisal* (Farmer *et al.*, 1996).

2.7.2 Mobile Agent Threats

Threats against mobile agents involve the protection from the remote host, other mobile agents and entities outside the mobile agent system, such as attacks on the transport mechanisms. These type of attacks are difficult to guard against because of the fact that traditional protection mechanisms were developed to address threats stemming from attacks on the execution environment by the application and not the other way around (Jansen, 2000).

In providing a secure framework for mobile agents, the category of threats stemming from attacks imposed by a malicious host onto a mobile agent is of main concern in this research. These threats are discussed and classified in the next section.

2.8 Threats in Mobile Agent Security

As a first step in designing a secure mobile agent framework, we organise the possible threats by a malicious host on a mobile agent in different criteria according to the method of attack.

The criteria by which a mobile agent has to be protected against a malicious host, is based on the five fundamental concerns or requirements of users gaining access of computer network services, namely *integrity*, *availability*, *confidentiality*, *authentication* and *non-repudiation* (ISO (7498-2), 1988). By using these fundamental security requirements, the criteria that has to be incorporated in the design of a mobile agent system, is defined as integrity, availability, confidentiality, and authentication. Each of these is described in detail below.

2.8.1 Integrity

The integrity of a mobile agent must be protected from tampering by a malicious host. This includes the protection from tampering of the mobile agent's code, state and data. In order to protect the integrity of the mobile agent, the security design has to incorporate the following sub-criteria:

Integrity interference: The mobile agent has to be protected from the executing host interfering with the mobile agent's execution mission. In this scenario the host does not alter any information, but interferes with the execution of the mobile agent. Examples include transmitting the mobile agent incorrectly, not executing the mobile agent completely, transmitting the agent to a host that is not specified in the itinerary, or executing the agent arbitrarily.

Information modification: This sub-criteria includes several possible actions, namely altering, corrupting, manipulating, deleting, misinterpreting or incorrect execution of the agent's code, data, control flow or status. Another example of *information modification* occurs when the executing host interferes with the interaction between different agents and alters the communication between them for its own benefit.

2.8.2 Availability

When a mobile agent arrives at a host, it must be given privileges and access to resources that are necessary for its design goals. If an authorised mobile agent is prevented from accessing objects or resources to which it should have legitimate access, *availability refusal* occurs. Acts of *availability refusal* are mostly deliberate actions performed by the executing nodes, in order to obstruct the agent. Three sub-criteria are defined, namely *denial-of-service*, *delay-of-service* and *transmission refusal*.

Denial-of-service: Under normal networking conditions, this kind of attack occurs when a network system crashes because it has been flooded with network traffic. In the case of mobile agents, *denial-of-service* simply means that the requested resources needed by the agent to accomplish its mission are denied. However, it is also possible for a malicious host to bombard the agent with so much irrelevant information, that the agent finds it impossible to complete its goals. Attacks relating to non-repudiation, where the agent platform denies that it has received an agent, is also included here.

Delay-of-service: This type of attack occurs when the host lets the mobile agent wait for the service and only provides the service or access to the required resources after a certain amount of time. This delay can have a negative effect on the actual purpose of the mobile agent.

Transmission-refusal: When a host with malicious intentions disregards the itinerary of the mobile agent and refuses to transmit the agent to the next host specified in its itinerary, *transmission-refusal* occurs.

2.8.3 Confidentiality

When the assets of the mobile agent are illegally accessed or disposed by its host, the privacy of the mobile agent is not respected and comes under attack. Three subclasses of *confidentiality attacks* are described, namely *eavesdropping*, *theft*, and *reverse engineering.*

Eavesdropping is an invasion of privacy that mostly occurs when the host spies on the agent and gathers information about the mobile agent or about the intercommunication between agents. The access of the remote host to the mobile agent's code, state and data present an opportunity for the host to monitor the agent for other purposes than protecting itself and its own resources. Although the host may not attempt to alter the agent, it can use this information for its own benefits.

Theft and *eavesdropping* are closely related. In this subclass, the malicious host not only spies on the agent, but also removes information from the agent. The malicious host may also “steal” the agent itself and use it for its own purposes, or simply kill it.

Reverse engineering occurs when the malicious host captures the mobile agent and analyse its data and state in order to manipulate future or existing agents. Different to a *theft* attack, a *reverse engineering* attack enables the host to construct its own similar agents, or update the profile of information to which the agent gets access.

2.8.4 Authentication

In the case of the malicious host problem, the agent must be able to correctly identify and authenticate its executing host. Hiding its own identity or refusal to present its own credentials, the host may jeopardise the intended goal of the mobile agent. There are two subclasses of *authentication* attacks, namely *masquerading* and *cloning.*

Masquerading: If a remote host masks itself as one of the destinations on the mobile agent’s itinerary when, in fact, it is not, *masquerading* occurs. A remote host can also masquerade itself as a trusted third party and by doing so accept mobile agents in order to extract sensitive information from them. The masquerading remote host can harm both visiting mobile agents as well as the host whose identity it has used (Jansen & Karygiannis, 1999).

Cloning: Each agent carries its own credentials in order to gain authorised access to the services of its executing hosts. If a host creates a clone of the mobile agent, it will cause unique agent authentication problems.

2.9 Mobile Agent Threats Model

For the previous discussion of threats against the mobile agent, we can safely argue that a mobile agent needs to be protected from its execution environment. In Tables 2.1 to 2.4 we show the impact of each threat on the different aspects of a mobile agent. The purpose of these arrangements is to understand the specific effects of suggested countermeasures, which are discussed in the next chapter.

As stated earlier in the chapter, the mobile agent consists of code, state and data. In protecting the mobile agent against possible attacks from a malicious host, it is also necessary to include the control flow (as specified in the code) as a separate component. The data of the mobile agent is divided into the *identification* of the agent, the *itinerary* of the agent, *initial* data (added at creation) to be used in the attainment of its goals on subsequent hosts, *aggregated* data acquired at previous hosts not to be used subsequently, *aggregated essential* data acquired at previous hosts on the itinerary to be used at subsequent hosts in attaining its goal, and *required* data as acquired (or to be acquired) at the current host.

For the purpose of defining a threat model, the different categories and specific threats within the categories are seen as enclosed threats. Attacks initiated by a malicious host can be a combination of different categories of threats. For example a remote host masquerading as a legitimate receiver of the mobile agent can, after receiving the agent perform a number of integrity or confidentiality violations, such as the altering or copying of sensitive information.

2.9.1 Integrity Threats

Table 2.1 provides the impact of integrity threats on the different components of the mobile agent. The incorrect transmission as well as the modification of the mobile agent poses threats to all components of the mobile agent, while the transmission of the agent to a host not specified on the itinerary interferes with the defined route of the mobile agent. Integrity interference in relation to the incorrect or arbitrarily execution of the agents, only poses threats to the state and control flow of the agent.

Table 2.1: Integrity Threats

	Integrity Interference	
	Integrity modification	
	Transmitting mobile agent incorrectly	*Transmitting agent to*

host not on itinerary (must make sure the host honours itinerary) Not executing the
mobile agent completely Executing mobile agent arbitrarily Deleting, corrupting,
manipulating, altering, misinterpreting, incorrect execution.

Code	Threat	No effect	No effect	No effect	
	Threat				
State	Threat	No effect	Threat	Threat	
	Threat				
Control Flow	Threat	No effect	Threat	Threat	
	Threat				
Data	**ID**	Threat	No effect	No effect	
No effect	Threat				

ItineraryThreatThreatNo effectNo effectThreat **Initial data** ThreatNo effectNo effectNo effectThreat **Aggregated data** ThreatNo effectNo effectNo effectThreat **Aggregated essential data**ThreatNo effectNo effectNo effectThreat **Required data**ThreatNo effectThreatThreatThreat

A *threat* implies that the specific part of the mobile agent is threatened by the particular offence listed in the column, while *no effect* (grey cells) implies that the type of offence listed in the column will not affect the particular part of the mobile agent. For example, if an agent is wrongfully transmitted to a host that is not on the itinerary (offence listed in 3rd column), such an incorrect transmission will have no direct impact on the code of the agent (the agent itself is not damaged), but this contempt threatens the intended itinerary of the mobile agent. Consider the 4th column offence, namely incomplete execution of the mobile agent, as a second example. Once again this offence does not endanger the existence (thus code) of the agent, but it threatens its state, control flow and required data, as incomplete execution might generate a false state and misleading data.

2.9.2 Availability Threats

Denial and delay-of-service threats have an influence on the code, state and required data of the mobile agent. Refusing to transmit the mobile agent, threatens all components of the mobile agent. The impact is outlined in Table 2.2.

Table 2.2: Availability Threats

	Availability				
	Denial of Service		**Delay of service**		
	Transmission refusal				
	Execution resources (memory & CPU denied)			*Data denied /*	

Bombarded with irrelevant information *Execution resources (memory & CPU delayed)* *Data is delayed* *Transmission refusal*

Code	Threat	No effect	Threat	No effect	
	Threat				
State	Threat	No effect	Threat	No effect	
	Threat				
Control Flow	No effect	No effect	No effect	No effect	
	Threat				

Data		**ID**	No effect	No effect	No effect	No

effect Threat **Itinerar** No effectNo effectNo effectNo effectThreat **Initial data** No effectNo effectNo effectNo effectThreat **Aggregated dat** No effectNo effectNo effectNo effectThreat **Aggregated essential dat** No effectNo effectNo effectNo effectThreat **Required dat** No effectThreatNo effectThreatThreat

2.9.3 Confidentiality Threats

Confidentiality threats are a concern to all components of the mobile agent except the data to be acquired at the current host (malicious host). See Table 2.3.

Table 2.3: Confidentiality Threats

		Confidentiality		
		Eavesdropping	*Theft*	*Reverse*

Engineer

Code		Threat	Threat	Threat
State		Threat	Threat	Threat
Control Flow		Threat	Threat	Threat
Data	**ID**	Threat	Threat	Threat
	Itinerary	Threat	Threat	Threat
	Initial data	Threat	Threat	Threat
	Aggregated data	Threat	Threat	Threat
	Aggregated essential data		Threat	Threat

Threat

	Required data	No effect	No effect	No effect

2.9.4 Authentication Threats

Authentication threats as depicted in Table 2.4, indicate that masquerading threats only affects the data to be required, while the cloning of the mobile agent threatens the identification of the mobile agent.

Table 2.4: Authentication Threats

		Authentication	
		Masquerading	*Cloning*
Code		No effect	No effect
State		No effect	No effect
Control Flow		No effect	No effect
Data	**ID**	No effect	Threat
	Itinerary	No effect	No effect
	Initial data	No effect	No effect
	Aggregated data	No effect	No effect
	Aggregated essential data		No effect No effect

	Required data	Threat	No effect

2.10 Conclusion

This chapter described the mobile agent paradigm as well as all the elements that form part thereof. Security is seen as one of the major factors that prohibits the implementation and widespread use of mobile agent systems. The first step in designing a secure framework for mobile agents in mobile agent systems is the categorising of all possible threats, which has been done in Section 2.8. This leads to the creation of a threat model depicting the attacks on the different components of the mobile agent. Chapter 3 seeks to categorise possible countermeasures for the different categories of threats.

CHAPTER 3

COUNTERMEASURES

3.1 Introduction

Different countermeasures have been proposed and a small number implemented in mobile agent applications. We have also found that there are few measures implemented to specific categories of threats (as we have proposed in Chapter 2), and as a result many of the implemented measures fail to address specific needs of different types of applications. The purpose of this chapter is twofold, namely to discuss the countermeasure structure of mobile agents and to combine these measures into a number of countermeasure classes to enhance their applicability.

3.2 Countermeasure Structure

Countermeasures directed toward the protection of the remote host are a direct evolution of traditional mechanisms employed by trusted hosts, while countermeasures directed towards the protection of the mobile agent are radically different from traditional lines. This is due to the fact that traditional mechanisms were not devised to address threats stemming from attacks on the application by the execution environment, which is exactly the situation faced by a mobile agent executing on a remote host that it may not completely trust (Jansen, 2000).

Sander & Tschudin (1998) broadly divide the malicious host problem into two categories, namely tampering *detection* and tampering *prevention*.

Countermeasures in the tampering detection category aim to detect mobile agent tampering *after* the tampering has occurred. It furthermore includes tracing the identity of the malicious host as well as proving the malicious act.

The tampering detection category includes countermeasures that prevent mobile agent tampering *before* the tampering can occur. Prevention mechanisms attempt to make it impossible (or very difficult) to access or modify the mobile agent in a meaningful way

(Vigna, 1998). Kotzanikolaou *et al.* (2000) further categorise prevention mechanisms as either passive or active. *Passive* prevention mechanisms protect the mobile agent by employing organisational or architectural solutions, such as letting the mobile agents only be employed in a trusted domain. An example is the creation of a network of trusted hosts in which the mobile agents are to be deployed (Sander & Tschudin, 1998). Passive prevention approaches either rely strongly on the trustworthiness of the mobile agent platform, or concede the core features of the mobile agent such as autonomy and migration.

Active prevention mechanisms provide the mobile agent with adequate protection without compromising the features and advantages of the mobile agent paradigm. These types of countermeasures can either be hardware based, such as the incorporation of special *trusted hardware components* (see Smith & Austel, 1998; Wilhelm *et al.*, 1998) or software based, such as the *obfuscation of code* (Hohl, 1997, 1998) and the use of *encrypted functions* (Sander & Tschudin, 1998).

3.3 Countermeasure Classes

Countermeasures that have been implemented, or proposed to reduce the vulnerability of the mobile agent against malicious hosts, can be categorized into a number of classes according to the protection technique being used. We propose four different classes, namely *trust-based computing*, countermeasures based on *recording and tracking*, countermeasures based on *cryptographic techniques*, and countermeasures based on *obfuscation and time techniques*.

3.3.1 Class 1: Trust-based computing

The creation of a trusted environment in which a mobile agent roams freely and fearlessly without being threatened by a potential malicious host can possibly alleviate most of the categories of threats that have been discussed. According to Ordille (1996), the central security concern is how to establish trust between entities and how to limit the risk for the different entities. Once the level of trust is established, the risk for the entities (in this case the mobile agent and the remote host) can also be established.

The *trust* that a mobile agent has in a particular host can be *blind*, based on *reputation*, based on *control and punishment*, or based on *policy enforcement* where an agent has prior (contractual) relationship with the host (Yee, 1997). Each of the different classes of trust has its own advantages and disadvantages. For example trust based on *reputation* is easy to implement because no special mechanisms are required, while trust based on *control and punishment* can have cost implications for the individual if punishment is seek amongst judicial lines, especially if the malicious entity is located in another country with a different or unknown law-system (Wilhelm *et al.*, 1998).

In order to implement countermeasures based on a notion of trust, a security policy must be created and used by the remote host. Wilhelm *et al.* (1998) define such a policy as a set of rules that constrains the behaviour of a host for all conceivable situations. They then define trust in a host as the belief that it will adhere to its published security policy.

Ordille (1996) categorises the travelling of mobile agents into three types, namely o*ne-hop agents*, *two-hop boomerang agents* and *multi-hop agents*, where a hop defines a trip from one host to another. A *one-hop* agent only travels from its local host to a remote host; a *two-hop boomerang* agent travels from its local host to a remote host and back to its local host, while *multi-hop agents* travel to multiple remote hosts. According to this classification, different levels of trust are established according to a pre-defined policy. For example, establishing trust for one-hop agents is simpler than for multi-hop agents, in that only one remote host is visited and the agent does not travel any further. The level of trust in these types of agents is determined by a risk policy; if the mission of the agent is only to carry data to the destination, then the agent owner only has to trust the remote host to accept the agent and its data.

Swarup & Fábrega (1999) describe aspects of trust in open distributed systems. They argue that computational models and mechanisms be produced that can enable trust between entities. Trust between agents can be established in a variety of ways such as blind trust, deterrence-based trust, knowledge-based trust, identification-based trust and social trust.

Trust benefits include enabling cooperation between agents, the lowering of access barriers for the protection of resources and entities, as well as the creation of trusted communities.

Swarup (1997) states that the critical problem in mobile agent security is the assessment of trust in mobile agents and hosts. Three trust appraisals are identified, namely *authentication* to deduce which principal made a specific request, *code appraisal* to ensure that is safe to execute a mobile agent and *state appraisal* to ensure that a mobile agent has not become malicious due to alterations in its state.

Countermeasures that make use of the notion of trust that have been researched for the mobile agent paradigm include the following:

Tamper resistant hardware (prevention)

Installing *tamper resistant hardware* is a method well suited to implement the notion of trust in agent-to-host relationships. This method uses the concept of a secure coprocessor model, where physically secure (tamper-detecting /-responding) hardware is added to conventional computing systems. These are computational devices that are trusted to execute their software correctly, despite physical attack. The distribution of trusted hardware components throughout a hostile environment enables secure distributed applications (Smith & Austel, 1998).

Wilhelm *et al.* (1998, 1999, 1999a, 2000) created the *Cryptographically Protected Objects (CryPO) Protocol,* which makes use of a Tamper-Proof Environment (TPE) in order to provide a secure execution environment for the execution of mobile agents on untrusted hosts. The TPE is a complete microcomputer and its main task is to run a virtual machine where the mobile agent platform (Agent Executor) can be installed. An underlying operating system controls the access to resources on the host where the TPE resides. A private key is contained in a cryptographic library that forms part of the TPE and is accessible only to this environment. The components contained in the TPE are protected which make it impossible to access or manipulate mobile agents executing in this environment. A Tamper-Proof Environment Manufacturer (TM) produces the TPE, and it guarantees the information contained in the TPE to be tamperproof.

The *CryPO* protocol consists of an *initialisation* and a *usage* phase. The initialisation phase is only executed once and consists of the TM publishing its certification key and sending it to the Agent Executor located on a remote host. The Agent Executor registers its TPE with one or several brokers. The usage phase of the protocol can only be executed once the initialisation phase is completed. In the usage phase the owner of the mobile agent contacts the broker for information regarding the Agent Executor it wants to interact with. This is done by the verification of the published certificate of the TPE. Once satisfied, the owner encrypts the mobile agent with the public key of the TPE and sends the encrypted agent to the Agent Executor. The Agent Executor doesn't possess the decryption key and has to forward the mobile agent to the TPE where it will be decrypted and executed. Once finished with its task on the specified TPE, it can request migration back to its owner or to the next remote host as specified on its itinerary. Mobile agents migrating from the TPE are encrypted and the TPE provides the certificate of the designated receiver of the mobile agent.

Although all computations on the mobile agent are protected and executed within the TPE, the Agent Executor is still responsible for sending the mobile agent to the next remote host. This can lead to the possibility of the mobile agent not being sent to the correct remote host as specified in the itinerary of the mobile agent (Wilhelm *et al*., 1998). A possible solution to this problem is the introduction of an itinerant safe policy, whereby the mobile agent is serialised within the TPE (Wilhelm *et al*., 1998). Other limitations to this protocol include the violation of the TPE if adequate time and resources are available. In the case where the private key of the TPE is compromised, the attacker will have complete control over all mobile agents sent to the TPE. Solutions to these limitations include periodic inspections of the TPE by an independent appraisal organisation (Wilhelm *et al*, 1998). A main feature of a mobile agent is autonomy (as specified in Chapter 2), and the *CryPO* protocol, which creates a trusted network, violates this feature in terms of regulating the environment in which a mobile agent can be deployed. Ma & Yen (2002) emphasise this disadvantage and also state that the use of special hardware reduces the usefulness of a mobile agent system as a middleware component. The costs implied by installing TPE's and thus creating a trusted network is also a deterrent for service providers (Borselius, 2002).

Another type of tamper-resistant hardware is proposed by Fünfrocken & Mattern (1999), which makes use of a Java Card as a trusted computing base. The card is able to run Java code and is added to the remote host running the agent platform, as a specialised hardware component. A Java Card owns a private key, which implies that the mobile agent can be encrypted with the corresponding public key and only the Java Card can decrypt and execute the code. An encrypted mobile agent moves its code from the agent platform to the Java Card and subsequently can be executed in a highly secure environment.

Kilian-Kehr & Posegga (2002) proposed the use of a smart card platform for the execution of mobile code. The smart card implements an interpreter for mobile code and the execution platform implements key management facilities.

According to Schneier (2000), it is not really possible to manufacture a device that is absolutely tamperproof. He suggests that instead of focussing on how tamper resistant a specific device is, the focus must shift to how much tamper resistance is needed in terms of the cost to "break" a tamper resistant device.

Trusted execution environment (prevention)

According to Sander & Tschudin (1998) a *trusted execution environment* can be achieved by setting up a trusted set of network nodes by using encryption and authentication techniques. This is done by encrypting the mobile agent as it is sent between remote hosts and by authenticating the host before the mobile agent is transported to it. Finally, the mobile agent has to be authenticated before it enters the host. However, this method goes against the notion of a mobile agent to a certain extent. If a mobile agent has a predefined itinerary, the advantage of the vast amount of resources available on the Internet, may be lost or severely obstructed.

Another drawback is that a method needs to be specified in order to create a *trusted* *execution environment*, by just authenticating the hosts it will still be possible for a malicious host to masquerade as a trusted legitimate one. The encryption of the agent only protects the mobile agent during transportation and not during execution on the remote host. The creation of a *trusted execution environment* can prevent some attacks against mobile agents and can be useful in a small environment (such as an Intranet), but is not viable in an open environment.

Trusted third party (prevention)

A *trusted third party* is a separate entity in the mobile agent system environment employed for secure and safe computations by mobile agents and hosts. Feigenbaum & Lee (1997) define the services performed at the trusted third party, as examination of the mobile agent and also the writing of an auxiliary program to monitor the destination host. This program determines whether the remote host can be regarded as a safe execution environment for the visiting mobile agent. The trusted third party digitally signs the mobile agent as well as the auxiliary program upon which the destination host can decide to accept or reject the mobile agent.

A trusted third party may also act as a Certification Authority for the generation of private and public key pairs. Furthermore, a mobile agent on route can also divert after migration from a host to the trusted entity in order to perform secure computations (see for example the FILIGRANE project (Jalali, 2000)). Advantages of using a trusted third party are that it can alleviate large increases in network traffic when making use of cryptographic algorithms in order to encrypt mobile agents between remote hosts (Piessens *et al.*, 2000). In the case where the mobile agent is forced to visit the trusted third party after executing at a remote host, is limits the autonomy of the mobile agent.

3.3.2 Class 2: Countermeasures based on recording and tracking techniques

Countermeasures based on recording make use of the itinerary information of a mobile agent, either by manipulating the migration history or by keeping it hidden.

Countermeasures that make use of tracking techniques make use of for example agents or servers that cooperate in order to reach the goal of the mobile agent.

Path histories (detection)

A *path history* is a countermeasure that is strongly used in the malicious agent problem where it is needed to maintain record of the agent's travels that can be substantiated. According to *path histories*, a record of all prior hosts visited by a mobile agent is maintained.

The computation of a *path history* requires that each host add a signed entry to the itinerary carried by the mobile agent. Ordille (1996) defines two methods to establish the level of a host's trust in a mobile agent. The first technique requires each host to add its identification to the itinerary of the mobile agent and forwards a copy of the added information to the next remote host specified on the itinerary. The next remote host can then determine whether it trusts the previous hosts that the mobile agent visited, either by simply reviewing the list of identities provided or by individually authenticating the signatures of each entry in the path history. In the second technique the added identification of the current host is signed before added to the itinerary and sent to the next specified host.

Although this method is used for the detection of malicious agents, the possibility exists that it can be implemented to detect malicious hosts as well. The generated record (of hosts visited) can be used by the mobile agent owner (once the agent has returned to the local host), to detect at which host possible tampering occurred. By keeping histories of the hosts visited, a current host can also detect if the mobile agent has been manipulated before accepting the agent for execution.

While the technique does not prevent a host from behaving maliciously, it serves as a strong deterrent, since the host's signed path entry is non-repudiatable. Disadvantages of *path histories*, is that it becomes more costly (in terms of size and thus the validation process) as the *path history* increases and the success of the scheme is dependent on whether the current host is able to determine the level of trust of the previous hosts visited by the agent (Jansen, 2000).

Detection objects (detection)

Detection objects as a countermeasure to detect modifications by a malicious host is based on prevention of *storage jamming* techniques. Storage jamming is the disruption of information systems by the unauthorised modification of data (McDermott & Goldschlag, 1996). One of the techniques introduced to combat storage jamming is the use of detection objects, which can also be used in the mobile agent environment.

Detection objects such as dummy data items or attributes accompany the mobile agent and are used to determine whether the host in question can be trusted. If the *detection objects* have not been modified, then reasonable confidence exists that legitimate data has not been corrupted also. According to McDermott & Goldschlag (1996), *detection objects* must satisfy two properties, namely hosts must not be able to distinguish between *detection objects* and real data, and a high probability that an unexpected *detection object* state indicates malicious modification.

One of the primary disadvantages to this technique is that it is very application specific as the *detection objects* must be believable enough to fool host systems and at the same time must not affect the result of the query returned by the mobile agent. Another disadvantage is the added computational cost as it is necessary to update the detection object often (Meadows, 1997).

Itinerary recording with replication and voting (detection)

Minsky *et al.* (1996) and Schneider (1997) proposed *itinerary recording with replication and voting* as a countermeasure by which multiple copies of a mobile agent are used to perform the computations as needed to reach the goal of the mobile agent. The idea behind the method is that although a malicious host may corrupt a few copies of the mobile agent, enough replicas to successfully complete the computation will still exist. For each stage of the computation, the host ensures that the mobile agent has not been tampered with.

The technique of *replication and voting* commences with a mobile agent being created at the local host. Upon migration the agent is replicated into a pre-determined number (*n*) of mobile agents that is sent to *n* different remote hosts. The replication of the mobile agent can be an exact copy, or the computations can be divided into different agents, with each agent responsible for computing a certain section of the ultimate goal. Upon arrival at a remote host, a voting system is used in order to determine the validity of the *n* mobile agents that are received by checking if the credentials of the agents are valid. Remote hosts that are involved in a particular stage of a computation are expected to know the set of acceptable hosts for the previous stage. The host propagates onto the next stage only a subset of the replica mobile agents it considers valid, based on the inputs it receives.

This technique can be used to ensure a critical message is delivered and is appropriate for tasks that can be safely duplicated as it guarantees the computation integrity by identifying trusted hosts. In duplicating agents, additional resources are consumed and the network traffic increases. As one of the advantages of mobile agents (as stated in Chapter 2), is that it can be used with success in order to alleviate bandwidth problems, using the technique of *replication and voting,* as a security measure will work against this advantage. It is also unrealistic to presume that two or more hosts exist that can execute a specific mobile agent in the same manner without being from the same provider. Jansen & Karygiannis (1999) also mentioned additional drawbacks such as the cost of setting up the authenticated channel and the inability for the peer to determine which of two platforms is responsible if the agent is killed.

Mutual itinerary recording (detection)

According to *mutual itinerary recording*, the itinerary of the mobile agent is recorded and tracked by another cooperating agent, while a mobile agent moves between hosts. The cooperating agent serves as a backup agent and is executed in a trusted environment. There also exists a secure communication channel between the mobile agent and its cooperating agent. The mobile agent will convey the information about the last host visited, the current host and the next host on its itinerary to the cooperating mobile agent through the authenticated channel. The cooperating agent maintains a record of the mobile agent's itinerary and takes appropriate action when inconsistencies are noted (Roth, 1998).

According to Roth (1998), the protocol assumes three categories of hosts (namely white hosts, grey hosts and red hosts) that are categorised according to their level of trust. A *white* host is completely trusted; hosts that are not completely trusted and which may potentially perform some malicious acts are categorised as *grey*. *Red* hosts may collaborate with at least one other host in order to launch an attack on a mobile agent.

Critical operations in the mobile agent are performed in the cooperating agent and secret data is distributed between the mobile agent and the cooperating agent. The cooperating agent records the actual route of the mobile agent. This is achieved by the mobile agent sending the address of the previous host as well as the address of the next host through the authenticated channel, to its cooperating agent (this is done on every remote host). The cooperating agent verifies the addresses as sent by the mobile agent and if a malicious host transfers the mobile agent to an incorrect host, it will be able to detect this and take appropriate actions.

Because the path records are maintained at the agent level, this technique can be incorporated into any appropriate application. Some drawbacks include the cost of setting up the authenticated channel and the inability of the peer to determine which of the two platforms is responsible if the agent is killed (Roth, 1998). This technique only detects modifications on the route of the agent and not on the agent itself. It also goes against the notion of a mobile agent as being autonomous; the same effect will be achieved by letting the agent migrate to a trusted third party after visiting a remote host.

Reference states (detection)

Reference states as a countermeasure are a variation of the *itinerary recording with replication and voting technique* (Minsky *et al.*, 1996; Schneider, 1997). Hohl (2000) suggests the idea of using *reference states*, which are mobile agent states produced by non-attacking or reference hosts in order to detect interference attacks. This countermeasure initially made use of a referenced trusted host to execute the mobile agent in parallel, however this was nothing more than a client-server set-up and the author presented an improved protocol (Hohl, 1999). In this protocol every host on the itinerary of the mobile agent receives the initial state, final state and input data from the previous host. The current host is then responsible for re-executing the mobile agent to check for indifferences.

In this protocol the local host computes and signs the initial state of the mobile agent, which is then transferred together with the mobile agent's code and initial state to the next host as specified in the itinerary. The receiving host checks the signature, if this cannot be verified then the local host is informed. In the case that the signature is valid, the mobile agent is executed, whereupon the host signs the resulting state. Upon migration the signatures together with the states, input and code is marshalled and sent to the next host. This protocol is repeated at every host on the itinerary. Every host re-computes the mobile agent with the input provided from the previous host and compares the results received. If the results differ then the previous node acted maliciously and can be acted on. This protocol has been implemented in the Mole mobile agent system (Baumann *et al*., 1998).

Reference states can detect attacks such as writing or modification in the state of the mobile agent. This mechanism is however not able to detect confidentiality attacks.
Advantages of this protocol include the presentation of the complete state of the mobile agent, which can be used in order to prove modifications done on a specific host. A disadvantage is the increase in costs by the extra overhead required for the computations (Hohl, 1999).

Phone home (prevention)

Grimley & Monroe (1999) propose that before leaving each host, a mobile agent transfers the data that it has required at the specific host to its owner.

The transfer of data to the local host can involve the transfer of any data it has acquired, thereby preventing its loss or its disclosure to future hosts, or it can act as a method to let a user know that the agent is still functioning.

Phoning home can prevent data acquired at remote hosts from malicious modifications by future hosts, but does not prevent tampering. Tampering can be detected if the state of the mobile agent after execution at a remote host is also sent to the owner of the agent. This countermeasure defies the autonomy property of a mobile agent, in that it needs to have direct contact with its owner.

Using a mobile agent system (prevention)

Yee (1997) proposed using a distributed *mobile agent system*, where a specific task is split into several mobile agents, based on the method of *recording and voting* as proposed by Schneider (1997). In a *mobile agent system*, the collaborative effort of all the agents accomplishes the task instead of a single agent assigned with the user's wishes.

By using a distributed *mobile agent system*, two mobile agents can for example be sent to the same list of remote hosts, but in a different order. The mobile agents in this case will have the same goal and would visit the same providers. This can alleviate problems such as a malicious host modifying, for example, the lowest price of goods as contained in the agent. It is however not in all cases viable to send replicas of agents to the same hosts and it does not allow for changes in the itinerary of the agent.

State appraisal (detection)

The goal of the state appraisal countermeasure is to ensure that an agent's state has not been tampered with. Although this countermeasure has been proposed as a possible solution to the malicious agent problem, it can be applied to the malicious host problem. The *state appraisal* function of a mobile agent uses authentication and authorization techniques to calculate a set of privileges as a function of the agent's state. Such a function can then be used to predict and thus detect certain state alterations (Farmer *et al.*, 1996). Although not all state alterations can be detected, it can protect the mobile agent against state modifications.

In the protocol Farmer *et al.* (1996), distinguish between a *program* and the *mobile agent* that will be responsible for executing the *program*. The program contains the source code and is signed by the author, while the mobile agent contains the data and state also signed by its owner. The *state appraisal* function for the program is computed which will calculate the maximum safe permissions to be granted to the agent as a function of its state. A message digest of the result of the compiled program as well as the *state appraisal* function is created and the owner signs this with its private key. Upon preparing the program for sending a second *state appraisal* function is attached which contains the permissions the sender wants an agent running the program to have.

Before sending the agent the owner attaches its name and computes a message digest for the program, the message digest of the program, the *state appraisal* function of the sender and the owner's name. This message is signed with the owner's private key. Upon migration of the agent between platforms the current platform constructs a message containing the agent, the current state of the mobile agent, the current interpreter, the principal on whose behalf the interpreter is executing and the principal on whose behalf the next interpreter should execute the agent (from its current state).

A remote host receiving the agent uses the *state appraisal* functions to verify that predecessor hosts have not changed the state of the mobile agent. *State appraisal*, as a countermeasure, has not been proven in practice and thus the possible implications on for example, processing costs have not been determined. The success of this technique also relies on the extent to which harmful alterations to an agent's state can be predicted and on the extent to which the state appraisal functions can be prepared before using the agent (Jansen, 2000). If it is possible to provide a mobile agent with *state appraisal* functions, it will allow manipulations of the agent's state to be detected during its execution (Wilhelm *et al.*, 1999). Westhoff (2001) also mentions that by using this approach a malicious act can be detected, but not the identity of the attacker.

Proof-carrying code (detection)

Proof-carrying code is a technique proposed by Necula & Lee (1998) as a countermeasure for the malicious agent problem, but can also prove useful to protect the mobile agent against code modifications by malicious hosts. The author of the mobile agent creates a formal safety proof that proves adherence of the mobile agent to the safety rules. The receiving host uses a proof validator to check if the proof is valid and safe to execute. Any modifications to the code of agents constructed with *proof-carrying code* will result in the rejection of the agent by the host.

Proof-carrying code is checking the built-in properties of the code and does not make use of cryptography or trusted third parties. The proofs are verified statically before the code is executed. Necula & Lee (1998) define the steps of *proof-carrying code* as (1) the specification of the safety policy for the interaction with the mobile agent, (2) the host receives the mobile agent and extracts a safety proof from it, (3) the safety proof is sent to the proof producer who is responsible for the proving of the agent's code and returning the proof to the host (the host can also acts as the proof producer) and (4) the validity of the proof is checked by the host by making use of a proof checker. If the proof is valid the agent can execute, if not, it will be rejected.

Proof-carrying code programs are tamperproof in the sense that any modification to the code will result in the proof not being valid. This can be used in the malicious agent problem to prevent unauthorised modifications of the mobile agent's code. Disadvantages of the technique include that the proofs are extensible in size and will have a large impact on computing transfer costs. Borselius (2002) adds that the difficulty in generating such formal proofs in an automated and efficient way, is a major drawback.

3.3.3 Class 3: Countermeasures based on cryptographic techniques

Techniques under this type of countermeasures utilise *encryption/decryption algorithms, private and public keys*, *digital signatures, digital timestamps,* and *hash functions* to address different threat aspects.

Anonymous itinerary (prevention)

Westhoff *et al.* (1999) proposed *anonymous itinerary* as a prevention method used to protect the route of the mobile agent. This method entails the encryption of the pre-defined itinerary of the mobile agent and in doing this, hides the agent's route from all other entities (including all remote hosts specified in the itinerary).

The protocol starts with the author of the mobile agent specifying the initial itinerary of the mobile agent as a concatenated list of Internet addresses. The local host (home) address is stored (in plaintext) separately from the rest of the itinerary. The reason for this is to provide a means for the remote hosts (if needed) to abort the agent. After defining the addresses of the remote hosts to be visited, the itinerary is then encrypted by making use of a public-key infrastructure. Concatenation or encapsulation techniques are also used in order for a specific host to only be able to decrypt data related to it. The local host signs the data of the mobile agent intended for a specific host. Before migration to the next host, the current host is deleted from the itinerary. Westhoff *et al.* (1999) present four different combinations of encryption and signature schemes that can be used for implementing the encryption and signature parts of the countermeasure. These combinations are:

Atomic encryptions and Signatures: The itinerary of the mobile agent is signed in an atomic way by using a public-key encryption method. By doing this the current host can verify the itinerary for modifications as well as decrypt the address of the next platform to be visited. All the other destinations specified in the itinerary are hidden. A *trip-marker* that uniquely identifies the mobile agent's journey (such as the time of creation) is added to the itinerary in order to prevent replay attacks.

Atomic Encryptions and Nested Signatures: All the addresses as well as the signatures in the itinerary are encrypted, which implies higher computational complexity. The signature contains the address of the current host, the previous and next hosts, the trip-marker as well as the encrypted texts (of all addresses) to be used by later hosts.

Nested Encryptions and Atomic Signatures: The remote host receives cipher text from the previous hosts, when decrypted reveals the next host, the signature from the local host and the encrypted address of the previous host. In this approach the signature does not contain the address of the previous host.

Nested Encryptions and Signatures: The current host decrypts the address of the next host. A signature and the encrypted itinerary is sent to the next host that contains the current host address, the address of the next host, the trip-marker and the complete remainder of the itinerary.

The above combinations can be used with varying results for different applications. For example *atomic encryptions and signatures* operates at a lower cost, making it ideal for short routes and less sensitive services, while *nested encryptions and signatures* detect an attack as early as possible.

Although this method is based on a pre-defined itinerary, it is possible to extend the algorithm in order to include new remote hosts during execution. The mobile agent has the ability to change its pre-defined itinerary and in doing so it has to include a signed confirmation of the changes.

Anonymous itinerary is a countermeasure that can be used effectively for applications that deem necessary to keep the itinerary of the mobile agent hidden. This can alleviate security problems based on competition by different hosts (where a host will sent an agent to a destination not on the itinerary or change for example prices according to the businesses listed in the itinerary). A possible disadvantage is the computational costs that are involved.

Execution tracing (detection)

Execution tracing is a technique for detecting unauthorised modifications of an agent through the recording of the agent's behaviour during its execution on each host. Vigna (1998) suggests a procedure that compels the executing host to produce a *trace*. A trace T_P of the execution of a mobile agent p consists of a sequence of pairs $<n,s>$ where n is a unique identifier of a statement and s is a signature.

Statements in the code of the mobile agent are classified as either *black* or *white*. A statement is *black* if it modifies the state using information received from the external execution environment (for example read(x) is classified as a black statement). A statement is classified as *white* if the mobile agent's execution state is modified on the basis of the value of the agent's internal variables only (for example x:=y+z is classified as a white statement). In the case of a *black* statement the signature contains the new values of the internal variables as a result of the statement execution. In the case of a white statement the signature is empty.

The protocol makes use of public/private key encryption and one-way hash functions to produce cryptographically secure messages. Upon requesting migration to the first host (Host A) on the itinerary, a signed message (containing the mobile agent's code and the initial state in encrypted form as well as the *mobile agent token*) is sent from the local host to the remote host. The *mobile agent token* contains the ID of the agent, a timestamp, a hash value of the code and the identity of the trusted third party to be used. The local host using its secret key signs this mobile agent token.

Host A receives the message and checks the signature on the message as well as the agent token. It examines the information contained in the message and on this basis makes a decision whether to refuse or accept the mobile agent. In both cases the host responds by sending a signed message containing a rejection or acceptance notice to the local host. If it contains an acceptance, the key used for encrypting the mobile agent is sent to Host A, protected with the public key of the host. Upon receiving the key, Host A can decrypt the agent and execute it.

If the mobile agent requests migration to the next remote host (Host B), the mobile agent is wrapped and Host A sends two consecutive signed messages to Host B. The first message, contains the names of the sender and receiver, the agent token, a hash value of the trace produced by the agent execution on Host A, a hash value of the current state on Host A and a timestamp. The second message, consists of the code of the mobile agent and the current state encrypted by a random key chosen by Host A, as well as a hash of the previous message. Host B checks the signatures and hash values and verifies that the local host sent the mobile agent at the indicated time. Host B sends a signed message containing an acceptance or rejection notice.

This protocol is repeated for every address listed in the itinerary of the mobile agent. The final host on the itinerary retrieves from the agent token, the name of the local host and contacts it to request for delivery of the agent.

In order to detect possible modifications on the mobile agent's code and state, the local host can check the execution sessions logged in the trace after the agent terminates, by requesting the traces from the corresponding hosts. This check requires the owner of the mobile agent to compute a hash of the received trace and compare it with the data to its disposal. This comparison enables the owner to identify possible security breaches.

This method can detect all possible manipulations of the mobile agent (code, state & execution flow) after suspicion has been aroused and only after the agent has terminated. It also relies on the hosts to be honest about their input information to the agent.

The approach has a number of drawbacks, the most obvious being the size and number of traces to be retained and the fact that the detection process is triggered occasionally, by suspicious results or other factors. Other more subtle problems identified include the lack of accommodating multi-threaded agents and dynamic optimisation techniques (Jansen & Karygiannis, 1999). Westhoff (2001) states another disadvantage, namely that in case of an attack, the identity of the attacker cannot be revealed. It furthermore places an extra burden on the hosts who have to dedicate large amounts of resources to the storage of enforcement information (Wilhelm *et al.*, 1999).

Computing with encrypted functions (prevention)

Sander & Tschudin (1998) suggest the use of encrypted functions, which prohibits the executing host from learning anything substantial about the agent. This approach is based on the *tamper-resistant hardware technique* (Wilhelm *et al.,* 1998), but with the difference that it only relies on software.

According to this technique, a mobile agent is composed of several decomposable functions. Each function, which should remain secret, is encrypted and sent on its way to perform a particular task. For example, if f_i is such a function, then the function $g_i = E(f_i)$ is the encrypted version of f_i, which is created by the sender. The sender also creates a program $P_i,(g_i)$ which implements the function g_i. Function $P_i(g_i)$ is sent to the host where it is deciphered and executed by a program, *P,* to determine $P(g_i)(x)$. The host will thus be able to see clear text instructions about a small part of the mobile agent, but will not be able to understand the goal of g_i. Because the mobile agent owner knows the decryption algorithm, he/she is able to decipher and hence determine the value of $f_i\ (x)$. This way, the host executes a set of instructions that do not portray the real meaning of the agent.

Computing with encrypted functions permanently prevents attacks on the entire agent and doesn't make use of a trusted functionality. This technique, while very powerful,

does not prevent denial of service, replay, experimental extraction and other forms of attack against the mobile agent (Jansen, 2000). According to Wilhelm *et al.* (1999) it is however in its current form not possible to implement.

Environmental key generation (prevention)

Riordan & Schneier (1998) describe a model in which the malicious host problem is countered with the introduction of *clueless* agents. Clueless agents carry a cipher text message, as well as a method for searching through a host environment for specific information. If this information is found, predetermined environmental conditions become true and allow the generation of a key that can decipher the mobile agent’s cipher text message. The cipher text message can be private data or part of the mobile agent’s code. The environmental conditions are hidden through a one-way hash function or a public key encryption of the hidden message. This ensures that a malicious host cannot uncover the message or the response action, by directly reading the mobile agent's code.

Clueless agents scan the environment for their activation keys on a fixed data channel such as web pages, mail messages and file systems. The *environmental key generation* protocol has three phases, namely the host sends an encryption key to the initiator of the agent, the initiator gives the mobile agent the encrypted message, part of the data needed to decrypt the message as well as the location of the rest of the data needed for decryption and finally, the mobile agent retrieves the data needed to derive the decryption key from the host and decrypts the message.

Time based constructions allow key generation based on time. Three different time-based constructions are used, namely *forward-time hash functions*, *forward-time public keys* and *backward-time hash functions*. Forward-time constructions permit key generation only after a given time, while backward-time constructions permit key generation only before it. Another form of *environmental key generation* is general server constructions that make use of one-way functions and a symmetric encryption algorithm.

This countermeasure can be used effectively to protect parts of the mobile agent against integrity and confidentiality attacks. One weakness of this approach is that a platform that completely controls the agent could simply modify the agent to print out the executable code upon receipt of the trigger, instead of executing it. Another drawback is that a host typically limits the capability of an agent to execute code created dynamically, since it is considered an unsafe operation (Jansen & Karygiannis, 1999).

Partial result encapsulation (prevention)

Partial result encapsulation makes use of a public key to encrypt the result of the mobile agent's action at each executing host platform. The encapsulated data bits are incrementally accumulated until an intermediate point(s) is reached or until the mobile agent returns to its point of origin, where the private key is used to decrypt the layers of data. By employing encryption and digital signatures, encapsulation of the results of an agent's visit to each host can, respectively, provide confidentiality and integrity (Chess *et al*., 1995; Jansen, 2000; Yee 1997).

In general, there are three alternative ways to encapsulate partial results (Jansen & Karygiannis, 1999):

- Provide the agent with a means for encapsulating the information,
- rely on the encapsulation capabilities of the agent platform, or
- rely on a trusted third party to timestamp a digital fingerprint of the results.

The following countermeasures are all sub-forms of partial result encapsulation, namely *partial result authentication codes, sliding encryption,* and *partial results along the way*.

Partial result authentication codes

Yee (1997) presents a variation to the *partial result encapsulation.* This technique requires the agent and its originator to maintain or incrementally generate a list of secret keys used in the p*artial result authentication codes* computation. Once a key is applied to encapsulate the information collected, the agent destroys it before moving onto the next platform, guaranteeing forward integrity. If one of the hosts visited by the mobile agent is malicious, then the previous set of results obtained by the mobile agent will remain valid. Only the owner of the mobile agent can verify the result, because no other copies of the secret key remain.

This technique has a number of limitations. The most serious occurs when a malicious platform retains copies of the original keys or key generating functions of an agent. If the agent revisits the platform or visits another host conspiring with it, a previous partial result entry or series of entries could be modified without the possibility of detection. Since *partial result authentication codes* is oriented towards integrity and not confidentiality, the accumulated set of partial results can also be viewed by any host visited, although this is easily resolved by applying sliding key or other forms of encryption (Jansen & Karygiannis, 1999).

Sliding encryption

Often the amount of information gathered by an agent is rather small, in comparison to the size of the encryption key involved and the resulting cipher text. A special form of encryption is implemented, namely *sliding encryption* that encrypts the mobile agent piecewise, which in turn yields small pieces of cipher text (Young & Yung, 1997).

The agent carries a public key and encrypts the information as it is accumulated at each host visited. When the agent returns home the information is decrypted using the private key maintained at the local host. While the purpose of sliding encryption is confidentiality, an additional integrity measure could be applied as well, before encryption occurs (Jansen & Karygiannis, 1999).

Sliding encryption aims at saving space rather than time, which can be crucial for an application where the mobile agent collects small amounts of data on a large number of different hosts (Loureiro *et al.*, 2000).

Partial results along the way

Another method is to require each host to encapsulate partial results along the way, rather than relying on the agent to encapsulate the information. The distinction is not only one of where the encapsulation mechanisms are retained, either with the agent or a platform, but also one of responsibility and associated liabilities (Jansen, 2000).

Karjoth *et al.* (1998) devised a platform-oriented technique for encapsulating partial results, which reformulated and improved on the *partial result authentication* technique. The method first constructs a chain of encapsulated results (as computed on every remote host) that binds all the results obtained at the different hosts together. Each host digitally signs its entry using its private key and uses a secure hash function to link the results and identities within an entry.

This technique provides forward integrity and confidentiality by encrypting each piece of accumulated information with the public key of the originator of the agent.

Yee (1997) mentioned that forward integrity could also be achieved using a trusted third party that performs digital time stamping. A digital timestamp allows an entity to verify that the contents of a file or document existed at a particular point in time. A problem associated with this method is the general availability of a trusted time-stamping infrastructure (Jansen & Karygiannis, 1999).

Digital signatures (detection)

Sander & Tschudin (1998) introduced the concept of an un-detachable *digital signature* that allows a mobile agent to produce a *digital signature* inside a malicious host without the host being able to deduce the agent's secret or to reuse the signature routine.

Digital signatures can also be used to provide integrity by signing the data acquired or computed at a host, which is not to be used at subsequent hosts. A disadvantage of this approach is that the size of the mobile agent grows linearly as it gathers results (Loureiro *et al.*, 2000).

3.3.4 Class 4: Countermeasures based on obfuscation and time techniques

The basis of countermeasures based on obfuscation and time techniques is to add restrictions on the lifetime of the mobile agent as well as scrambling the code in order to make it difficult or impossible to understand.

Code obfuscation (prevention)

Hohl (1997, 1998) suggested a mechanism called Blackbox Security. The strategy behind this technique involves the scrambling of the code in such a way that it is impossible to gain a complete understanding of the code's function (i.e., specification and data). It is also impossible to modify the resulting code without detection. Hohl (1997, 1998) proposed *code-obfuscation* as a combination of two mechanisms. The first mechanism dynamically generates a new and less understandable version of the mobile agent code, while the second mechanism restricts the lifetime of the mobile agent's code and data.

Before migration the local host scrambles the code of the mobile agent, mixes the data elements of the mobile agent and adds expiry dates to the data elements by making use of digital signatures. The local host then signs the code of the mobile agent together with another expiry date. Once migrated, a malicious host can still manipulate the code (although it will take a certain extra amount of time). On sending the manipulated mobile agent to the next host on the itinerary, the receiving host will reject the mobile agent due to the fact that the validity dates have expired. Three different techniques are commonly defined for code obfuscation, namely *variable re-composition, structure dissolving* and *conversion of compile time control flow elements into run-time data dependent jumps.* In *variable re-composition,* the set of program variables are scrambled, new variables are created that contain some data from the original variables and are adapted in the program code. *Structure dissolving* consists of the program structure being eliminated by the replacement of procedure calls by procedure code, the substitution of blocks by goto statements or the dissolvement of small variable scopes into global ones. In the *conversion of compile-time control flow elements into run-time data dependent jumps*, the control flow elements such as if-and-while statements are converted into a form that is dependent on the content of variables.

Code obfuscation techniques prevent attacks temporarily on the agent's code and it doesn't make use of a trusted functionality. Since an agent can become invalid before completing its computation, obfuscated code is suitable for applications that do not convey information intended for long-lived concealment. Furthermore, no techniques are currently known for establishing the lower bounds on the complexity for an attacker to reverse engineer an agent's code (Jansen & Karygiannis, 1999).

This approach will cost both execution time, space and communication bandwidth and will require some time-critical restrictions, but gives the agent the possibility to do some security sensitive work without the danger of an immediate explosion of sensitive data by the host (Hohl, 1997).

Westhoff (2001) also states that this approach requires a large number of interactions with a trusted host at every destination specified on the itinerary.

Code transformations (prevention)

An *et al*. (2002) proposed the method of *code transformations* whereby a compiler-based approach is used to conduct *code transformations* for the purpose of obstructing static analysis. Prevention of tampering of the state of the mobile agent is done by *transforming* the data-flow and control-flow information as well as the relationship between them.

The transformation of the control-flow is performed in two steps. Firstly the high-level control structures are converted into if-then-go statements. Secondly the *goto* statement is modified in such a way that the target address is determined dynamically by the switch variable computed in each block of code.

The data-flow transformation can be conducted by either of two methods namely *dynamic computation of branch targets* and *alias through pointer manipulation*. Branch targets are the latest definition of the switch variable. In dynamic computation of branch targets an array is defined whereby the value of the switch variable is computed.

Time sensitive agents (prevention)

Time sensitive agents make use of the fact that it takes time for a malicious host to evaluate an executing agent. If the amount of time needed to execute a mobile agent on a host is limited, then the chance that it will be tampered with is minimized. It is necessary to determine the maximum amount of time needed by a mobile agent to execute safely on an untrusted host. Once this maximum time has elapsed, the agent can be programmed for example to shut down or to move to the next host specified on the itinerary (Grimley & Monroe, 1999).

At the moment, the implementation of one specific countermeasure is seldom enough to provide acceptable security against a malicious host, but it is essential to select and combine solutions according to the needs of the user and the sensitivity of the mobile agent. *Time sensitive agents* can be used to prevent malicious host attacks by combining them with for example code obfuscation techniques.

3.4 Analysis of Threats and Countermeasures

In order to provide a framework for the protection of mobile agents against malicious hosts, it is necessary to analyse the different countermeasures according to the categories of threats as discussed in Chapter 2. Each of the countermeasures is evaluated according to the protection that they provide for each part of the mobile agent. The analysis is discussed in the following sections according to the information provided in the tables.

3.4.1 Integrity Interference

The four different integrity interference threats namely, *transmitting the mobile agent incorrectly, transmitting the mobile agent to a host not on the itinerary, not executing the mobile agent completely* and *executing the mobile agent arbitrarily* as well as the countermeasures that can be used to prevent/detect these attacks are outlined in Table 3.1 and Table 3.2. The tables follow the same outline as in Chapter 2, whereby the grey cells indicate that the type of offence listed in the column does not affect the particular part of the agent and thus no countermeasure/s are needed. *Tamper resistant hardware* (Wilhelm *et al.*, 1998) together with the creation of a *trusted environment* (Sander & Tschudin, 1998**)** is the only two countermeasures that provide the most preventative protection against possible integrity interference attacks. *Tamper resistant hardware* (Wilhelm *et al.,* 1998) can only provide protection against attacks on the migration process if the serialisation and de-serialisation process of the mobile agent is done inside the boundaries of the trusted hardware. Detection methods, such as *itinerary recording with replication and voting* (Minsky *et al.*, 1996) as well as *execution tracing* (Vigna, 1998) also provide protection against integrity interference attacks. One of the problem-areas is to make sure that the current host provides the mobile agent with the correct information as it requested.

Table 3.1: Integrity Interference

	Integrity Interference			
	Transmitting mobile agent incorrectly		*Countermeasures*	*Transmitting*

agent to host not on itinerary *Countermeasures*

Code	Threat	Tamper resistant hardware	Trusted execution	environment

Itinerary recording with replication & voting Execution tracing Reference states No effect

State	Threat	Tamper resistant hardware	Trusted execution	environment

Itinerary recording with replication & voting Reference states State appraisal Execution tracing No effect

Control Flow	Threat	Tamper resistant hardware	Trusted execution	environment

Itinerary recording with replication & voting Reference states Execution tracing No effect

Data	**ID**	Threat	Tamper resistant	hardware

Trusted execution environment Itinerary recording with replication & voting Execution tracing No effect

	Itinerary	Threat	Tamper resistant hardware	Trusted execution	environment

Itinerary recording with replication & voting Execution tracing Threat Tamper resistant hardware Trusted execution environment Path histories Itinerary recording with replication &

voting Mutual itinerary recording Anonymous itinerary

	Initial data	Threat	Tamper resistant hardware Trusted execution environment		

Itinerary recording with replication & voting Digital signature Execution tracing No effect

	Aggregated data		Threat	Tamper resistant hardware	

Trusted execution environment Itinerary recording with replication & voting Partial result encapsulation Digital signature Phone home No effect

	Aggregated essential data		Threat	Tamper resistant hardware	

Trusted execution environment Itinerary recording with replication & voting Partial result encapsulation Digital signature No effect

	Required data	Threat	Tamper resistant hardware Trusted execution environment		

Itinerary recording with replication & voting No effect

Table 3.2: Integrity Interference (Cont.)

	Integrity Interference			
	Not executing the mobile agent completely		*Countermeasures*	*Executing*

mobile agent arbitrarily *Countermeasures*

Code	No effect		No effect	
State	Threat	Tamper resistant hardware Trusted execution environment		

Itinerary recording with replication & voting Reference states State appraisal Execution tracing

Threat Tamper resistant hardware Trusted execution environment

Itinerary recording with replication & voting Reference states Execution tracing

Control Flow	Threat	Tamper resistant hardware Trusted execution environment		

Itinerary recording with replication & voting Execution tracing Threat Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Execution tracing

Data		**ID**	No effect		No effect

	Itinerary	No effect		No effect	
	Initial data	No effect		No effect	
	Aggregated data		No effect		No effect

	Aggregated essential data		No effect		No effect

	Required data	Threat	Trusted execution environment Reference states		

Threat Trusted execution environment Reference states

3.4.2 Integrity modification

Table 3.3 provide the threats and countermeasures relating to integrity modification. As indicated in the table, a number of countermeasures provide protection against the *deletion*, *corruption, manipulation, alteration, misinterpretation and incorrect execution* of the mobile agent. *Tamper resistant hardware* (Wilhelm *et al.*, 1998), *code transformations* (An *et al*., 2002), *code obfuscation* (Hohl, 1997, 1998), *computing with encrypted functions* (Sander & Tschudin, 1998) and providing a *trusted execution environment* (Sander & Tschudin, 1998) provide preventative protection against all integrity modification attacks. Detection methods that can be used include *execution tracing* (Vigna, 1998), *proof-carrying code (*Necula & Lee, 1998), *itinerary recording with replication and voting* (Minsky *et al.*, 1996), *detection objects* (McDermott & Goldschlag, 1996) *and path histories* (Ordille, 1996). As indicated in the table possible countermeasures exist for the whole agent against integrity modification attacks.

Table 3.3: Integrity Modification

		Integrity modification	
		Deleting, corrupting, manipulating, altering, misinterpreting, incorrect execution.	*Countermeasures*
Code		Threat	Tamper resistant hardware Trusted execution environment Path histories Detection objects Itinerary recording with replication & voting Proof carrying code Execution tracing Computing with encrypted functions Code obfuscation Code transformations Time sensitive agents Environmental key generation
State		Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Reference states State appraisal Execution tracing Computing with encrypted functions Code transformations Time sensitive agents
Control Flow		Threat	Tamper resistant hardware Trusted execution environment Path histories Detection objects Itinerary recording with replication & voting Execution tracing Computing with encrypted functions Code obfuscation Code transformations Time sensitive agents
Data	**ID**	Threat	Tamper resistant hardware Trusted execution environment Path histories Itinerary recording with replication & voting Time sensitive agents Environmental key generation
	Itinerary	Threat	Tamper resistant hardware Trusted execution environment Path histories Itinerary recording with replication & voting Mutual itinerary recording Anonymous itinerary Time sensitive agents Environmental key generation
	Initial data	Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Digital signature Environmental key generation
	Aggregated data	Threat	Tamper resistant hardware Trusted execution environment Trusted third party Itinerary recording with replication & voting Phoning home Partial result encapsulation Digital signature Time sensitive agents Environmental key generation
	Aggregated essential data	Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Partial result encapsulation Digital signature Time sensitive agents Environmental key generation
	Required data	Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Reference states Partial result encapsulation Digital signature

3.4.3 Availability

Availability threats consist of *denial of service*, *delay of service* and *transmission refusal*. The countermeasures to combat the availability threats are given in Table 3.4, 3.5 and 3.6. Providing protection against denial of service attacks can be obtained by the use of *tamper resistant hardware* (Wilhelm *et al.,* 1998), the creation of *a trusted execution environment* (Sander & Tschudin, 1998) and by using *time sensitive agents* (Grimley & Monroe, 1999). It must be noted that *time sensitive agents* (Grimley & Monroe, 1999) are to be used in collaboration with other countermeasure techniques such as *code obfuscation* (Hohl, 1997, 1998) in order to make it more viable. Table 3.4 again outlines the lack of countermeasures for attacks that denies the mobile agent information as requested.

Table 3.4: Availability (Denial of service)

		Availability			
		Denial of Service			
		Execution resources (memory & CPU denied)			
		Countermeasures	*Data denied / Bombarded with irrelevant information*		
		Countermeasures			
Code		Threat	Tamper resistant hardware Trusted execution environment		
Time sensitive agents			No effect		
State		Threat	Tamper resistant hardware Trusted execution environment		
Time sensitive agents			No effect		
Control Flow		No effect		No effect	
Data		**ID**	No effect		No effect
	Itinerary	No effect		No effect	
	Initial data	No effect		No effect	
	Aggregated data		No effect		No effect
	Aggregated essential data		No effect		No effect
	Required data		No effect		Threat
	Trusted execution environment				

Table 3.5 provides the countermeasures of the delay-of-service attacks. *Tamper resistant hardware* (Wilhelm *et al.,* 1998), providing a *trusted execution environment* (Sander & Tschudin, 1998) and *time sensitive agents* (Grimley & Monroe, 1999) are the only countermeasures that can possibly protect the mobile agent against delay-of-service attacks.

Delaying the provision of data as requested by the mobile agent once again proves to be difficult to protect against.

Table 3.5: Availability (Delay-of-service)

	Availability			
	Delay of service			
	Execution resources	*(memory & CPU delayed)*		
	Countermeasure	*Data is delayed*	*Countermeasures*	
Code	Threat	Tamper resistant hardware Trusted execution environment		
Time sensitive agents		No effect		
State	Threat	Tamper resistant hardware Trusted execution environment		
Time sensitive agents		No effect		
Control Flow	No effect		No effect	
Data	**ID**	No effect		No effect
Itinerary	No effect		No effect	
Initial data	No effect		No effect	
Aggregated data		No effect		No effect
Aggregated essential data		No effect		No effect
Required data		No effect		Threat
Trusted execution environment Time sensitive agents				

In Table 3.6 the list of countermeasures providing protection against the host refusing to transmit the agent are given. Prevention techniques include *tamper resistant hardware* (Wilhelm *et al.*, 1998), *trusted execution environment* (Sander & Tschudin, 1998) and *time sensitive agents* (Grimley & Monroe, 1999). *Mutual itinerary recording* (Roth, 1998) and *itinerary recording with replication and voting* (Minsky *et al.*, 1996) are two detection methods that can be used.

Table 3.6: Availability (Transmission Refusal)

		Availability	
		Transmission refusal	
		Transmission refusal	*Countermeasures*
Code		Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
State		Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
Control Flow		Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
Data		**ID**	Threat Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
	Itinerary	Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
	Initial data	Threat	Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
	Aggregated data		Threat Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
	Aggregated essential data		Threat Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording Time sensitive agents
	Required data		Threat Tamper resistant hardware Trusted execution environment Itinerary recording with replication & voting Mutual itinerary recording

3.4.4 Confidentiality

Countermeasures for confidentiality attacks are provided in Table 3.7 and Table 3.8. *Tamper resistant hardware* (Wilhelm *et al.*, 1998), *trusted execution environment* (Sander & Tschudin, 1998), *code obfuscation* (Hohl, 1997, 1998), *code transformation* (An *et al*., 2002), and *environmental key generation* **(**Riordan & Schneier, 1998) are prevention mechanisms that prove viable against eavesdropping attacks. This is because they either make use of a trusted environment or the different components of the mobile agent are encrypted. Theft of the mobile agent is difficult to protect against and can only be protected in a trusted network or when making use of techniques where the agent is duplicated, such as *using a mobile agent system* (Yee, 1997).

Table 3.7: Confidentiality

		Confidentiality			
		Eavesdropping	*Countermeasures*	*Theft*	*Countermeasures*
Code		Threat	Tamper resistant hardware Trusted execution environment Code obfuscation Code transformations Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
State		Threat	Tamper resistant hardware Trusted execution environment Code transformations	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
Control Flow		Threat	Tamper resistant hardware Trusted execution environment Code obfuscation Code transformations Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
Data	**ID**	Threat	Tamper resistant hardware Trusted execution environment Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
	Itinerary	Threat	Tamper resistant hardware Trusted execution environment Environmental key generation Anonymous itinerary	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
	Initial data	Threat	Tamper resistant hardware Trusted execution environment Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
	Aggregated data	Threat	Tamper resistant hardware Trusted execution environment Trusted third party Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
	Aggregated essential data	Threat	Tamper resistant hardware Trusted execution environment Environmental key generation	Threat	Tamper resistant hardware Trusted execution environment Using a mobile agent system
	Required data		No effect		No effect

Reverse engineering of the mobile agent can only be protected when making use of *tamper resistant hardware* (Wilhelm *et al.,* 1998), a *trusted execution environment* (Sander & Tschudin, 1998), *time sensitive agents* (Grimley & Monroe, 1999) and methods that incorporate encryption techniques (such as *environmental key generation* **(**Riordan & Schneier, 1998)).

Table 3.8: Confidentiality (cont.)

	Confidentiality	
	Reverse Engineer	*Countermeasures*
Code	Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents Code obfuscation Code transformation Environmental key

generation

State		Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents
Control Flow		Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents Environmental key generation
Data	**ID**	Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents
	Itinerary	Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents Anonymous itinerary
	Initial data	Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents Environmental key generation
	Aggregated data	Threat	Tamper resistant hardware Trusted execution environment Trusted third party Time sensitive agents Partial result encapsulation
	Aggregated essential data	Threat	Tamper resistant hardware Trusted execution environment Time sensitive agents
	Required data	No effect	

3.4.5 Authentication

Authentication attacks (namely *masquerading* and *cloning*) can only be prevented when deploying the mobile agent in a *trusted execution environment* (Sander & Tschudin, 1998) (see Table 3.9).

Table 3.9: Authentication

		Authentication			
		Masquerading	*Countermeasures*	*Cloning*	
		Countermeasures			
Code		No effect		No effect	
State		No effect		No effect	
Control Flow		No effect		No effect	
Data	**ID**	No effect		Threat	Trusted execution environment
	Itinerary	No effect		No effect	
	Initial data	No effect		No effect	
	Aggregated data		No effect		No effect
	Aggregated essential data		No effect		No effect
	Required data		Threat	Trusted execution environment	

Digital signatures No effect

3.5 Conclusion

This chapter provided a detailed look at the possible countermeasures for protecting a malicious agent against attacks by a malicious host. The countermeasures were divided into four different classes, namely *trust-based computing, recording and tracking, cryptographic techniques* and *obfuscation and time.* Section 3.4 analysed the different countermeasures by creating matrixes of threats against countermeasures. This analysis is essential in providing a framework for the protection of a mobile agent and is referenced again in later chapters. Chapter 4 provides information and discussions on current mobile agent systems as well as applications that incorporate security techniques.

CHAPTER 4

MOBILE AGENT MODELS, FRAMEWORKS, ARCHITECTURES, SYSTEMS AND APPLICATIONS

4.1 Introduction

Working towards a framework for the protection of mobile agents against malicious hosts, it is essential to study proposed mobile agent models, frameworks and architectures as well as current mobile agent systems and mobile agent applications. A large number of such systems are available in literature but only a few incorporate security methods into their designs. Insights into the different proposals, systems and applications that have integrated security techniques into their designs, are offered in this chapter as well as detail of the designs. The analysis of the frameworks, systems and applications will guide us towards establishing a set of criteria for a mobile agent security framework and ultimately the requirements for such a framework

The study of the designs of the systems listed in this chapter consists of a description of each system followed by a short summary detailing the types of countermeasures incorporated into the designs as well as the advantages and disadvantages of the designs. Each section ends with a short integrated discussion on the analysis of the different systems.

4.2 Mobile Agent Models and Frameworks

For the purpose of this research, we have studied more than thirty mobile agent models, frameworks and architectures as described in literature. The literature that was studied do not necessarily agree on (or distinguish between) terminology such as *frameworks, models, architectures* and *systems.* For example, some describe their work as a “framework”, whilst others use the term “model” to structure similar research efforts. It is outside the scope of this research to get absorbed in the definitions of these terms. Our aim is rather to recapitulate the essence of the different research efforts that were investigated and therefore, we merely use the same terminology offered by the authors. These proposals and implementations introduce interactions between different countermeasures as well as measures not discussed as part of the previous chapters. In the following paragraphs, we describe the most prominent of these systems in terms of the basic functionality of the security implementation of each, which include the type of countermeasures used. We conclude the discussion on each system by an evaluative remark on the most salient points as well as the drawbacks regarding the security implementation of each system.

The proposed mobile agent models and frameworks are divided into those that make use of some kind of *trusted environment* and those that can operate in an *open environment* (where no trusted environment is created or specified).

4.2.1 Trusted environment

The mobile agent models and frameworks in this section make use of the concept of creating a trusted environment. This is achieved by either using trusted hardware, a trusted third party or by authenticating the hosts.

Police office model (POM)

Guan *et al.* (2000) presented a mobile agent security model by setting up special hosts called police offices within defined regions. These police offices are based on the concept of police stations in the real world. The idea of *POM* is to prevent a large number of attacks that can be performed by malicious hosts against mobile agents. This is achieved by the separation of critical components (of the agent) and only allowing non-critical components to be executed at the remote hosts.

Regions are defined that consist of a number of special hosts connected to each other. Regions may not overlap and hosts within a region have high-speed connections relative to low speed connections to hosts outside the region. A *police office* is a special host inside a predefined region, with certain characteristics, namely, it is a trusted host, it is responsible for controlling all hosts in the region and it is accessible through any host specified in the region. All mobile agents are divided into distinct parts, namely a *master* part that is security critical and a *slave* part that is security-free. The slave part can only migrate between the host and the *police office*.

Once a mobile agent needs to migrate to a specific host (*$Host_1$)* in a region, it first migrates to the *police office* of the specific region where-in *$Host_1$ is* located. The master part of the mobile agent remains at the police office and sends the slave part to *$Host_1$* where it will perform security-free actions. After completion the slave part returns to the police office with the results obtained. Computations with the returned results are performed by the master part at the police office and on completion the mobile agent can migrate to the next host on its itinerary.

The countermeasures used in this model are based on the creation of a trusted execution environment by using the notion of a trusted entity for secure computations of the agents. Although the autonomy and mobility aspects of the mobile agent are restricted within this model, the model can be implemented successfully in applications that are reliant on a trusted environment for the secure execution of mobile agents. One of the advantages of the model is that the security critical data (such as keys for encryption / decryption) are only migrated between trusted entities and is thus protected from malicious acts by foreign hosts. Computational cost implications for the remote hosts are minimal due to the agent only retrieving information at the different hosts and the computations completed at the trusted entity.

Although *POM* offers a number of inspiring security benefits to tackle the malicious host problem, there are some notable weaknesses. For example, the partitioning of the mobile agent in a master and slave part can be difficult depending on the type of application to be implemented. The creation of regions can also pose problems because they are required not to overlap and for every region a trusted entity needs to be established. The model does not supply the relation between the number of hosts and the trusted entity within a domain. This can lead to a bottleneck at the trusted entity, if a large number of hosts are defined within a region. The size of the itinerary of the agent will be extensive due to the inclusion of the trusted entities as well as the hosts within a domain to be visited. The model also defines the police office as a separate entity, which causes problems such as establishing the responsible entity for the creation and maintenance of the police offices (both in terms of hardware and software). The communication sessions within the model also increases substantially due to the mobile agent being split and it not being migrated as a whole to the list of remote hosts.

Security enhanced mobile agents

Varadharajan (2000) proposed a security model whereby the notion of a *security enhanced mobile agent* is introduced. The security enhanced agent carries a passport that contains its security credentials and related security code.

Each host in this model contains a *trusted security management* component (SMC), which maintains security policy information as well as public and private keys. Hosts that obey the same security policies are also grouped together to form a *domain*. Each domain has a security authority, namely the *security management authority* (SMA) that is responsible to interact with the SMC's in the domain in order to establish and maintain security policies. It is also responsible to interact with SMAs in other domains. Each SMC and SMA in the system has public-private key pairs and they are trusted entities.

The *security enhanced mobile agent* has a structure consisting of an *identifier, privilege token, data store, agent code* and *security tags.* The *identifier* field consists of a unique identifier assigned at creation, a creator-principal certificate that refers to the creator of the mobile agent, a creator-SMC certificate (signed by the SMA), a timestamp when the mobile agent was created and the intended lifetime of the mobile agent. The p*rivilege token* contains privileges to be used in conjunction with the policy at the host in order to determine whether a request by a mobile agent is to be allowed or disallowed. Each privilege in the token consists of an identifier, a timestamp and a lifetime. The d*ata store* contains the execution state and itinerary of the mobile agent, while the *agent code* is divided into two types of code, namely *application code* as specified by the creator of the mobile agent and *security code.* Security code is a default set of methods automatically added when the security enhanced mobile agent is created. Two *security tags* are identified, namely one created by the owner that contains the hash value of the original security enhanced mobile agent and one which is generated by the sending host and contains the hash value of the first certificate and the *data store*.

When creating a mobile agent, a unique *identifier* is generated and the creator-principal certificate as well as the creator-SMC certificate is added. The privileges of the mobile agent are defined, the default security code is added and the security tags are generated. The migration request from one host to the next host as specified in the itinerary contains the identity of the sending host, its SMC certificate, the target host, the operation being requested and the valid time period.

The static part of the mobile agent (code and creator granted privileges) is signed using the private key of the creator. The dynamic part is signed using the private key of the sending host. The receiving host can verify the authenticity of the sending platform as well as check the integrity of the application code. If all checks are successful, the mobile agent is executed and a copy of the results is stored in *data store*. The SMC of the executing host produces a signed hash digest of the results along with a timestamp using its private key.

The countermeasures incorporated into the design of *Security enhanced mobile agents* consist of policies, encryption, digital signatures and time techniques in order to protect the agent against malicious modifications. These countermeasures combined provide protection for the mobile agent's code and data against integrity modification and confidentiality attacks. The creation of keys for encryption and decryption purposes is managed within a security manager contained in every remote host. The creation of a secure software component within every host as well as the creation of a trusted security management authority can prove this model viable in a small closed environment.

This model creates a trusted environment by authenticating the hosts as well as the agents before migration. Aggregated data is also signed at every host in order to protect the results obtained. The use of a time stamp within the concept of a trusted environment seems to provide additional protection against attacks such as *reverse engineering* and *delay of execution*.

The disadvantages of this model are the autonomy and mobility restrictions placed on the agent, due to the agent only being released in a trusted domain. The establishments of trusted entities as well as specifying the domains place additional restrictions on the mobile agent.

Flexible IPR for software agent reliance (FILIGRANE)

The purpose of the *FILIGRANE* project is to develop a security framework for mobile code commerce. Jalali *et al.* (2000) proposed the project, which makes use of the standard *Java Cryptography Architecture* and *Java Cryptography Extension* as the underlying security infrastructure.

The model consists of a number of entities, namely a *certification authority*, a *smart card issuer*, the *producer* of the mobile code, the *provider* who sells services or information, the *end user* who is registered to download software, a *rights clearing house* which is responsible for the definition of rights between entities, a *fee collecting agency*, a *quality label service* and an *E-notary* which acts as a trusted repository for all entities.

A mobile agent is protected in *FILIGRANE* by means of a *signature* (more than one entity can sign the agent as a whole or individual files that form part of the agent), *encryption* (the agent is encrypted to firstly avoid reverse engineering and secondly to control the execution and reading there-of), *rules* (used for describing the contracts between different entities and are checked by the host to detect breaching), *watermarks* (are embedded in the agent for the purpose of identification of code, integrity checks and avoidance of reverse engineering), *obfuscation* (modifications of the code in order to make the process of reverse engineering difficult), *tagging* (the identification of the mobile agent). All the mentioned protection mechanisms are combined inside a package called a *code envelope*.

The *FILIGRANE* framework uses smart cards as secure physical tokens for storage and usage rights. The operations on the mobile agent are controlled by a security engine, which are embedded in the hosts in order to control mobile agent packaging and execution. The security engine consists of a number of configurable name services. A session manager registered and manages these services.

The *FILIGRANE* framework makes use of a number of techniques in order to create a trusted environment for a specific mobile agent application. These techniques include the use of smart cards, creation of policies, encryption of the agent, watermarking and code obfuscation. The *FILIGRANE* project introduced a new security concept within the mobile agent environment, namely *watermarking* that can be used in the protection of the agents. Watermarking is the practice of imperceptibly, altering a cover to embed a message about that cover. The integration of watermarking, code obfuscation and encryption techniques, is a new direction taken in mobile agent security technology and can protect an agent against integrity modifications and confidentiality attacks (especially eavesdropping and reverse engineering). The cryptography aspects are done via an extension to the Java language, which relates to no additional costs if the mobile agent system is based on the mentioned language.

A drawback to this framework is the use of trusted hardware in the form of smart cards. This is however, an application specific security framework for the trading of mobile code on the Internet. Additional requirements for the implementation of this framework include the trusted entity within a domain as well as the security engine embedded at every remote host.

M&M Mobile agent framework

Marques *et al.* (2001) have developed a component framework for the deployment of mobile agents by making use of components that allow applications to become able to send and receive mobile agents. In *M&M* there are no mobile agent platforms and the security aspects must be integrated with security mechanisms already in place in the applications. The reason being that the mobile agents migrates between applications and not hosts.

Marques *et al.* (2001) distinguish between different application domains that run on different environments. Two types of environments are considered, namely *closed* environments where all nodes belong to the same authority (for example network management and software upgrading) and *open* environments where the nodes may belong to different authorities (for example electronic commerce and information gathering). The *M&M* mobile agent framework operates in an *agent-accountable* environment, which is a mix between a closed and open environment. An agent accountable environment is defined as a set of cooperating organisations that deploy a mobile agent infrastructure for supporting their operations. These organisations form a trusted environment.

The security model consists of three modules, namely the *mobility component* (that provides the basic infrastructure for the migration and management of the mobile agents), the *security component* (responsible for handling tasks such as public and private key management, user and host authentication and authorisation) and the *security manager* (this component must be instantiated to allow for permission-based mechanisms to be employed).

The security features of the framework are *authentication and authorisation, confidentiality and privacy, accountability, installable services and remote management* as well as *cryptographic primitives*. Each of the principals in the *M&M* framework has a private and public key pair stored on a local key store. Upon creation of a mobile agent, the owner has to provide a password used for uniquely connecting the mobile agent to the owner. An object containing the identity information of the mobile agent (such as itinerary, name, hash of code, creation and expiry time) is created and signed with the private key of the owner to prevent tampering. The owner can only create mobile agents in the nodes where it is registered. Key distribution is locally done from a key-store and a LDAP server stores all keys for a whole system. When the mobile agent migrates to a host, its identity is firstly sent whereupon the *security component* will validate and either accept or reject the mobile agent.

SSL sockets are used to protect the mobile agent while migrating and the data and execution state is kept private. In terms of accountability, log reports are kept where each log entry consists of a security level, an origin and a timestamp. Cryptographic primitives such as partial result authentication codes are used for the implementation of secure information gathering protocols.

The *M&M* framework integrates countermeasures such as digital signatures, encryption, passwords, policies, partial result authentication codes and log reports for protecting the agent. The *M&M* framework can be useful in applications that require a trusted domain for implementation. A main feature of the *M&M* framework is that it distinguishes between types of environments (open, close or agent-accountable). A security manager is also built-in at every host for handling authentication and key management related tasks.

M&M does not require an execution environment on each host, but transfers agents between applications. This has additional requirements for the host as well as the agent in terms of software needed for development.

Distributed transactions

Vogler *et al.* (1997) present an architecture for a mobile agent system that guarantees protection of the hosts as well as protection of the mobile agents. This architecture makes use of a trusted third party that contains information about all instances of the closed system. The trusted third party is also responsible for key generation, as well as the logging of data about the mobile agents and hosts (such as the identities of the mobile agent and the host, as well as time intervals). For acceptance in this architecture, it is assumed that every principal possesses a certified public key pair and is registered at the trusted third party.

The initiator of the mobile agent registers the agent at the trusted third party, which generates a unique mobile agent identification. This communication is protected by public key encryption. The next host on the itinerary is determined with the aid of a special trader for agents. Once a contract between the target host and agent is negotiated, a copy of the mobile agent is sent to the host by making use of two different mechanisms namely *distributed transaction* processing and encryption.

The protocol steps for the transfer of the mobile agent starts off with the originator host initiating a *distributed transaction* involving the target host and the trusted third party. A session key is requested by the originator host for the secure transfer of the mobile agent. The session key is generated and sent to the originator and target host by the trusted third party. The copy of the mobile agent is encrypted with the session key and transferred to the target host, whereupon it acknowledges and decrypts the mobile agent.

This architecture guarantees that no other entity besides the trusted hosts have access to the mobile agent. If one of the trusted hosts attacks the agent, the logging facilities can be used to detect the breach and trace it back to the host.

The countermeasures employed by the *Distributed transactions* architecture are a trusted third party for key distribution and logging as well as the encryption of agents between entities. A closed environment of trusted hosts is created by requiring the remote hosts to register at a trusted third party within a certain domain. The creation and distribution of a session key for the encryption of the agent can be used effectively, with the only disadvantage being the additional communication sessions between the hosts and the trusted third party. The trusted third party is also required to keep logs of the execution of the agent for detection purposes.

Mansion

Mansion provides a logical model for designing distributed multi-agent applications. Van't Noordende *et al.* (2002) present the security architecture of *Mansion*, which provides protection for hosts as well as mobile agents.

The *Mansion* framework consists of a *logical* model that is used to structure an application and a *physical* model that consists of a network of hosts on which the logical model is mapped. An application is modelled as a closed *world* containing a set of hyper-linked *rooms*, which determines how they are connected. Entities in a room (such as mobile agents) are injected into a world through a *daemon*, which does some security and consistency checks. Global services that are accessible to agents in all rooms are contained in an *attic*. A *basement* in each world keeps track of the information needed to make the world function, such as location tracking. *Zones* are used to express the distribution and security properties of the hosts and are protected by public-key cryptography.

The owner of the mobile agent provides only a list of trusted hosts on its itinerary in order to make sure that it does not migrate to a host in an untrusted zone. If information located on an untrusted host is needed, a helper agent is created with minimal functionality to retrieve the data and return to its parent (the mobile agent). An *agent container* is used to store the agent's data and code. Upon migration to a next host the contents of the mobile agent is signed with the private key of the host. An audit trail of visited hosts is kept in order to determine illegal changes to the mobile agent.

The *Mansion* architecture protects the mobile agent against malicious modifications by making use of a trusted network of hosts. The countermeasures employed include the use of a trusted entity for distributing the agent into a specified domain, digital signatures, encryption of the agent as well as keeping path histories of the hosts visited. As *Mansion* only lets the agent be deployed amongst a pre-defined set of hosts, it will be suitable for applications that require a trusted domain for deployment.

The creation of slave agents for retrieving data means that the mobile agent is not executed at the remote host but on a trusted entity. This implies additional communication sessions for retrieving data.

Planet

Kato *et al.* (1996) designed a distributed computing system, called *Planet* for a worldwide network. The *Planet* system model uses five basic abstractions namely, an *object* (an entity that encapsulates data as well as programs to manipulate the data), *activity* (a computational entity that encapsulates the current state of computation), *place* (computational resources), *repository* (worldwide object store) and *protection domain* (control object accesses).

Two techniques are used in the protection domain mechanism, namely the use of *virtual-memory management* hardware units and providing each protection domain with a distinct *virtual address space*. A host sends an object to another host in a secure way by following a number of steps. The *server* object requests an *authenticator* object in order to register its service in the service directory. The authenticator object registers the service and generates a wrapper and un-wrapper pair. The wrap operation encrypts an object and attaches a unique object name to the object. It also unloads the secure object from the protection domain, while the unwrap operation loads and decrypts the object. The authenticator object passes the un-wrapper to the server object that waits until an object arrives that can be unwrapped (by making use of the provided un-wrapper). The client object makes a request for authentication of itself as well as a request for service from the authenticator's object service directory. The authenticator object passes the wrapper to the client object, upon which the client object creates an object and uses the provided wrapper to encrypt the object. This secure object is sent to the server object that uses the un-wrapper to unwrap the object.

Planet makes use of cryptographic protocols as well as protection domains and specific hardware components to protect agents. An advantage is that the required hardware component is not specialised, and is already available on virtually all modern computers. The creation of a trusted domain is achieved by the use of a trusted entity, authentication of the hosts as well as the encryption of the agent between hosts. This model is based on a distributed architecture and can be used in an environment that requires a trusted domain for distributed applications

The authentications and world-wide object store required by this model involves additional hardware/software to be created within the system.

Proxy-agents and trusted domains

Mitroviæ & Arribalgaza (2002) proposed an architecture for secure mobile agent systems by using trusted domains and proxy agents. The architecture introduces the concept of *security proxy agents* as facilitators of security services for both mobile agents and mobile agent systems. The security proxy agents contain a set of security and cryptographic mechanisms that can be used.

There is a *proxy factory* present in every mobile agent system that is responsible for the creation of security proxy agents as well as the association of these with mobile agents. One or more security proxy agents guard the entrance to the mobile agent systems. The proposed architecture relies on trusted domains, where every domain has one or more mobile agent systems that deal with security. The authentication of mobile agents and mobile agent systems as well as the application of trust policies is done by one domain. By checking the signature of the mobile agent or modules within the mobile agent systems, it is possible to detect alteration of the mobile agent's code.

Upon migration from one domain to another, the mobile agent will move to the *domain* *controller* where the proxy factory service is located. A security proxy agent is created and assigned to the mobile agent, equipped with the credential of the mobile agent. Both the mobile agent and the security proxy agent move to the domain controller of the destination host, where the security proxy agent checks both for alterations. If no alteration is detected then the security proxy agent of the mobile agent and the destination domain controller negotiates possible cooperation. Once cooperation is agreed-on, the mobile agent is decrypted and can move freely within the domain. The security proxy agent will reside on the domain controller, waiting for the mobile agent to finish its journey within the domain. Upon completion the mobile agent is again encrypted and the journey is continued to the next domain.

The *Proxy agents and trusted domains* architecture implements the use of a trusted entity for the creation of the proxy agents, the creation of security domains, as well as the use of policies and authentication techniques. This architecture can typically be deployed in an environment that allows for the creation of multiple domains with a secure entity present in every domain as well as applications that need a trusted environment for implementation.

Additional communication sessions is however a downside to this architecture, due to the cooperation agreement between the hosts and the proxies. The autonomy of the mobile agent is limited due to the creation of trusted domains. Additional cost implications can also have an affect if the trusted entities do not exist within the specified domains.

Electronic supermarkets

Wu (2000) proposed the use of *electronic supermarkets* in order to solve or minimise the security problems experienced in mobile agent systems. An electronic supermarket is a database owned by a trusted authority that is responsible for the management thereof. In order to secure visiting mobile agents, trusted hardware is implemented on all *electronic supermarkets.* Upon entering an electronic supermarket, the identity of the mobile agent is checked and recorded to a databank. This will alleviate problems encountered in the case where a mobile agent is cloned.

Electronic supermarkets create a trusted set of hosts by requiring the implementation of tamper resistant hardware devices on every host. Domains are specified and a trusted entity is required within every domain. The purpose of the trusted entity is to allow for the registration of mobile agents before entering the domain. This feature will assist in protection of the host and can assist in detecting if a malicious host has made an exact copy of the agent. *Electronic supermarkets* is an application specific model that can be used for applications that require the agent to conduct e-commerce related functions.

The use of trusted hardware within the model protects the agent from a malicious host but does however have additional cost implications for the remote hosts.

Domain name exchange (DNX)

Schütz *et al.* (2000) presented different security techniques designed for the *Domain Name eXcgange (DNX)* platform. Cryptographic tools are used for the safe routing of mobile agents between hosts and a secure Java-based platform guarantees the safe execution of the mobile agents.

Every host in the mobile agent network is structured with three main parts, namely *an agent space* for the execution of the mobile agents, *a services space* that provides the interface between the mobile agent and the mobile agent platform, and the *security space* which controls the incoming and outgoing mobile agents. The security space consists of a *sender agent*, which is a stationary agent responsible for encrypting and signing the outgoing agents and a *receiver agent* responsible for decrypting and verifying the incoming agents.

The *JavaSeal* agent platform (Vitek *et al*., 1998) is used to ensure the secure execution of agents, controlled communications between agent environments and enforcing the security policy of the agents.

The *DNX* platform integrates encryption techniques as well as the notion of a secure platform for the safe execution of agents. The notion of a protection domain is achieved by using *JavaSeal* (Vitek *et al*., 1998), which is a micro-kernel mobile agent system developed using Java. Applications that are required to execute within a trusted environment and that are based on the Java language can be implemented by using the *DNX* model. It provides secure transmission of agents between hosts and the authentication of agents and hosts.

The disadvantages are the restrictions placed on the autonomy of the mobile agent as well as the required space necessary on the remote hosts.

Supervisor-worker framework

Fischmeister (2000) described a framework whereby confidential data is left at a secure place and slave agents are sent to untrusted hosts carrying a limited amount of information.

The framework makes use of *supervisors*, who are responsible for dividing tasks into sub-tasks, controlling the *workers,* and generating reports after merging the outputs as received from the worker agents. The workers implement all methods needed for accepting and fulfilling tasks and sending reports to the supervisor. Upon initialisation, the user creates the *supervisor* and delegates a task. The supervisor splits the task into different subtasks and moves to a host in the area where the *workers* will perform their subtasks. The supervisor then creates the workers and delegates the subtasks, upon which the workers move to the destined hosts and relay the results back to the supervisor. The supervisor merges the results and conveys it back to the user.

The creation of a trusted execution environment for the execution of agents within the *Supervisor-worker framework* is achieved by creating secure domains. Every domain is equipped with a trusted third party for accepting incoming agents within the domain as well as the creation of slave agents in order to retrieve results from hosts within the domain. This trusted entity is also responsible for the computation of the agent once all the required information is retrieved. The *Supervisor-worker framework* will be adequate within applications that require a trusted environment, where the creation of trusted hosts is replaced by the computation referred to a trusted third party.

The supervisor has to reside on a trusted host in the vicinity of the workers, which is a major disadvantage of this framework. The strategies for dividing the task into subtasks, as well as the protection of the slave agents, are also unclear.

4.2.2 Open environment

A small number of frameworks and models are available in literature that combine security techniques without the creation of a trusted environment. The proposed frameworks and models for an open environment are discussed in this section.

Secure and automatic wrapper for mobile agents (SAWMA)

Luo (2001) proposed a detection approach for the protection of mobile agents by making use of three layered techniques, namely *secret spreading*, *obfuscation*, and *Java watermarking*. Before migration to a host the mobile agent uses a *wrapper* to converts the clear text of the mobile agent code cryptographically. By making use of these techniques the code of the mobile agent will be more difficult to be attacked by a malicious host.

The sharing of secrets is done by attaching time limits to the lifetime of the agent as well as making use of distributed agents, in order to share the secrets amongst more than one agent. Methods to evoke secret sharing can be for example *recursive encryptions* and the *split of variables*. The author developed a technique called *class evolution* that enables the encryption to change throughout the execution cycle. This technique also includes a method whereby mobile agents don't execute in clear text form on the remote host.

Code watermarking attempts to detect any manipulation of the mobile agent's code. Watermarks embedded in Java code are dynamic; meaning the detection of the watermark requires the execution of the code. The execution status of the mobile agent is observed by its owner and detection of modification is done by the verification of the watermark.

Combining watermarking, code obfuscation and time techniques forms the basis of *SAWMA*. As also proposed in the *FILIGRANE* project (Jalali *et al*., 2000), the combination of these three techniques is worth looking into for the purpose of protecting the mobile agent. Additional countermeasures within the *SAWMA* model include the encryption of the agents as well as time limits. Secret data carried by the agent is protected by means of creating a certain number of mobile agents, each containing a sub-set of the secret data. By using this method the data that is declared as secret is spread amongst a number of agents and is not contained in a single agent. This framework can be used for implementation within an open environment where a trusted set of hosts is not required and where the mobile agent can be split into a number of distributed agents.

The drawback is the creation of the distributed agents for the purpose of secret sharing. The migration paths for these agents is also unclear (in terms of whether they will visit the same hosts), as well as the increase in communication sessions due to the increase of mobile agents within the system. The actual creation of the distributed agents for secret sharing requires additional computational costs from the owner or creator of the agent.

Agent factory

Brazier *et al.* (2002) presented an approach whereby a blueprint of the mobile agent's functionality is migrated instead of the code, data and state. The mobile agent is designed to have a compositional structure and the resulting specification of the mobile agent is the blueprint. An *agent factory* is responsible for the generation of the mobile agent from the received blueprint for a specific platform. An agent platform requires libraries of re-usable mobile agent components as well as ways to describe the functionality of the mobile agent components.

The blueprint contains descriptions of the interfaces of the components within an agent factory as well as additional information regarding the relation between these components on two levels of detail. These two levels are a *conceptual description*, which is the blueprint of the components, as well as interactions and interfaces between them. A *detailed description* includes the code and definitions.

The blueprint of a mobile agent does not change during the existence of the agent and by adding an integrity check (such as a digital signature) to a blueprint it is possible to detect whether the blueprint has been changed. The advantage of making use of agent factories is that mobile agents are able to migrate between non-identical platforms.

The *agent factory* makes use of a separate entity required for the creation of the agent from a conceptual description. Within the framework the conceptual description is migrated and not the agent itself. The only protection provided is digital signatures added to the blueprint. This framework provides useful mechanisms for mobile agent applications within an open environment, due to a specific language-related execution environment not being required on remote hosts.

The question arises whether the problem of protecting a mobile agent is shifted to the protection of the blueprint. Another disadvantage is the existence of the agent factory responsible for the creation of the detailed description from the specifications contained in the conceptual description.

Security framework for a mobile agent system

Bryce (2000) describes a distributed security infrastructure for mobile agents, in which the mobile agents themselves are used to enforce security *properties*. The security properties are *believability*, meaning that mechanisms are provided for authentication and *survivability*, meaning the agent can be programmed to survive attacks by malicious hosts.

The framework makes use of encryption as well as replication and voting techniques. The mobile agent carries signed credentials that verify its properties and its security policy is designed to adapt to a possible changing environment. An agent defines a set of access *groups* that represents a set of access rights. Keys for encryption and decryption purposes are carried within a predefined class of the agent.

The countermeasures used within the *Security framework for a mobile agent system* is based on the creation of cooperating agents. The replication and voting techniques used in the framework, can detect manipulations of the agent and the encryption of the agent (or parts there-of) provides safety against confidentiality and integrity attacks. Applications that consist of a small number of mobile agents can be implemented.

Disadvantages of this scheme is the protection of the key carried by the mobile agent for decryption purposes as well as the additional communication sessions created by the cooperating agents. The overhead costs of the system also increase with the use of replication and voting techniques.

Mobile code security framework

Tan & Moreau (2002) described a method by which execution traces are enhanced through a trusted third party called a verification server. The construction of a mobile agent is simplified by making use of mobile agent templates.

The execution tracing protocol as proposed by Vigna (1998) is changed in this framework through the introduction of a verification server that is responsible for verifying the traces, instead of the local host. The framework consists of a *certification authority*, responsible for the issuing of certificates to other entities in the framework as well as the management of keys and a *verification server*, which is a trusted third party responsible for the verification of execution traces submitted by hosts on behalf of the agent owner. Two types of certificates are used, namely *capability certificates* and *execution certificates*. Capability certificates associate the identity of the host with its capability of correctly executing the mobile agent template. A mobile agent template identifier replaces the public key present in a normal certificate. The private key of the verification server signs these certificates. Execution certificates identify the success of the validation process and are generated by the verification server for a host. An execution certificate contains a hash of the agent's code and state, a timestamp, identity of the verification server, identity of the host and the results of the trace. A record of all invalid execution traces that were detected is kept in a *capability certificate revocation list*. The verification server submits an entry containing the identity of the server, identity of the host platform, fault detected in the trace and a timestamp.

Before migrating to a new host, the following occurs: The mobile agent contains a list of template identifiers that represents the templates it is composed from. This as well as the code of the mobile agent is signed by the agent owner platform. The identifiers are sent to the new host platform, which checks if it possesses capability certificates containing some or all identifiers specified. The capability certificates are sent back to the mobile agent for review in order to decide to migrate or not.

After migration to the new host platform, the mobile agent is executed and an execution trace is created by the platform. The created trace is submitted to the verification server where it is validated and an execution certificate is prepared and sent back to the host platform (once the trace is validated). The host platform keeps a copy of the execution certificate and the original is sent with the mobile agent to the next host platform. If the trace is not validated no certificate is issued and an entry is written to the capability certificate revocation list.

The *Mobile code security framework* is advancing on the current *execution traces* technique through the use of a trusted certification authority. This trusted entity is not only responsible for key management but also the verification of the traces and the subsequent issuing of certificates upon validation. This framework can effectively be implemented in environments where such certification authorities exist.

Possible network congestion at the verification server can be a problem if the number of hosts and mobile agents compared to the number of verification servers are high. Execution tracing as a countermeasure also causes extensive extra overhead in terms of computational resources as well as additional communication sessions.

Self-protecting mobile agents

D'Anna *et al.* (2003) state in a final report, the development of a distributed mobile agent system, whereby the mobile agent is converted into a set of tamper-resistant *agentlets*.

Every mobile agent is partitioned into a set of communicating programs called *agentlets* that executes on independent hosts. Critical information contained in the mobile agent is spread across the *agentlets* that limits their vulnerability. The code and data of each *agentlet* is obfuscated by using a number of techniques. A time limit is also added so that a successful attack on an *agentlet* cannot be accomplished before the agent expires. *Agentlets* are self-monitoring by using challenge/response techniques. Compromised *agentlets* are automatically excluded, lost *agentlets* are replaced and the identities of malicious nodes are reported.

Self-protecting mobile agents employs the use of distributed agents as well as code obfuscation and time techniques for protecting the agent. A mobile agent is split into a number of cooperating agents for the purpose of spreading the secret parts of the mobile agent amongst a number of different agents. The agents are protected by the use of code obfuscation and time techniques, which combined, provide adequate protection for the agent against hosts employing reverse engineering techniques. Self-protecting agents can be used in an environment that relates to the use of cooperating agents.

The disadvantages of this framework are the actual creation of the different agents as well as the additional communication sessions required due to the use of additional agents. The additional agents can also influence the computational costs of the remote hosts if they are required to be executed on the same set of remote hosts.

Plain text algorithm

An *et al.* (2002) proposed a method whereby the code of the mobile agent is sent in plain text and the data and state is encrypted. The protocol relies on the assumption that every host must be able to handle public key encryption and decryption.

Upon migration, the current host on the itinerary sends the identification of the mobile agent's owner, the mobile agent's code as well as a hash of the mobile agent's code encrypted with the secret key of the current host, to the next host. The receiving host checks if the hash code of the mobile agent's code is equal to the encrypted hash code as sent by the previous host. If it is equal the mobile agent is executed, if not the mobile agent is sent back to its local host.

The *Plain text algorithm* method provides protection against integrity and authentication attacks, but not against confidentiality attacks.

An *et al.* (2002) also distinguish between agents employed in a trusted or untrusted environment and applications are defined as either critical or non-critical (according to the security levels that are required by the agent).

The countermeasures employed by the *Plain text algorithm* are based on encryption / decryption techniques. A new way of protecting the agent is proposed by only encrypting the state and the data of the agent and migrating the code in plain text format. A hash of the mobile agent's code is created and validated by the next host. This proposed method would be viable in an application that requires the retrieval of information or similar applications where the data and state can be migrated in an encrypted form.

The classification of mobile agent applications depends on the type of environment (trusted or untrusted) and type of application (critical and non-critical) and is criteria that needs to be incorporated into the design of a mobile agent security framework.

Protocol for detecting a mobile agent clone

Baek (1999) proposed a protocol that can prevent the cloning of a mobile agent. Through the use of this protocol, it is also possible to detect whether a clone of a mobile agent exists as well as identify the malicious host that was responsible for the cloning.

Cloning is seen as an exact replica of a current mobile agent, including the same unique identifier. The protocol makes use of a trusted party namely a *coordinator*, which upon receiving messages from hosts, determines the existence of a mobile agent clone by executing a clone detection algorithm. The creation, execution, migration and deletion of a mobile agent can only be done by a host if the coordinator grants permission. This gives control of the life cycle of the mobile agent over to the coordinator, and is also used as a method to detect possible cloning. The coordinator is in a position to predict the next step in the life cycle of the mobile agent, and any request not within this prediction indicates that a clone is in operation.

The problem of duplicating mobile agents is a threat that can only be prevented by making use of a trusted execution environment. The protocol presented by Baek (1999) provides a new method by which the threat of cloning can be prevented. The additional requirements of the proposed protocol are the establishment of a trusted third party for the purpose of clone detection. This proposal needs to be integrated with other countermeasure techniques in that it only provides a method for detecting the creation of exact replicas by malicious hosts.

The disadvantages of the proposal lie in the additional communication sessions required between the hosts and the trusted entity responsible for the clone detection.

Three-tier protection model

Sameh & Fakhry (2002) presented a solution to security in mobile agent systems by using a combination of code obfuscation, encryption and time techniques.

The code obfuscation part is achieved through the involvement of three major parts namely, the *insertion of dummy code* (the aim of this part is to make the mobile agent code more complex for any attacker to understand), the *alteration of the values of numeric variables* (every numeric value such as integer and float is changed by multiplying the value with a random seed kept at the home host. This random seed value is used to regenerate the original numerical value of the mobile agent (by making use of an inverse operation) and the *alteration of the values of string variables* (every index in the string is changed by multiplying its numerical value by the generated random seed and kept at the home host).

The Data Encryption Standard (DES) algorithm is used to encrypt the secret data of the mobile agent. This protection model was tested by making use of the Concordia mobile agent system (Kiniry & Zimmerman, 1997).

The countermeasures implemented within the *Three-tier protection model* consist of code obfuscation, encryption and time techniques. The integration of these methods is to be used in applications that need to prevent the reverse engineering of the mobile agent.

The disadvantages include the additional computational costs needed by the creator of the agent in order to implement code obfuscation techniques. The use of code obfuscators for this purpose can also mean additional financial costs for the creator of the agent.

4.2.3 Evaluative summary of mobile agent frameworks, architectures and models

The analysis of the different proposed mobile agent frameworks; architectures and models provided a step forward in determining the requirements of a mobile agent security framework. Other mobile agent protocols in the open environment, which are not specifically discussed in this thesis, include the *Multi-Agent Cryptographic Protocols* (Tate & Xu, 2003) and the *One-Round Secure Computation and Secure Autonomous Mobile Agents* (Cachin *et al.*, 2000) This section summarises the investigation into the proposed systems and highlights the findings.

Security levels

One of the main issues that became noticeable through the discussion of the various models and frameworks, is the apparent inability of these systems to distinguish between different levels of security, depending on the type of environment where the mobile agents are deployed. The main criteria obtained from the analysis is that a mobile agent security framework needs to provide for different security levels. For example, *M&M* (Marques *et al.*, 2001) introduced the concept of distinguishing between different types of environments in which the agent is to be deployed. This proves to be a valuable input, due to the countermeasures used within a framework being dependent on the environment in which the agent operates in. The *Plain text algorithm* (An *et al.*, 2002) also classifies the mobile agent applications according to the type of environment (trusted or untrusted) as well as the type of application (critical and non-critical). The different proposed systems were also distinguished in terms of those operating in a closed environment (such as *POM* (Guan *et al.*, 2000) and *FILIGRANE* (Jalali *et al.*, 2000)) and those that can be implemented in an open environment (for example *Self protecting mobile agents* (D'Anna *et al.*, 2003) and *Mobile code security framework* (Tan & Moreau, 2002)).

It is thus essential that our security framework provide for different levels of security according to the *environment* in which the agent is to be deployed as well as the type of mobile agent system *application*.

Autonomy

Another aspect that is essential to mobile agent applications is the *autonomy* of the mobile agent. A large number of the discussed mobile agent models and frameworks are based on the creation of a trusted execution environment. Although trusted environments restrict the mobility and autonomy aspects of the mobile agent, it seems that (depending on the type of application), there may be a demand for these types of environments and hence in some cases it is necessary to establish trusted environments for the deployment of mobile agents.

The autonomy and mobility aspects of a mobile agent are also restricted in systems that implement techniques where the agent is split into a number of cooperating agents. Systems such as *POM* (Guan *et al.*, 2000), *Mansion* (Van't Noordende *et al.*, 2002) and *Supervisor-worker framework* (Fischmeister, 2000) only allow the computation of the agent on a trusted entity with slave agents created in order to retrieve data from remote hosts needed by the mobile agent. *SAWMA* (Luo, 2001) requires the secret part of the agent to be split into a number of distributed agents in order to protect the secret part of the agent within an open environment, while *Security framework for mobile agent systems* (Bryce, 2000) and *Self protecting mobile agent* (D'Anna *et al.*, 2003) requires the agent to be split into a number of cooperating agents for secret spreading.

The creation of cooperating agents has a number of disadvantages (such as additional communication sessions and additional overhead) and should only be used in applications that require a distributed design.

Trusted third party

A large number of the proposed systems employ the use of a trusted entity (especially the systems that are based on the notion of a trusted execution environment). The objectives of such an entity range from key management to providing a place for secure computation of the agent. Systems that use a trusted third party for secure computations are *POM* (Guan *et al.*, 2000), *Mansion* (Van't Noordende *et al.*, 2002), *Planet* (Kato *et al.*, 1996) and *Supervisor-worker framework* (Fischmeister, 2000).

The use of a trusted third party for the establishment of trusted domains by software methods (which include key management and distribution duties), are used by systems such as *Security enhanced mobile agents* (Varadharajan, 2000) and *Distributed transactions*

(Vogler *et al.*, 1997).

Systems that utilise a trusted third party to perform other duties include, *Distributed transactions* (Vogler *et al.*, 1997) that require the trusted entity to create logs of the execution of the agent for detection purposes; *Proxy agents and trusted domains* (Mitroviæ & Arribalgaza, 2002) for the creation of proxy agents as well as the distribution of the proxies to the hosts; *Electronic supermarkets* (Wu, 2000) for registering agents within a domain; *Agent factory* (Brazier *et al.*, 2002) requires the existence of an agent factory that is responsible for the creation of the mobile agent from a conceptual description; *Mobile code security framework* (Tan & Moreau, 2002) for validating cryptographic traces and *Protocol for detecting a mobile agent clone* (Baek, 1999) requires a trusted third party to detect if a malicious host created a clone of the mobile agent.

The use of a trusted entity for either the creation of a trusted environment, for securing some computational results within an open environment, or for key management / certification features seems essential within a security framework.

Countermeasures

A number of different countermeasures form part of the proposed systems. The authentication of the different remote hosts as well as the mobile agent is done mainly by the use of digital signatures (for example, *Security enhanced mobile agents* (Varadharajan, 2000), *Mansion* (Van't Noordende *et al.*, 2002), *Planet* (Kato *et al.*, 1996) and *Mobile code security framework* (Tan & Moreau, 2002)).

Keys for encryption and authentication purposes include the creation of public / private key pairs by models such as *FILIGRANE* (Jalali *et al.*, 2000), *M&M* (Marques *et al.*, 2001) and *SAWMA* (Luo, 2001)), as well as the use of session keys by for example *Distributed transactions* (Vogler *et al.*, 1997).

The protection of the mobile agent against reverse engineering attacks as well as the modification of the code of the agent, involves the use of individual or the integration of techniques such as watermarking, code obfuscation and time techniques (for example, *FILIGRANE* (Jalali *et al.*, 2000), *SAWMA* (Luo, 2001), *Self protecting mobile agent* (D'Anna *et al.*, 2003) and *Three-Tier protection model* (Sameh & Fakhry (2002)).

The protection of computational results as well as data retrieved by the agent is performed by *partial results authentication code* (Yee, 1997) techniques in systems such as *Security enhanced mobile agents* (Varadharajan, 2000) and *M&M* (Marques *et al.*, 2001).

Proposals such as *Mansion* (Van't Noordende *et al.*, 2002) generate an audit trail of hosts visited in order to detect if the itinerary of the mobile agent was followed, while the *Security framework for mobile agent systems* (Bryce, 2000) make use of *replication and voting techniques* (Minsky *et al*, 1996; Schneider, 1997) for protecting the mobile agent.

It is essential that our proposed security framework not only provide for the use of individual countermeasures, but also the *integration* of specific countermeasures. For example, the use of *code obfuscation* (Hohl, 1997, 1998) and *time techniques* (Grimley & Monroe, 1999) will provide additional protection for code manipulations if combined.

Policies

Policies provide a manner to define the security requirements of the agent. Although the mobile agent is dependant on the remote host for its execution, the *definition of security policies* together with the implementation of certain countermeasures can aid in providing protection for the agent. Models that integrate security policies in their designs are *POM* (Guan *et al.*, 2000), *FILIGRANE* (Jalali *et al.*, 2000), *M&M* (Marques *et al.*, 2001), *Proxy agents and trusted domains* (Mitroviæ & Arribalgaza, 2002) and *Security framework for mobile agent systems* (Bryce, 2000).

Additional requirements

Additional security implementations are required in some models (of which some can have additional cost implications when implemented), for example the creation of a trusted entity, as well as trusted hardware components (such as *FILIGRANE* (Jalali *et al.*, 2000) and *Electronic supermarkets* (Wu, 2000)). *Planet* (Kato *et al.*, 1996) also requires additional secure hardware.

Software requirements for remote hosts include the creation of a security manager present at the foreign hosts (for example *Security enhanced mobile agents* (Varadharajan, 2000), *FILIGRANE* (Jalali *et al.*, 2000) and *M&M* (Marques *et al.*, 2001)).

Agent factory (Brazier *et al.*, 2002) also requires the creation of a mobile agent according to a provided conceptual design. Systems that make use of encryption techniques as well as authentication techniques will require certain additional encryption software.

The communication sessions required for the implementation of some of the models are a concern, especially with the use of trusted entities for computations as well as cooperating agents. For instance *POM* (Guan *et al.*, 2000) lets the agent only migrate to a trusted entity and slave agents are sent to the hosts in the domain for obtaining the information. Additional communication sessions are also needed for the creation and distribution of public / private keys by means of a trusted entity as well as the creation and distribution of session keys.

In determining the requirements of a mobile agent security framework, the *additional requirements* in terms of computational overhead, communication sessions and financial implications, need to be taken in consideration.

4.3 Mobile Agent Systems

In Section 4.2 we considered frameworks, architectures and models proposed by other researchers that have security methods incorporated into their designs. In this section we consider a number of mobile agent systems and tools, both research-based and commercial, which have been developed for the purpose of assisting in the creation and deployment of mobile agent system applications. Some of these were developed for applications in a closed environment and thus no security mechanisms were included in the design and implementation, while others included some security techniques. Examples of systems developed that do not incorporate agent security into their designs are *Mole* (Baumann *et al.*, 1998), *Tracy* (Braun *et al.*, 2000), *AMASE* (Pascotto, S.a.), *Grasshopper* (Breugst *et al.*, 1999) and *Voyager* (ObjectSpace, 1997).

This section focuses on providing detailed descriptions of mobile agent systems that include mobile agent security features. The main objective of these mobile agent systems is to aid in the development of applications within the mobile agent environment, and is not integrated frameworks as the systems analysed in Section 4.2. The analysis of these systems thus differs from the previous section by providing an integrated discussion on the detailed systems in Section 4.3.1.

Agent Development Kit (ADK)

The *Agent Development Kit (ADK)* is a mobile component-based development platform that allows Java Developers to easily build, deploy and manage secure, large-scale distributed solutions that operate regardless of location, environment or protocol. *ADK* is a commercial package developed by Tryllian, but is freely available for research purposes.

The security features of *ADK* is built on the Java security model, and is accomplished by digitally signing the agent’s class files with a private key as well as including a certificate. The certificate itself consists of a public key and personal data. The public key is used for encryption purposes. If agents need to send the host computer their certificate, they will do so using the host's public key. Personal data is included in order to make it possible to trace the owner of an agent's certificate. If the agent's owner gives their certificate to an unauthorised third party, it will be possible to trace it back to them.

With the signature, a host can check that nobody has tampered with the agent. It can also find out who created the agent and who trusts the developer of the agent. Assigning permissions to certificates allows the host to determine what an agent is allowed to do when it enters the host.

When an agent requests to enter the host, the agent is inspected to see who created it and to check if nobody has changed the contents somehow. It does this by inspecting the certificate included in the agent. This certificate contains the builder of the agent, his public key and an agent checksum. The checksum can only be created with the private key, which is only known to the builder.

Agent Tcl

Agent Tcl is a simple, flexible and secure mobile agent system based on the Tcl scripting language developed for research purposes at Dartmouth College. The architecture builds on systems such as Telescript (Tardo & Valente, 1996), Ara (Peine, 1998) and TIAS (Harker, 1995) and consists of a *server*, which is responsible for keeping track of agents, migration, communication and non-volatile storage.

A mobile agent is requested to register at a server of which the process is digitally signed using the owner's private key, encrypted using the server's public key and sent to the server. Upon migration the mobile agent is signed with the private key of the current server and encrypted with the public key of the destination's server (Gray, 1996).

Agent Tcl only provides authentication and encryption facilities, with no additional protection for the remote hosts.

Aglets

Aglets is an open source mobile agent system developed by IBM Japan and is currently only being upgraded by the open source community. *Aglets* as described by (Karjoth *et al.*, 1997) are Java objects that can move from one host on the Internet to another. The aglet security model supports the flexible definition of various security policies, which are defined in terms of a set of rules. These policies specify conditions such as the authentication required for all entities and the communication security between aglets and between hosts. The main focus of the aglets security model is the protection of the host against malicious agents and the protection of aglets from other aglets. Protection of aglets against modifications by other aglets is achieved by making use of proxy agents.

Ajanta

Ajanta is a mobile agent programming system being developed at the University of Minnesota. It allows agents written in Java to securely migrate between hosts on the Internet. The *Ajanta* project is aimed at building an infrastructure for mobile agent execution that integrates security and robustness features as an integral part of the design.

The *Ajanta* architecture includes a generic mobile agent server, which provides a secure mobile agent execution platform. The entities in the *Ajanta* mobile agent system are a *principal* on whose behalf actions are performed, the *creator* of the mobile agent, the human *owner* and the *guardian*. A guardian object is assigned to each mobile agent by the application in order to deal with exception conditions.

A mobile agent in *Ajanta* carries *credentials* which is a tamperproof certificate containing its name, the name of its owner, the name of its creator and the name of its guardian. A *code base server* provides the code for the classes required by the mobile agent. The code base server is normally the creator of the mobile agent and the mobile agent also carries the URL of this server. The credentials object also contains the hash-value of the read-only data contained in the mobile agent, which together with the credentials is signed by the creator.

In order to protect the mobile agent from tampering by other agents on a specific host, the mobile agent is isolated in a protected domain. Two Java mechanisms for creating protected domains are used, namely thread grouping and class loading. The Agent Transfer Protocol for the migration of mobile agents between hosts employs standard cryptographic mechanisms such as hashing and digital signatures.

Ajanta implements three methods for the detection of tampering by malicious hosts. Firstly, part of the mobile agent's state can be declared as read-only and is cryptographically protected. A read-only container object contains a vector of objects along with the owner's digital signature for these objects. The owner computes the digital signature by firstly using a one-way hash function (SHA) to digest the vector of objects to a 128-bit value and then encrypting this by making use of the private key supplied by the constructor.

The second method is the keeping of append-only logs in the case of data obtained and not to be used subsequently. Data is digitally signed by the current host and inserted into the append-only log.

The third method protects items in the mobile agent so that they are only accessible to certain hosts. The targeted state contains a vector of objects that are individually encrypted using the public key of the host for which it is targeted. The corresponding identities of the hosts are inserted into a separate vector. These two vectors are then hashed together and signed by the owner of the mobile agent.

In order to prevent the copying and masquerading of mobile agents, a copy of the itinerary is inserted into the read-only container as well as making use of a name service, which is implemented as a group of autonomous registries. A name registry entry contains the location information for the resource it represents (Karnik & Tripathi, 2000).

Asynchronous Message Transfer Agent System (AMETAS)

The *Asynchronous Message Transfer Agent System (AMETAS*), was developed by Zapf *et al.* (1998) and is implemented in Java. Mobile agents and hosts are required to apply security rules as stated in the different security policies. The security services of the agent platform, the programming language as well as the infrastructure are responsible for the enforcements of the security policies. The mobile agent code (after compilation) is digitally signed by its owner, which allows for subsequent mobile agent platforms to validate the integrity of the mobile agent code. The owner of the mobile agent may specify the hosts who are allowed or denied to execute the mobile agent. These rights are added to the mobile agent and digitally signed.

A Certification Authority forms part of the security framework for the storing and issuing of public-private key pairs. Before migration between hosts, the two hosts authenticate each other. The hosts use their public keys to identify themselves and prove the ownership of the corresponding private key by encrypting some random data. The mobile agent is sent between the two hosts in encrypted form. Both agent platforms also have to confirm the sending or receiving of the mobile agent data.

Anchor toolkit

The *Anchor toolkit* was developed by Mudumbai *et al*. (1999) at the Imaging and Computing Sciences Division, University of California as a mobile agent system intended for research purposes.

According to Mudumbai *et al.* (1999), the *Anchor toolkit* handles the transmission and secure management of mobile agents in a heterogeneous distributed computing environment. The mobile agent system model is based on Aglets (Karjoth *et al.*, 1997), whereby mobile agents are created within a *context*. Mobile agents are grouped together in a context and they can only be accessed through a *proxy*. A proxy handles all communication and actions directed towards the mobile agent. The IAIK-SSLtoolkit together with Java Cryptographic Extensions are used to encrypt mobile agents migrating between hosts. X.509 certificates are used for mutual authentication between hosts. Hosts are also responsible for signing the mobile agent's persistent state before migration.

Agents for remote action (ARA)

Agents for Remote Action (ARA) is a platform for the portable and secure execution of mobile agents, currently under development at the Unversity of Kaiserslautern.
Peine (1998) provides a detailed description of the security architecture of the *ARA* mobile agent platform. The programming model consists of the three components, namely *places* which is the entity that the mobile agent migrates to in order to use services provided, *service* which is only accessible to mobile agents at a place and the *mobile agents*. The entities in the system are the *mobile agent users* (the person initiating the mobile agent), *mobile agent manufacturers* and the *host machines.*

Associated with each mobile agent is its *passport* that contains relevant identification information such as the *identity of the mobile agent*, *time of creation* and appropriate *certificates*. Upon creation of the agent, the mobile agent is signed and the mobile agent initiates a host trace that is responsible for keeping a list of all hosts visited. The host trace is incrementally signed by the receiving host systems on every hop. The mobile agent is divided into two parts, namely *changing* and *unchanging*. The unchanging part consists of the mobile agent code and its passport that is signed upon creation of the mobile agent. The changing part is not signed and thus not protected from malicious host actions.

Hosts are grouped into regions that are managed on a lower level without providing any of this knowledge to the mobile agent. At migration encryption of the mobile agent can be omitted if the destination host resides in the same region as the current host.

Almost zero infrastructure mobile agent system (aZIMAs)

The *almost Zero Infrastructure Mobile Agent system (aZIMAs)*, is developed by Nalla *et al*. (2002). *aZIMAs* is a simple mobile agent system based on HTTP for the deployment of Java-based mobile agents. A mobile agent is protected from other agents by the system creating a separate namespace for each executing agent. Security provisions are the declaration of parts of the mobile agent as read-only and the assumption that mobile agents are only executed on trusted hosts.

Bee-gent

Bee-gent is a software development framework developed at the Systems and Software Research Laboratories at Toshiba Corporation, that allow developers to build flexible open distributed systems that make optimal use of existing applications (Toshiba, 2001). The *Bee-gent* framework consists of two types of agents, namely *agent wrappers* that are used to convert existing applications to mobile agent systems and *mediation agents* that handle inter-application coordination. The mediation agents support digital fingerprint authentication as well as secret key encryption. The mediation agents move from the site of an application to another where they interact with the agent wrappers. The agent wrappers themselves manage the states of the applications they are wrapped around, invoking them when necessary. Thus inter-application coordination is handled through the agent wrappers generating and receiving requests, which are transported around by the mediation agents. The mediation agents do more than just transport the messages; they are able to respond to the nature of the request to determine the best course of action.

Concordia

Concordia is a Java-based mobile agent system designed specifically to support enterprise computing and mobile platforms and was developed by the Mitsubishi Electric Information Technology Center.

Concordia is implemented in Java and the framework consists of multiple components, each of which is responsible for handling a specific task. These components are the *Concordia Server, Administration Manager*, *Security Manager*, *Persistence Manager*, *Event Manager*, *Directory Manager*, and the *Queue Manager*. The Security Manager handles all security related issues within the system and also supports strong authentication through certificates. Mobile agents are protected while migrating through SSLv3. The mobile agent's state is encrypted while in persistent storage in order to prevent unauthorised access and modification (Kiniry & Zimmerman, 1997).

D'Agents

D'Agents is a mobile agent system that is capable of initiating and executing mobile agents written in Tcl, Java and Scheme. It is developed at Dartmouth College as an advancement of the *agent Tcl* project. The core system of the *D'Agents* architecture consists of four levels, namely an *interface* to the transport mechanisms, a *server* running on every host, the *execution environment* (which are just interpreters for the different languages) and the *mobile agents.* The tasks of the server is to provide communication facilities, receiving and authenticating mobile agents as well as restarting the mobile agent in the appropriate execution environment.

Each host has a public-private key pair and PGP is used for digital signatures and encryption. When a mobile agent migrates from its home host to a new destination, the state is digitally signed with the private key of the owner and optionally encrypted with the public key of the destination host. The receiving host verifies the signature and decrypts the agent. Currently *D'Agents* only provide protection during migration as well as protection for the host against malicious agents (Gray *et al*., 1998).

Jumping Beans

Jumping Beans®, Inc. provides solutions to mobile corporate wireless systems using *Jumping Beans*, mobile agent system.

Jumping Beans implement a client-server architecture, whereby an agent always returns to a secure central host first before moving to any other platform. Jumping Beans has four layers of security, namely *traditional distributed security, multi-jump security*, *trusted* *source* as well as *monitoring and intervention. Jumping Beans* employs all of the standard security techniques used in traditional distributed computing systems, such as digital signatures, encryption, passwords and audit logs. The system administrator assigns a level of trust to each host, and ensures the code executed by an application comes from a known trusted source, even if an untrusted host launched the application. The system administrator can also track the activity in the entire *Jumping Beans* system to help detect unwanted activity. In addition, the system administrator can control applications to help prevent or stop unwanted activity.

S-agent

Makino *et al.* (2000) proposes a secure mobile agent system that provides an architecture for the protection of attacks against mobile agents as well as protection against hosts. The design is based on the Java security model and all entities in the system must contain RSA (Rivest *et al.*, 1978) key pairs.

To detect modifications of the mobile agent by a malicious host, three functions are implemented, namely an *agent ticket, state signing* and *logging.* An agent ticket is created as part of the mobile agent and consists of the date of creation, a sequence number, name of the source host and the hash value of the agent Class object. All hosts on the itinerary of the mobile agent must digitally sign the ticket sequentially.
Upon migration the current host must sign the current state of the mobile agent. The serialised form of the mobile agent and the destination address of the next host on the itinerary are stored in the source host for a period of time.

Secure mobile agents (SeMoA)

Secure Mobile Agents (SeMoA) is developed by using Java and focuses on all aspects of mobile agent security (Roth & Jalali, 2001). The security architecture of *SeMoA* consists of a number of layers through which a visiting mobile agent has to pass before being accepted for execution onto the host.

The different layers are the *transport layer security protocol*, *security filter* and *sandbox*. The purpose of the transport layer security protocol is to provide mutual authentication of the hosts, as well as encryption and integrity protection. The implementation used is SSL and migration requests are accepted or denied according to a specified security policy. Different security filters exist for accepting or rejecting incoming and outgoing mobile agents. *SeMoA* make use of two complementary filters to handle digital signatures and the encryption of mobile agents. The signatures of incoming mobile agents are verified and the mobile agent is decrypted. Outgoing mobile agents are signed and encrypted. A sandbox is created for the incoming mobile agent to protect the host against mobile agent attacks. A dedicated class-loader loads the classes for use by the mobile agent. All loaded classes are verified against a set of trusted hash functions signed by the mobile agent's owner to prevent the loading of unauthorised classes. Mobile agents are also separated from other mobile agents and each has its own view of the environment.

In *SeMoA* the mobile agents are transported between hosts as Java Archives, with signature files added to the contents of the ZIP archive. The JAR format is extended to provide support for the selective encryption for multiple recipients. Two digital signatures are attached to mobile agents, which is the static part signed by its owner as well as each host signing the complete mobile agent. By doing this the host commit to state changes that occurred while it executed the mobile agent.

The Secure Hash Algorithm (SHA1) digest algorithm is applied to the mobile agent's owner signature. This leads to globally unique names and anonymity for the mobile agent.

Secure and open mobile agent (SOMA)

Corradi *et al.* (1999) describes the *Secure and Open Mobile Agent (SOMA)* which offers a number of tools and mechanisms aimed at protecting mobile agents as well as hosts against malicious behaviour. The *SOMA* framework supports flexible security policies in order to administrate interactions of mobile agents and mobile agent platforms. Several different principals are modelled namely the *mobile agent*, the *place* that represents the execution site, the *agent creator*, the *agent owner*, the *place creator* and the *place owner*. Every principal owns specific, tamperproof credentials which are needed for authentication and authorisation. X.509 certificates are used to bind the unique identities of agent / place owners and creators to a cryptographic public key pair in a secure way.

Agent owners are associated with specific roles and each agent carries a set of exclusive credentials as part of their state. These credentials act as proof that the agents behaved correctly.

The *SOMA* security infrastructure is composed of a number of building blocks, such as a *policy server* used for management of domain policies, a *domain server* which maintains references to resources, a *role server* for managing roles, a *certification authority*, a *directory service* responsible for distributing certificates, an *authentication server* and an *authorisation server* for granting access to resources.

The mobile agent is encrypted and digitally signed before migration between hosts. The destination host verifies the authenticity of the previous site’s credentials and accepts or denies the mobile agent. *SOMA* implemented two solutions aimed at the detection of modification attacks on the state of the mobile agent, namely *trusted third party* and *multiple hops*.

The trusted third party acts as a trusting environment where the mobile agents can conduct secure computations. After the mobile agent has visited a possible malicious site, it migrates to the trusted third party in order to check for any inconsistencies. The trusted environment is responsible for maintaining the proof of all mobile agent computations at different sites.

In the multiple hops approach mobile agents can autonomously migrate through the network without having to interact with trusted third parties. Each host must provide a short proof of the mobile agent's computation, which is stored as part of the mobile agent's state. Each proof attached is cryptographically linked with the ones computed at previous mobile agent platforms. This prevents the modification of one proof from influencing all the previous proofs. On arriving at the home platform, the cryptographic proofs are verified in order to detect any integrity violation.

Tromsø and cornell moving agents (TACOMA)

Johansen *et al.* (1995) proposed the *Tromsø And COrnell Moving Agents (Tacoma)* project that focuses on operating system support for mobile agents. A *briefcase* that contains collections of named *folders* is associated with each mobile agent for the carrying of data. *Broker agents* are used to maintain databases of service providers. The broker agents are contacted by a mobile agent to identify hosts offering a specific service. They are also used to enforce the policies of a protected mobile agent by arranging meetings with other agents. Upon migration to a new host, a *rear guard* agent is created and stored on the current host. This agent is responsible for the launching of a new mobile agent in the case of an agent that is killed, as well as terminating itself once a mobile agent is safely executed and has migrated from the new host. *TACOMA* is implemented on Tcl, where every host is required to run a Tcl interpreter as part of the mobile agent platform.

Web agent-based service providing platform (WASP)

Fünfrocken & Mattern (1999), developed the *Web Agent-based Service Providing* platform which provides support for resource management, mobility, agent execution, communication and security. These tasks or services are achieved by making use of distributed Java concepts. Mobile agent platforms are integrated into World Wide Web servers by making use of server extension modules. The *WASP* platform provides basic security mechanisms for protection of the host against malicious mobile agents.

In order to protect the mobile agent, *WASP* makes use of a trusted environment. Fünfrocken & Mattern (1999) are currently experimenting with a smart-card that contains a Java byte-code interpreter (Java Card). The purpose of the Java Card is to provide authorisation for mobile agents acting on behalf of their initiators as well as using the Java Card as a trusted computing base for the execution of mobile agents. The Java card contains a private key that will provide a mobile agent the opportunity to be encrypted. The mobile agent can only be decrypted and executed by the Java card.

4.3.1 Evaluative summary of mobile agent systems

The mobile agent systems analysed in 4.3 provide security mechanisms for the development of secure mobile agent applications by incorporating a number of different techniques. Realised from the analysis is the large number of systems that make use of the Java security model as the primary security model. Examples of such systems are *ADK* (ADK), *Aglets* (Karjoth *et al.*, 1997), and *Concordia* (Kiniry & Zimmerman, 1997).

Encryption and authentication

A large number of the evaluated systems feature authentication techniques based on the creation of digital signatures. The creation of the digital signatures ranges from the signing of the class file of the agent (*ADK* (ADK)) to the signing of the agent itself (*Agent Tcl* (Gray, 1996), *Concordia* (Kiniry & Zimmerman, 1997) and *AMETAS* (Zapf *et al.*, 1998)).

Encryption techniques are included in almost all the evaluated systems that provide protection of the agent while in transit. Examples are *Ajanta* (Karnik & Tripathi, 2000), *Anchor* (Mudumbai *et al.*, 1999), *Jumping Beans* (Jumping Beans®) and *SOMA* (Corradi *et al.*, 1999).

The provision of *authentication* and *encryption* techniques for the agent as well as the host seems essential to a security framework.

Trusted third party

A few mobile agent systems provide for the use of a trusted third party for secure computations within the design. *SOMA* (Corradi *et al.*, 1999) provides for the use of such a trusted entity not only for secure computations but also for detection of modifications by the host (although this requires the mobile agent to be migrated to the trusted entity after leaving a remote host). Most of the systems that provide cryptography methods also include a certification authority in the design for the purpose of key generation and distribution. Examples of such systems are *Agent Tcl* (Gray, 1996), *Concordia* (Kiniry & Zimmerman, 1997), and *D'Agents* (Gray *et al.*, 1998).

The creation of a trusted environment by using either hardware (*WASP* (Fünfrocken & Mattern, 1999)) or software (*aZIMAs* (Nalla *et al*. (2002)) means is supported by a few systems. The creation of proxy agents to assist in protecting the agent by providing some sort of barrier for accessing the agent is provided by *Aglets* (Karjoth *et al.*, 1997) and *Ajanta* (Karnik & Tripathi, 2000).

Countermeasures

A few systems include logging capabilities whereby either the data required at the host or the identity of the host is sent to the agent's owner, for detection purposes (for example *Anchor toolkit* (Mudumbai *et al*., 1999), *S-agent* (Makino *et al.*, 2000) and *SOMA* (Corradi *et al.*, 1999)).

Ajanta (Karnik & Tripathi, 2000) and *Jumping Beans* (Jumping Beans®) provide for the creation of logs that can be appended to the agent for providing audit trail information. The *ARA* system (Peine, 1998) supplies methods for creating logs of the hosts visited, while encryption of the current state of the agent is possible within *S-agent* (Makino *et al.*, 2000), *SeMoA* (Roth & Jalali, 2001) and *SOMA* (Corradi *et al.*, 1999). *Ajanta* (Karnik & Tripathi, 2000) also include a method by which certain data carried by the agent is only made accessible to specific hosts, as well as incorporating a read-only container for protection of data.

ARA (Peine, 1998) also allows for dividing the agent into changing and unchanging parts, of which each can be protected separately.

Security credentials

A number of systems include the notion of security credentials as part of the design of a mobile agent. For example in *ARA* (Peine (1998) a passport is added that contains information such as the time of creation and *S-agent* (Makino *et al.*, 2000) make use of a ticket that contains the time of creation as well as the name of its owner.

Policies

Mobile agent systems that provide for the creation of policies in order to aid in the protection of the mobile agent, include *Aglets* (Karjoth *et al.*, 1997), *AMETAS* (Zapf *et al.*, 1998) and *SOMA* (Corradi *et al.*, 1999). As indicated in the analysis of the mobile agent frameworks and models, the inclusion of security policies for the agent is seen as an added measure for providing protection for the mobile agent.

Additional Requirements

As mentioned in Section 4.2.3 for determining the techniques to be integrated within the framework, it is essential to analyse the additional requirements imposed by the chosen techniques. The mobile agent systems evaluated provide for a number of countermeasures that are included in the design of the systems and are thus present and usable within the provided execution environments.

Systems that make use of trusted authorities for various purposes means additional communication sessions required for the use thereof. The encryption and creation of digital signature also has an influence on the computational overhead for the creation as well as the verification of the agent. The use of path histories as a method to detect variations on the itinerary of the agent also has size implications on the agent itself (and thus bandwidth implications).

4.4 Mobile Agent System Applications

In Section 4.2 we analysed existing frameworks, architectures and models within the mobile agent environment that incorporated measures for the protection of the mobile agent into the respective designs. Section 4.3 followed the same trend by studying available mobile agent tools that can be used for the development of mobile agent applications.

A number of applications have been developed by making use of existing mobile agent technologies. Examples of developed applications that provide no protection for the mobile agent are, *Using mobile agents for analysing intrusion in Computer Networks* (Aslam *et al*., 2001), *An intrusion detection system for Aglets* (Vigna *et al*., 2002), *Combining world wide web and wireless security* (Claessens *et al*., 2001), *Mobile agents* *supporting secure GRID environments* (Robles *et al*., 2002) and *A method of tracing intruders by use of mobile agents* (Asaka *et al*., 1999). A small number of applications do however provide security techniques to prevent or detect malicious modifications on the mobile agent. They are subsequently analysed of which a summary is provided in 4.4.1.

Attack resistant distributed hierarchical intrusion detection system

Mell & McLarnon (1999) used mobile agents to cast internal nodes of a distributed intrusion detection system. These mobile agents randomly move around the network in such a way that an attacker is not able to locate their position. In the case where an attacker demobilises a host, the remaining agents estimate the location of the attacker and automatically avoid those networks. Mobile agents that are killed are re-introduced into the system by a group of backups that retain all or partial state information.

Cherubim

The System Software Research Group at the University of Illinois, developed *Cherubim*, that employs a secure node architecture by using mobile agents and customised security policies (Campbell & Qian, 1998). Security functions are embedded in *smart security packets* in active networks. These smart packets implement user-level policies or capabilities as scripts.

The architecture has a pre-configured core security service, which provides basic public-key encryption, authentication and auditing facilities upon which the meta-*level* structure is built. This core security service along with a set of default meta-level components forms a security manager with basic facilities supporting dynamic security agents. Thereafter new security measures can be dynamically injected into this basic system.

MAgNET

Dasgupta *et al.* (1999) developed an e-commerce system that provides protection for both the mobile agent and the host. Every host is provided with a public-private key pair, and the mobile agent is divided into three portions, namely a *header*, *code* and *data*. The header consists of the mobile agent’s identifier and the identifier of the owner, which are encrypted with the owner's private key. The code, which can only be executed with a license, is obtained from the owner. The license is attained once the host acknowledges receipt of the mobile agent. The data obtained at different hosts on the itinerary is encrypted with the private key of the host. The host also computes a checksum of the data and sends this to the owner.

Mobile agent based transactions in open environments

De Assis Silva & Popescu-Zeletin (2000) developed a transaction model for open environments based on mobile agents. In this protocol, if an agent executing at a host becomes unreachable for a certain period of time, the agent (and its execution) can be recovered and executed at another host. The protocol is based on the division of the mobile agent task/s into a number of subtasks that is each executed by a different agent.

Secure electronic transactions

Kotzanikolaou *et al.* (1999) proposed a mobile agent-oriented model for collecting and evaluating purchasing contracts, signed by Internet merchants. A master-slave distributed agent architecture is used as well as making use of permission-tokens. For every host on the itinerary, a static master agent creates a mobile slave agent. The slave agents are collaborative agents and each are provided by a permission-token used as authentication proof for hosts. Each slave agent migrates to a specific host and negotiates on behalf of the buyer. Upon execution completion on the host, each slave agent returns home with purchase contracts signed by the hosts. The master agent is responsible for the evaluation of the contracts as well as initiating the buying of the selected products. Authentication of hosts is done by sending the host’s certificate to the master agent, along with the signature of the permission-token. Slave agents do not carry any secret keys or sensitive information and is thus not likely to be manipulated by the hosts.

SIAS

Shopping Information Agent System (SIAS) is a web-based mobile agent system that conducts information searches on products in an electronic marketplace. *SIAS* is implemented on top of the Concordia API using Java as programming language. The *SIAS* system makes use of a public-key infrastructure. Each host and mobile agent in the mobile agent system is required to own a pair of keys for encryption and decryption. Each mobile agent or host can encrypt or digitally sign the data of the mobile agent for protection. *SIAS* makes use of a *key* server for facilitating public key cryptography. The RSA encryption algorithm is used for encrypting the mobile agent's data. Each mobile agent and host must have a public key certificate registered at the key server for encryption purposes. By doing this, a closed network of hosts is established.

Each host is required to encrypt the results of the mobile agent with the agent's public key and can only be decrypted by the owner. These results are also digitally signed by the hosts in order to provide integrity. In protecting the itinerary of the mobile agent, every host has to encrypt the itinerary by using its private key in order to form a chain of encrypted itineraries (Chan *et al*., 2000).

Virtual Internet Pets

Gupta *et al.* (2001) developed an application whereby life-like pets are simulated by using Java-enabled mobile agents. The architecture of the application consists of two core components, namely a *mobile agent monitor* and a *graphical user interface*. The mobile agent monitor consists of an agent server responsible for executing the mobile agents and a stationary master agent instantiated by the agent server and used for inter-agent communication, interaction with the user and creating local agents. A proxy is used to act as a shield that protects an agent from malicious hosts.

4.4.1 Evaluative summary of mobile agent applications

The mobile agent applications that provide security techniques for the protection of the mobile agent from malicious modifications by the remote host were analysed in the previous section, and the results are subsequently summarised:

Encryption and authentication

Applications such as *Cherubim* (Campbell & Qian, 1998), *MAgNET* (Dasgupta *et al.* (1999), *Secure electronic transactions* (Kotzanikolaou *et al.* (1999) and *SIAS* (Chan *et al.*, 2000) makes use of digital signatures as a means of providing authentication abilities within the applications. Public / private encryption methods are also used in the mentioned systems for the encryption and decryption of the mobile agents.

Auditing

Auditing facilities are implemented in most of the applications for the purpose of detecting possible malicious manipulations. The auditing facilities include the encryption and digital signing of aggregated results as well the digital signing of the remote hosts visited by the mobile agent. Examples of applications that provide such measures are *Cherubim* (Campbell & Qian, 1998), *MAgNET* (Dasgupta *et al.* (1999) and *SIAS* (Chan *et al.*, 2000).

As *auditing facilities* are present in the analysis of the mobile agent frameworks, systems, models and applications, it is deemed necessary that the security framework include techniques for providing such facilities

Additional requirements

Secure electronic transactions (Kotzanikolaou *et al.* (1999) and *Mobile agent based transactions in open environments* (De Assis Silva & Popescu-Zeletin, 2000) require the subdivision of the mobile agent into a number of slave agents in order to split the secret data amongst a number of agents. As indicated in the analysis of mobile agent frameworks, models and systems, the use of coordinated agents results in additional communication costs as well as computational overhead. *SIAS* (Chan *et al.*, 2000) also requires the inclusion of a trusted entity to act as a key server for the management and distribution of keys used for encryption / decryption purposes.

4.5 Conclusion

The purpose of this chapter was to analyse mobile agent frameworks, systems and applications (that are either proposed or implemented), for providing insights into the development of criteria for designing a mobile agent security framework. Notable from the analysis is the dependence of a mobile agent security framework on the application and the distinction between open and closed (or trusted) environments. Furthermore, it also seems to be important to distinguish between the use of different countermeasures instead of having an all-inclusive type of security package, which could heavily burden a system's performance.

A worrying aspect in many of the studied models / frameworks / systems, is the restraints that the security framework of a particular mobile agent system often imposes on autonomy of the agents. As hinted before, there are situations where trusted environments, with known hosts are desirable, but to impose a trusted environment restriction on mobile agent behaviour in general, goes against the very basis of mobile agent definitions. Another point of concern in the studied systems is the additional requirements that are often demanded from remote hosts as well as the mobile agent system as a whole in order to implement acceptable security measures. In the next chapter, we take the discussion of this chapter further as a step towards proposing a security framework for mobile agents against malicious hosts.

CHAPTER 5

SECURITY ISSUES IN MOBILE AGENTS

5.1 Introduction

The security issues in mobile agent systems as outlined in the previous chapters accentuate the need for an integrated framework to address mobile agent security. In this chapter we investigate criteria for such an integrated security framework. These criteria will then form the pillars of our proposal as outlined in Chapter 6. We also examine the available standards for mobile agent systems, to place our research in context. Although security issues are not addressed per sé in these standards, we nevertheless find it essential to the understanding of the system components involved that can be affected by security threats.

5.2 Mobile Agent Standards

There are currently two standards specified for mobile agent technology, namely the *Mobile Agent System Interoperability Facility (MASIF)* developed by the Object Management Group (OMG) and the *Foundation for Intelligent Physical Agents (FIPA)* developed by companies and universities under Swiss law.

The purpose of MASIF is to address interoperability between agent systems and not between agent applications and agent systems. It defines the interfaces at the agent system level of the agent rather than focusing at the agent level itself. Four main areas of standardisation are contained in MASIF namely: *Agent Management*, *Agent Transfer*, *Agent and Agent Systems Names* and *Agent System Type and Location* Syntax (MASIF version 1.0). MASIF's security is based on the CORBA security model. Currently there exists no specific standardisation model for mobile agent security (Milojicic *et al*., 1998).

The purpose of FIPA is to enable interoperability of intelligent agents. The FIPA specifications are grouped into five categories, namely *Applications*, *Abstract Architectures*, *Agent Communication*, *Agent Management* and *Agent Message Transport*. As can be deduced from the name of these categories, the FIPA standards are primarily focused on agent communication languages, agent services and supporting management ontology's for agent systems in general. No specific emphasis is placed on mobile agent systems and hence agent mobility and many other features specific to mobile agents are excluded from this standard. Obviously this also excludes any FIPA standards for mobile agent security.

The specification for agent communication saw the development of an *Agent Communication Language (ACL)*, which is a high level language that allows communication between agents. Inter-agent communication is conducted via messages that consist of two parts. The first part is an envelope that conveys information necessary for transportation, while the second part is the actual message contained in a message body. The agent communication language is based on the *Speech Act Theory*, for more information see Poslad *et al*. (2000).

According to Poslad & Calisti (2000) there are several reasons contributing to the current lack of mobile agent security standards. Some of these reasons include:

Security issues are complex and cannot be developed by ordinary mobile agent programmers, as specific skills and expertise are required for security programming.

Security cover generally falls outside the scope of current mobile agent architectures. This is due to the general conception of programmers and users that the software architecture of the mobile agent platform will take responsibility of security coverage.

Because security is both domain and platform dependent, it would be naïve to think that a general architecture will be suitable for applications and implementations.

5.3 Challenges in Mobile Agent Security

The protection of mobile agents against malicious hosts has introduced a new field in the security arena. For the first time it is deemed necessary to protect an application (in this case the mobile agent) from manipulations by the executor (the host) of the application (Jansen, 2000). As can be summarised from previous chapters in the mobile agent paradigm, the agent is sent between hosts in order to achieve its goal. At every host the agent is executed, information is exchanged between the agent and the host and the state of the agent is updated accordingly. This execution at a foreign site introduces specific security challenges in relation to the protection of the agent. In the next few paragraphs specific challenges in the deployment of adequate security techniques for the protection of mobile agents against malicious entities are described.

5.3.1 Requirement for sound autonomy and mobility

According to Chan & Lyu (1999), the autonomy property of a mobile agent makes security of an agent the most challenging area of mobile code security. A truly autonomous agent is required to make independent itinerary decisions, based on its current environment and aggregated data. The trust model is one of the most popular techniques that are implemented in mobile agent computing to secure both the agent and the host. According to this model, security is applied according to the level of trust allocated to each host visited by the agent. This implies, that the hosts to be visited are known beforehand (and trusted to various degrees), which compromises the requirement of autonomy. Furthermore, the *autonomy* of an agent is inextricably related to its mobility. Therefore, it is important that integrated security techniques do not compromise its *mobility* either. These two characteristics of mobile agents (and also mobile agent systems) increase the complexity inherent to the design and maintenance of such a system. Added to the mentioned complexity is the fact that multifaceted systems with many components have a higher possibility of failure or breach; on the other side, simplistic systems can be vulnerable (Mitroviæ & Arribalzaga, 2002).

5.3.2 Tolerating changing network and application environments

Mobile agents have to operate in a dynamic communication environment, which contributes to specific challenges in the development of an adequate mobile agent security framework. According to Campbell & Qian (1998) a mobile security system should accommodate changes in security schemes that are imposed by changes in the network environment. The dynamic communication environment challenge also implies that different mobile agents (either involved in various or in the same mobile agent system) may require dissimilar security protection mechanisms, which leads to different levels of security.

5.3.3 Anticipating remote host support

In order to protect the mobile agent from malicious entities, a host has to conform to the security policy of the mobile agent. The challenge in this is that the mobile agent, as a single entity cannot provide adequate protection for itself and relies on its environment and its host for providing the required protection. It is therefore considered necessary that a host possesses intrinsic mechanisms to support the security requirements of the agent. Different to the predefined policies and means of trust that are provided on *trusted host environments*, these types of support measures, provide a way for agents to anticipate security support from its execution environment, without necessarily trusting the host. It implies the procurement of additional security functions and services, according to the needs of the application and hence the agent (Albayrak & Wieczorek, 1999). Examples of support measures include the availability of decryption algorithms or digital signature procedures on a host that are made available to the agent; an ability (for the agent) to audit its host's services (Jansen, 2000), et cetera.

5.3.4 Anticipating the required level of security support

According to Jansen & Karygiannis (1999), there are a number of factors that play an important role in determining the required level of security for a mobile agent application. Examples of these factors include available security mechanisms, performance requirements, costs, sensitivity of the mobile agent's code and data, the maximum acceptable risk, et cetera. It therefore makes sense to consider these factors when evaluating the application in order to determine its required level of protection, as these will inevitably influence the efficiency as well as computational costs required to establish a secure execution environment.

5.3.5 Avoiding multiple communication sessions

Current mobile agent systems and proposed countermeasures employ multiple communication sessions either between mobile agent and local host or between local host and remote hosts. The purpose of these multiple sessions are usually either to establish a handshake protocol (such as exchanging session keys) between the agent and its remote host, or to convey aggregated results back to the local host. However, the need for multiple sessions compromises the advantage of minimum bandwidth requirements that are typically associated with mobile agents as described in earlier chapters.

5.3.6 Minimising the computational cost for the deployment countermeasures

Whilst most applications have insufficient countermeasures to create a secure execution environment for the mobile agent, runtime efficiency can seriously be compromised with the deployment of unnecessary countermeasures. It is therefore a challenge to find the balance between the required number of security measures and an over-burdened system. For example, many *detection mechanisms* require excessive computations at the local host once the detection trace data is accumulated from the different remote hosts. This implies that if a specific application does not require the full extent of detection mechanisms, it is a waste of computational time and cost.

5.4 Requirements for a Mobile Agent Security Framework

As stated in Chapter 2, the criteria that protects a mobile agent against a malicious host is based on the fundamental concerns or requirements of users gaining access of computer network services, namely *integrity, availability, confidentiality* and *authentication.* These concerns together with the challenges discussed in the previous section, are used as the basis for establishing the requirements for an integrated mobile agent security framework. We propose the following eight requirements for an integrated mobile agent security framework:

1. The framework must provide different levels of security, depending on the *type of implementation environment* in which the mobile agent would be deployed.
2. The framework must incorporate different levels of security depending on the *type*

of application and *agent objectives*.

3. The framework must maintain and not hamper the autonomy and mobility factor of the agent.
4. *Additional security implementations* on the remote hosts (and the system as a whole) must be kept to the minimum, to reduce cost and time. This includes both additional hardware and software requirements.
5. The number of *communication sessions* between the remote hosts (and between remote hosts and other entities) must be minimised. There also needs to be no permanent connection between the agent and the local host.
6. *Computational cost* of implementing countermeasures and maintenance thereof must be as low as possible.
7. The cost of implementation should be affordable or at least minimised. The *financial costs* of implementing countermeasures need to be in direct relation with the degree of security required.
8. The host must possess intrinsic mechanisms to support the security requirements of the agent. This implies the *provision and integration of additional security functions and services*, according to the needs of the application and hence the agent.

5.5 Evaluation of proposed and current countermeasures, frameworks, architectures, models, systems and applications

The purpose of this section is to evaluate the different proposed and current countermeasures, frameworks, architectures, models, systems and applications against the requirements as described in 5.4.

5.5.1 Evaluation of countermeasures

Chapter 3 provides a detailed view on the countermeasures on mobile agent security threats that are currently available in literature. We use the requirements of an integrated mobile agent security framework (as described in 5.4) to evaluate the different countermeasures and assign an applicability property to the different countermeasures for specific requirements. Also included in the evaluation is the countermeasure that came forward from the discussion of the mobile agent frameworks, architectures and models (see Chapter 4), namely the use of *watermarking* techniques.

Requirements 1, 2, 8:

The first two requirements insist on the framework offering different levels of security that are based on the agent execution environment as well as the type of application. Since these two requirements involve the incorporation of various countermeasures or potentially the implementation of different degrees thereof, it is not sensible to evaluate them in terms of individual countermeasures. The requirements for different levels of security (requirements 1 & 2), as well as the requirement for the procurement of additional services (requirement 8) are discussed in greater detail in 5.5.2, 5.5.3 & 5.5.4, as it is not sensible to evaluate individual countermeasures in terms of these requirements

Requirement 3: Autonomy and mobility

A large number of countermeasures for protecting mobile agents against malicious acts from a remote host rely on the creation of a trusted environment. As mentioned before, one of the main disadvantages of a trusted domain is that it impedes the autonomy of the mobile agent by placing restrictions on the choice of service providers to be visited.

To remind the reader, a trusted environment can either be created through the installation of tamper resistant hardware or specific software methods. *Tamper resistant hardware* such as proposed by Wilhelm *et al.* (1998, 1999, 1999a, 2000) and Fünfrocken & Mattern (1999), restricts the autonomy of the mobile agent because the agent can only migrate to a list of pre-defined hosts, which have installed the compulsory specialised hardware.

The use of *software methods* to create a trusted environment entails the deployment of encryption and authenticating techniques, such as proposed by Sander & Tschudin (1998). As described before, the implementation of measures that enforce the use of specific (trusted or pre-listed) hosts restrict the autonomy of the agent.

In the case of a countermeasure employing encryption techniques, a *trusted third party* may be needed (depending on the type of implementation) as a certification authority for the management and maintenance of keys for encryption purposes. Such a mechanism does not inhibit the autonomy of the agent, but in the case of the trusted entity being used for secure computations (Feigenbaum & Lee, 1997), the autonomy and mobility are negatively affected.

Countermeasures based on recording and tracking techniques such as *path histories* (Ordille, 1996), *detection objects* (McDermott & Goldschlag, 1996), *reference states* (Hohl, 1999), *state appraisal* (Farmer *et al.*, 1996) and *proof carrying code* (Necula & Lee, 1998) have no limiting effect on the autonomy of the agent. Countermeasures such as *itinerary recording with replication and voting* (Minsky *et al*, 1996; Schneider, 1997) and *mutual itinerary recording* (Roth, 1998) restrict the autonomy, as it requires the replication or tracking of the mobile agent by cooperating agents. *Mutual itinerary recording* (Roth, 1998) can potentially inhibit the autonomy of a mobile agent by requiring the establishment of an authenticated channel between the two agents.

Phone home (Grimley & Monroe, 1999) places a requirement for a continuous link between the executing host and the local host, as results that are obtained at every host are sent back to the local host. Not only does this mechanism violate many of the requirements established in 5.4, but it also affects the autonomy negatively as the local host requires persistent communication with its agent. Splitting the agent into *cooperating agents* (Yee, 1997) also limits the autonomy of the agent as the itinerary of such an agent is pre-defined and no dynamic changes are possible without direct communication channels between the cooperating agents.

Cryptographic techniques that have no influence on the autonomy and mobility of the mobile agent include *anonymous itinerary* (Westhoff *et al.*, 1999), *cryptographic traces* (Vigna, 1998), *computing with encrypted functions* (Sander & Tschudin, 1998), *environmental key generation* (Riordan & Schneier, 1998), *partial result encapsulation* (Chess et al., 1995; Jansen, 2000; Yee, 1997) and *digital signatures* (Sander & Tschudin, 1998).

Countermeasures based on code obfuscation and time techniques (namely *code obfuscation* (Hohl, 1997, 1998), *code transformation* (An *et al.*, 2002) and *time* *sensitive agents* (Grimley & Monroe, 1999)), as well as *watermarking* (Jalali, *et al.*, 2000) do not restrict the autonomy or mobility of the mobile agent.

Requirement 4: Additional requirements for implementation

The use of *tamper resistant hardware* (see Wilhelm *et al.* (1998, 1999, 1999a, 2000) and Fünfrocken & Mattern (1999)) requires each host to incorporate specialised equipment in order to provide a secure environment for incoming mobile agents. Hosts that are not able to provide guarantees for the implementation of such required hardware will not be able to host mobile agents. Tamper resistant hardware therefore places additional requirements for the implementation of security countermeasures, which is undesirable for secure autonomous mobile agents.

Software methods to create a trusted environment (Sander & Tschudin 1998) rely on encryption and authentication methods and require the host to provide methods for supporting the agent. The additional requirements for cryptographic methods are related to the algorithms and protocols being used for implementation. For example, if the system requires encryption algorithms that are based on the underlying virtual machine, then the additional requirements are minimum. The countermeasures discussed in Chapter 3 are merely propositions and lack implementation details; in cases where cryptographic techniques are incorporated into the countermeasures, additional requirements for implementation are positively stated. Examples of such measures are *anonymous itinerary* (Westhoff *et al.*, 1999), *cryptographic traces* (Vigna, 1998), *environmental key generation* (Riordan & Schneier, 1998), et cetera.

Trusted third parties (Feigenbaum & Lee, 1997) require extra hardware to be implemented (for both certification authority as well as secure computations). In the case of the third party acting as a certification authority, the additional requirements can be a minimum if these entities already exist within the specified domains.

Detection objects (McDermott & Goldschlag, 1996) are application specific and require the creation and maintenance of the objects within the agent as additional requirements. *Itinerary recording with replication and voting* (Minsky *et al*, 1996; Schneider, 1997) requires some method of duplication for execution, while *mutual itinerary recording* (Roth, 1998) requires a duplicate of the agent, a trusted environment for the duplicated agent as well as the creation of an authenticated channel between the two agents.

Phone home (Grimley & Monroe, 1999) requires a direct link between the remote host and the local host of the owner in order to convey results retrieved and the use of a *mobile agent system* (Yee, 1997), needs the creation of distributing cooperating agents as additional implementation requirements. A number of the measures listed, require only additional methods to be implemented on the local host and not any of the remote hosts. Examples of these measures are *code obfuscation* (Hohl, 1997, 1998), *proof-carrying code* (Necula & Lee, 1998), *code transformations* (An *et al*., 2002) and *watermarking techniques* (Jalali, *et al.*, 2000), where the code of the mobile agent needs to be transformed or proof's added before migration to the first host on the itinerary.

Requirement 5: Number of communication sessions

Remote hosts that make use of *tamper resistant hardware* (Wilhelm *et al.* (1998, 1999, 1999a, 2000); Fünfrocken & Mattern (1999)) to protect mobile agents, accept the mobile agent after migration, upon which the agent is sent to the specialized hardware component. No additional communication sessions need to be established between remote hosts during the migration process but depending on the location of the physical trusted component, extra sessions between the host and its tamper resistant hardware may be required.

When using *software methods* (Sander & Tschudin, 1998), to create a trusted environment, the communication sessions can increase noticeably depending on the implemented approach. For example, when the chosen algorithm requires a public/private key pair for implementation, additional communication sessions may be needed for the establishment of keys, especially if the certification authority is an external entity.

The amount of communication sessions between a remote host and the trusted third party will also increase if either *trusted entities* are used as certification authorities, or if secure places of computations are established. In this scenario, the mobile agent requires communication sessions with the trusted entity during execution at a specific remote host for the purpose of transferring secret data or for requiring keys for encryption/decryption.

Recording and tracking techniques such as *itinerary recording with replication and voting* (Minsky *et al*, 1996; Schneider, 1997) and *mutual itinerary recording* (Roth, 1998) require the establishment of additional communication sessions in order to allow for the sending of information between the cooperating agents. Another example is *phone home* (Grimley & Monroe, 1999) where it is necessary to establish a communication link between the agent and the local host for the transfer of aggregated data after execution at every host.

Countermeasures that are based on some form of cryptographic techniques may possibly require the establishment of additional communication sessions depending on the implementation details. For example, systems that rely on certification authorities for the distribution of keys, will require communication lines between the remote hosts and the trusted entity. *Digital signatures* (Sander & Tschudin, 1998) and *anonymous itinerary* (Westhoff *et al.*, 1999) are examples of such systems. In *execution traces* (Vigna, 1998) a receiving host first has to authenticate the mobile agent before acceptance and execution, which leads to a considerable increase in communication between subsequent remote hosts.

Proof-carrying code (Necula & Lee, 1998) does not make use of cryptography or trusted third parties, but does however require links with proof validators for authenticating the agent.

Measures based on *code obfuscation* (Hohl, 1997, 1998) and *time techniques* (Grimley & Monroe, 1999) require no additional communication sessions between entities in the system. The same is true for *watermarking* techniques (Jalali, *et al.*, 2000)

Requirement 6: Computational costs

Tamper resistant hardware (see Wilhelm *et al.* (1998, 1999, 1999a, 2000) and Fünfrocken & Mattern (1999)), have no implication of computational costs since the agent is executed as normal, but only in the context of specialised hardware.

Computational costs involved in *software methods* depend on the implementation algorithm used. Computational costs implications for software methods used in the creation of a *trusted environment* (such as Sander & Tschudin, 1998) depend on the chosen cryptographic algorithms. This is also true for any countermeasures that use *cryptographic methods* for signing or encrypting the agent or parts thereof, being that different encryption algorithms require different amounts of computational resources. Examples of such systems are *environmental key generation* (Riordan & Schneier, 1998), *anonymous itinerary* (Westhoff *et al.*, 1999), digital *signatures* (Sander & Tschudin, 1998) and *execution traces* (Vigna, 1998).

The use of *trusted third parties* (Feigenbaum & Lee, 1997) does not have an affect on the computational cost of the remote host or the system as a whole, being that the computation is transferred to the trusted entity.

Path histories (Ordille, 1996) and *reference states* (Hohl, 1999) require the host to be able to digitally sign the agent or parts thereof. If any, the costs involved are dependent on the methods required for signing purposes. After a host has digitally signed an agent (or parts thereof) in the *path histories* countermeasure, it is sent to the next host. The receiving host then has the opportunity to authenticate the agent before acceptance into the system. As the number of hosts on the itinerary of the agent increases, so does the computational costs required from each host.

Countermeasures based on the detection of malicious acts such as *partial result encapsulation* (Chess et al., 1995; Jansen, 2000; Yee, 1997) and *detection objects* (McDermott & Goldschlag, 1996), cause large increases of overhead costs on the local host of the agent. This is due to the log reports and reference information being encapsulated and sent back to the owner for determining if malicious modifications had occurred.

Countermeasures based on tracking techniques such as *mutual itinerary recording* (Roth, 1998) require the creation of an authenticated channel between the cooperating agents. In this case, added cost in terms of computations is also required as the agent is executed both at the remote host and the trusted entity. With *reference states* (Hohl, 1999) the agent is re-computed and the initial and final state digitally signed - these mechanisms obviously also increase the computational cost of the method.

Code obfuscation (Hohl, 1997, 1998) and *code transformations* (An *et al*., 2002) imply an increase in computational costs for the owner or creator of the agent due to the scrambling of the code being done at the local host. The creation of a *watermark* (Jalali, *et al.*, 2000) requires additional computational costs at the local host. The validation thereof also requires additional costs for the remote hosts.

Requirement 7: Financial implications

Trusted hardware components (Wilhelm *et al.*, 1998, 1999, 1999a, 2000; Fünfrocken & Mattern, 1999), are costly to implement and has deterred service providers from using this technology due to the financial implications.

Costs involved for *software methods* depend on the implementation algorithm used. The use of *trusted third parties* acting as certification authorities for the management and distribution of keys, can have extra financial implications if such trusted units do not exist within a specified domain. For use as a secure environment for computations such entities need to be created and maintained and can thus instigate additional costs.

Countermeasures based on tracking techniques such as *mutual itinerary recording* (Roth, 1998) require the creation of an authenticated channel between the cooperating agents.

The financial implications of using cryptographic techniques depend on the implementation software being used. Some software is available free of charge and commercial versions negatively affect the financial costs of these countermeasures. It is also noted that some cryptography methods are available from the core system being used (such as the underlying Java virtual machine). Examples of measures that incorporate cryptographic techniques are *anonymous itinerary* (Westhoff *et al.*, 1999), *cryptographic traces* (Vigna, 1998) and *encrypted functions* (Sander & Tschudin, 1998). *Path histories* (Ordille, 1996) and *reference states* (Hohl, 1999) only require the host to be able to digitally sign the agent or parts thereof. If any, the costs involved are dependent on the methods used for signing purposes.

For *code obfuscation* (Hohl, 1997, 1998), *code transformation* (An *et al*., 2002) and *proof carrying code* (Necula & Lee, 1998), specialised software is needed only by the local host for the scrambling of the agent. *Proof carrying code* (Necula & Lee, 1998) also falls in this category, due to the creator of the agent requiring special software.

The different countermeasures evaluated against the established requirements for an integrated mobile agent security framework are summarised in Table 5.1.

Table 5.1: Evaluation of countermeasures

	Requirements				
Countermeasures	**Additional communication sessions**	**Inhibits autonomy & mobility**	**Additional computational costs**	**Additional requirements**	**Additional financial costs**
Tamper resistant hardware	Yes	Yes	No	No	Yes
Trusted execution environment	Yes	Yes	Yes	Yes	Yes
Trusted third party - certification authority	No	Yes	Yes	No	Yes
Trusted third party - computations	Yes	Yes	Yes	No	Yes
Path Histories	No	Yes	No	Yes	Yes
Detection objects	No	Yes	No	Yes	No
Itinerary recording	Yes	Yes	Yes	Yes	Yes
Mutual itinerary recording	Yes	Yes	Yes	Yes	Yes
Reference states	No	Yes	No	Yes	Yes
Phone home	Yes	Yes	Yes	Yes	No
Mobile agent system	Yes	Yes	Yes	Yes	No
State appraisal	No	Yes	No	Yes	Yes
Proof carrying code	No	Yes	Yes	Yes	Yes
Anonymous itinerary	No	Yes	No	Yes	Yes
Cryptographic traces	No	Yes	Yes	Yes	Yes
Encrypted functions	No	Yes	No	Yes	Yes
Environmental key generation	No	Yes	No	Yes	Yes
Partial result encapsulation	No	Yes	No	Yes	Yes
Digital signatures	No	Yes	No	Yes	Yes
Code obfuscation	No	Yes	No	Yes	Yes
Code transformation	No	Yes	No	Yes	Yes
Time sensitive agents	No	No	No	No	No
Watermark techniques	No	Yes	No	Yes	Yes

5.5.2 Frameworks, architectures and models

Whilst we have evaluated the available or proposed countermeasures against our requirements for an integrated security framework in 5.5.1, we will continue this trend in this section and evaluate proposed and implemented mobile agent security frameworks, architectures and models. (These models were described in Chapter 4). A summary of the evaluation is depicted in Table 5.2.

Requirement 1: Type of implementation environment

As illustrated in Chapter 4, we categorised the different mobile agent frameworks, architectures and models into proposed systems based on the notion of a *trusted* environment and those that can be employed in an *open* environment. A number of the proposed systems that were discussed indicated the necessity to distinguish between different security mechanisms for dissimilar implementation environments in which the agent will be deployed. Proposals such as *M&M* (Marques *et al,* 2001), *Mansion* (Van't Noordende *et al*, 2002) and *Plain text algorithm* (An *et al,* 2002) incorporate various security approaches into their designs based on the implementation environment in which the mobile agent is to be deployed. The same indication is also further emphasised if one considers the variation in security methods employed in proposed systems that operate in a trusted environment as opposed to an open environment.

Proposed frameworks in an open environment (such as *SAWMA* (Luo, 2001) and *Agent factory* (Brazier *et al.*, 2002)) do not provide for the choice between different implementation environments.

Requirement 2: Type of mobile agent application areas

Various proposed frameworks, models and architectures are designed specifically for unique mobile agent applications. This leads to the incorporation of different security mechanisms for different types of applications. For example, *FILIGRANE* (Jalali *et al,* 2000), *M&M* (Marques *et al,* 2001), *Mansion* (Van't Noordende *et al*, 2002), *PLANET*

(Kato *et al*, 1996), *Distributed transactions* (Vogler *et al*, 1997) and *Plain text algorithm* (An *et al*, 2002) are all proposed systems for providing mobile agent security within a specific application. According to the different applications (such as information retrieval for obtaining flight information versus the actual booking agent for a flight), different security technologies are incorporated within the design.

Although proposed systems such as *FILIGRANE* (mobile commerce), *Mansion* (distributed applications) and *Self-protecting mobile agents* (distributed) are application specific systems, different levels of security are not incorporated into their designs.

Requirement 3: Autonomy and mobility

Although the notion of a trusted environment goes against the autonomy and mobility feature of the mobile agent, some systems, due to restraints such as costs and sensitivity, require deployment in a trusted setting.

A large number of the proposed frameworks, architectures and models require a trusted environment as a core feature on which it is built. Different methods of creating a trusted environment are used, for example *FILIGRANE* (Jalali *et al*, 2000) and *Electronic supermarkets* (Wu, 2000) incorporate *trusted hardware* (Wilhelm *et al.,* 1998, 1999, 1999a, 2000) while the *POM* (Guan *et al*, 2000) and *Supervisor-worker* (Fischmeister, 2000) frameworks make use of a trusted entity within domains for accepting agents into the domain as well as to create slave agents for acquiring information on behalf of the mobile agent. *Mansion* (Van't Noordende *et al*, 2002) only allows the agent to migrate according to a pre-defined list of trusted hosts. The use of a trusted entity to act as a certification authority is implemented by a number of systems such as *Security enhanced mobile agents* (Varadharajan, 2000), *M&M* (Marques *et al*, 2001) and *Distributed transactions* (Vogler *et al.*, 1997), for the distribution and management of keys. This feature does not inhibit the autonomy or mobility of the agent, as certification authorities are used as a general means of providing security in computer systems.

Systems such as *FILIGRANE* (Jalali *et al,* 2000) require the use of smart cards by the hosts before a mobile agent is able to migrate to the host. This limits the autonomy of the agent, due to the implementation specifics of the design. *Planet* (Kato *et al.*, 1996) also makes use of trusted hardware component, with the advantage that it is virtually already available on all computers. *PLANET* (Kato *et al.*, 1996) and *Proxy agents* (Mitroviæ & Arribalgaza, 2002) also requires a trusted entity as a point of entry into a domain, in which the hosts are required to register and from which the agent is allowed to migrate to the specified hosts. This feature places a prominent restriction on the autonomy of the agent.

Systems developed for an open environment such as *SAWMA* (Luo, 2001) and *Three-tier protection model* (Sameh & Fakhry, 2002) do not influence the autonomy of the mobile agent.

Requirement 4: Additional requirements for implementation

The analysis of the current and proposed frameworks, architectures and models revealed that proposed systems that are based on the creation of a trusted environment require a large number of additional costs (in some instances) in terms of implementation. Systems such as POM (Guan *et al.*, 2000) and *Supervisor-worker framework* (Fischmeister, 2000) require a trusted host to secure computations in every domain. Where a trusted entity is used to serve as a certification authority, the additional requirements depend on the existing infrastructure within a specific domain. Examples include the *Security enhanced mobile agents* (Varadharajan, 2000), *M&M* (Marques *et al*, 2001) and *Distributed transactions* (Vogler *et al.*, 1997). The implementation of trusted entities rely on the size of these domains, the number of mobile agents within a domain, as well as the current existence of such units within a domain.

The use of specialised *trusted hardware* (Wilhelm *et al.,* 1998, 1999, 1999a, 2000) requires hosts to have the devices installed before acceptance into a domain. Examples are *Electronic supermarkets* (Wu, 2000), *PLANET* (Kato *et al.,* 1996) and *FILIGRANE* (Jalali *et al*, 2000). The latter also requires code obfuscation and watermarking methods to be available.

As discussed in the analysis of the countermeasures contained in the previous section, systems that incorporate cryptographic techniques require appropriate software for the implementation of these techniques. Examples of frameworks that include cryptographic techniques are *Mansion* (Van't Noordende *et al*, 2002), *Distributed transactions* (Vogler *et al.*, 1997), and *DNX* (Schütz *et al.*, 2000).

Some of the suggested frameworks or proposed systems incorporate countermeasures not discussed in the previous section. For example *SAWMA* (Luo, 2001) uses *code obfuscation* (Hohl, 1997, 1998) and *Java watermarking techniques.* The latter requires additional software for the local host in order to obfuscate code and to create legitimate watermarks.

Requirement 5: Number of communication sessions

As described earlier, it is desirable that the number of communication sessions between local hosts, executing remote hosts as well as external hosts (for example a trusted third party) must be kept to a minimum. In the case of systems operating in a trusted environment, the communication sessions between the different entities depend on the methods used in the creation of the trusted environment. Proposed systems such as *POM* (Guan *et al.*, 2000) and *Supervisor-worker framework* (Fischmeister, 2000) require a trusted entity for secure computations (Wilhelm *et al.*, 1998, 1999, 1999a, 2000), which doubles the sessions as opposed to the agent just migrating to the remote hosts as specified on the itinerary. When incorporating a trusted entity as certification authority (for key management), the communication session between the remote host and the certification authority will at least double. Other factors such as the encryption protocol might even escalate this figure. Examples of these types of proposed systems are *Security enhanced mobile agents* (Varadharajan, 2000) and *Distributed transactions* (Vogler *et al.*, 1997).

FILIGRANE (Jalali *et al*, 2000) and *Electronic supermarkets* (Wu, 2000) that make use of *trusted hardware* (Wilhelm *et al.*, 1998, 1999, 1999a, 2000) do not require additional communication sessions if the specialised hardware is located on site of the host. *M&M* (Marques *et al.*, 2001) requires additional communication sessions due to the agent being authenticated first (by sending the identification of the agent), before migration to the remote host. *Distributed transactions* (Vogler *et al.*, 1997) requires the establishment of session keys, which increases the number of communication sessions between hosts.

In the case of an agent that is only allowed to migrate to a pre-defined list of trusted hosts such as *Mansion* (Van't Noordende *et al*, 2002), no extra communication sessions are recorded. *DNX* (Schütz *et al.*, 2000) only requires the agents to be encrypted before sending and depending on whether a certification authority is used; no extra communication sessions are required.

In the *Security framework for a mobile agent system* (Bryce, 2000) replication and voting is used. Furthermore, the *Mobile code security framework* (Tan & Moreau, 2002)) requires *cryptographic traces* (Vigna, 1998)), which leads to large increases in communication session. *Self-protecting mobile agents* (D'Anna *et al.*, 2003) requires additional communication sessions in order to allow communication between *agentlets.*

Requirement 6: Computational costs

Where computations are performed within a trusted hardware component, no additional computational costs are incurred (for example *FILIGRANE* (Jalali *et al.*, 2000) and *Electronic supermarkets* (Wu, 2000). However, mobile agents that are required to conduct their computations on a trusted entity (separate from the remote host), cause a decrease in computational cost on the remote host. Due to the computation being moved to the trusted third party, the cost of conducting the computation is transferred from the remote host to the trusted entity. Proposed frameworks, architectures and models in this category are for example *POM* (Guan *et al.*, 2000), and *Supervisor-worker framework* (Fischmeister, 2000).

Systems that incorporate encryption/decryption techniques require the host to perform encryption/decryption on the agent or parts thereof before and after execution. This might also increase the computational costs (depending on the specific algorithm). Examples are *Mansion* (Van't Noordende *et al*, 2002), *Distributed transactions* (Vogler *et al.*, 1997), *DNX* (Schütz *et al.*, 2000) and *Planet* (Kato *et al.*, 1996). In *Proxy agents* (Mitroviæ & Arribalgaza, 2002) the agents are decrypted within the trusted third party present in every domain. Once decrypted and authenticated the agent can move freely within the domain. This means that the computational costs are moved from the host to the trusted entity.

M&M (Marques *et al.,* 2001) incorporates *partial results authentication codes* (Yee, 1997) as a method of creating log reports, which leads to an increase in computation at the remote host.

Requirement 7: Financial implications

In general, the inclusion of specialised hardware for the secure computation of mobile agents incurs additional financial costs (example *FILIGRANE* (Jalali *et al*, 2000), *Electronic supermarkets* (Wu, 2000)).

Systems that require the incorporation of a trusted host for the purpose of secure computations such as *POM* (Guan *et al.,* 2000) and *Supervisor-worker framework* (Fischmeister, 2000) induce additional financial costs on the creation of the security system. This is due to a dedicated machine being set-up per domain, and depending on the size of the domains and the number of mobile agents that need to be processed, the possibility of increased expenses arises. The maintenance of these machines is also something to keep in mind. In terms of creating a trusted entity for key management (in systems such as *Security enhanced mobile agents* (Varadharajan, 2000) and *Distributed transactions* (Vogler *et al.*, 1997)) additional costs are implied, if no such an authority currently exists. If such an authority does exist, additional costs can occur when making use of the service

The use of methods to encrypt the mobile agent or parts of the agent can require additional financial costs depending on the encryption methods to be used (or if underlying system such as Java is used). Examples are *Mansion* (Van't Noordende *et al.,* 2002), *Distributed transactions* (Vogler *et al.,* 1997), *DNX* (Schütz *et al.,* 2000), and *Planet* (Kato *et al.,* 1996).

The incorporation of *code obfuscation* (Hohl, 1997, 1998) and *watermarking* techniques into systems (such as *SAWMA* (Luo, 2001)) require the use of specific software, which have financial cost implications.

Requirement 8: Choices of countermeasures

Mobile agent frameworks, architectures and models (as discussed in Chapter 4) that require a trusted environment for deployment, are usually not permitted to make a decision on specific countermeasures to be used for a specific type of application. In general, choices regarding the countermeasures are based on the creation of the trusted environment. Once the desired environment is created no additional protection is required for the mobile agent. Examples of such systems are *POM* (Guan *et al*, 2000), *Planet* (Kato *et al.*, 1996) and *Supervisor-worker framework* (Fischmeister, 2000). Systems such as *Security enhanced mobile agents* (Varadharajan, 2000) that incorporates the use of trusted entities to act as a certification authority provide the possibility of the use of different measures to combat attacks, for example the use of different encryption algorithms.

A number of proposed systems make use of security mechanisms (depending on the goal of the agent), which are applicable to all agents without any selection opportunities. Example are FILIGRANE (Jalali *et al.,* 2000), which uses encryption, smart cards, code obfuscation and watermarking techniques; *M&M* (Marques *et al.,* 2001), and *Distributed transactions* (Vogler *et al.,* 1997) both use encryption and the creation of log reports.

Some of the proposed frameworks, architectures and models are based on security mechanisms that are provided by the underlying infrastructure of the system in which it is implemented. For example, *FILIGRANE* (Jalali *et al.,* 2000), *M&M* (Marques *et al.,* 2001) and *DNX* (Schütz *et al.,* 2000) make use of the Java security manager as a basis for providing cryptographic techniques.

Different types of mobile agent frameworks, proposed systems and implementations are evaluated against the established requirements for an integrated mobile agent security framework are summarised in Table 5.2.

Table 5.2: Evaluation of frameworks and models

	Requirements

Frameworks / architectures / models			Provide environment levels	
Provide application levels		Inhibits autonomy and mobility		Additional requirements
POM	No	No	Yes	Yes
Security enhanced mobile agents		No	No	Yes
Yes				
FILIGRANE	No	No	Yes	Yes
M&M	Yes	Yes	Yes	Yes
Distributed transactions		No	No	Yes
Yes				
Mansion	No	No	Yes	Yes
DNX	No	No	No	No
Planet	No	No	Yes	Yes
Proxy agents	No	No	Yes	Yes
Electronic supermarkets		No	No	Yes
Yes				
Supervisor worker	No	No	Yes	Yes
SAWMA	No	No	No	Yes
Agent Factory	No	No	No	No
Security framework for mobile agent system		No	No	No
No				
Mobile code security framework		No	No	No
Yes				
Self protecting mobile agents		No	No	No
Yes				
Plaintext algorithm	Yes	Yes	No	Yes
Clone	No	No	No	Yes
Three tier protection model		No	No	No
Yes				

Table 5.2: Evaluation of frameworks and models (cont.)

Requirements				
Frameworks / architectures / models			Additional communication	
sessions	Additional computational costs		Additional financial costs	
Counter choices				
POM	Yes	No	Yes	No
Security enhanced mobile agents		Yes	Yes	Yes
Different encryption methods				
FILIGRANE	No	No	Yes	No
M&M	Yes	Yes	Yes	Yes
Distributed transactions		Yes	Yes	Yes

	No				
	Mansion	No	Yes	Yes	No
	DNX	CA?	Yes	Yes	No
	Planet	Yes	Yes	Yes	No
	Proxy agents	Yes	No	Yes	No
	Electronic supermarkets		No	No	Yes
	No				
	Supervisor worker	Yes	No	Yes	No
	SAWMA	Yes	Yes	Yes	No
	Agent Factory	No	No	No	No
	Security framework for mobile agent system		Yes	Yes	Yes
	No				
	Mobile code security framework		Yes	Yes	No
	No				
	Self protecting mobile agents		Yes	No	No
	No				
	Plaintext algorithm	No	Yes	No	No
	Clone	Yes	Yes	Yes	No
	Three tier protection model		Yes	Yes	Yes
	No				

5.5.3 Mobile agent systems and tools

Chapter 4 saw discussions on a number of mobile agent systems that can be used as a basis for the generation of mobile agent applications. A large number of these systems are the result of research projects initiated by academic and research institutions. As the acceptance of mobile agent systems is reliant on their ability to provide protection for the mobile agent, it is essential to evaluate the described mobile agent systems against the requirements for a security framework (see 5.4), in order to aid in the process of defining such a framework. This section provides the mentioned analysis of which a summary is listed in Table 5.3.

Requirement 1: Type of implementation environment

The analysis of the mobile agent systems and tools as described in the previous chapter, displayed that none of the systems provide for different levels of security depending on the type of implementation environment. A large number of these systems are built on the security designs of the underlying operating system, language or virtual machine and only make use of encryption and digital signature algorithms for providing security to the agent.

Requirement 2: Type of mobile agent application areas

None of the mobile agent systems and tools analysed, integrate different levels of security according to the type of application for which the mobile agent will be used.

Requirement 3: Autonomy and mobility

A large number of systems (such as *ADK* (ADK), *D’Agents* (Gray *et al.*, 1998) and *SOMA* (Corradi *et al.*, 1999)) do not inhibit the autonomy and mobility of the agent. Systems that do however place a restriction on the autonomy or mobility of the mobile agent are for example *Agent TCL* (Gray, 1996), which requires the agent to register at the remote host before migration, *aZIMAs* (Nalla *et al*., 2002), that makes use of a trusted set of hosts and *Jumping Beans* (Jumping Beans) that entails the agent being transferred to a trusted central host between migrations.

Requirement 4: Additional requirements for implementation

Mobile agent systems such as (*ADK* (ADK), *Aglets* (Karjoth *et al.*, 1997), *Ajanta* (Karnik & Tripathi, 2000), *AMETAS* (Zapf *et al.*, 1998), *Anchor Toolkit* (Mudumbai *et al.*, 1999), *aZIMAs* (Nalla *et al*., 2002), Concordia (Kiniry & Zimmerman, 1997), *D’Agents* (Gray *et al*., 1998), *S-agent* (Makino *et al.*, 2000), *SeMoA* (Roth & Jalali, 2001) and *WASP* (Fünfrocken & Mattern, 1999)) are built on the Java platform, which require the installation of the Java virtual machine before the implementation of the agent systems. *Agent TCL* (Gray, 1996) and *TACOMA* (Johansen *et al.*, 1995) are built on the Tcl scripting language.

Systems such as *ADK* (ADK) and *AMETAS* (Zapf *et al.*, 1998) incorporate digital signing of parts of the agent, which will require a certification authority for the provision of private/public key pairs. It is also possible that the certification authority can form part of the functions of the current host. *Agent Tcl* (Gray, 1996) requires an additional server within a domain for registration and key management purposes of the mobile agent.

Requirement 5: Number of communication sessions

Additional communication sessions for the distribution of keys will depend on the location (or use) of a certification authority. For example *ADK* (ADK) and *AMETAS* (Zapf *et al.*, 1998), make use of digital signing and will require the generation of public / private key pairs either by the host (no additional communication sessions) or a certification authority (additional communication sessions). *Agent Tcl* (Gray, 1996) requires the agent to first register at a server for encryption and signing purposes, before being sent to the first remote host. This implies additional communication sessions.

Requirement 6: Computational costs

Additional costs in terms of computations are considered in cases where the mobile agent system makes use of cryptography techniques for encryption and signing purposes. Examples of mobile agent systems that incorporate digital signing and certificates are *ADK* (ADK), *Agent Tcl* (Gray, 1996) and *AMETAS* (Zapf *et al.* (1998). *Ajanta* (Karnik & Tripathi, 2000) also incorporates the use of logs for detection purposes that have added computational costs.

Requirement 7: Financial implications

A number of systems are being developed as research projects at various institutions, of which some progressed to become commercial systems. A mobile agent system that can be used for research purposes (but needs to be paid for if used commercially) is *ADK* (ADK). Examples of systems that are available for deploying mobile agent applications free of charge are *Aglets* (Karjoth *et al.*, 1997) and *Agent Tcl* (Gray, 1996) of which the latter also requires an additional server (such as a certification authority) for registering and signing the agent.

Requirement 8: Choices of countermeasures

A large number of systems don't provide the owner or developer of the mobile agent with a choice of possible countermeasures. Systems such as *ADK* (ADK), only provide for the digital signing of parts (or whole) of the agent, while systems such as *Agent Tcl* (Gray,

1996) also incorporates encryption techniques.

It is however possible to incorporate possible additional countermeasures based on the system used for development and deployment of the mobile agent system. For example Java provides a number of possibilities such as encryption as well as different encryption algorithms and programs. *Ajanta* (Karnik & Tripathi, 2000) provides three layers of protection, namely read-only containers, append-only logs and only accessible to certain hosts.

Table 5.3: Evaluation of mobile agent systems

	Requirements				
Mobile agent systems			Provide environment levels		Provide
application levels		Inhibits autonomy and mobility		Additional requirements	
	ADK	No	No	No	Yes
	Agent TCL	No	No	Yes	Yes
	Aglets	No	No	Yes	Yes
	Ajanta	No	No	Yes	Yes
	AMETAS	No	No	No	Yes
	Anchor	No	No	Yes	Yes
	ARA	No	No	No	No
	aZIMAs	No	No	Yes	Yes
	Bee-gent	No	No	No	No
	Concordia	No	No	No	Yes
	D’Agents	No	No	No	Yes
	Jumping Beans	No	No	Yes	No
	S-agent	No	No	No	Yes
	SeMoA	No	No	No	Yes
	SOMA	No	No	No	Yes
	TACOMA	No	No	No	Yes
	WASP	No	No	Yes	Yes

Table 5.3: Evaluation of mobile agent systems (cont.)

	Requirements				
Mobile agent systems			Additional communication sessions		Additional
computational costs		Additional financial costs		Counter choices	
	ADK	Yes	Yes	Yes	No
	Agent TCL	Yes	Yes	Yes	No
	Aglets	Yes	Yes	No	No
	Ajanta	Yes	Yes	No	Yes
	AMETAS	No	Yes	No	No
	Anchor	No	Yes	No	No
	ARA	No	Yes	No	No
	aZIMAs	No	No	No	No
	Bee-gent	No	Yes	No	No
	Concordia	No	Yes	No	No
	D’Agents	No	Yes	No	No
	Jumping Beans	No	Yes	Yes	No
	S-agent	No	Yes	No	No
	SeMoA	No	Yes	No	No
	SOMA	No	Yes	Yes	No
	TACOMA	No	No	Yes	No

	WASP	No	No	Yes	No

5.5.4 Mobile agent system applications

The process of analysing mobile agent countermeasures, models, frameworks, architectures and systems against the requirements for a security framework, is continued in this section with the analysis of mobile agent system applications (as detailed in Chapter 4). Table 5.4 provides a summary of the analysis results.

Requirement 1: Type of implementation environment

Current applications developed within the mobile agent paradigm, are mostly developed for a specific environment. This has the effect that none of the systems that were evaluated make provision for different levels of security according to the environment in which the agent are deployed.

Requirement 2: Type of mobile agent application areas

All of the analysed mobile agent applications were developed for a specific application. This means that the security techniques incorporated within the applications are related to a specific environment and multi-levels of security are not catered for.

Requirement 3: Autonomy and mobility

The autonomy and mobility of the agent is restricted in systems such as *MAgNET* (Dasgupta *et al.*, 1999) *that* requires the licensing of the agent, *Mobile agent based transactions in open environments* (De Assis Silva & Popescu-Zeletin, 2000) that incorporate the split of the agent into multi-agents and *SIAS* (Chan *et al*., 2000) that requires a trusted set of hosts.

Requirement 4: Additional requirements for implementation

SIAS (Chan *et al*., 2000) incorporates the use of a key server for the distribution and management of keys, while *MAgNET* (Dasgupta *et al.*, 1999) requires the licensing of mobile agent code. Applications such as *Mobile agent based transactions in open* *environment* (De Assis Silva & Popescu-Zeletin, 2000) and *Secure electronic transactions* (Kotzanikolaou *et al.,* 1999) requires the mobile agent to be split into multi-agents of which each are required to complete a sub-task of the mobile agent.

Requirement 5: Number of communication sessions

Additional communication sessions are required by *Mobile agent based transactions in open environment* (De Assis Silva & Popescu-Zeletin, 2000) and *Secure electronic transactions* (Kotzanikolaou *et al.,* 1999) due to the inter-agent communication sessions between the slave agents. *SIAS* (Chan *et al*., 2000) also requires additional communication sessions with the key server that is used for key distribution and management.

Requirement 6: Computational costs

Additional computational costs are reflected in applications that incorporate digital signatures or encryption techniques. Examples of such applications are *MAgNET* (Dasgupta *et al.,* 1999), *Secure electronic transactions* (Kotzanikolaou *et al.,* 1999) and *SIAS* (Chan *et al.,* 2000).

Requirement 7: Financial implications

SIAS (Chan *et al*., 2000) makes use of a key server to manage key distributions. This aspect can have additional financial implications if such a server does not exist within a specified domain.

Requirement 8: Choices of countermeasures

Cherubim (Campbell & Qian, 1998) is the only analysed mobile agent system application that allows for the incorporation of different countermeasures. The architecture has a pre-configured core security service; after which new security measures can be dynamically injected into this basic system.

Table 5.4: Evaluation of mobile agent system applications

	Requirements				
Mobile agent systems and tools				Provide environment levels	
	Provide application levels		Inhibits autonomy and mobility		Additional

requirements

Attack Resistant Distributed Hierarchical IDS	No	No	No	No
Cherubim	No	No	No	Yes
MAgNET	No	No	Yes	Yes
Mobile agent based transactions	No	No	Yes	Yes
Secure Electronic Transactions	No	No	Yes	Yes
SIAS	No	No	Yes	Yes
Virtual Internet Pets	No	No	Yes	Yes

Table 5.4: Evaluation of mobile agent system applications (cont.)

	Requirements			
Mobile agent systems and tools	**Additional communication sessions**	**Additional computational costs**	**Additional financial costs**	**Counter choices**
Attack Resistant Distributed Hierarchical IDS	No	No	No	No
Cherubim	Yes	Yes	No	Yes
MAgNET	Yes	Yes	No	No
Mobile agent based transactions	Yes	No	No	No
Secure Electronic Transactions	Yes	No	No	No
SIAS	Yes	Yes	Yes	No
Virtual Internet Pets	Yes	No	No	No

5.6 Conclusion

The requirements for a mobile agent security framework were proposed upon which the different countermeasures, frameworks, architectures, models, mobile agent systems and applications were evaluated against the proposed criteria. This provides essential analysis information for the creation of a mobile agent security framework, which forms the focus of the next chapter.

CHAPTER 6

PROPOSED FRAMEWORK

6.1 Introduction

The criteria and requirements of a mobile agent security framework were outlined and described in Chapter 5. We used these requirements to evaluate the individual mobile agent countermeasures as well as mobile agent systems, frameworks, architectures, models and applications. These evaluation results directed us to the proposal of a mobile agent security framework, as outlined in this chapter.

6.2 Establishing Security Levels

The identification of the challenges (see 5.3) and the requirements (see 5.4) of a mobile agent security framework brings us closer to establishing a security framework that is appropriate for different mobile agent applications. As the first step in providing an integrated security framework, we distinguish between different levels of security. In doing this (depending on the application and the environment), the mobile agent can be deployed in various degrees of a trusted environment.

Classification of mobile agent application areas

The different uses and applications of mobile agents, as described in Chapter 2, lead to the following grouping into three categories:

> *Information retrieval:* Applications in this category are responsible for searching and retrieving information from different hosts and then to convey these results <u>back to the owner</u> of the mobile agent. Examples of such applications are search engines and requesting prices for specific goods at different vendors. These types of applications request a mere lookup of a database or table and return the information to its owner.
>
> *Information conveying and retrieval:* This category includes the functionality of the previous category, with additional roles. Applications that not only retrieve information

but also convey results between the different hosts, or between hosts and the mobile agent's owner. Examples include network testing and the confirmation of information. As can be seen from the examples, these types of applications take the information that they gather along to a next host where the host is able to make informed decisions based on the information that it has received. As stated above, such a host is not necessarily the mobile agent's owner.

Computations: Besides the retrieval and transport of information, this category also has the added ability of performing some computations (to different degrees) on the different hosts. E-commerce applications and the well-known airline ticket reservation system are examples of this category. Different to the previous category, where the gathered information enabled a host to make informed decisions, the gathered information is usually used (in this case) by the mobile agent to make informed decisions.

The different groups of applications necessitate different levels of security that must be incorporated in the mobile agent security framework.

Classification of implementation environments

As suggested by the analysis of the mobile agent system frameworks and models in Chapter 4, an agent security framework is not just dependent on the applications but also on the environment in which the applications operate. We categorise mobile agent applications as being able to operate in three different implementation environments, namely in a *trusted* environment, in a *pre-defined* environment and in an *open* environment.

A *trusted* environment is a network consisting of trusted nodes and a mobile agent is only deployed amongst these trusted hosts. The level of trust between the owner of the mobile agent and the hosts can vary according to the requirements of the application. An example of such an environment is an intranet environment of a company or organisation and mobile agents are deployed within this intranet environment. The mobile agent may for example be required to migrate between service providers belonging to the same organisation. In this specific scenario, the mobile agent has reason to trust the hosts that it visits.

A *pre-defined* environment on the other hand consists of a number of pre-defined hosts that a mobile agent should visit. In such a case, the mobile agent owner simply specifies the itinerary of hosts to be visited before the mobile agent is deployed. The hosts indicated beforehand are not necessarily seen as trusted entities, but the mobile agent owner might have a better idea of where problems could have been encountered once the mobile agent returns home. Examples of this category are environments in which the service providers to be visited are determined beforehand, such as a flight booking between a choice of airlines (ex. only Virgin, British Airways or American Airlines).

An *open* environment is seen as the World Wide Web (WWW), in which the agent roams freely without its owner specifying a pre-determined itinerary. The mobile agents are able to autonomously migrate between hosts and make decisions in order to reach their goals. This type of environment is the most difficult to protect against and extensive security methods have to be incorporated.

Framework security levels

According to the criteria of a mobile agent security framework, as well as the analysis and discussions in the previous chapters regarding the countermeasures, frameworks, systems and applications, we propose the following six levels of security within the framework:

> *Basic closed*: The basic closed level is a trusted environment in which the mobile agent is deployed. This trusted environment is typically a local area network (such as an Intranet) within a specific organisation. The level of trust in this environment is high. The mobile agent system executing on the *basic closed level* will mainly be used for information conveying and retrieval with no computations taking place on the different hosts.
>
> *Extended closed*: This security level is a local area network that can possibly be extended by incorporating two or more Intranets. It is basically a trusted network of nodes and the mobile agent deployed at the *extended closed security level* is used not just for information searching and conveying, but also for computations. The level of trust is high.

Basic Restricted: Applications operating on the *basic restricted level* will make use of hosts on the Internet, where the hosts are pre-determined by the owner. Applications operating in the *basic restricted level* are mainly for information retrieval and conveying, but with no computations. The level of trust on this level is low.

Extended Restricted: As in the previous framework level, hosts to be visited in this type of framework are predefined. Additional to the information retrieval and conveying, mobile agents will also have no restrictions on the functions executed at the different hosts, which means that computations are allowed. As before, the level of trust is low.

Basic Open: In the *basic open framework level*, Internet hosts are included without the restriction of a predefined itinerary. However, at this level, mobile agents are only used for information conveying and retrieval with no computations on the different hosts. The level of trust is nil.

Extended Open: As in the previous framework level, the *extended open* framework level includes Internet hosts without the restriction of a predefined itinerary. Applications in this environment have no restriction on accessing and computational functions on the different hosts. Since the level of trust is extremely low, it is important that all components of the agent are protected.

It is necessary to distinguish between mobile agents that is tasked to convey and / or retrieve information and mobile agents whose ultimate goal include computations on the different hosts. Agents that fall in the latter category will need additional protection in terms of information required throughout their journey as well as the protection of the actual computational results. Figure 6.1 summarises the discussion on the different security levels. In the diagram the x-axis depicts the different types of execution environments, while the application categories are depicted along the y-axis.

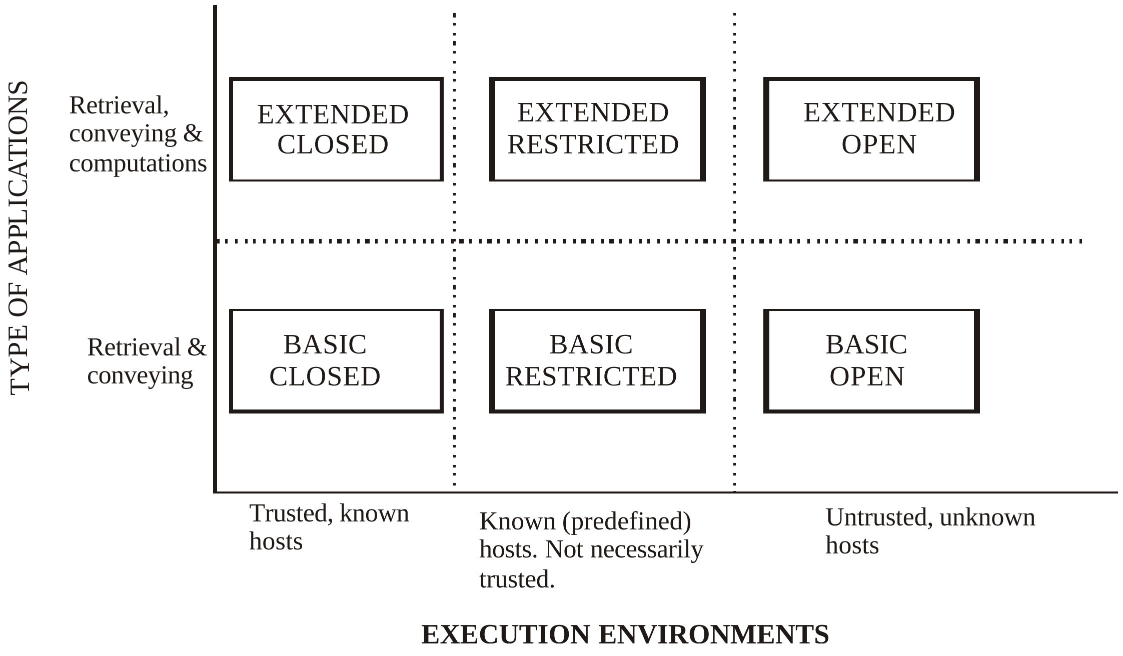

Figure 6.1: Proposed framework security levels

6.3 Countermeasures for Security Levels

As described in the previous section and depicted in Figure 6.1, the different *framework security levels* depend on the definition of specific application environments in which mobile agents are likely to be deployed. These application environments range from a highly trusted environment to an untrusted open environment. In the rest of this section, we consider the most appropriate countermeasures to be integrated into a particular environment in order to improve the security without risking performance. The suggested framework has a dynamic nature. Although the framework itself appears static in the number of solutions it seems to offer, the ability to use any countermeasure or combination of countermeasure for different security demands, lends a dynamic character to the proposed framework. This implies that although there may be various appropriate countermeasures available for a specific security level, only a selected few of these may be suitable for a particular application. This in turn depends on what degree of security the application is expecting from the framework.

Closed Security Level

The creation of a highly trusted environment for the *closed* security level, can be achieved by either hardware or software methods. Hardware methods involve the implementation of tamper resistant components on each host (see Wilhelm *et al.* (1998, 1999, 1999a, 2000) and Fünfrocken & Mattern (1999)). The main disadvantage of introducing specialised hardware components is the costs involved. This method does however provide a high level of computational trust for the mobile agent and protects the agent against attacks when executing at a remote host. These types of countermeasures are recommended within environments that require a high level of security, where the protection of the agents verifies the high implementation costs.

Software methods for the creation of a trusted execution environment can also be achieved by setting up a trusted set of network nodes. This can be done for example by using encryption and authentication techniques (Sander & Tschudin, 1998). Yee (1997) also introduced a number of alternative methods to achieve trust in a mobile agent system, namely *blind* trust, trust based on *reputation*, trust based on *control and punishment*, or trust based on *policy enforcement* where an agent had a prior (contractual) relationship with the host.

In a specific local area network such as an Intranet, trust can be achieved by being part of a certain corporation or company. In this case the mobile agent is deployed amongst entities with the same goal in mind and possible malicious intent by a remote host is diminished. In this type of environment introduction of additional hardware or software methods might not be necessary, because of the high level of trust that is already present.

A trusted third party used as a certification authority as well as for the distribution and management of keys, can be incorporated into the *basic closed.* This trusted entity could also be extended to the *extended closed* level with the trusted third party being used for secure computations. The transferring of data and computation results from each remote host to the local host is also incorporated into the *closed* levels as a method to provide for a trusted execution environment. In determining the countermeasures for the different levels of security within the framework, distinction is made between measures that can *detect* malicious behaviour and those that *prevent* malicious behaviour. Countermeasures included in the *closed* security levels provide preventative measures for protecting the agent. The list of techniques incorporated within the *closed* security level is depicted in Table 6.1.

Table 6.1: Countermeasures for closed security level

Basic closed ***Prevention methods:*** Trusted	execution environment Tamper resistant

hardware Trusted third party - (certification authority) **Extended closed** ***Prevention methods:*** Trusted execution environment Tamper resistant hardware Trusted third party - (certification authority) Trusted third party - (computations) Phone home

The creation of the *basic closed* level as well as the *extended closed* level is divided into two different ways, namely by using trusted hardware components or by incorporating software techniques. The steps for the creation of a trusted execution environment by using tamper resistant hardware are shown in Figure 6.2, and involves the following steps:

(1) Upon creation of the mobile agent, its itinerary is defined which only include remote hosts that have a tamper resistant hardware device installed. This step also includes the creation of a security policy for the mobile agent that specifies its security definitions and requirements.

(2) Once the itinerary of the mobile agent is defined, the public key of the first remote host is requested from the tamper resistant module located on the specified remote

host.

(3) Using the public key of *Remote Host A*, the mobile agent is encrypted and transferred to the tamper resistant hardware module located on *Remote Host A*.

(4) *Remote Host A* decrypts the received mobile agent by using its private key and subsequently executes the agent.

This process is repeated at every remote host on the itinerary encrypting the mobile agent by using the public key of the next host, until the itinerary is exhausted and the agent returns to the local host.

Figure 6.2: Creation of closed level by using tamper resistant hardware

The use of software methods to create a trusted environment entails the creation of security policies as well as certification and authentication techniques. Figure 6.3 depicts the communication sessions that take place between the local host and *Remote Host A*.

(1) Upon creation of the mobile agent, a security policy is defined for the mobile agent outlining its security requirements. The list of hosts to be visited is scheduled within the itinerary of the agent. A public / private key pair is created for every remote host either by using a certification authority or by the remote hosts themselves. Figure 6-3 outlines the key creation process as done by the remote hosts.

(2) The mobile agent authenticates itself by creating a digital certificate and sends the certificate to *Remote Host* A.

(3) Upon verification of the agent, *Remote Host A* responds by sending a digital certificate authenticating itself, back to the mobile agent.

(4) The local host encrypts the mobile agent and sends it to *Remote host A*.

(5) *Remote Host A* decrypts the agent and continues with the execution of the agent.

This process continues with every remote host listed on the itinerary required to authenticate itself, as well as to encrypts / decrypts the mobile agent.

Figure 6.3: Creation of closed level by using software methods

The protection of the computational results on the *extended closed* level by means of either a trusted computing entity or by conveying the results back to the local host requires the following steps (as outlined in Figure 6.4):

(1) The mobile agent migrates from the local host to *Remote Host A*, where it is executed.

(2) Secure computations are completed at the trusted entity and the results are conveyed back to the agent.

(3) Results obtained at *Remote Host A* are conveyed back to the local host.

(4) The mobile agent migrates to *Remote Host B.*

(5) Secure computations are completed at the trusted entity and the results are attached to the agent.

(6) Results obtained at *Remote Host B* are conveyed back to the local host.

(7) The mobile agent migrates to *Remote Host C,* and the same process for secure computations is followed.

(8) The mobile agent migrates from *Remote Host N* back to the local host.

Figure 6.4: Trusted computing base and phone home

Restricted security level

The *restricted* security levels of the framework operate in a predefined environment. The requirement for this environment is that the itinerary of the mobile agent be pre-determined by its owner (or creator) before migration to the first remote host. Although the hosts are known beforehand, the restricted security level is not seen as a trusted environment and a trusted environment is not created, as was the case in the closed levels.

The list of countermeasures that can assist in the creation of the *restricted* security level is shown in Table 6.2. These countermeasures are divided into methods that can be used for detection purposes and methods that can be incorporated for the prevention of mobile agent threats.

Table 6.2: Available countermeasures for restricted level

Basic restricted ***Detection methods:*** Detection objects Digital signatures Itinerary recording Path histories Proof carrying code Watermarking ***Prevention methods:*** Anonymous itinerary Code obfuscation Code transformations Computing with encrypted functions Environmental key generation Mobile agent system Partial results encapsulation Time sensitive agents **Extended restricted** ***Detection methods:*** Detection objects Digital signatures Execution tracing Itinerary recording Path histories Proof carrying code Reference states State appraisal Watermarking ***Prevention methods:*** Anonymous itinerary Code obfuscation Code transformations Computing with encrypted functions Environmental key generation Mobile agent system Partial results encapsulation Time sensitive agents

The analysis of the measures indicated in Table 6.2 according to the requirements of a mobile agent security framework (see 5.5.1) will assist us in determining methods to be included within the framework. Countermeasures to be included in the *restricted level* of the framework are subsequently discussed.

Detection objects (McDermott & Goldschlag, 1996) provide a way for detecting changes within the code of the mobile agent. It requires the creator of the agent to insert the dummy values and also implies additional computational overhead at the local host when determining if the inserted values have been modified. The limitations of detection methods are the increase in computational costs, and because it is deemed necessary that the creator of the mobile agent is provided with a choice of detection methods, it is included in both the *basic restricted* as well as the *extended restricted* levels.

The encryption and authentication of the agent is present as a feature in most of the mobile agent systems and from the analysis of these systems as well as the frameworks and applications (Chapter 5), is thus seen as a primary requirement for all levels of the mobile agent security framework.

As indicated in the Chapter 4 & Chapter 5, techniques that make use of cooperating agents in order to protect the mobile agent, have limiting effects on the autonomy and mobility properties of the mobile agent. Although the *restricted* security levels make use of a predefined itinerary and thus inhibits the autonomy and mobility of the mobile agent in a certain sense, *Itinerary recording with replication and voting* (Minsky *et al.*, 1996) requires the establishment of a set of cooperating agents and is in violation of all the requirements of the security framework and are thus not included within the framework.

The *Path Histories* (Ordille, 1996) countermeasure is included within the *basic restricted* as well as the *extended restricted* levels for providing a trail of hosts visited. The downside once again of a detection method is the additional computational costs in validating the histories.

Proof-carrying code (Necula & Lee, 1998) requires the existence of a proof-validator within the domain in which the mobile agent will be deployed, as well as additional communication sessions between the remote hosts and the mentioned validator. This method is thus not included within the framework. *Watermarking* techniques (Jalali *et al.*, 2000) provide a way of assessing if the mobile agent is valid and require the creator of the agent to create and insert the watermark, which implies additional overhead costs. Watermarking techniques need to be available on the *basic restricted* as well as the *extended restricted* levels of the framework.

Prevention mechanisms include the use of *anonymous itinerary* (Westhoff *et al.*, 1999) in order to provide anonymity with regards to the remote hosts to be visited. The prevention of attacks aimed at the code of the mobile agent can be achieved by combining *code obfuscation* (Hohl, 1997, 1998) and *time techniques* (Grimley & Monroe, 1999) as well as *code transformations* (An *et al*., 2002) and *time techniques* (Grimley & Monroe, 1999). *Computing with encrypted functions* (Sander & Tschudin, 1998) does provide protection for the agent and is included within the *basic restricted* and *extended restricted* levels of the framework.

Environmental key generation (Riordan & Schneier, 1998) is a method of encrypting the agent and providing means of key generation for decryption purposes. The use of a *mobile agent system* (Yee, 1997) restricts the autonomy and mobility property of the mobile agent as well as demanding a huge increase in computational costs and is not included within the framework. *Partial result encapsulation* (Chess *et al*., 1995; Jansen, 2000; Yee, 1997) is included in the framework on both the *basic restricted* and the *extended restricted* levels.

A summary of the countermeasures included in the *basic restricted* security level is depicted in Table 6.3.

Table 6.3: Countermeasures for basic restricted security level

Basic Restricted	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Auditing* Path histories *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques *Keeping the agent secret:* Computing with encrypted functions *Protecting itinerary:* Anonymous itinerary *Auditing:* Partial result encapsulation

On the *extended security* level protection needs to be provided for the state of the mobile agent. Detection methods for protecting state information include *cryptographic traces* (Vigna, 1998), *reference states* (Hohl, 2000) and *state appraisal* (Farmer *et al.*, 1996). *Cryptographic traces* (Vigna, 1998) induces a huge amount of additional communication sessions as well as computational costs, which implies restricting two of the requirements of a security framework and is thus not included within the framework. *State appraisal* (Farmer *et al.*, 1996) and *reference states* (Hohl, 2000) are included in the *extended restricted* level.

Table 6.4 provides a summary of the measures included in the *extended restricted* security level.

Table 6.4: Countermeasures for extended restricted level

Extended Restricted	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Audit trail:* Path histories *State protection:* Reference states State appraisal *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques Environmental key generation *Keeping the agent secret:* Computing with encrypted functions *Protecting itinerary:* Anonymous itinerary *Auditing:* Partial result encapsulation

Open Security Level

Any remote host can form part of the list of hosts visited by a mobile agent in the *open* security levels. The open environment is for example the Internet and the mobile agent has the ability to migrate between any of the hosts available. No assumptions are made regarding the level of trust of the remote host before migration.

Table 6.5 provides the lists of current appropriate countermeasures for the *open* security levels.

Table 6.5: Available countermeasures for open level

Basic open	***Detection methods:*** Detection objects Digital signatures Itinerary recording Mutual itinerary recording Path histories Proof carrying code Watermarking ***Prevention methods:*** Code obfuscation Code transformations Computing with encrypted functions Environmental key generation Partial results encapsulation Time sensitive agents
Extended open	***Detection methods:*** Detection objects Digital signatures Execution tracing Itinerary recording Mutual itinerary recording Path histories Proof carrying code Reference states State appraisal Watermarking ***Prevention methods:*** Code obfuscation Code transformations Computing with encrypted functions Environmental key generation Partial results encapsulation Time sensitive agents

The only countermeasure not discussed on the previous levels is *mutual itinerary recording* (Roth, 1998). This method inhibits the autonomy and mobility of the agent (which is essential in an open environment), it is not included in the framework. Table 6.6 provides the countermeasures included within the *basic open* security level.

Table 6.6: Countermeasures for basic open level

Basic Open	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Auditing* Path histories *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques *Keeping the agent secret:* Computing with encrypted functions *Auditing:* Partial result encapsulation

The countermeasures included within the *extended open* level have been reviewed as part of the discussions surrounding the *restricted* security levels. Table 6.7 lists the countermeasures for the *extended open* level.

Table 6.7: Countermeasures for extended open level

Extended Open	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Audit trail:* Path histories *State protection:* Reference states State appraisal *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques *Keeping the agent secret:* Computing with encrypted functions *Auditing:* Partial result encapsulation

The different countermeasures listed within the different security levels were discussed in detail in Chapter 3, while the shortcomings resulting from implementing and testing of the measures are outlined in Chapter 8.

6.4 Conclusion

In the previous chapters we have studied and discussed the available literature on mobile agent security with specific reference to the malicious host problem. Through this research we were able to identify the most salient characteristics in available security frameworks and mobile agent systems, but also isolate the drawbacks, which up to this point, still leaves a mobile agent vulnerable for malicious hosts attacks. In Chapter 6, the accumulated background knowledge and arguments were used to describe a dynamic mobile agent security framework that is based on the definition of multiple security levels, depending on the type of deployment environment as well as type of application. Under these conditions, it is possible to assess the security requirements of a particular mobile agent system and to assemble a custom-made security plan for the particular mobile agent system that would not interfere with the system's performance or make the deployment of such a system expensive. In the next chapter, we describe the implementation and analysis of our multi-level security framework.

CHAPTER 7

IMPLEMENTATION

7.1 Introduction

In Chapter 5 the environment in which a mobile agent will or can be deployed, as well as the type of mobile agent application have been evaluated against a set of criteria that have been developed earlier for efficient mobile agent systems. This evaluation led to the definitions of different security levels, which form the basis of the proposed mobile agent security framework. In the research effort pertaining to this thesis, the proposed framework has been prototyped and tested against current and proposed countermeasures, systems and implementations (Chapter 6). The implementation specifics are described in this chapter, that include the prototype being implemented and through experimentation tested for different scenarios in order to ascertain the practicality of the proposed framework. The line of reasoning is then continued with an analysis and interpretation of the testing results.

7.2 Summary of the Proposed Mobile Agent Security Framework

In the previous chapter a mobile agent security framework has been propositioned based on eight requirements that have been established to enable secure, yet effective mobile agent systems. In summary, the following requirements are imposed on a security framework for mobile agent systems, which render a secure yet efficient and goal driven system:

1. The framework must provide different levels of security, depending on the *type of implementation environment* in which the mobile agent would be deployed.
2. The framework must incorporate different levels of security depending on the *type of application* and *agent objectives*.
3. The framework must maintain and not hamper the autonomy and mobility factor of the agent.
4. *Additional security implementations* on the remote hosts (and the system as a whole) must be kept to the minimum, to reduce cost and time. This includes both additional hardware and software requirements.

5. The number of *communication sessions* between the remote hosts (and between remote hosts and other entities) must be minimised. There also needs to be no permanent connection between the agent and the local host.
6. *Computational cost* of implementing countermeasures and maintenance thereof must be as low as possible.
7. The cost of implementation should be affordable or at least minimised. The *financial costs* of implementing countermeasures need to be in direct relation with the degree of security required.
8. The host must possess intrinsic mechanisms to support the security requirements of the agent. This implies the *provision and integration of additional security functions and services,* according to the needs of the application, and hence the agent.

Based on these requirements, available countermeasures as well as available mobile agents systems (and frameworks) or proposed mobile agent systems / frameworks were evaluated. This resulted in the proposal of six levels of security for mobile agents and mobile agent systems. These levels are summarised as follows:

1. *Basic closed:* a trusted execution environment that allows information retrieval and conveying without computations.
2. *Extended closed:* a trusted execution environment that allows information retrieval and conveying with computations.
3. *Basic restricted:* a potentially untrusted, but predefined execution environment that allows information retrieval and conveying without computations.
4. *Extended restricted:* a potentially untrusted, but predefined execution environment that allows information retrieval and conveying with computations.
5. *Basic open:* an untrusted and also unknown execution environment that allows information retrieval and conveying without computations.
6. *Extended open:* an untrusted and also unknown execution environment that allows information retrieval and conveying with computations.

These predefined execution environments enable the careful evaluation of available countermeasures and selection of applicable measures based on the specific objectives of the mobile agent (system) as well as the anticipated execution environment.

The practical implication of these different levels of security is that it becomes possible to identify appropriate security countermeasures for particular deployment environments. Table 7.1 summarises the options that are typically available to a mobile agent system programmer when designing a secure mobile agent system that is protected against malicious hosts attacks.

Table 7.1: Countermeasures for security levels

Basic Open	**Extended Open**
Detection methods: Detection objects Watermarking Digital signatures Path histories *Prevention methods:* Code obfuscation Code transformation Time techniques Computing with encrypted functions Partial result encapsulation	*Detection methods:* Detection objects Watermarking Digital signatures Path histories Reference states State appraisal *Prevention methods:* Code obfuscation Code transformation Time techniques Computing with encrypted functions Partial result encapsulation
Basic Restricted	**Extended Restricted**
Detection methods: Detection objects Watermarking Digital signatures Path histories *Prevention methods:* Code obfuscation Code transformation Time techniques Anonymous itinerary Computing with encrypted functions Partial result encapsulation	*Detection methods:* Detection objects Watermarking Digital signatures Path histories Reference states State appraisal *Prevention methods:* Code obfuscation Code transformation Time techniques Environmental key generation Anonymous itinerary Computing with encrypted functions Partial result encapsulation
Basic Closed	**Extended Closed**
Prevention methods: Trusted execution environment Tamper resistant hardware Trusted third party - (certification authority)	*Prevention methods:* Trusted execution environment Tamper resistant hardware Trusted third party - (certification authority) Trusted third party - (computations) Phone home

7.3 Implementation Specifications

To test whether the mobile agent security framework as proposed in the previous chapter can be implemented, we could either have constructed a new mobile agent development platform or use an existing mobile agent development system that allows for the creation and management of agents. We chose the latter, as the research question pertaining to this study is focussed on the security aspects and not to improve current creation, control or migration capabilities of mobile agents. To test our propositions, we searched for an existing mobile agent development system that could form the basis from where our framework could be implemented and tested. Our system requirements for such a basis included the following:

As a basis, the selected system had to provide the infrastructure for the initialisation and controlling of mobile agents

As a basis, the selected system also had to provide for, or facilitate an execution environment with capabilities such as migration and communication.

The selected system's code had to be available for modification so that specific security measures could be implemented as desired.

7.3.1 Primary software environment

As discussed in Chapter 4, a number of mobile agent systems are currently available to aid the development of mobile agent applications. An analysis of the mentioned systems against our specific requirements has lead to the choosing of the *Aglets* software development kit, developed by IBM Japan. Although *Aglets* are not one of the newest mobile agent systems around, the choice of *Aglets* was further supported by (1) its availability as open source on the Internet and (2) the number of applications that are already developed by using *Aglets.* The latter provided us with a certain level of persistence and confidence in the technology, especially at this level, where we deemed it unnecessary to waste coding time on already existing technology. The first version of *Aglets* was released in 1996, with the latest version (v. 2) being available as open source. The *Aglets* system is written in Java and requires the Java virtual machine for implementation. The Aglet API is a set of Java classes and interfaces that allows for the creation and management of mobile agents. Network communication is done through the *Aglet Transfer Protocol (ATP).*

With the development kit of the Aglet system, a graphical user interface named *Tahiti* is also used to simplify the management and control of aglets that are created within a specified environment. *Tahiti* contains a network daemon that listens for incoming aglets as well as a security manager that include measures for protecting the host. Take note however, that the security manager is only concerned with the protection of the host and not the aglet.

The *Aglet* model consists of four basic elements, namely an *aglet*, which is a mobile agent, or as described in literature, a mobile Java object; a *proxy* representing the aglet; a *context*, which is the aglet's workplace and an *identifier,* which is globally unique and bound to each aglet (Lange & Oshima, 1998). Lange & Oshima (1998) describe the fundamental operations of an aglet as:

creation - occurring within an aglet *context*;
cloning - producing a copy of an aglet;
dispatching - *moving* an aglet between aglet contexts;
retraction - *removing* an aglet from the current aglet context;
activation / deactivation - temporary halt or restart of the aglet;
disposal - removing an aglet from the current context

The *Aglet* programming model is event-based where customised listeners are employed to catch events within the life cycle of the aglet and subsequently allow the developer to code appropriate actions. There are three different listeners defined in this context, namely

a *clone listener* - listening for *cloning* events;
a *mobility listener* - listening for *dispatch, retract* or arrival *messages* of an aglet;
a *persistence listener* - listening for *activation* or *deactivation* messages for a an aglet in order to facilitate specific actions based on the message it receives

The *Aglet* communication model is implemented by using message passing, which allows for the creation and exchange of messages in flexible ways. A proxy aglet is used to protect aglets against other aglets (not hosts). For this process, a proxy is initiated upon creation of an aglet. The proxy provides a way of accessing the aglet. Any aglet that instigates communication with another aglet first has to access the proxy of the aglet and then has to interact via the proxy.

The life cycle of an aglet is depicted in Figure 7.1 and shows the fundamental operations available on aglets as discussed above (Lange & Oshima, 1998).

Figure 7.1: Aglet life cycle

7.3.2 Experimentation environment and equipment

In our research effort, we set to create a simple implementation environment where the Aglet Software Development Kit (ASDK) could be deployed. The implementation environment consists of three hosts forming a network. Each host in our experiments has the following configuration: Pentium II, 200MHz processor with 128 MB RAM; Windows 98 operating system with Java Software Development Kit (version 1.4), Java Virtual Machine (version 2); Aglets Software Development Kit (version 2.1) installed. We further use the Aglet Transfer Protocol (version 1.2) and the Tahiti aglets server (version 1.0b5).

The implementation and testing of the proposed framework set to test whether existing countermeasures can be used to provide a dynamic set of measures in order to proof the viability of the framework. The purpose of the implementation is not to define new measures but to incorporate existing methods by making use of the Aglets Software Development Kit as a platform.

7.4 Aglet Security Model

In this section we describe the available security features of ASDK. The *Aglet* security model as described by Lange & Oshima (1998), is based on the definition of security policies as well as a description of how and where these policies are enforced. Furthermore, the model defines several *principals* (important entities) that can be authenticated to support the intended security. The primary principals are those in the *aglet system*, the *aglet context* and the *network domain*. The principals in the aglet system include the *aglet* itself, the *aglet manufacturer* and the *aglet owner,* while those in the aglet *context* are the *context* itself, the *context manufacturer* and the *context owner*. Finally, the principals in a network domain are the hosts.

The *Aglet* security model provides methods for the protection of the host and also protection of the aglet against other aglets. No capabilities for the aglet to protect itself against a malicious host are incorporated into the model. Even though our interest lies specifically in the protection against malicious hosts, we find it necessary to discuss the *Aglet* security model in order to specify the security policies that are required for the creation and distribution of aglets.

Permissions within the *Aglet* security model define the capabilities of executing aglets by setting access restrictions and also limitations on resource consumption. Permission is defined as a resource and the abstract syntax of permissions is based on the JDK policy file definition. The permission structure for aglets include the following types of permissions:

file permissions, controlling access to the local file system;
network permissions, controlling access to the network;
window system permissions, for controlling the opening of windows;
context permission, for granting permission to services provided by the context;
aglet permission, for controlling methods provided by an individual aglet.

Besides setting *permissions* for a particular aglet that is intended to keep a tight rein on the aglet, it is also possible to define a particular level of protection for the aglet. Even though *protections* are not defined in a way that they can safeguard the agent from a malicious host, they facilitate a minor degree of safety for the aglet. For example, a *protection* can be set to specify that only the owner of a specific aglet can dispose of the aglet.

Another component of the Aglet security is the definition of *security policies* by *authorities*. The authorities are typically the *aglet owner,* the *context owner* and the *network domain owner.* The security policies are sets of rules containing the protection level within the permission structure. For this purpose, a security policy file is defined and presented at each host that an aglet is to visit.

When setting up a security policy file, all permissions are initially allowed on the different hosts. Figure 7.2 shows an extract of a sample policy file (the complete file can be viewed in Addendum A).

```
 grant codeBase "atp://*:*/" {   permission java.io.FilePermission
"codebase", "read";   permission java.io.FilePermission "codebase",
"read, write,
execute";   permission java.util.PropertyPermission "browser",
"read";   permission java.util.PropertyPermission "java.rmi.*",
"read";   permission com.ibm.aglets.security.ContextPermission "*",
"create,receive,retract";   protection
com.ibm.aglet.security.AgletProtection "*",
"dispatch,dispose,deactivate,activate,clone,retract";   protection
com.ibm.aglet.security.MessageProtection "*", "*"; };
```

Figure 7.2: Aglet security policy file

7.5 Implementation of Framework

To implement and test our proposed framework (as detailed in Chapter 6), it is necessary to design and generate aglets for the different application levels of the framework (namely the three *basic* and the three *extended* levels).

Applications in the *basic* security category are responsible for retrieving information from the different remote hosts, as well as conveying results either between the different hosts or between the remote hosts and the local host of the owner. For the purpose of this particular implementation an aglet (named RetrievalAglet) is created. RetrievalAglet is tasked to collect the prices of specific goods at various host sites and once information at a particular host has been collected, the retrieved information is saved in a file and attached to the agent as part of its aggregated data. Upon its return to the local host, the data is viewed and printed by the owner / creator of the aglet.

In this specific implementation, the required information is contained in a file located at every host. The structure of RetrievalAglet is supplied in Figure 7.3, with the full source code listed in Addendum B. As illustrated in the outline given in Figure 7.3, the RetrievalAglet class contains three methods namely, *onCreation* that is initiated when an object of class RetrievalAglet is created, *run* that contains the steps to be completed by the agent on each remote host (in this case the retrieval of prices for specific goods) and *NextDestination()* that provides the steps for transferring the agent to the next remote host.

```
 public class RetrievalAglet extends Aglet {   //specifications of requested information    public
void onCreation(Object init) {       addMobilityListener(    new MobilityAdapter() {             public
void onArrival(MobilityEvent b) {    }    }  ); }   public void run() {   try {    //retrieve information
//add retrieved information to aglet    }    } catch (Exception e) {    System.out.println
(e.getMessage());   }  }     void NextDestination() {   try {    //migrate to next remote host   }   }
catch (Exception e) {   System.out.println(e.getMessage()); }  } }
```

Figure 7.3: RetrievalAglet

The implementation and testing in the *extended* category of the framework requires an application that besides the retrieval and transport of information also has the added ability of performing some computations (to different degrees) on the different hosts. To demonstrate this an aglet (named ComputationAglet) is created that is based on the concept of RetrievalAglet, with the added ability of not only commanding the price of goods available on each host, but also to make a computation (in order to find the lowest price between the hosts). The lowest price as well as the URL of the host at which it is obtained is saved in a file, attached to the mobile agent and printed at the local host.

Figure 7.4 lists the structure of ComputationAglet, of which the full source code is available in Addendum C. The methods of the ComputationAglet are similar to those of the RetrievalAglet. However, take note of the additional code required in the *run* method, which is necessary for the computational facilities of this class.

```
 public class ComputationAglet extends Aglet {                   //specifications of requested
information    public void onCreation(Object init) {        addMobilityListener(      new
MobilityAdapter() {              public void onArrival(MobilityEvent b) {     }     }    );   }   public void
run() {   try {    //Retrieve information                                    //Determine lowest price
//If lowest price, require aglet context and add to aglet         } catch (Exception e) {
System.out.println(e.getMessage());   }  }   void NextDestination() {   try {    //migrate to next
remote host   }catch (Exception e) {   //Failed to initialize next destination   System.out.println
(e.getMessage()); }  } }
```

Figure 7.4: ComputationAglet

7.5.1 Basic Closed Level

As portrayed in Chapter 6, the countermeasures incorporated into the *basic closed* security level are methods to create and sustain a trusted environment. To remind the reader, the measures are repeated in Table 7.2. Below we discuss the implementation issues of these prevention methods for the *basic closed* environment.

Table 7.2: Countermeasures for basic closed level

Basic closed	***Prevention methods:*** Tamper resistant hardware Trusted execution environment Trusted third party - (certification authority)

The implementation of tamper resistant hardware requires additional hardware components to be installed on the different remote hosts. As this implies additional financial costs, we are not implementing trusted hardware modules for testing purposes. Tamper resistant hardware components are however available commercially, examples are *nShield*, which is a secure server peripheral for the management of cryptographic keys and the protection of sensitive applications, as well as *utimaco*, that enables the use of trusted hardware platforms for secure mobile computing. These devices can be implemented directly on the remote hosts and thus require no additional communication sessions between the host and the trusted hardware module. The disadvantage of this type of countermeasure (according to the requirements of a mobile agent system framework) is the additional requirement in terms of the installation of specialised hardware, which in turn has financial implications.

The creation of a *trusted execution environment* (Sander & Tschudin, 1998) can also be achieved by using software techniques to provide a reliable environment. This method incorporates encryption and authentication techniques by requiring the remote hosts and the mobile agent to be authenticated before migration to the different entities, as well as the encryption of the mobile agent between hosts.

The first step in creating a trusted environment for RetrievalAglet to be dispatched in (according to the specifications provided in Chapter 6) is that the local host needs to digitally sign and encrypt the aglet.

A number of different implementation methods are available for digitally signing classes and objects. Java provides the ability of digitally signing code as well as the creation of private/public key pairs, with the use of the *Java Cryptography Architecture (JCA).* The JCA framework contains the *Digital Signature Algorithm (DSA),* which (by default) is used for the creation and verification of digital signatures. DSA is a public key algorithm where the secret key operates on the message hash generated by the *Secure Hash Algorithm* (*SHA-1)*. For verification of the signature, the hash of the message is re-computed, the public-key used to decrypt the signature and the results compared.

With Aglets being written on Java, it is possible to digitally sign the aglet class by using the Java Cryptographic Architecture. This process consists of creating a Java Archive File (JAR) containing the aglet, creating a private/public key pair and the creation of a certificate, which (by default) is valid for a period of 90 days. The created JAR file is subsequently signed with the generated private key; the generated certificate is attached and both are incorporated and sent as a signed JAR file to the first remote host. Figure 7.5 illustrates how a certificate can be generated in Java. The methods listed in Figure 7.5 are *jar* that creates an archive file by using the RetrievalAglet class, *keytool -genkey* that generates a private / public key pair which is saved within a keystore file, *jarsigner* creates a digital certificate of the created archive file by making use of the generated private key and *keytool -export* that exports the resulted certificate as well as the keys used in generating the certificate.

```
 //creation of JAR file            jar cvf RetrievalAglet.jar RetrievalAglet.class //creation of keys
keytool -genkey -alias localhostkey -keypass privpass -keystore                        C:\keys
\keyfile -storepass keyfilepass  //signing the file jarsigner - keystore C:\keys\keyfile -signedjar
RetrievalSigned.jar                RetrievalAglet.jar localhostkey //exporting the keys keytool -
export -keystore C:\keys\keyfile -alias localhost -file              SignedRetriev.cer
```

Figure 7.5: Signing RetrievalAglet

After receiving the signed aglet, the remote host needs to validate the certificate by using the generated public key. The validation steps are outlined in Figure 7.6 and consist of the method *keytool -import* required for importing the certificate as well as the public key for verification purposes.

```
 //import the certificate            keytool -import -alias remotehostkey -file SignedRetriev.cer -
keystore keyfile
```

Figure 7.6: Verifying signed RetrievalAglet

Another method that can be used for creating digital signatures is the incorporation of the *Java Cryptographic Extension (JCE)* package, which is an extension of the Java language. JCE provides a number of cryptographic services. These services include the creation and validation of digital signatures, encryption implementations such as DES, Triple DES, and Blowfish, as well as key generators for generating keys appropriate for the different encryption algorithms.

The SignedObject class provided with the JCE extension is used to digitally sign RetrievalAglet. Figure 7.7 illustrates the creation as well as the verification of the signature. The process of signing the object consists of the generation and initialisation of a *KeyPairGenerator* object that specifies the algorithm to be used as well as the key-size and a source for randomness.

The *Signature* class provides the functionality of a cryptographic digital signature algorithm and an object of the *SignedObject* class is instantiated for creating the digital certificate and subsequently verifying the signature.

```
 //Signing object   KeyPairGenerator genKey = KeyPairGenerator.getInstance("DSA","SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");  genKey.initailize
(1024,random);  KeyPair getkeys = genKey.generateKeyPair();  PrivateKey private_key =
getkeys.getPrivate();  PublicKey public_key = getkeys.getPublic();  Signature algorithm =
Signature.getInstance("SHAwithDSA", "SUN");  algorithm.initSign(private_key);   SignedObject
signedaglet = new SignedObject(init, private_key, algorithm);    //Verifying signature   boolean
verify = signedobject.verify(public_key, algorithm);  signedaglet.getObject();
```

Figure 7.7: Digital signature with SignedObject class

A security manager class (SecMan) is available as an extension to the ASDK that provides encryption/decryption and keystore services for aglets. This class is based on the JCE and the digital signing of the aglet is done in the same way as specified in Figure 7.7. It also includes methods for public, private and session key generation and the object is signed with a SignObject method.

The authentication of the remote host can be incorporated for example, by creating a stationary aglet on the server to act as a security manager for verifying incoming agents. The procedure of creating such a digital signature uses the same procedures as specified for the authentication of an aglet.

The generation and verification of a digital certificate by using either of the methods specified above lead to an increase in computational costs. The additional requirements are available as part of either the Java or the Aglet package, with additional communication sessions only required if the keys sent are not a part of the serialized object. As the public key is used for verification of the generated certificates, no additional security threats are implied if the public key is migrated along with the aglet. The methods implemented for digital signatures require no additional financial cost in that the available packages within the creation environment have been used.

The encryption of RetrievalAglet can also be achieved by using a variety of encryption algorithms within certain cryptography packages. The JCE package can be used to provide encryption/decryption according to specified providers such as DES and Blowfish. Open source toolkits such as *OpenSSL* can also be used to incorporate cryptographic functions, such as encryption and digital certificates. *OpenSSL* implements *Secure Sockets Layer* (SSL v2/v3) and *Transport Layer Security* (TLS v1) network protocols. Figure 7.8 outlines an example of a command for encrypting the aglet by making use of the *OpenSSL,* with Base64 as the encryption algorithm.

```
openssl base64 -in RetrievalAglet.class -out EncryptedRetrieval.class
```

Figure 7.8: Encryption of RetrievalAglet

Another method for encrypting the aglet is by using the SealedObject class, provided as part of the JCE package. This class enables a programmer to create an object and protect its confidentiality with a cryptographic algorithm. Figure 7.9 lists the code for encrypting and decrypting RetrievalAglet by using the SealedObject class.

The listing consists of initialising a *Cipher* object that provides the functionality of a cryptographic cipher for encryption and decryption purposes, the generation of a secret key by using the *SecretKey* class, as well as the sealing of the resulted encrypted object by making use of the *SealedObject* class. The decryption of the object consists of initialising a *Cipher* object and decrypting the object by using the generated secret key.

```
 //Encrypting and sealing object  Cipher des_encrypt;  SecretKey key_for_des;  KeyGenerator
genKey = KeyGenerator.getInstance("DES");  Key_for_des = genKey.generateKey();
des_encrypt = Cipher.getInstance("DES");  des_encrypt.init
(Cipher.ENCRYPT_MODE,key_for_des);  SealedObject sealed = new SealedObject(init,
des_encrypt);  //Decrypting object  Cipher des_decrypt  des_decrypt.init
(Cipher.DECRYPT_MODE, key_for_des);  try {   sealed.getObject(des_decrypt);  } catch
(Exception e) {   System.out.println(e.getMessage();  }
```

Figure 7.9: Encryption with SealedObject class

As with the creation of a digital signature, the SecMan class can also be used. This security manager class also provides methods for key generation and the encryption / decryption of an aglet.

The process of encrypting and decrypting RetrievalAglet indicate an increase in computational costs. Upon revisiting the requirements for a mobile agent system framework, the implementation of a *trusted execution environment* in Aglets and Java, have additional requirements in terms of computational costs, and with the implementation being without a trusted third party, no additional communication sessions are required. No additional requirements have been incorporated due to existing toolkits and packages being used, which also relates to no financial implications.

A trusted third party can be used for the generation and management of keys and certificates (which are currently implemented as being on every remote host), which will lead to an increase in communication sessions within the framework. A number of such certification authorities are in place, (for example *Thawte* and *VeriSign*) which do have financial cost implications.

The implemented results for the basic closed security level, evaluated against the criteria of a mobile agent security framework are shown in Table 7.3. Cells within the table that contains a *yes* indicate that the specific countermeasure (listed on the different rows) do not meet the specified requirement as listed in the different columns. A *no* indicates that the specified requirement is met by the corresponding countermeasure. For example in the 3rd column, 3rd row, it is shown that tamper resistant hardware inhibits the autonomy and mobility of the mobile agent, while the 5th column, 4th row indicates that a trusted execution environment requires no additional communication sessions.

Table 7.3: Implementation results of basic closed level.

	Requirements				
Countermeasures	**Inhibits autonomy & mobility**	**Additional requirements**	**Additional communication sessions**	**Additional computational costs**	**Additional financial costs**
Tamper resistant hardware	Yes	Yes	No	No	Yes
Trusted execution environment	Yes	No	No	Yes	No
Trusted third party - certification authority	Yes	Yes	Yes	Yes	Yes

The creation and management of a trusted set of hosts is thus possible with the use of tamper resistant hardware as well as software techniques. The points of concern for software methods used in the creation of a trusted environment, is the distribution of keys between the different hosts. Although the a certification authority can be used for this purpose, the mobile agent still depends on the host for decryption and possible malicious behaviour can still occur.

7.5.2 Extended closed security level

The objective of the extended-closed level is to create a trusted environment for applications that require information retrieval as well as computations on every remote host, if such an environment does not exist. Table 7.4 lists the countermeasures incorporated for this purpose (as taken from Chapter 6).

Table 7.4: Countermeasures for extended closed level

Extended closed	***Prevention methods:*** Tamper resistant hardware Trusted execution environment Trusted third party - (certification authority) Trusted third party - (computations) Phone home

The use of tamper resistant hardware for the creation of a trusted environment is the same as discussed in the *basic closed* level. The digitally signing and encryption of ComputationAglet is done in the same manner as for RetrievalAglet, with the same methods being used.

Additional countermeasures listed on the extended closed security level are *Phone home* and the use of a *trusted entity for secure computations*. Phone home requires the sending of computational results (as computed at each remote host) directly to the local host before migration of the aglet to the next remote host. In order to achieve this, a class is created from which an aglet is instantiated that is sent to the local host, before every migration. Figure 7.10 lists the code for the *Phone-Home* class. The listing contains the creation of a *proxy* for the aglet (by using the *AgletProxy* class) for handling the communication with the local host. Upon completing the required computation at the remote host, a message (method *sendMessage)* is sent to the local host containing the results of the computation.

```
public class PhoneHomeClass extends Aglet {    File results = null;  AgletProxy proxy = null;
public void onCreation(Object init) {   dir = (File)((Object[])init)[0];   proxy = (AgletProxy)((Object
[])init)[1];      addMobilityListener(    new MobilityAdapter() {     public void onArrival
(MobilityEvent me) {      try {       proxy.sendMessage(new
Message("Result",results));      }catch (Exception e) {       dispose();      } } } ); } }
```

Figure 7.10: Phone home class

The instantiation of the servant aglet that is responsible for conveying the information back to the local host is done within ComputationAglet by making use of the methods listed in Figure 7.11. The *Phonehome* method includes the creation of an aglet (*createAglet* method) as well as a proxy for the created aglet (*AgletProxy* class) and the dispatching of the proxy to the local host (*dispatch* method). The *handleMessage* class is responsible for handling the results received from the remote host.

```
void Phonehome() {  try {  URL homeaddress = new URL("atp://RemoteC.tut");  File resultsfile
= new File("C:/data/resultsfile.dat");  AgletContext context = getAgletContext();  AgletProxy
thisProxy = getProxy();  Object[] init = new Object[] {directory, thisProxy};  AgletProxy proxy =
context.createAglet(getCodeBase(),"PhoneHomeClass", init);  proxy.dispatch(homeaddress);  }
catch (Exception e) {   System.out.println(e.getMessage());}  }   public boolean handleMessage
(Message msg) {  if(msg.sameKind("Results")) {   String[] list = (String[])msg.getArg();   for(int
i=0; i<list.length;)    System.out.println(i+": "+list[i++]);   return true;  } else   return false; }
```

Figure 7.11: Phone home method

As shown, implementation of the *Phone home* countermeasure is relatively straightforward, with only a message containing the data send to the local host and not the agent itself. However, it has the downside of increased communication sessions as well as the availability of communication lines between the local host and the different remote hosts.

The same method that is used for the *Phone home* implementation is also deployed to test the use of a *trusted host for computations*. In this case the local host is seen as a trusted host and the results obtained is sent to the local host where the computations are completed at the originator site of the agent. This method leads to an increase in computational costs for the local host as well as additional communication sessions.

According to the discussion and the implementation results, the analysis of the countermeasures in the extended closed level according to the requirements of a mobile agent security framework (as detailed in Chapter 6), is shown in Table 7.5.

Table 7.5: Implementation results of extended closed level

	Requirements				
Countermeasures	**Inhibits autonomy & mobility**	**Additional requirements**	**Additional communication sessions**	**Additional computational costs**	**Additional financial costs**
Tamper resistant hardware	Yes	Yes	No	No	Yes
Trusted execution environment	Yes	No	No	Yes	No
Trusted third party - certification authority	Yes	Yes	Yes	Yes	Yes
Phone Home	Yes	Yes	Yes	No	No
Trusted third party - computations	Yes	Yes	Yes	No	Yes

7.5.3 Basic restricted security level

The restricted security level consists of the mobile agent migrating according to a pre-defined itinerary. Although the hosts are known beforehand, the restricted security level is not seen as a trusted environment and a trusted environment is not created, as was the case in the closed levels. The detail of the basic restricted level (as determined in Chapter 6) is once again outlined in Table 7.6.

Table 7.6: Countermeasures for basic restricted level

Basic Restricted	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Auditing* Path histories *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques Environmental key generation *Keeping the agent secret:* Computing with encrypted functions *Protecting itinerary:* Anonymous itinerary *Auditing:* Partial result encapsulation

As explained in an earlier chapter, the use of *detection objects* as a method to detect illegal tampering of the mobile agent entails the insertions of dummy data items within the mobile agent code, upon creation of the agent. Once the mobile agent has returned to its home environment the *detection objects* are checked and verified if they have changed. If they are still intact, then the agent is assumed to be unmodified. As *detection objects* form a complex research field on their own, for the purpose of this research, *detection objects* have mainly been implemented in a database scenario and not as part of the source code or data of an aglet. The details of the creation and updating of such *detection objects* fall outside the scope of this thesis. It is however noted that *detection objects* can be used to successfully detect manipulations of the agent and the agent's data. A possible implementation can be for example the insertion of small parts of code or data items into the aglet at creation.

A number of *watermarking* tools are available for use in devising as well as verifying a watermark. A number of these tools also include *code obfuscation* and *code transformation* techniques in the process of creating a *watermark*. One such tool, namely *Sandmark* was developed by the University of Arizona and provides features for software *watermarking, tamper-proofing* and *code obfuscation* of Java programs. By using the *Sandmark* tool, a *watermark* was added to RetrievalAglet upon creation and verified at the subsequent remote hosts. The results indicated increases in computational costs as well as the watermarking tool being an additional requirement. The financial implications of using this particular tool were none due to the tool being non-commercial.

The creation and verification of a digital signature has been implemented on the closed security levels and is thus not shown again. It is however necessary to note that with the itinerary being set before migration of the agent, it is possible for the agent and the host to be digitally authenticated. The authentication of the hosts can also be done upon arrival of the agent at the remote host. If the current host is invalid then the aglet is disposed. The code segment for authentication of the host is listed in Figure 7.12, and contains the *AuthenticateHost* method that obtains the context in which the aglet has migrated to, as well as the remote host listed in the itinerary of the aglet. If the obtained two addresses do not correspond the aglet is disposed.

```
 void AuthenticateHost() {   URL getCurrentHostURL;   AgletContext CurrentContext =
getAgletContext();   getCurrentHostURL = CurrentContext.getHostingURL();   if (!
(destination.equals(getCurrentHostURL)))   {    dispose();   }  }
```

Figure 7.12: Authentication of host

The creation of a *path history* pertains the signing of the itinerary in order to ensure that the aglet migrated to the remote host as specified on the itinerary of the aglet. This countermeasure is implemented by forcing the current remote host to add its context to the itinerary and digitally sign the URL. Upon arrival at the next remote host, the signature is verified in order to check for inconsistencies. The implementation code for creating path histories is listed in Addendum D.

The creation and verification of digital signatures is resource extensive, with increases in computational costs. No additional requirements are needed for the implementation thereof because an extension to the Java framework is used.

Code obfuscation and *code transformation* techniques and programs are widely available as both commercial products and open source products. These programs make use of a number of techniques in order to scramble the code into an illogical format. Examples of such programs are *Retroguard*, *Smokescreen* and *Sandmark*. As it is beyond the scope of this thesis to devise methods for *code obfuscation* and *code transformations*, existing programs were used and modified in order to test its ability to be incorporated into the framework. The *Sandmark* tool has once again been used for introducing the *code* *transformation* and *code obfuscation* to the aglet.

As *code obfuscation* and *code transformation* techniques are optimised with the inclusion of *time* techniques, our aglet was only allowed a certain amount of time to complete its tasks. Upon creation of an aglet, the system time was added to the *AgletInfo* class. By using the creation time (Figure 7.13), constraints can be added to an aglet in order to retract or dispose the aglet once the time has expired. The creation time of the aglet is retrieved (in the *OnCreation* method) and can be measured against the system time of the remote host in order to determine the current existence time of the aglet.

```
 public void onCreation (Object init) {  try {  AgletProxy proxy = getProxy();  AgletInfo info =
proxy.getAgletInfo();  long time createTime = info.getCreationTime(); }
```

Figure 7.13: Time sensitive aglet

The results that were obtained implied additional computational costs and also added requirements in terms of the required software.

No implementation of computations with *encrypted functions* as defined by Wilhelm et al. (1999) could be found. It seems that encrypted functions offer a mathematically sound, but quite complex method to protect against certain aspects of the malicious host problem. Furthermore, it seems that the complexity of the proposition is steep and as a result it hinders the method’s implement-ability. Further investigation into this specific countermeasure is beyond the scope of this thesis.

Environmental key generation relies on the encryption of the aglet. Decryption is only done once the slave aglet has retrieved some environmental data from the remote host. At this stage, it becomes possible to allow the decryption of the key and subsequently the aglet code. The encryption and decryption possibilities for an aglet have been discussed earlier and are not covered again. The code for the creation of a slave agent that determines the environmental data is listed in Figure 7.14. The *onArrival* method lists the code whereby the specified environment variable is obtained and the aglet is subsequently decrypted if the required state of the environment variable has been reached.

```
 public class Environmental extends Aglet {    File Keyfile=null;  AgletProxy proxy = null;
public void onCreation(Object init) {   Keyfile = (File)((Object[])init);   proxy = (AgletProxy)
((Object[])init);      addMobilityListener(    new MobilityAdapter() {     public void onArrival
(MobilityEvent me) {                   getEnvironmentvar();
if (true)                                                          decrypt();       dispose();      }
}   }   );  } }
```

Figure 7.14: Environmental key generation

Partial results encapsulation requires the retrieved data to be encrypted at each host. The local host of the agent then decrypts the layers of encrypted data once the agent has returned. The source code for *encapsulating partial results* is available in Addendum E.

The encryption and decryption of the aggregated data shows an increase in computational costs, with no additional requirements in terms of tools and software.

The encryption of the itinerary of the aglet in order to hide the destinations is a possible countermeasure that can be implemented. The implementation of this countermeasure is done in the same manner as the encryption and decryption of the aggregated results (Addendum E). The security manager class (SecMan), also provides for the encryption of a static itinerary.

The countermeasures that are incorporated as well as the implementation results are detailed in Table 7.7.

Table 7.7: Implementation results of basic restricted level

	Requirements				
Countermeasures	**Inhibits autonomy & mobility**	**Additional requirements**	**Additional communication sessions**	**Additional computational costs**	**Additional financial costs**
Path Histories	No	No	No	Yes	No
Detection objects	No	Yes	No	Yes	No
Proof carrying code	No	Yes	Yes	Yes	Yes
Anonymous itinerary	No	No	No	Yes	No
Partial result encapsulation	No	No	No	Yes	No

	Digital signatures	No	No	No	No	Yes
	Code obfuscation	No	No	Yes	No	Yes
	Code transformation	No	No	Yes	No	Yes
	Watermark	No	Yes	No	Yes	No
	Time sensitive agents	No	No	No	No	No

7.5.4 Extended restricted level

A large number of the countermeasures listed, discussed and implemented in the basic restricted level, also form part of the extended restricted level (as listed in Table 7.8). The implementation results of those countermeasures are thus not discussed again.

Table 7.8: Countermeasures for extended restricted level

Extended Restricted	
Detection methods	**Prevention methods**
Detecting code modifications: Detection objects / Watermarking *Authentication:*	

Digital signatures *Audit trail:* Path histories *State protection:* Reference states State appraisal *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques Environmental key generation *Keeping the agent secret:* Computing with encrypted functions *Protecting itinerary:* Anonymous itinerary *Auditing:* Partial result encapsulation

Additional countermeasures for this level include *Reference states* and *State appraisal* as detection methods. *Reference states* can be included by signing the state of the aglet. A way to accomplish this is by capturing the state of the object for signing purposes. The code of achieving this is listed in Figure 7.15, where an object of class *ByteArrayOutputStream* is used as input to an object of class *ObjectOutputStream.* As *Aglets* implements weak migration, the final state needs to be signed and recomputed at the next remote host.

```
ByteArrayOutputStream bout = new ByteArrayOutputStream();    ObjectOutputStream out
= new ObjectOutputStream(bout);
```

Figure 7.15: Capturing the state of an aglet

The use of *state appraisal* functions requires the creation of these functions to verify the state as well as the code of the aglet. The creation of a *state appraisal function* is captured by first requiring the mobile agent to define a security policy (containing permissions to be followed by the remote hosts). The created security policy is added to the agent and the mobile agent is digitally signed. Upon arrival at the remote host the agent as well as its state is verified by using the attached policy. An example of such a security policy is contained in Figure 7.16, where only the owner of the aglet is allowed to dispose of the aglet.

```
 grant codeBase http://*:*/, ownedby "owner", {
protection.com.ibm.aglet.security.AgletProtection
"owner" "dispose";  };
```

Figure 7.16: Aglet policy file

The digital signing of the aglet is covered in detail in previous sections and the same methods are followed to authenticate the agent for implementation of the *state appraisal* countermeasure. Figure 7.17 lists the code for the *stateAppraisal* method. Upon arrival at the remote host (method *onArrival*), the remote host validates the agent as well as its state, by computing the message digest of the aglet's state (method *VerifySignature*) as well as enquiring if set permissions in the aglet security file has been violated by the previous host (methods *RetrieveAgletPolicy* and *VerifyAgletPolicy*).

```
 public class StateAppraisal extends Aglet {    public void onCreation(Object init) {
addMobilityListener(     new MobilityAdapter() {             public void onArrival(MobilityEvent b) {
VerifySignature();
RetrieveAgletPolicy();                                         VerifyAgletPolicy();
}   }  );  }
```

Figure 7.17: State appraisal

The implementation results for the extended restricted level are listed in Table 7.9.

Table 7.9: Results of extended restricted level

	Requirements					
Countermeasures			**Inhibit sautonomy & mobility**		**Additional requirements**	
	Additional communication sessions			**Additional computational costs**		**Additional financial costs**
	Path Histories	No	No	No	Yes	No
	Detection objects		No	Yes	No	Yes

	No					
	Reference states		No	No	No	Yes
	No					
	State appraisal	No	Yes	No	Yes	Yes
	Anonymous itinerary		No	No	No	Yes
	No					
	Environmental key generation			No	Yes	Yes
	Yes	Yes				
	Partial result encapsulation		No	No	No	Yes
	No					
	Digital signatures		No	No	No	Yes
	No					
	Code obfuscation		No	Yes	No	Yes
	No					
	Code transformation		No	Yes	No	Yes
	No					
	Watermark	No	Yes	No	Yes	No
	Time sensitive agents		No	No	No	No
	No					

7.5.5 Basic Open Security Level

The countermeasures included in the basic open security level as well as the extended open level are listed in Table 7.10 and Table 7.11 respectively. The included measures have been implemented and discussed in the previous sections and are not discussed again. The countermeasures available to provide protection of the agent in the open levels, is a concern. This problem can however be alleviated by the maturity of countermeasures such as *computing with encrypted functions,* for example.

Table 7.10: Countermeasures for basic open security level

Basic Open	
Detection methods	**Prevention methods**
Detecting code modifications: Detection	objects / Watermarking *Authentication:*

Digital signatures *Auditing* Path histories *Preventing code modifications:* Code obfuscation & time techniques Code transformation & time techniques *Keeping the agent secret:* Computing with encrypted functions *Auditing:* Partial result encapsulation

Table 7.11: Countermeasures for extended open security level

Extended Open	
Detection methods	**Prevention methods**
Detecting code modifications: Detection	objects / Watermarking *Authentication:*

Digital signatures *Audit trail:* Path histories *State protection:* Reference states
State appraisal *Preventing code modifications:* Code

obfuscation & time techniques Code transformation & time techniques *Keeping the agent secret:* Computing with encrypted functions *Auditing:* Partial result encapsulation

For implementation results of the open security levels, the reader is referred to Tables 7.7 and 7.9.

7.6 Evaluation of Framework

The countermeasures incorporated within the framework as well as the implementation results of the countermeasures in the different levels is evaluated against the analysis of threats and countermeasures as discussed in Chapter 3. The results are depicted and subsequently discussed in the next few tables and sections.

Integrity Interference

The protection provided against integrity interference attacks on the different framework levels are shown in Table 7.12, Table 7.13, Table 7.14 and Table 7.15. A *yes* within the tables indicate that the specific level of the framework provides adequate protection for the specified part of the agent that is threatened, while a *no* specifies that no protection is provided. For example (Table 7.12) if an agent is incorrectly transmitted, the 3rd column, 3rd row implies that the Basic Closed level provide protection for the code of the agent, while the 5th column, 4th row, states that the Basic Restricted level provide no protection for the state of the agent. Grey areas stipulate the parts of the agent that is not affected by the mentioned threat.

Table 7.12: Integrity interference protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Integrity Interference						
	Transmitting mobile agent incorrectly				*BC*	*EC*	*BR ER*
	BO	*EO*					
Code	Threat	Yes	Yes	Yes	Yes	Yes	Yes
State	Threat	Yes	Yes	No	Yes	No	Yes
Control Flow	Threat	Yes	Yes	No	Yes	No	Yes
Data	**ID**	Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Itinerary	Threat	Yes	Yes	Yes	Yes	No	No

	Initial data	Threat	Yes	Yes	Yes	Yes	Yes	Yes
	Aggregated data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Aggregated essential data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Required data	Threat	Yes	Yes	No	No	No	No

The closed security levels provide security protection against the incorrect transmission of the mobile agent (illustrated in columns BC, BR and BO). The basic restricted and basic open levels only provide protection for the initial, aggregated and aggregated essential data (illustrated in columns BR and BO and relevant rows).

The ASDK provides a way of protecting the aglet from other aglets by making use of a proxy. This method can be extended into protecting the aglets' information, such *ID* and *time of creation*, by only allowing access to this information via the proxy. By certifying the aglet, the aglet can also be protected in that the verifying host would detect discrepancies. Through encrypting the itinerary, it can be protected, although this is not possible within the open environments.

Table 7.13: Integrity interference protection (cont.)
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Integrity Interference						
	Transmitting agent to host not on itinerary				*BC*	*EC*	*BR ER*
	BO	*EO*					
Code	No effect						

State No effect **Control Flow**No effect **DataID**No
effect **Itinerary**ThreatYesYesYesYesYesYes
Initial data No effect **Aggregated data** No effect **Aggregated essential data**No
effect **Required data**No effect

The protection against transmitting the agent to a host not on the itinerary is possible on all levels (illustrated in columns BC, EC, BR, ER, BO and EO) either by using *path histories* or *anonymous itinerary*.

Table 7.14: Integrity interference protection (cont.)
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

Integrity Interference

	Not executing the mobile agent completely				*BC*	*EC*	*BR ER*
	BO	*EO*					
Code	No effect						
	StateThreatYesYesNoYesNoYes						
Control Flow	Threat	Yes	Yes	No	No	No	No

Data	ID	No effect					

ItineraryNo effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**ThreatYesYesNoYesNoYes

The closed levels provide protection against not executing the agent completely (illustrated in columns BC and EC and relevant rows), while the basic restricted (illustrated in columns BR and BO and relevant rows) and basic open levels (illustrated in columns ER and EO and relevant rows) provide no protection. The extended restricted and extended open levels provide protection due to the inclusion of measures such as *reference states* and *state appraisal* (illustrated in columns ER and EO and relevant rows).

Table 7.15: Integrity interference protection (cont.)
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Integrity Interference						
	Executing mobil	*e agent ar*	*bitrarily*	*BC*	*EC*	*BR*	*ER BO*

EO

Code	No effect						

StateThreatYesYesNoYesNoYes

Control Flow	Threat	Yes	Yes	No	No	No	No
Data	**ID**	No effect					

ItineraryNo effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**ThreatYesYesNoYesNoYes

The mobile agent can be protected from the host executing it *arbitrarily* on the closed levels (illustrated in columns BC and EC and relevant rows), with the basic restricted and basic open levels providing no protection (illustrated in columns BR and BO and relevant rows). The extended restricted and extended open levels provide protection for the agent's state and required data (illustrated in columns ER and EO and relevant rows) with the inclusion of measures such as *reference states*.

Integrity modification

Integrity modification protection as provided on the different levels is listed in Table 7.16.

Table 7.16: Integrity modification protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Integrity Modification						
	Deleting, corrupting, mani	*pulating,*	*altering,*	*misinter*	*preting,*	*incorrect*	

execution. *BC* *EC* *BR* *ER* *BO* *EO*

Code		Threat	Yes	Yes	Yes	Yes	Yes	Yes
State		Threat	Yes	Yes	Yes	Yes	Yes	Yes
Control Flow		Threat	Yes	Yes	Yes	Yes	Yes	Yes
Data		**ID**	Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Itinerary	Threat	Yes	Yes	Yes	Yes	Yes	Yes
	Initial data	Threat	Yes	Yes	Yes	Yes	Yes	Yes
	Aggregated data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Aggregated essential data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Required data	Threat	Yes	Yes	Yes	Yes	Yes	Yes

All levels of the framework can protect against the deletion, manipulation, alteration, misinterpretation and incorrect execution of the agent (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows). This is achieved by the inclusion of techniques such as *code obfuscation* and *reference states*.

Availability

The protection that the different levels of the framework provide against availability attacks is detailed in the Tables 7.17, 7.18, 7.19, 7.20 and 7.21.

Table 7.17: Availability protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Availability						
	Execution resources (memory & CPU denied)				*BC*	*EC*	*BR*
	ER	*BO*	*EO*				
Code	Threat	Yes	Yes	Yes	Yes	Yes	Yes
State	Threat	Yes	Yes	Yes	Yes	Yes	Yes

Control Flow No effect **DataID**No effect **Itinerary**No effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**No effect The *denial of execution resources* by the remote host can be countered by adding time limitations to the existence of the agent. All the levels of the framework thus provide protection against these types of attacks (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows).

Table 7.18: Availability protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)
Availability

	Data denied / Bombarded with irrelevant information	*BC*	*EC*	*BR*

ER *BO* *EO*

Code	No effect						

State No effect **Control Flow**No effect **DataID**No effect **Itinerary**No effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**ThreatYesYesNoNoNoNo

The host bombarding the agent with irrelevant information can only be prevented within a trusted environment (illustrated in columns BC, EC, BR, ER, BO and EO).

Table 7.19: Availability protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Availability		
	Execution resources (memory & CPU delayed)	*BC*	*EC*	*BR*

ER *BO* *EO*

Code	Threat	Yes	Yes	Yes	Yes	Yes	Yes

State	Threat	Yes	Yes	Yes	Yes	Yes	Yes

Control Flow No effect **DataID**No effect **Itinerary**No effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**No effect

The protection provided against the host delaying execution resources is the incorporation of time limitation techniques as well as the use of trusted environment. All the levels of the framework thus provide protection for the agent (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows).

Table 7.20: Availability protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)
Availability

	Data is delayed	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*

Code	No effect						

State No effect **Control Flow**No effect **DataID**No effect **Itinerary**No effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**ThreatYesYesYesYesYesYes

Protection against the host delaying the supply of requested information is countered by the inclusion of time constraints on the lifetime of the agent or by releasing the agent only within a trusted environment. All the levels of the framework provide protection for the agent against the delay of data (illustrated in columns BC, EC, BR, ER, BO and EO).

Table 7.21: Availability protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Availability						
	Transmission refusal	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code	Threat	Yes	Yes	Yes	Yes	Yes	Yes
State	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Control Flow	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Data	**ID**	Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Itinerary	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Initial data	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Aggregated data		Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Aggregated essential data		Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Required data	Threat	Yes	Yes	Yes	Yes	Yes	Yes

All levels of the framework provide protection against the host refusing to transmit the agent by including *time sensitive agents* on the restricted and open levels and the use of technologies for creating a *trusted environment* (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows).

Confidentiality

The protection provided against confidentiality attacks are shown in the next number of tables (Tables 7.22, 7.23 and 7.24)

Table 7.22: Confidentiality protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Confidentiality						
	Eavesdropping	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code	Threat	Yes	Yes	Yes	Yes	Yes	Yes
State	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Control Flow	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Data	**ID**	Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Itinerary	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Initial data	Threat	Yes	Yes	Yes	Yes	Yes	Yes
Aggregated data		Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Aggregated essential data		Threat	Yes	Yes	Yes	Yes	Yes
Yes							
Required data	No effect						

All levels of the framework provide protection against eavesdropping attacks (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows). This is achieved by the inclusion of techniques such as *code obfuscation* and *environmental key generation.*

Table 7.23: Confidentiality protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)
Confidentiality

	Theft	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code	Threat	Yes	Yes	No	No	No	No
State	Threat	Yes	Yes	No	No	No	No
Control Flow	Threat	Yes	Yes	No	No	No	No
Data	**ID**	Threat	Yes	Yes	No	No	No
No							
Itinerary	Threat	Yes	Yes	No	No	No	No

	Initial data	Threat	Yes	Yes	No	No	No	No
	Aggregated data		Threat	Yes	Yes	No	No	No
	No							
	Aggregated essential data		Threat	Yes	Yes	No	No	No
	No							

Required data No effect

The only way, in which the agent can be protected against theft by the host, is by creating a trusted environment (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows).

Table 7.24: Confidentiality protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)
Confidentiality

		Reverse Engineer	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code		Threat	Yes	Yes	Yes	Yes	Yes	Yes
State		Threat	Yes	Yes	Yes	Yes	Yes	Yes
Control Flow		Threat	Yes	Yes	Yes	Yes	Yes	Yes
Data		**ID**	Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Itinerary	Threat	Yes	Yes	Yes	Yes	Yes	Yes
	Initial data	Threat	Yes	Yes	Yes	Yes	Yes	Yes
	Aggregated data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							
	Aggregated essential data		Threat	Yes	Yes	Yes	Yes	Yes
	Yes							

Required data No effect

The inclusion of techniques such as *code obfuscation* and *code transformation* in the framework counter reverse engineering attacks. All the levels of the framework provide protection for the reverse engineering of the mobile agent (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows).

Authentication

The protection of authentication techniques is list in the next two tables (Tables 7.25 & 7.26).

Table 7.25: Authentication protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)
Authentication

	Masquerading	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code	No effect						

State No effect **Control Flow**No effect **DataID**No effect **Itinerary**No effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**ThreatYesYesYesYesYesYes

A host masquerading as another can supply incorrect data as requested by the agent. Protection techniques for these types of threats include authentication of the host, as well as the digital signing of aggregated data. All the levels of the framework provide protection for the agent (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows) against masquerading attacks.

Table 7.26: Authentication protection
(BC= Basic Closed, EC=Extended Closed, BR=Basic Restricted, ER=Extended Restricted, BO=Basic Open, EO=Extended Open)

	Authentication						
	Cloning	*BC*	*EC*	*BR*	*ER*	*BO*	*EO*
Code	No effect						

State No effect **Control Flow**No effect **DataID**ThreatYesYesYesYesYesYes
ItineraryNo effect **Initial data** No effect **Aggregated data** No effect **Aggregated essential data**No effect **Required data**No effect

The aglet is protected against cloning attacks (illustrated in columns BC, EC, BR, ER, BO and EO and relevant rows), being that a unique identifier (ID) is assigned via a proxy and on a cloning event the ID of the clone is changed.

7.7 Conclusion

Our proposed mobile agent security framework has been implemented by using the *Aglets Software Development Kit (ASDK).* In this chapter we used simple aglets and implemented various types of countermeasures based on the different levels of security that were established in our proposed framework. A number of the specified countermeasures could not be implemented due to the methods only proposed in theory (such as *computing with encrypted functions*). The implementations were tested and as a result we found a security structure that allows for the dynamic integration of various types of countermeasures based on an evaluation of the deployment area and type of application. This framework has the added benefit that new countermeasures that are defined by other researchers, or methods that mature over time (such as encrypted functions) can be added to this structure. In the next chapter we discuss the implementation and test results in the context of other similar research and conclude our findings.

CHAPTER 8

RESEARCH SUMMARY, EVALUATION AND CONCLUSIONS

8.1 Summary of Propositions

The introduction of mobile agents as a computing paradigm established new possibilities for conducting business in a network and especially the Internet environment. The paradigm introduces several advantages such as alleviating bandwidth problems (Suri, *et al.*, 2000) and providing means for intelligent information retrieval (Aerts, *et al.*, 2002).

The development of applications based on mobile agent technology has however been burdened by the security problems introduced by the paradigm itself. Jansen (2000) categorises mobile agent threats into four distinct classes, namely threats imposed by (1) an agent on a host; (2) a host on an agent; (3) an agent on another agent; and (4) network entities on an agent. (In each case, an “agent” refers to a mobile agent.) The protection of hosts against malicious agents is based on security techniques in the subject field of *General Computer Security*. However, threats imposed by malicious hosts on agents introduced a new research area since current computer security solutions cannot simply be transferred to resolve these types of threats. This is largely due to the *autonomy* and *mobility* characteristics of mobile agents, which imply that an agent carries its code, data, attributes and state from site to site, where the site itself might be an unsafe execution or hosting environment. Whilst current computer security solutions enable practitioners to safeguard a particular site against malicious agent attacks, these solutions are unable to protect mobile code travelling to potentially unsafe environments. It is this very property of a mobile agent to be executed at various (potentially unsafe) sites, which is often most desirable of this specific technology.

The framework proposed in this study has been designed through several research phases, which we consequently summarise. As a first step, a *mobile agent threat model* has been established based on the five fundamental concerns or requirements of users gaining access of computer network services, namely *integrity, availability, confidentiality, authentication* and *non-repudiation* (ISO (7498-2), 1988). The intention of this model was to categorise the different types of threats that a malicious host could impose on an agent.

In the second research phase, current countermeasures for protecting a mobile agent against malicious host attacks were analysed and categorised into particular countermeasure classes, including *trust based computing, recording and tracking techniques, cryptographic techniques* and *obfuscation and time techniques*. This analysis provided specific information regarding the protection of specific components of the mobile agent.

In the third research phase, mobile agent systems, models, frameworks and architectures were studied and evaluated. This evaluation provided insights into the most salient security elements of the studied structures as well as their inherent drawbacks.

These insights led us to the fourth research phase, which continued research on available solutions to identify the security challenges that we are currently faced with.

The observations from the previous two research phases were combined to establish a set of requirements for a security framework (research phase 5). These requirements stipulate what a security framework has to adhere to in order to provide comprehensive security measures without inhibiting application performance or introducing unnecessary financial implications. In summary, these requirements insist on -

1. distinguishing between different types of *deployment environments*;
2. distinguishing between different *application objectives* (types of mobile agent applications);
3. preserving the *autonomous* and *mobile* character of a mobile agent ;
4. limiting requirements for additional hardware and software;
5. restraining the number of communication sessions;
6. regulating computational cost;
7. minimizing implementation costs;
8. providing a dynamic structure that could include security functions and services as required.

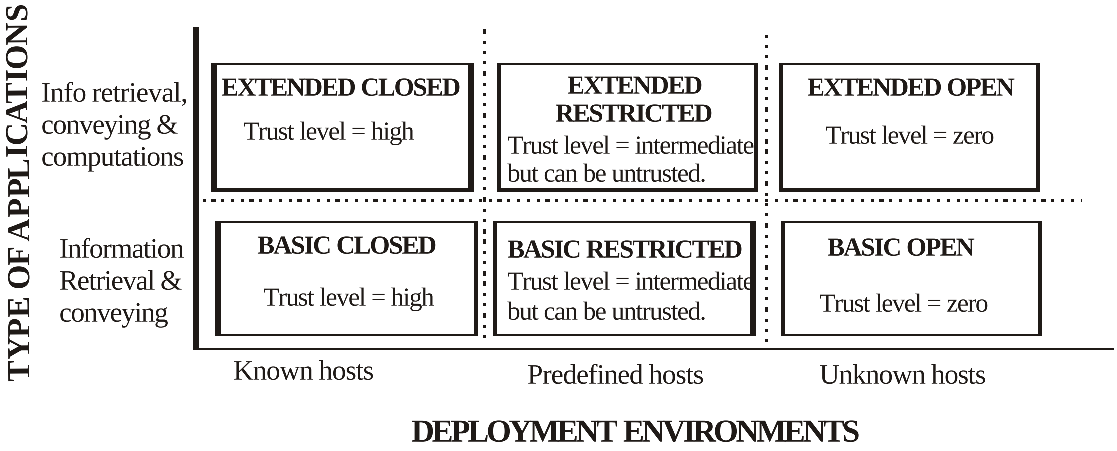

Figure 8.1: Security levels forming the basis of the proposed framework

Notable from these requirements is the importance to distinguish between different kinds of mobile applications, as well as the kinds of execution environments where these applications are to be deployed. These distinctions were captured in the definition of security levels (research phase 6), which are summarised in Figure 8.1. In a seventh research phase, the outcomes of the analysis of the countermeasures (phase 2) were integrated into the different security levels. The implication of phase 7 is illustrated in Figure 8.2. This led to the establishment of a dynamic mobile agent security framework that could be used by practitioners when designing mobile agent systems. (We describe the full advantages and significance of the proposed model in the following section.) Implementation and testing were done in research phase 8.

After implementing and testing the proposed framework, we returned our attention to the studied mobile agent frameworks, architectures, models, systems and applications (phase 3) to (a) determine if a security framework exists that adhere to requirements defined in research phase 5, and (b) evaluate the proposed security framework against current mobile agent security solutions. The investigation, done in phase 9, revealed that no framework, architecture, model or system could be found in literature that is based on a dynamic security framework adhering to the stated requirements.

Figure 8.2: Evaluation of countermeasures for specific security levels

The research phases described above are summarised in Figure 8.3.

The results obtained from the implementation and testing revealed that the proposed framework provide protection for a mobile agents on a number of levels, including against *integrity modification attacks*, certain *availability attacks,* certain *confidentiality attacks* and *authentication attacks*. However, the framework is only intended to provide guidelines to use current available countermeasures. Since the framework is a dynamic structure, it is the responsibility of the mobile agent developer / owner to carefully evaluate the deployment environment as well as the mobile agent system's objectives in order to make informed decisions regarding countermeasures to be integrated into the intended system.

8.2 Evaluation of Proposed Framework

Before considering the outcomes of our research effort, the objective and relevance of this study are revisited for a moment. To determine whether the objective of this study is scientifically sound, we asked ourselves why it was necessary to design a security framework, rather than, for example, design new countermeasures for the malicious host problem? The necessity for a framework can best be described by using an analogy to that of the processes involved in the building of a factory (industrial unit). The development of an industrial unit involves steps such as the design, construction of the structure, building or assembling of walls, interior decoration, et cetera. The inclusion of the *design plan* and *building of the structure* processes is essential to the ultimate assembly of the factory. A *structure* is needed to establish the outlines and requirements of the design, as well as to provide specifications of how and where the building blocks have to be placed. Without such a structure (in our case the framework), the building blocks (in our case the countermeasures) will be unorganised modules lying in disarray.

The importance of a mobile agent security framework can be seen in the large number of proposals in this area (see Mobile Agent List (2003)). The process of proposing a mobile agent security framework necessitates the establishment of criteria and subsequently a set of requirements to which the framework needs to abide to (Fischmeister *et al*., 2001). Yet, literature reveals the inexistence of such a set of requirements and as a result, also a lack of a comprehensive model that is based on such a set of requirements. In fact, our literature review described the details of many different countermeasures for malicious host attacks, without much interaction and integration possibilities. Furthermore, literature pointed out that different degrees of protection are required for the malicious host problem, but lack due to the non availability of requirements to aid mobile agent developers to design of secure systems (Orso *et al.*, 2001). Our proposal thus answers to a research problem that has been expressed by more than one researcher in the field of mobile agent technology.

In this study, the search for an existing security framework that adhere to our set of requirements, proved to be futile. However, as illustrated in Chapters 4 and 6, a number of proposals for an integrated security framework do indeed exist, but an analysis indicated that they don't provide adequate protection for all components of the mobile agent. This corresponds with Orso *et al.*, (2001) about the requirement for an integrated security framework that provides protection for the mobile agent against malicious hosts. In the next three subsections, we briefly point out why our proposed framework overcomes deficiencies of current solutions by considering problems in current systems, problems in countermeasures and security level issues.

8.2.1 Problems in current systems

The study of mobile agent systems (to act as tools for the creation and maintenance of mobile agent applications), revealed a number of useful insights into the creation of an agent security framework. One of the main problems encountered by most of these systems is the fact that these systems have generally been designed without considering agent security (see Mobile Agent List (2003)). It almost seems as though security against malicious hosts is generally not catered for.

Mobile agent systems rely on an operating system or virtual machine to provide an installation platform where an execution environment can be established. Furthermore, the security methods that are incorporated into the design of mobile agent systems are mostly an extension of techniques provided by the underlying operating system or virtual machine (for example, *Aglets* (Lange & Oshima, 1998)). These underlying systems (operating system or virtual machine) are biased towards securing the system, rather than visiting code (the mobile agent). Hence, the resulting mobile agent system that relies on these types of security mechanisms often fails to provide adequate protection methods for the mobile agent. The processes involved in creating mobile agent systems are quite extensive and are often the result of years of research (see Mobile Agent List (2003)). Thus, instead of developing new mobile agent systems that adhere to our list of security requirements, it is more beneficial and less expensive to provide a security framework that can be integrated into existing mobile agent systems. In this way, developers of mobile agent applications can incorporate our proposed security framework into their designs to provide much more protected mobile agents without creating new systems.

At this stage, mobile agent application developers are currently forced to develop their own execution environments in order to implement adequate security techniques (for example *Cherubim* (Campbell & Qian, 1998)). This situation is not ideal and defies the computing requirement of reuse.

There exist quite a large number of mobile agent system tools that can be used for the development of mobile agent applications (see Mobile Agent List (2003)). The number of these systems that provide security related measures to be used in current applications is however small in relation to the number of tools available. This has the effect that a limited number of applications have been developed due to their inability to provide protection measures for the agent, and this has an influence on the development of the paradigm as a whole (Green & Hurst, 1997).

8.2.2 Countermeasures

The discussion on current and proposed mobile agent frameworks, architectures and models (Chapter 4), as well as the evaluation of these systems against the proposed requirements of a security framework (Chapter 6), indicate that currently there exist no system that possesses an integrated system of security methods in order to provide optimum protection for the mobile agent against malicious hosts.

As shown in Chapter 3, countermeasures could be categorised by distinguishing between *detection* and *prevention* mechanisms. Literature shows that current mobile agent frameworks mostly integrate encryption and authentication techniques for *prevention* purposes, whilst the authentication of results is used for *detecting* potential interferences with, or damages done to the agent. As the distribution of keys within a cryptographic environment remains a question (Algesheimer *et al*., 2001; Fung *et al*., 2001), the prevention provided in these systems seems to be inadequate.

Our research also indicates that the results obtained by the application of protection mechanisms can be improved substantially by integrating such techniques with other countermeasures (such as *code obfuscation* and *time techniques*, as well as *code obfuscation*, *watermarking* and *time techniques*).

Because the security techniques of current systems are based on traditional security mechanism, the countermeasures used in these systems are mainly *authentication* and *encryption* techniques. Furthermore, these techniques provide better agent protection during transmission than during execution (where the malicious host has control) (Campbell & Qian, 1998). The proposed framework uses encryption methods to create trusted environments.

One of the most popular mechanisms to protect the mobile agent is to perform some sort of partitioning, such as splitting the agent into a security sensitive part and a no-security part, or making use of a distributed design where the agent is cloned and sent to the various hosts, or splitting the objective of the agent amongst several agents. In the proposed framework the mobile agent is considered as a single entity with its own goals and requirements. However, this implies that although an agent is most probably part of a mobile agent system consisting of many cooperating agents, the agent can cooperate with, and make use of stationary agents to complete its objective, the agent is not split into several agents to achieve a particular goal.

8.2.3 Security levels

As different types of applications call for diverse security mechanisms, it is vital that the mobile agent developers are enabled with tools to build more secure mobile agent applications. Distinguishing between types of applications and deployment environments is at the base of the proposed framework. Such a distinction offers several advantages including -

> it allows the agent developer to do a proper evaluation of the potential threats of a specific deployment environment;
>
> it allows the agent developer to select only specific and necessary countermeasures that could defend against the potential threats;
>
> it allows for the construction of a more light-weight and secure mobile agent application that target only threats that are a reality instead of carrying along superfluous countermeasures;
>
> considering the above advantages, it allows for a more cost effective, yet protected mobile agent system;
>
> once again, by considering the above advantages, it allows for the construction of an application with improved performance, where no unnecessary computations are conducted.

A large number of current mobile agent security frameworks is designed and developed for specific mobile agent applications (for example *Electronic supermarkets* (Wu, 2000)). As a result specific designs can often not easily be reused, extended or transferred to other, different applications. This is where the proposed framework is significant, being both dynamic (adaptable) and not application specific. Even though, existing systems often use security measures inherent to the underlying operating system or virtual machine, the proposed framework offers the opportunity to add additional (needed) countermeasures based on environmental and application evaluation without redesigning the entire application.

8.3 Practical Implementation of the Framework

In assessing the proposed framework it is vital to evaluate the proposal against current frameworks by using the devised requirements. Examples of applications that distinguish between different types of environments and different types of applications are minimal (see *M&M* (Marques *et al.,* 2001) and *Plain text algorithm* (An *et al*, 2002)). Unfortunately (as pointed out in Chapter 5), these systems do not uphold some of the most salient characteristics of mobile agents such as autonomy and mobility, which are essential to be preserved.

It is these characteristics that form the essence of the mobile agent paradigm. To protect against malicious host attacks, many of the current systems require the creation of a trusted environment. Such a requirement inherently restricts the mobile agent paradigm. Although some applications might benefit from the establishment of a set of trusted hosts (such as *SIAS* (Chan *et al.,* 2000)), it remains necessary for a security framework to provide for environments in which the agent can roam freely.

Figure 8.4 provides graphical information on the systems that restrict the agent as well as those that allow for the agent to visit any host. The x-axis portrays the different frameworks as discussed, while the y-axis depicts the restrictions placed by the different frameworks in terms of mobility and autonomy. These restrictions are sectioned into three categories, namely *low, intermediate* and *high*. A *low* indicates that either the mobility or the autonomy of the agent is constrained (for example *Security enhanced mobile agents*), *intermediate* specifies that the mobility and the autonomy of the agent is restricted (for example *FILIGRANE*) and a *high* points out that the autonomy and mobility of the agent is limited as well as the agent only allowed to roam within a trusted environment (for example *Supervisor-worker*). A zero level is indicated on the y-axis if no autonomy and mobility restrictions are enforced by the specific framework (for example our proposed framework).

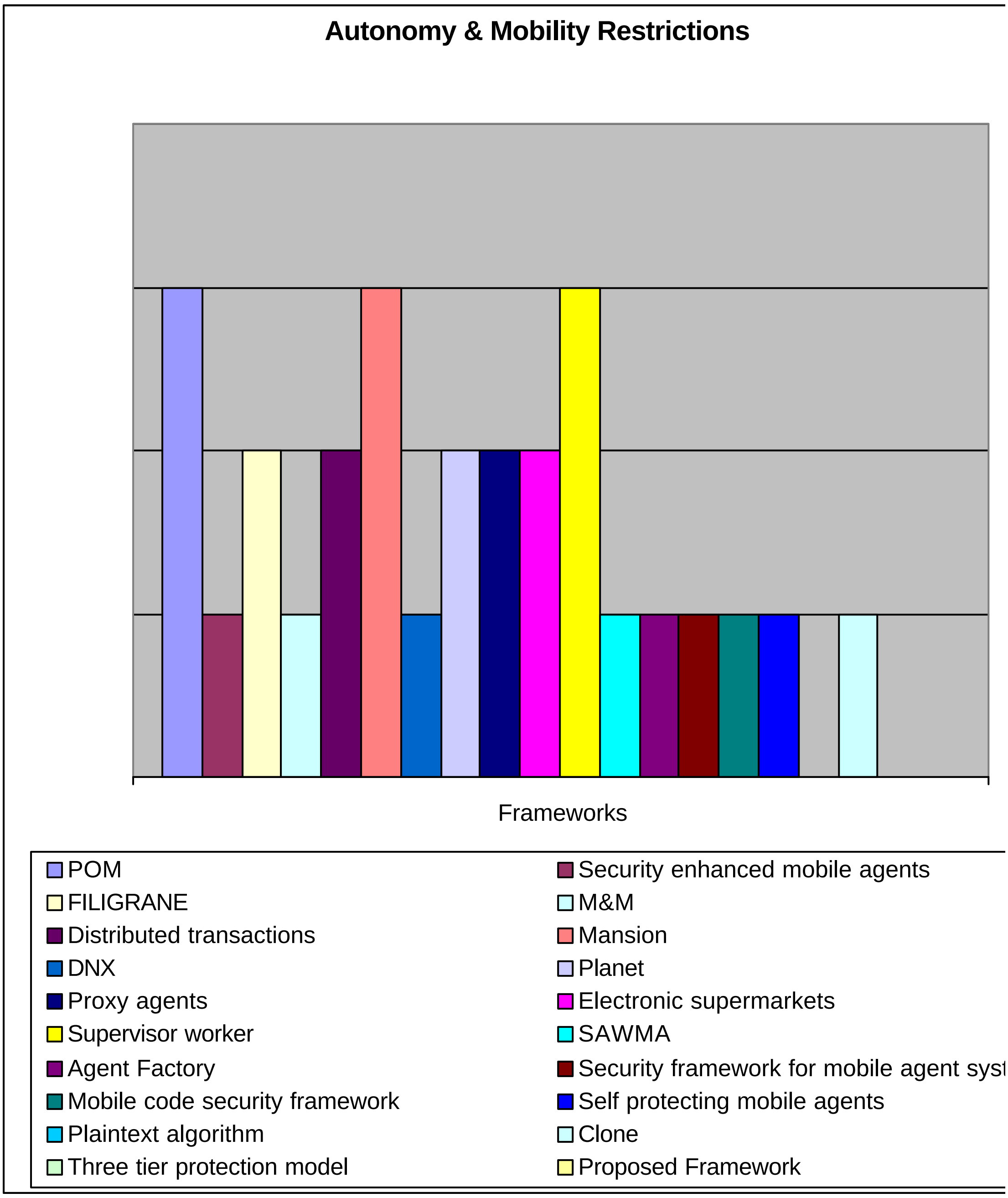

Figure 8.4: Autonomy & mobility restrictions

The costs of implementing different countermeasures must be in direct relation to the degree of security required. These types of costs include financial costs (in acquiring additional hardware or software) as well as computational costs (for example additional CPU and memory requirements). Figure 8.5 provides information with regards to the additional computation costs as required by the current frameworks, with the x-axis listing the different frameworks and the y-axis indicating three levels of computational costs incurred. The additional computational costs required by the implementation of the different frameworks are measured within three categories, namely *low, intermediate* and *high.* Frameworks within the *low* category such as *DNX* only make use of cryptographic methods to protect the agent, while frameworks within the *intermediate* category such as *FILIGRANE* incorporates two additional methods (both cryptographic methods and watermarking techniques). Frameworks categorized as *high* integrate into their designs three or more countermeasures that require additional hardware or software, such as our proposed framework (which include for example cryptographic measures, watermarking, code obfuscation and detection methods).

Our proposed framework as implemented has no additional financial implications but we did experience increases in computational costs. The high level of computational costs for our framework reflects the burden placed on the system if all the countermeasures on all the security levels (*basic, restricted* & *open*) are implemented in one application. This increase will be lessened if the developer of a mobile agent application makes informative decisions regarding the measures to be incorporated into the design.

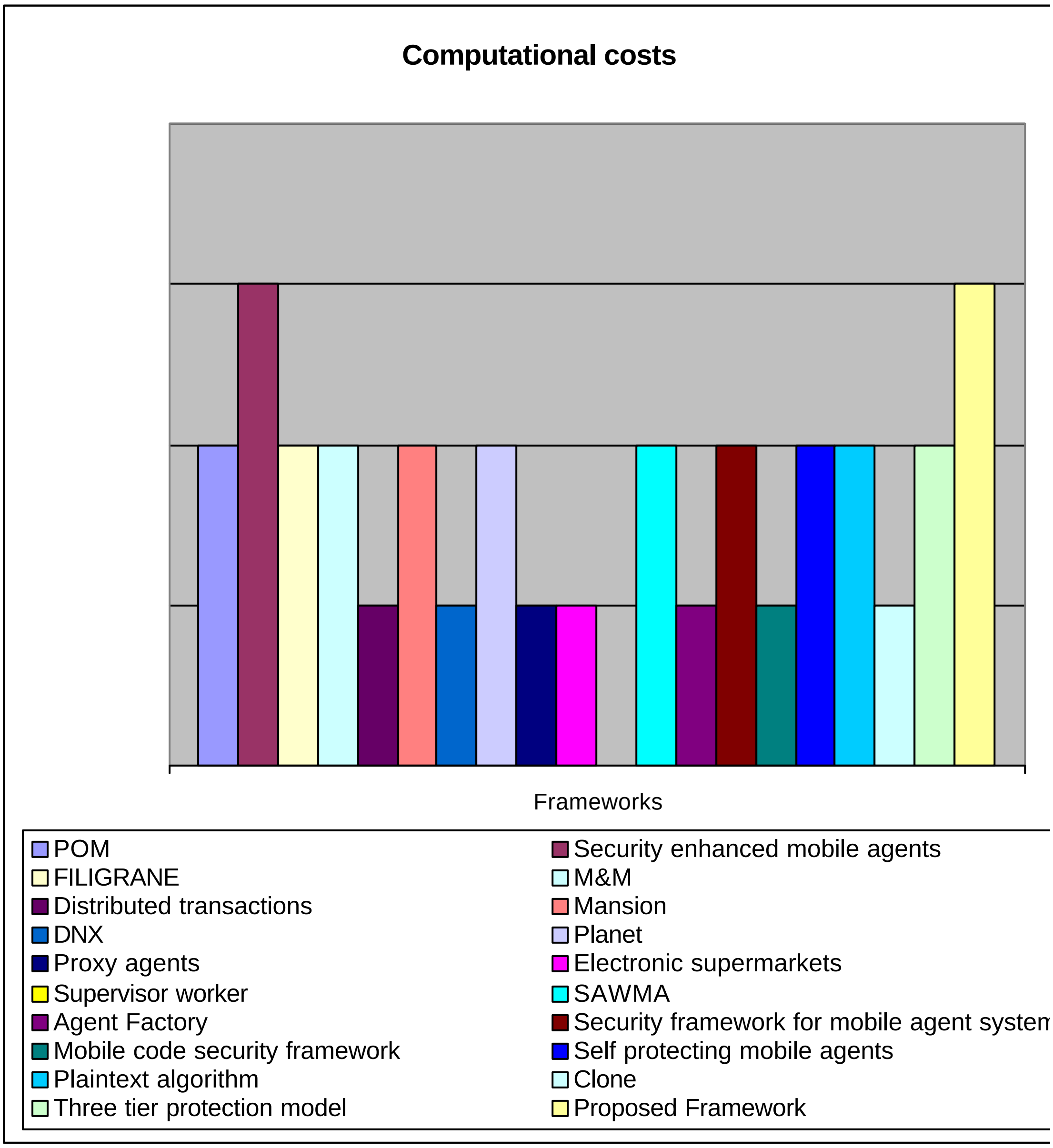

Figure 8.5: Cost implications

Besides additional computational and financial costs, the implementation of countermeasures can also imply added requirements, such as a certification authority. Figure 8.6 details these added requirements, with the x-axis portraying the different frameworks and the y-axis three categories of additional requirements, namely *low* (which indicates only one additional requirements), *intermediate* (which indicates two additional requirements) and *high* (which indicates three or more additional requirements). For example *Security enhanced mobile agents* only make use of trusted entities, while *Self protecting mobile agents* include code obfuscation software as well as software used to divide the mobile agent into different subagents. *FILIGRANE* need code obfuscation, watermarking as well as encryption software for implementation. As seen our proposed framework has some additional requirements (specialised hardware in the creation of a trusted environment).

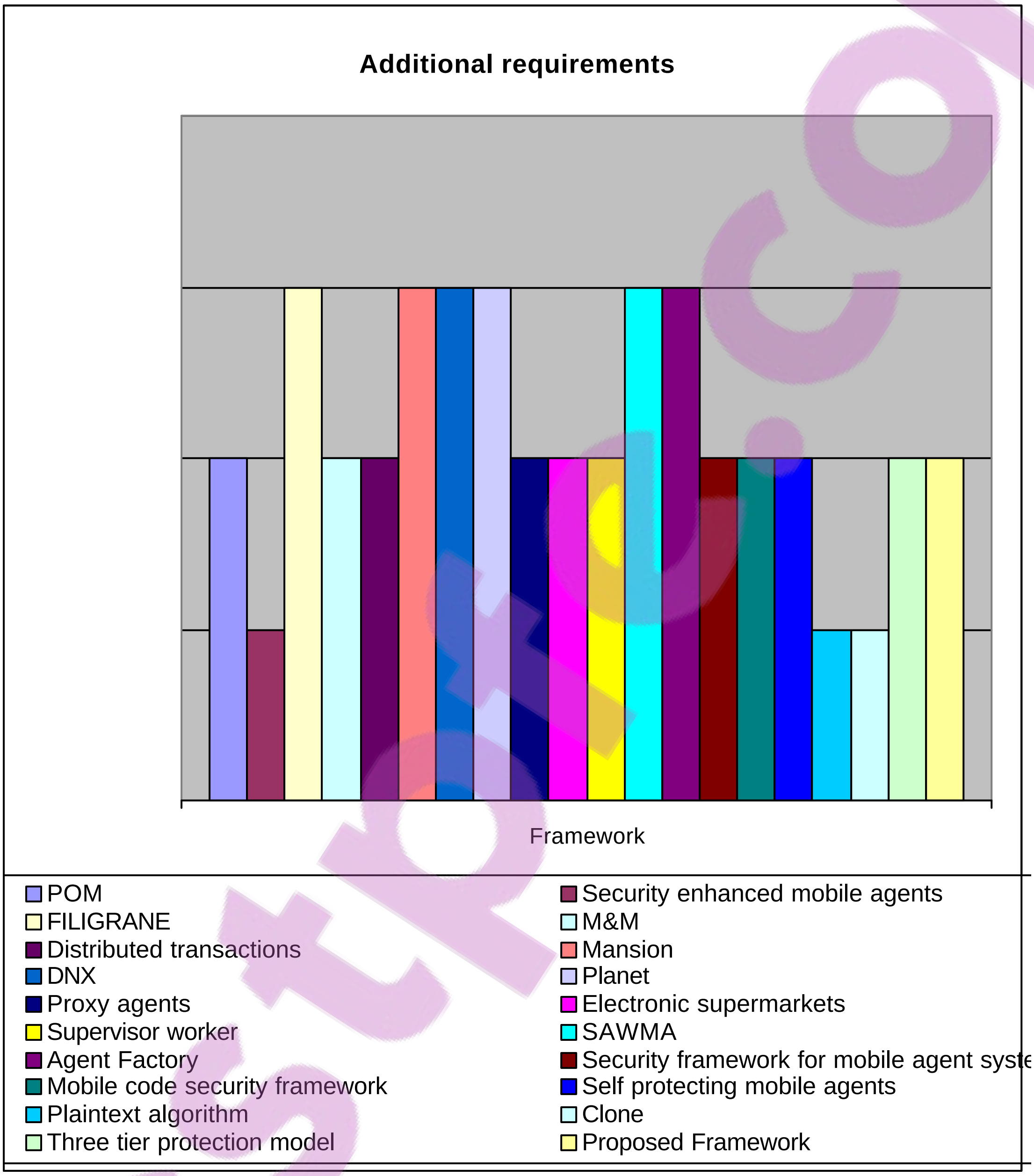

Figure 8.6: Level of additional requirements

The different frameworks require additional communication sessions for purposes such as the exchange of session keys, or to convey aggregated results back to the local host. However, the need for multiple sessions compromises the advantage of minimum bandwidth requirements that are typically associated with mobile agents. Figure 8.7 outlines the additional communication sessions of the different frameworks by distinguishing between *low* (only additional communication sessions in terms of sending the agent to a trusted entity or by using techniques whereby results obtained are conferred to the local host) and *high* (additional communication sessions required by distributed agents as well as trusted entities). For example *POM* need additional communication sessions between trusted entities as well as different parts of the mobile agent, while *M&M* make use of a trusted entity and *FILIGRANE* illustrate no additional communication sessions upon implementation. As illustrated in Figure 8.7, our framework requires no additional communication sessions between different entities within the system.

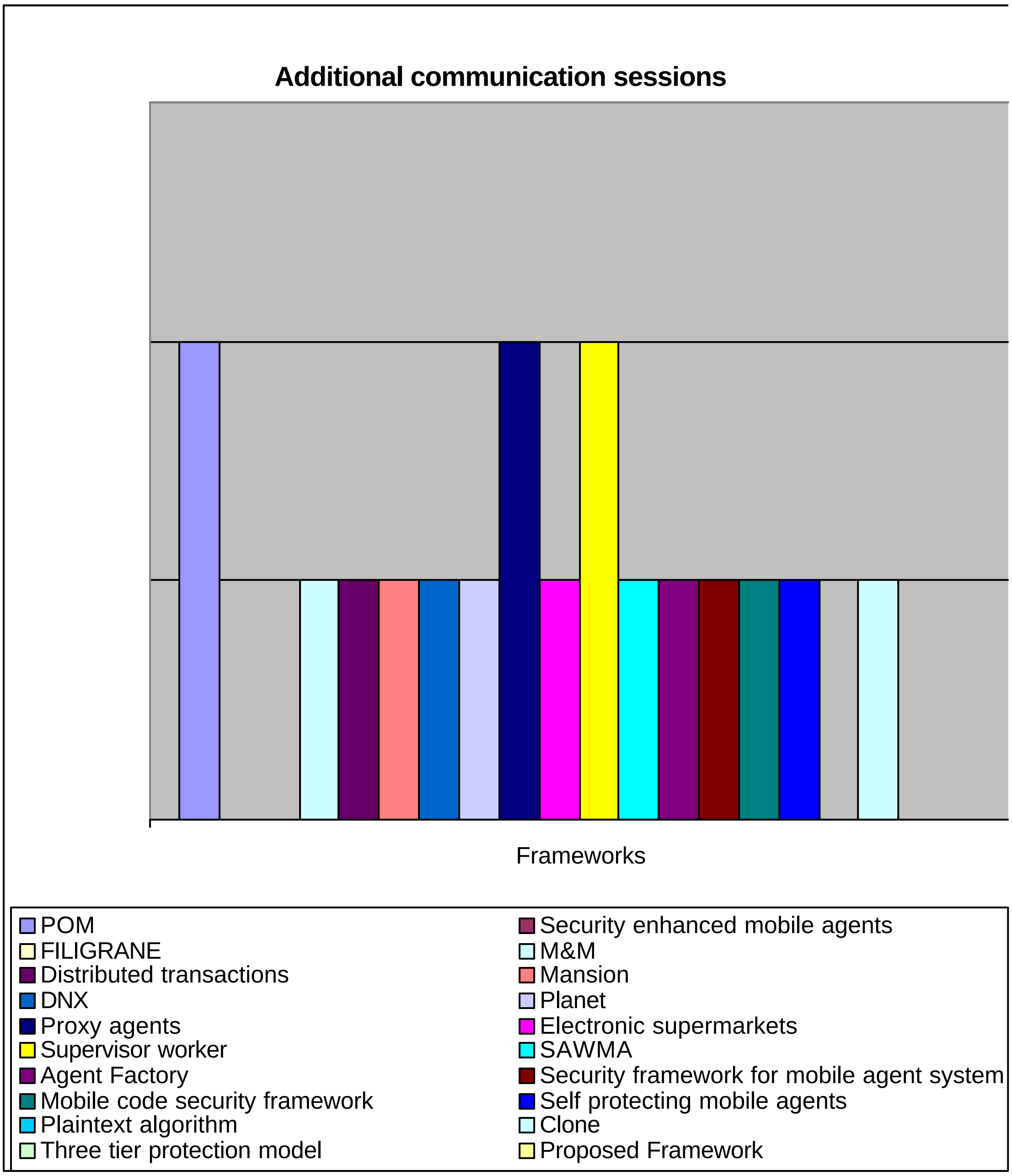

Figure 8.7: Increase in communication sessions

Figure 8.8 provides a graphical interpretation for the countermeasures that are integrated into the different frameworks. As illustrated, the countermeasures are categorised into *trust, recording & tracking*, *cryptography* and *time* techniques. The evaluation of currently available mobile agent systems according to the number of countermeasures that are incorporated into each category is illustrated. As shown, a large number of systems rely on a trusted environment by using techniques such as *authentication* and *encryption.* (For sake of clarity, the use of conventional encryption to protect the agent between hosts is classified as being part of trust-based computing.). In figure 8.8 the x-axis contains the different frameworks grouped within the four mentioned categories of countermeasures, while the y-axis provide information regarding the number of countermeasures included within the four categories. For example, *POM* has incorporated into its design one trusted measure (trusted entities) with no measures from the other three categories included.

Our framework integrates methods within every countermeasure class and is thus provides more options in terms of protecting the agent.

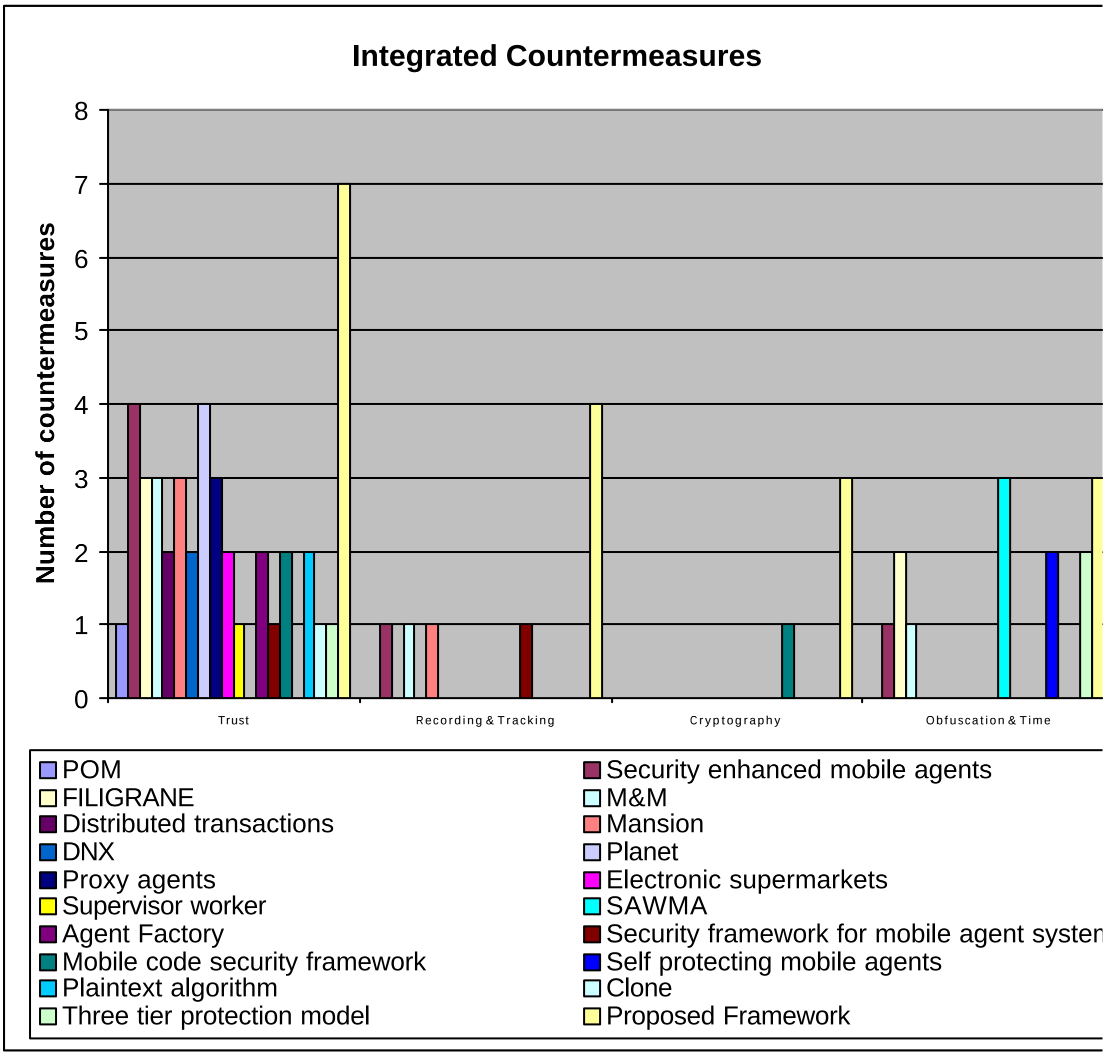

Figure 8.8: Integrated countermeasures

8.4 Drawbacks of the Proposed Framework

A number of drawbacks of our proposed framework are noted. Although the framework enables mobile agent application developers to secure an agent against many types of malicious host attacks, some parts of the agent are still vulnerable, specifically under certain conditions. For example, the discussed solutions are vulnerable when a host floods the agent with irrelevant information, or even steals the agent. This can partly be attributed to the fact that certain countermeasures (such as computing with encrypted functions) have not yet reached maturity. Furthermore, countermeasures that provide comprehensive protection in an open environment are minimal and research is desperately required for the expansion of this field (Roth, 2001).

Another drawback is the implication on computational costs if an agent is to operate in extreme conditions. Extreme conditions are those where the application requires and thus implements all countermeasures that are listed on the framework in order to safeguard the agent against malicious host attacks. It is indeed possible, that certain applications operating in open environments, especially where fraud is a potential threat, might require extreme measures. Further research is required in this area to attempt a reduction in the computational or financial costs.

A possible limitation of the current implementation of the proposed framework is the fact that the framework was only tested in the Java environment. With the development of new technologies the implement-ability thereof need to be tested in other environments such as .NET, where Java is not necessarily the language of choice.

Another implementation drawback arises from the current immature state of certain countermeasures (as hinted above). Although these measures are listed in the framework, they have to mature before implementation tests can be conducted. This implies that, considering the objectives of this research, it was not possible to test all the possible countermeasures against malicious host attacks.

The last potential drawback is at the same time one of the most salient elements of the proposed framework. As the framework is a dynamic structure, its correct implementation depends on the mobile agent developer /owner to correctly evaluate the type of application as well as the deployment environment. The selection of particular countermeasures not only relies on this, but also on the free will of the developer.

As mentioned before, the framework protects the mobile agent on a number of levels, such as against *integrity modification attacks*, certain *availability attacks,* certain *confidentiality attacks* and *authentication attacks.* Points of concerns are the protection of the mobile agent against theft; refusal of aggregated data and denial of data on the restricted and open levels of the framework as well as not executing the agent completely on the *closed levels.* Although the proposed framework might provide an improved measure of security, an agent might still be vulnerable to attacks from these types of threats.

8.5 Future Research and Possible Extensions

At the moment, the main target of the proposed framework is secure mobile agent applications. This implies, that the proposed framework offers value especially to new developments that adhere to Internet demands (e-commerce applications). However, the framework can also be extended to assist developers of mobile agent system tools. For these types of developments, the evaluation of the *application objective* has to be reconsidered, since it might not always be predictable beforehand.

A particularly useful extension to this framework will be the development of an intelligent computerised framework that could assist in the evaluation of the *application objective* (type of application) and *deployment environment*, based on inputs from the mobile agent developer / owner. The objective of such a system would be to use the given inputs, evaluate the security levels and make useful suggestions for countermeasures to be incorporated into the new mobile agent system. What would be even more useful is if this computerised framework can generate protection objects that could be integrated into the new system.

As described above, the proposed framework is particularly significant to practitioners and mobile agent system developers. However, the framework is also useful to fellow researchers within the mobile agent security field. The proposed framework provides the opportunity to researchers to determine methods to integrate their countermeasures into the framework. It also offers to an opportunity continue this research with the inclusion of an automatic intelligent component, that would reduce the chances of incorrect human evaluation.

Future research on specific countermeasures is required, as current measures seem to be inadequate in providing protection against all threats. Research into the role and specific requirements of trusted entities and if (and where) they exist determine whether they can provide services such as secure computations.

BIBLIOGRAPHY

AERTS, A.T.M., SZIRBIK, N.B. & GOOSSENAERTS, J.B.M. 2002. A flexible, agent-based ICT architecture for virtual enterprises. *Computers in Industry,* 49(3): 311-328.

AGENT DEVELOPMENT KIT (ADK). [Online]. Available at: <http://www.tryllian.com/development/DOCS2_1/devguide/ch12.html>. Accessed: 08/01/2004.

ALBAYRAK, S. & WIECZOREK, D. 1999. JIAC - A Toolkit for Telecommunication Applications. In: *Proceedings of the Third International Workshop on Intelligent Agents for Telecommunication Applications. Sweden*:1-18.

ALGESHEIMER, J., CACHIN, C., CAMENISCH, J. & KARJOTH, G. 2001. Cryptographic Security for Mobile Code. In: *The 2001 IEEE Symposium on Security and Privacy, Oakland, California.*

AN, L., JIANG, Q., LUO, X. & REN, Z. 2002. Protecting Mobile Agents against Malicious Hosts. Term Paper.

ASAKA, M., OKAZAWA, S., TAGUCHI, A. & GOTO, S. 1999. A Method of Tracing Intruders by Use of Mobile Agent. In: *Proceedings of the 9th Annual Internetworking Conference (INET99).* San Jose, California.

ASLAM, J., CREMONINI, M., KOTZ, D., & RUS, D. 2001. Using Mobile Agents for Analyzing Intrusion in Computer Networks. In: *Proceedings of the Workshop on Mobile Object Systems. ECOOP 2001.*

BAEK, J. 1999. A Design of a Protocol for Detecting a Mobile Agent Clone and its correctness proof using Coloured Petri Nets. In: *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing.* Atlanta, Georgia.

BAUMANN, J., HOHL, F., ROTHERMEL, K. & STRABER, M. 1998. Mole -

Concepts of a Mobile Agent System. *World Wide Web Journal*, 1(3):12-137.

BERNERS-LEE T., HENDLER J. & LASSILA, O. 2001. The Semantic Web. *The Scientific American.com.* [Online]. Available at: <http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21>. Accessed: 10/6/2003.

BORSELIUS, N. 2002. Mobile agent security. *IEEE Electronics & Communication Engineering Journal,* 14(5):211-218, October.

BRADSHAW, J. M. (ed.). 1997. *Software Agents.* AAAI Press / MIT Press: Menlo Park, California.

BRAUN, P., EISMANN, J., ERFURTH, C. & ROSSAK, W. 2000. Concepts and Architecture of the Mobile Agent System Tracy. [Online] Whitepaper. Available at: < http://www.the-agent-factory.de/src/paper.pdf>. Accessed: 4/11/2003.

BRAZIER, F.M.T., OVEREINDER, B.J., VAN STEEN, M. & WIJNGAARDS, N.J.E. 2002. Agent Factory: Generative Migration of Mobile Agents in Heterogeneous Environments. In: *Proceedings of the ACM Symposium on Applied Computing (SAC 2002):*101-106.

BREUGST, M., CHOY, S., HAGEN, L., HOFT, M. & MAGEDANZ, T. 1999. Grasshopper - An Agent Platform for Mobile Agent Based Services in Fixed and Mobile Telecommunications Environments. In: Bigham, J. (ed.). *Proceedings of the Software Agents for Future Communication Systems Workshop.* Springer Verlag:326-257.

BRYCE, C. 2000. A security framework for a mobile agent system. In: *6th European Symposium on Research in Computer Security.* Toulouse, France, Springer: 273-290.

CACHIN, C., CAMENISCH, J., KILIAN, J. & MÜLLER, J. 2000. One-Round Secure Computation and Secure Autonomous Mobile Agents. In: Montanari, U., Rolim, J.D.P. &

Welzl, E. (eds.). *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP).* Lecture Notes in Computer Science, vol. 1853. Springer-Verlag, New York.

CAMPBELL, R. & QIAN, T. 1998. Dynamic Agent-based Security Architecture for Mobile Computers. In: *Proceedings of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98)*, Australia: 291-299.

CHAN, A.H.W. & LYU, M.R. 1999. The mobile code paradigm and its security issues. *World Wide Web: Technologies and Applications for the New Millennium*. G.H. Young (ed.), CSREA Press, Athens, Georgia: 353-357.

CHAN, A.H.W., WONG, T.Y., WONG, C.K.M. & LYU, M.R. 2000. SIAS: A Secure Shopping Information Agent System. In: *Proceedings of the Fourth International Conference on Autonomous Agents.* Barcelona, Spain: 257-258.

CHESS, D., GROSOF, B., HARRISON, C.G., LEVINE, D., PARRIS, C. & TSUDIK, G. 1995. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2 (3):34-49.

CLAESSENS, J., PRENEEL, B. & VANDEWALLE, J. 2001. Combining World Wide Web and Wireless Security. *Network Security*:153-172.

COMER, D.E. & STEVENS, D.L. 1999. *Internetworking with TCP/IP, Vol III: Client-Server Programming and Applications.* Second Edition. Prentice Hall.

CORRADI, A., MONTANARI, R. & STEFANELLI, C. 1999. Security Issues in Mobile Agent Technology. In: *7th IEEE workshop on future trends of distributed computing systems.* Tunisia, December 1999.

D'ANNA, L., MATT, B., REISSE, A., VAN VLECK, T., SCHWAB, S. & LEBLANC,

P. 2003. Self Protecting Mobile Agents - Final Report. [Online]. Available at: < http://opensource.nailabs.com/jbet/papers/obfreport.pdf>. Accessed: 03/01/2004.

DASGUPTA, P., NARASIMHAN, N., MOSER, L.E. & MELLIAR-SMITH, P.M. 1999. MAgNET: Mobile agents for networked electronic trading. *IEEE Transactions on Knowledge and Data Engineering,* 11(4):509-525.

DE ASSIS SILVA, F. M. & POPESCU-ZELETIN, R. 2000. Mobile Agent-Based Transactions in Open Environments. *IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems,* E38-B(5), May.

D'INVERNO, M. & LUCK, M. 2001. *Understanding agent systems.* Springer, Berlin.

ETZIANI, O. & WELD, D. 1995. Intelligent Agents on the Internet: Fact, fiction and forecast. *IEEE Expert*, 10(4):44-49.

FARMER, W., GUTTMAN, J. & SWARUP, V. 1996. Security for Mobile Agents: Authentication and State Appraisal. In: *Proceedings of the 4th European Symposium on Research in Computer Security.* Rome Italy:118-130.

FEIGENBAUM, J. & LEE, P. 1997. Trust Management and Proof-Carrying Code in Secure Mobile Code Applications. Position Paper. DARPA Workshop on Foundations for Secure Mobile Code.

FISCHMEISTER, S. 2000. Building Secure Mobile Agents: The Supervisor-Worker Framework. Masters Thesis. Technical University of Vienna.

FORD, B. & KARMOUCH, 1997. An Architectural Model for Mobile Agent-Based Multimedia Applications. In: *Proceedings of the Canadian Conference on Broadband Research.* Ottawa.

FRANKLIN, S. & GRAESSER, 1997. Is it an agent or just a program? A taxonomy for autonomous agents. In: Muller, J.P., Wooldridge, M., & Jennings, N.R. (ed.) *Intelligent*

Agents III - Proceedings of the 3^{rd} *International Workshop on Agent Theories, Architectures and Languages*. Lecture Notes in Artificial Intelligence: Berlin: Springer Verlag: (1193): 21-36.

FUGETTA, A., PICCO, G.P. & VIGNA, G. 1998. Understanding code mobility. *IEEE transaction on software engineering*: 24(5).

FÜNFROCKEN, S. & MATTERN, F. 1999. Mobile Agents as an Architectural Concept for Internet-based Distributed Applications - The WASP project approach-. In: *Proceedings of KiVS'99*. Steinmetz. Springer:32-43.

FUNG, W., GOLIN, M. & GRAY, J. 2001. Protection of keys against modification attacks. In: *The 2001 IEEE Symposium on Security and Privacy, Oakland, California.*

GENESERETH, M.R. & KETCHPEL, S.P. 1994. Software Agents. *Communications of ACM*, 37:48-53.

GRAY, R.S. 1996. Agent Tcl: A flexible and secure mobile agent system. In: *Proceedings of USENIX Tcl/Tk Workshop*. Monterey, California:9-23.

GRAY, R.S., KOTZ, D., CYBENKO, G. & RUS, D. 1998. *D'Agents: Security in a multiple-language, mobile-agent system.* In: Vigna, G. (ed). *Mobile Agents and Security*. Springer Verlag:154-187.

GRAY, R.S., CYBENKO, G., KOTZ, D., PETERSON, R.A. & RUS, D. 2002. D'Agents: Applications and performance of a mobile agent system. *Software: Practice And Experience*, 35(6):543-573.

GREEN, S. & HURST, L. 1997. Software Agents: A review. LAG Report, Trinity College, Dublin.

GRIMLEY, M.J. & MONROE, B.D. 1999. Protecting the integrity of agents: an exploration in letting agents loose in an unpredictable world. [Online] *ACM Crossroads*, ACM. Available at: <http://www.acm.org/crossroads/xrds5-4/integrity.html>. Accessed: 4/11/2003.

GUAN, X., YANG, Y., & YOU, J. 2000. POM - A Mobile Agent Security Model against Malicious Hosts. In: *Proceedings of the fourth International Conference on High Performance Computing in the Asia-Pacific Region*: 2(2000):1165-1166.

GUPTA, G., SUNNY, S., NAUTIYAL, S., GANERIWALLA, S. & HSU, W. 2001. Virtual Internet Pets based on Java-enabled Mobile Agents. In: *The International Workshop on Agent Technologies over Internet Applications.* Taipei, Taiwan.

HARKER, K.E. 1995. TIAS: A Transportable Intelligent Agent System. [Online]. Available at: < http://www.cs.dartmouth.edu/reports/abstracts/TR95-258/>. Accessed: 08/01/2004.

HOFFMANN, M., PETERS, J. & PINSDORF, U. 2002. Multilateral Security in Mobile Applications and Location Based Services. In: *Europe's Independent IT Security Conference (ISSE 2002)*. Paris, France: The European Forum for Electronic Business: October 2002.

HOHL, F. 1997. An Approach to Solve the Problem of Malicious Hosts. *Universität Stuttgart*, Fakultät Informatik, Bericht Nr. 1997/03.

HOHL, F. 1998. Time Limited Blacbox Security: Protecting Mobile Agents from Malicious Hosts. In: G. Vigna (ed.). *Mobile Agents and Security.* Springer Verlag, Lecture Notes in Computer Science:1419:92-113.

HOHL, F. 1999. A Protocol to Detect Malicious Host Attacks by Using Reference States. Universität Stuttgart.

HOHL, F. 2000. A Framework to Protect Mobile Agents by Using Reference States. In: *Proceedings of the 20th International Conference on Distributed Computing Systems.* Taipei, Taiwan.

HORVAT, H., CVETKOVIÆ, D., MILUTINOVIÆ, D., KOÈOVIÆ, P. & KOVAÈEVIÆ, V. [CD-ROM]. 2000. Mobile Agents and Java Mobile Agent Toolkits. In: *Proceedings of the 33rd Hawaii International Conference on System Sciences.* Hawaii:10.

IBM Intelligent Agents Whitepaper. 1998. [Online]. Available at: <http://www.networking.ibm.com/iag/iaghome.html>. Accessed: 23/2/2002.

INTERNATIONAL STANDARDS ORGANIZATION. 1988. ISO 7498-2. *Security Architecture*.

JALALI, M. 2000. FILIGRANE Protocol: A Security Protocol for Trading of Mobile Code in Internet. In: *6th International Conference on Information Systems, Analysis, and Synthesis (ISAS/SCI).* Orlando, USA.

JALALI, M., HACHEZ, G. & VASSEROT, C. 2000. FILIGRANE. A security framework for trading of mobile code in Internet. In: *Autonomous Agents Workshop: Agents in Industry.* Barcelona, Spain.

JANSEN, W. A. 1999. Mobile Agents and Security. In: *Canadian Information Technology Security Symposium.*

JANSEN, W.A. 2000. Countermeasures for Mobile Agent Security. In: *Computer Communications, Special Issue on advanced security techniques for network protection*, Elsevier Science: 23(17) November 2000.

JANSEN, W. A. 2001 A Privilege Management Scheme for Mobile Agent Systems. In: *First International Workshop on Security of Mobile Multiagent Systems, at the Fifth International Conference on Autonomous Agents Conference.* Montreal, Canada.

JANSEN, W. & KARYGIANNIS, T. 1999. Mobile Agent Security. *NIST Special Publication 800-19*, National Institute of Standards and Technology: August 1999.

JOHANSEN, D., VAN RENESSE, R. & SCHNEIDER, F.B. 1995. Operating System Support for Mobile Agents. In: *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems.*

JUMPING BEANS. [Online]. Jumping Beans Incorporated. Available at: <http://www.jumpingbeans.com/>. Accessed: 04/11/03.

KAGAL, L., PERICH, F., CHEN, H., TOLIA, S., ZOU, Y., FININ, T., JOSHI, A., PENG, Y., COST, R.S. & NICHOLAS, C. 2003. Agents making sense of the Semantic Web. *NEC Research Institute CiteSeer*. [Online]. Available at: <http://citeseer.nj.nec.com/531295.html>. Accessed: 5/5/2003.

KARJOTH, G., ASOKAN, N. & GULCU, C. 1998. Protecting the Computation Results of Free-Roaming Agents. In: *Proceedings of the Second International Workshop on Mobile Agents.* Stuttgart, Germany.

KARJOTH, G., LANGE, D.B. & OSHIMA, M. 1997. A security model for aglets. *IEEE Internet Computing*, 1(4) pp 68-77.

KARNIK, N. 1998. Security in mobile agent systems. PhD Dissertation, University of Minnesota.

KARNIK, N. 2000. Security in Mobile Agent Systems. *Technical Report*. University of Minnesota.

KARNIK, N.M. & TRIPATHI, A. R. 2000. *Security in the Ajanta mobile agent system*. John Wiley & Sons, Ltd.

KATO, K., TOUMURA, K., AIKAWA, S., YOSHIDA, J., KONO, K., TAURA, K. & SEKUGUCHI, T. 1996. Protected and Secure Mobile Object Computing in PLANET. In: *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems.* Linz, Austria.

KILIAN-KEHR, R. & POSEGGA, J. 2002. Smart Cards in Interaction: Towards Trustworthy Digital Signatures. In: *Proceedings of the 5th Smart Card Research and*

Advanced Application Conference (CARDIS-02). San Jose, California.

KINIRY, J. & ZIMMERMAN, D. 1997. A Look at Mitsubishi's Concordia. [Online]. Available at: < http://www.computer.org/internet/v1n4/kiniry.htm>. Accessed: 30/04/03.

KOTZANIKOLAOU, P., KATSIRELOS, G. & CHRISSIKOPULOS, V. 1999. Mobile Agents for Secure Electronic Transactions. *Recent Advances in signal processing and communications*, World scientific and engineering society press:363-368.

KOTZANIKOLAOU, P., KATSIRELOS, G. & CHRISSIKOPOULOS, V. 2000. Secure Transactions with Mobile Agents in Hostile Environments. In: Dawson, E., Clark, A. & Boyd, C. (ed.). *Information Security and Privacy. ACISP.* Australia: LNCS 1841 (2000): 289-297.

LANGE, D.B. 1998. Mobile objects and mobile agents: the future of distributed computing. In: *Proceedings of the European Conference on Object Oriented Programming (ECOOP'98)*. Brussels, Belgium, Invited Talk.

LANGE, D.B. & OSHIMA, M. 1998. *Programming and Deploying Java^{TM} Mobile Agents with Aglets.* Massachusetts: Addison Wesley.

LOUREIRO, S., MOLVA, R. & ROUDIER, Y. 2000. Mobile Code Security. In: *Proceedings of ISYPAR 2000 Code Mobile.* Toulouse, France.

LUO, C. 2001. SAWMA, Secure and Automatic Wrapper for Mobile Agents. [Online]. Available at: <http://www.crcg.edu/research/projects/swama.php3> . Accessed: 02/09/02.

MARQUES, P., SANTOS, N., SILVA, L. & GABRIEL, J. 2001.The Security Architecture of the M&M Mobile Agent Framework. In: *Proceedings of the SPIE's International Symposium on The Convergence of Information Technologies and Communications. Denver, Colorado.*

MA, Q. & YEN, I. 2002. Secure and Survivable Agent Computation. Technical Report

UTD-CS-17-02. Department of Computer Science. University of Texas, Dallas.

MAKINO, S., OKOSHI, T., NAKAZAWA, J. & TOKUDA, H. 2000. S-agent: The Design of Secure Mobile Agent System. In: *IFIP Middleware*. New York

McDERMOTT, J. & GOLDSCHLAG, D. 1996. Storage Jamming. In: Spooner, D. Demurijan, S. & Dobson, J. (ed.). *Database Security IX: Status and Prospects.* Chapman & Hall.

MEADOWS, C. 1997. Detecting attacks on mobile agents. In: *DARPA Foundations for secure mobile code workshop.* Monterey, California.

MELL, P. & McLARNON, M. 1999. Mobile Agent Attack Resistant Distributed Hierarchical Intrusion Detection System. National Institute of Standards and Technology.

MILOJICIC, D., BREUGST, M., BUSSE, I., CAMPBELL, J., COVACI, S. FRIEDMAN, B., KOSAKA, K., LANGE, D., ONO, K., OSHIMA, M., THAM, C., VIRDHAGRISWARAN, S. & WHITE, J. 1998. MASIF: The OMG Mobile Agent System Interoperability Facility. *Personal Technologies*, 2(2):117-128.

MINSKY, Y., VAN RENESSE, R., SHEIDER, F., AND STOLLER, S. 1996. Cryptographic support for fault-tolerant distributed computing. In: *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems support for worldwide applications.* Connemara, Ireland:109-114.

MITROVIÆ, N. & ARRIBALZAGA, U.A. 2002. Mobile Agent security using Proxy-agents and Trusted domains. In: *Second International Workshop on Security of Mobile Multiagent Systems (SEMAS 2002).*

MOBILE AGENT LIST. [Online]. 2003. University of Stuttgart. Available at: <http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/preview/preview.html>. Accessed: 07/08/03.

MONTANARI, R., STEFANELLI, C. & DULAY, N. 2001. Flexible Security Policies for Mobile Agent Systems. *Microprocessors and Microsystems Journal, Elsevier Science.*

MUDUMBAI, S., ESSIARI, A. & JOHNSTON, W. 1999. Anchor Toolkit - A Secure Mobile Agent System. In: Springer. *Mobile Agents Conference.*

NALLA, A., HELAL, A. & RENGANARAYANAN, V. 2002. aZIMAs - almost Zero Infrastructure Mobile Agent System. In: *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC),* Orlando, Florida.

NECULA, G.C. & LEE, P. 1998. Safe, Untrusted Agents Using Proof-Carrying Code. In: Springer Verlag. *Mobile Agents and Security*. LNCS: 61-91.

OBJECT MANAGEMENT GROUP. 2000. *Agent Technology Green Paper*. Agent Working Group. OMG Document ec/2000-08-01.

OBJECTSPACE, INC. 1997. ObjectSpace Voyager Core Technology User Guide. [Online]. Available at: <http://www.objectspace.com/voyager/whitepapers/Voyager.pdf> Accessed: 27/03/03.

ORDILLE, J.J. 1996. When agents roam, who can you trust? In: *Proceedings of the First Conference on Emerging Technologies and Applications in Communications.*

ORSO, A. VIGNA, G. & HARROLD, M.J. 2001. MASSA: Mobile agent security through static / analysis. In: *Proceedings of the First ICSE Workshop on Software Engineering and Mobility (WSEM 2001).*

PAPAIOANNOU, T. 2000. On the Structuring of Distributed Systems: the Argument for Mobility. PhD Thesis. Loughborough University.

PASCOTTO, R. S.a. AMASE: Agent-based Mobile Access to Information Services. [Online]. Available at: <http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch3/amase.htm>

Accessed: 11/01/04.

PEINE, H. 1998. Security Concepts and Implementation in the Ara Mobile Agent System. In: *7th IEEE Workshop on Enabling Technologies: Infrastructure for collaborative Enterprises*, Palo Alto, California, USA.

PICCO, G.P. 1998. Understanding, Evaluating, Formalizing, and Exploiting Code Mobility. PhD Thesis, Politecnico di Torino, Italy.

PICCO, G.P. 2001. Mobile agents: an introduction. *Microprocessors and Microsystems*: 2(2): 65-74.

PIESSENS, F., DE DECKER, B., VON HOEYMISSEN, E. & NEVEN, G. 2000. On the trade-off between communication and trust in secure computations. In: *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*. France.

POSLAD, S., BUCKLE, P. & HADINGHAM, R. 2000. The FIPA-OS Agent Platform: Open Source for Open Standards. In: Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, UK: 355-368.

POSLAD, S. & CALISTI, M. 2000. Towards Improved Trust and Security in FIPA Agent Platforms. In: Autonomous Agents 2000 Workshop on Deception, Fraud and Trust in Agent Societies, Spain.

RIORDAN, J., AND SCHNEIER, B. 1998. Environmental key generation towards clueless agents. Vigna, G. (ed.) *Mobile Agents and Security, Springer Verlag. Lecture notes in computer science* no. 1419.

RIVEST, R.L., SHAMIR, A. & ADLEMAN, L.M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 2(21) (1978): 120-126.*

ROBLES, S., NAVARRO, G., PONS, J., RIFA, J. & BORRELL, J. 2002. Mobile Agents Supporting Secure GRID Environments. In: *Euroweb 2002 Conference. British Computer Society.* World Wide Web Consortium: 95-197.

ROTH, V. 1998. Secure Recording of Itineraries through Cooperating Agents. In: *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations:*147-154.

ROTH, V. 2001. On the Robustness of some cryptographic protocols for mobile agent protection. In: *Mobile Agent 2001, Technical Papers.*

ROTH, V. & JALALI. M.. 2001. Concepts and architecture of a security-centric mobile agent server. In: *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems, (ISADS 2001),* Dallas, Texas, U.S.A., March 2001:435-442.

SAMEH, A. & FAKHRY, D. 2002. Security in Mobile Agent Systems. In: *Proceedings of the Symposium on Applications and the Internet (SAINT 2002),* Japan.

SANDER, T. & TSCHUDIN, C.F. 1998. Protecting Mobile Agents Against Malicious Hosts. . In: Vigna, G. (ed.). *Mobile Agent Security.* Springer-Verlag, LNCS, 1419:44-60.

SCHNEIDER, F.B. 1997. Towards Fault-Tolerant and Secure Agentry. In: *Proceedings of the 11th International Workshop on Distributed Algorithms.*

SCHNEIER B. 2000. *Secrets and Lies*. *Digital security in a networked world*. John Wiley & Sons, Inc.

SCHÜTZ, F., GANNOUNE, L. & FRANCIOLI, J. 2000. Security Mechanisms for a Mobile Agent-Based Platform for Electronic Commerce of Internet Domain Names. In: *Proceedings of the 10th Annual Internet Society Conference.* Yokohama, Japan.

SMITH, D.C., CYPHER, A. & SPOHIER, J. 1994. Programming agents without a programming language. *Communications of the ACM*, 37(7):55-67.

SMITH, S.W. & AUSTEL, V. 1998. Trusting trusted hardware: Towards a formal model for programmable secure processors. In *3rd USENIX Workshop on Electronic Commerce*. Boston. Massachusetts.

SUNDSTED, T. 1998. An introduction to agents. Java World. [Online] Available at: <http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html>. Accessed: 5/4/2001.

SURI, N., BRADSHAW, J.M., BREEDY, M.R., GROTH, P.T., HILL, G.A., JEFFERS, R. & MITROVICH, T.S. 2000. An Overview of the NOMADS Mobile Agent System. In: Ciar & Bryce (eds.). *6th ECOOP Workshop on Mobile Object Systems.* France.

SWARUP, V. 1997. Trust Appraisal and Secure routing of Mobile Agents. In: *DARPA Workshop on Foundations for Secure Mobile Code*. Monterey, CA.

SWARUP, V. & FÁBREGA, J.T. 1999. Trust: Benefits, Models, and Mechanisms. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects.* Lecture Notes in Computer Science, Springer Verlag.

TAN, H.K. & MOREAU, L. 2002. Certificates for Mobile Code Security. In: *Proceedings of the 17th ACM Symposium on Applied Computing (SAC'2002):*76.

TARDO, J. & VALENTE, L. 1996. Mobile agent security and Telescript. In: *41st IEEE International Computer Conference,* San Jose, California.

TATE, S.R. & XU, K. 2003. Mobile Agent Security through Multi-Agent Cryptographic Protocols. In: *Proceedings of the 4th International Conference on Internet Computing*:462-468.

TOSHIBA. 2001. Bee-gent, Bonding and encapsulation enhancement agent. [Online]. Available at: <http://www2.toshiba.co.jp/beegent/index.htm>. Accessed: 11/01/04.

TRIPATHI, A.R., KARNIK, N.M., VORA, M.K, AHMED, T. & SINGH, R.D. 1999. Mobile Agent Programming in Ajanta. In: *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99).* Austin, Texas: 190-197.

TRIPATHI, A.R., AHMED, T. & KARNIK, N.M. 2001. Experiences and future challenges in mobile agent programming. *Microprocessors and Microsystems*, 25:121-129.

VAN'T NOORDENDE, G. BRAZIER, F.M.T. & TANENBAUM, A.S. 2002. A Security Framework for a Mobile Agent System. In: *Second International workshop on Security of Mobile Multiagent Systems.*

VARADHARAJAN, V. 2000. Security Enhanced Mobile Agents. In: *Proceedings of the seventh ACM conference on Computers and Communications Security.* Athens: 200-209.

VIGNA, G. 1998. Cryptographic Traces for Mobile Agents. *Springer-Verlag, LNCS,* 1419:137-153.

VIGNA, G., CASSELL, B. & FAYRAM, D. 2002. An Intrusion Detection System for Aglets. In: Suri, N (ed.). *Proceedings of the International Conference on Mobile Agents*. Barcelona, Spain.

VITEK, J., BRYCE, C. & BINDER, W. 1998. Designing JavaSeal or How to Make Java safe for Agents. In: Tsichritzis, D. (ed.). *Electronic Commerce Objects.* July:105-126.

VOGLER, H., KUNKELMAN, T. & MOSCHGATH, M. 1997. An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transactions. In: *International Conference on Parallel and Distributed Systems.* Seoul, Korea.

WESTHOFF, D. 2001. An Optimistic Third Party Protocol to protect a Mobile Agent's Binary Code. *International Journal of Software Engineering and Knowledge*

Engineering, 11(5):607-619.

WESTHOFF, D., SCHNEIDER, M., UNGER, C. AND KADERALI, F. 1999. Methods for Protecting a Mobile Agent's Route. In: *Proceedings of the Information Security Workshop,* LNCS 1729.

WHABE, R., LUCCO, S. & ANDERSON, T. 1993. Efficient Software-Based Fault Isolation. In: *Proceedings of the 14th ACM Symposium on Operating System Principles.* Asheville, NC: 203-216.

WILHELM, U.G., STAAMANN, S. & BUTTYAN, L. 1998. Protecting the Itinerary of Mobile Agents. In: *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*. Brussels:135-145.

WILHELM, U.G., STAAMANN, S. & BUTTYAN, L. 1999. On the problem of trust in mobile agent systems. In: *IEEE Network and Distributed Systems Security Symposium.* San Diego, California:11-13.

WILHELM, U.G., STAAMANN, S. & BUTTYAN, L. 1999a. Introducing trusted third parties to the mobile agent paradigm. In: J. Vitek, J & Jensen, C. (Eds.). *Secure Internet Programming: Security Issues for Mobile and Distributed Objects.* Springer-Verlag:471-491.

WILHELM, U.G., STAAMANN, S. & BUTTYAN, L. 2000. A Pessimistic Approach to Trust in Mobile Agent Platforms. *IEEE Internet Computing,* 4(5):40-48.

WOOLDRIDGE, M. 2002. *An Introduction to Multi-Agent Systems.* Chichesler, UK: John Wiley and Sons.

WU, C. 2000. Electronic Supermarket for Mobile Agents. In: *Proceedings of The 4th World Multi-Conference on Systematics, Cybernetics and Informatics (SCI2000),* Orlando, Florida, USA (3): 371-379.

YEE, B.S. 1997. A Sanctuary for Mobile Agents. In: *Proceedings of the DARPA Workshop on foundations for secure mobile code.*

YLITALO, J. 2000. Secure Platforms for Mobile Agents. *Web Report*. [Online]. Available at: <http://www/tml.hut.fi/Opinnot/Tik-110.501/1999/papers/mobileagents/mobileagents.html>. Accessed: 26/4/2002.

YOUNG, A. & YUNG, M. 1997. Sliding Encryption: A Cryptographic Tool for Mobile Agents. In: *Proceedings of the 4th International Workshop on Fast Software Encryption.*

ZAPF, M., MÜLLER, H. & GEIHS, K. 1998. Security requirements for Mobile Agents in Electronic Markets. In: *Working Conference on Trends in Distributed Systems for Electronic Commerce (TrEC'98)*, LNCS, Hamburg, Germany, Springer.

ZELTSER, L. 2000. The Evolution of Malicious Agents. Web Report. [Online]. Available at: <http://www.zelter.com/agents/>. Accessed: 05/07/02.

ADDENDUM A

Aglet Security Policy File

```
grant codeBase "atp://*:*/" {   permission java.io.FilePermission
"codebase", "read";   permission java.io.FilePermission "codebase",
"read, write,
execute";   permission java.net.SocketPermission "localhost:*",
"listen,resolve";   permission java.net.SocketPermission "codebase:
*", "connect";   permission java.awt.AWTPermission
"showWindowWithoutWarningBanner";   permission
java.util.PropertyPermission "awt.*", "read";   permission
java.util.PropertyPermission "hotjava.*", "read";   permission
java.util.PropertyPermission "apple.*", "read";   permission
java.util.PropertyPermission "file.*", "read";   permission
java.util.PropertyPermission "line.separator", "read";   permission
java.util.PropertyPermission "path.separator", "read";   permission
java.util.PropertyPermission "http.maxConnections",
"read";   permission java.util.PropertyPermission "user.timezone",
"read";   permission java.util.PropertyPermission "socksProxyHost",
"read";   permission java.util.PropertyPermission "socksProxyPort",
"read";   permission java.util.PropertyPermission "browser", "read";
permission java.util.PropertyPermission "java.rmi.*", "read";
permission java.util.PropertyPermission "sun.rmi.*", "read";
permission java.util.PropertyPermission "http.proxyHost", "read";
permission java.util.PropertyPermission "proxyHost", "read";
permission java.util.PropertyPermission "user.*", "read";
permission java.util.PropertyPermission "os.*", "read";   permission
java.util.PropertyPermission "java.*", "read";   permission
java.lang.RuntimePermission "createClassLoader";   permission
java.lang.RuntimePermission "accessClassInPackage.java.*";
permission java.lang.RuntimePermission
"accessClassInPackage.com.ibm.aglets.util.*";   permission
java.lang.RuntimePermission
"accessClassInPackage.com.ibm.aglets.AgletProxyImpl";   permission
java.lang.RuntimePermission "accessClassInPackage.com.ibm.aglet.*";
permission java.lang.RuntimePermission "loadLibrary.JdbcOdbc";
permission java.lang.RuntimePermission
"accessClassInPackage.sun.jdbc.odbc";   permission
java.lang.RuntimePermission "accessClassInPackage.java.security.*";
permission java.lang.RuntimePermission
"accessClassInPackage.java.security.spec.*";   permission
java.security.AllPermission "*", "*";   permission
com.ibm.aglets.security.AgletPermission "*",
"dispatch,dispose,deactivate,activate,clone,retract";   permission
com.ibm.aglets.security.MessagePermission "*", "*";   permission
com.ibm.aglets.security.ContextPermission "*", "multicast,subscribe";
permission com.ibm.aglets.security.ContextPermission "*",
"create,receive,retract";   permission
com.ibm.aglets.security.ContextPermission "property.*", "read,write";
protection com.ibm.aglet.security.AgletProtection "*",
"dispatch,dispose,deactivate,activate,clone,retract";   protection
com.ibm.aglet.security.MessageProtection "*", "*"; }; grant codeBase
"http://*:*/" {   permission java.io.FilePermission "codebase",
"read";   permission java.io.FilePermission "codebase", "read, write,
execute";   permission java.net.SocketPermission "localhost:*",
"listen,resolve";   permission java.net.SocketPermission "codebase:
```

```
*", "connect";   permission java.awt.AWTPermission
"showWindowWithoutWarningBanner";   permission
java.util.PropertyPermission "awt.*", "read";   permission
java.util.PropertyPermission "hotjava.*", "read";   permission
java.util.PropertyPermission "apple.*", "read";   permission
java.util.PropertyPermission "file.*", "read";   permission
java.util.PropertyPermission "line.separator", "read";   permission
java.util.PropertyPermission "path.separator", "read";   permission
java.util.PropertyPermission "http.maxConnections",
"read";   permission java.util.PropertyPermission "user.timezone",
"read";   permission java.util.PropertyPermission "socksProxyHost",
"read";   permission java.util.PropertyPermission "socksProxyPort",
"read";   permission java.util.PropertyPermission "browser", "read";
permission java.util.PropertyPermission "java.rmi.*", "read";
permission java.util.PropertyPermission "sun.rmi.*", "read";
permission java.util.PropertyPermission "http.proxyHost", "read";
permission java.util.PropertyPermission "proxyHost", "read";
permission java.util.PropertyPermission "user.*", "read";
permission java.util.PropertyPermission "os.*", "read";   permission
java.util.PropertyPermission "java.*", "read";   permission
java.lang.RuntimePermission "createClassLoader";   permission
java.lang.RuntimePermission "accessClassInPackage.java.*";
permission java.lang.RuntimePermission
"accessClassInPackage.com.ibm.aglets.util.*";   permission
java.lang.RuntimePermission
"accessClassInPackage.com.ibm.aglets.AgletProxyImpl";   permission
java.lang.RuntimePermission "accessClassInPackage.com.ibm.aglet.*";
permission java.lang.RuntimePermission "loadLibrary.JdbcOdbc";
permission java.lang.RuntimePermission
"accessClassInPackage.sun.jdbc.odbc";   permission
com.ibm.aglets.security.AgletPermission "*",
"dispatch,dispose,deactivate,activate,clone,retract";   permission
com.ibm.aglets.security.MessagePermission "*", "*";   permission
com.ibm.aglets.security.ContextPermission "*", "multicast,subscribe";
permission com.ibm.aglets.security.ContextPermission "*",
"create,receive,retract";   permission
com.ibm.aglets.security.ContextPermission "property.*", "read,write";
protection com.ibm.aglet.security.AgletProtection "*",
"dispatch,dispose,deactivate,activate,clone,retract";   protection
com.ibm.aglet.security.MessageProtection "*", "*"; }; grant codeBase
"file://-/" {   permission java.security.AllPermission "*", "*";
protection com.ibm.aglet.security.AgletProtection "*",
"dispatch,dispose,deactivate,activate,clone,retract";   protection
com.ibm.aglet.security.MessageProtection "*", "*"; }; grant codeBase
"file://c:/-" {   permission java.io.FilePermission "*", "read,
write, execute";   permission java.security.AllPermission "*", "*";
protection com.ibm.aglet.security.AgletProtection "*",
"dispatch,dispose,deactivate,activate,clone,retract"; };
```

ADDENDUM B

Source code for RetrievalAglet

```
 //An aglet of this class sends is send to two destinations to obtain information  //and store the
retrieved values in a file.  public class RetrievalAglet extends Aglet {   File dir = new File
("C:/data/testFile.dat");  File ResultsFile = new File("C:/data/ResultsFile.dat");  String from =
"Anti-Virus";  int finish = 0;    public void onCreation(Object init) {      addMobilityListener(    new
MobilityAdapter() {             public void onArrival(MobilityEvent b) {          finish++;    }    }  ); }
public void run() {   try {    switch (finish) {     case 0: NextDestination();      break;     case 1:
getInfo(dir,from);       NextDestination();       break;     case 2: getInfo(dir,from);
NextDestination();       break;     case 3: PrintResultsFile();       dispose();    }        } catch
(Exception e) {    System.out.println(e.getMessage());   }  }  //Read information file, get value,
write value to ResultsFile  void getInfo(File file, String from) throws IOException {   FileReader
readfile = new FileReader(file);   FileWriter writefile = new FileWriter(ResultsFile, true);   URL
RetrievalURL;   String name;   int number;   int tokentype;   AgletContext RetrievalContext =
getAgletContext();   RetrievalURL = RetrievalContext.getHostingURL();   StreamTokenizer
inputStream = new StreamTokenizer(readfile);   PrintWriter results = new PrintWriter(writefile);
tokentype = inputStream.nextToken();   results.println("Information search for "+from+" on host
"+RetrievalURL);     while (tokentype != StreamTokenizer.TT_EOF)   {    name =
inputStream.sval;     inputStream.nextToken();    number = (int)inputStream.nval;    if
(name.equals(from)) {     results.println(number+"\t");    }    tokentype = inputStream.nextToken
();   }   readfile.close();   writefile.close();  }   //print results retrieved on every host  void
PrintResultsFile() {   try {   FileReader readfile = new FileReader(ResultsFile);   String name;
int number, retNum = 2;   int tokentype;   StreamTokenizer inputStream = new StreamTokenizer
(readfile);   String contents=new String();   inputStream.ordinaryChars(0x00,0x7F);   tokentype =
inputStream.nextToken();      while (tokentype != StreamTokenizer.TT_EOF)   {
contents=contents+String.valueOf((char)tokentype);    tokentype = inputStream.nextToken();   }
readfile.close();    System.out.println(contents);  } catch (Exception e) {   System.out.println
(e.getMessage());} }   //NextDestination contains the list of hosts to be visited.  void
NextDestination() {   try {    URL destination;   switch (finish) {      case 0: destination = new
URL("atp://RemoteA.tut");          dispatch(destination);       case 1: destination = new URL
("atp://RemoteB.tut");          dispatch(destination);       case 2: destination = new URL
("atp://RemoteC.tut");          dispatch(destination);   }   }catch (Exception e) {   //Failed to
initialize next destination   System.out.println(e.getMessage());  } } }
```

ADDENDUM C

Source code for ComputationAglet

```
 public class ComputationAglet extends Aglet {  File dir = new File("C:/data/testFile.dat");  File
LowestBidFile = new File("C:/data/LowestBidFile.dat");  String from = "Anti-Virus";  int finish =
0;  URL LowestBidURL;  int LowestBid = 10000;    public void onCreation(Object init) {
addMobilityListener(     new MobilityAdapter() {              public void onArrival(MobilityEvent b) {
finish++;     }    }   );   }   public void run() {    try {    switch (finish) {     case 0: NextDestination();
break;     case 1: getLowestBid(dir,from);       NextDestination();       break;     case 2:
getLowestBid(dir,from);       NextDestination();       break;     case 3: PrintResultsFile();
dispose();    }          } catch (Exception e) {    System.out.println(e.getMessage());   }  }  //Read
file, get value, determine lowest bid and write to file  void getLowestBid(File file, String from)
throws IOException {   FileReader readfile = new FileReader(file);   FileWriter writefile = new
FileWriter(LowestBidFile);   String name;   int number;   int tokentype;   int LowestBid = 10000;
AgletContext LowestBidContext = getAgletContext();   StreamTokenizer inputStream = new
StreamTokenizer(readfile);   PrintWriter results = new PrintWriter(writefile);   tokentype =
inputStream.nextToken();   while (tokentype != StreamTokenizer.TT_EOF)   {    name =
inputStream.sval;     inputStream.nextToken();    number = (int)inputStream.nval;    if
(name.equals(from)) {      if (number < LowestBid) {      LowestBid = number;      LowestBidURL
=                                                   LowestBidContext.getHostingURL();
results.println(LowestBidContext.getHostingURL());      results.println(number);     }    }
tokentype = inputStream.nextToken();   }   readfile.close();   writefile.close();  }     void
NextDestination() {   try {    URL destination;   switch (finish) {    case 0: destination = new URL
("atp://RemoteA.tut ");     dispatch(destination);    case 1: destination = new URL
("atp://RemoteB.tut ");     dispatch(destination);    case 2: destination = new URL
("atp://RemoteC.tut ");     dispatch(destination);     }   }catch (Exception e) {   //Failed to
initialize next destination   System.out.println(e.getMessage());  } }    //print results retrieved on
every host  void PrintResultsFile() {   try {   FileReader readfile = new FileReader(LowestBidFile);
String name;   int number;   int tokentype;   StreamTokenizer inputStream = new
StreamTokenizer(readfile);   String contents=new String();   inputStream.ordinaryChars
(0x00,0x7F);   tokentype = inputStream.nextToken();       while (tokentype !=
StreamTokenizer.TT_EOF)   {    contents=contents+String.valueOf((char)tokentype);
tokentype = inputStream.nextToken();   }    readfile.close();    System.out.println(contents);  }
catch (Exception e) {   System.out.println(e.getMessage());} }  }
```

ADDENDUM D

Source code for implementation of Path Histories

```
 //Sign Itinerary void CreateSignature() {  try {  FileOutputStream signaturefile = new
FileOutputStream(signature);  FileOutputStream keyfile = new FileOutputStream(keys);
//Create a keypair generator.  KeyPairGenerator generatekey =
KeyPairGenerator.getInstance("DSA","SUN");     //Initialize the keypair generator
SecureRandom random = SecureRandom.getInstance("SHA1PRNG",
"SUN");  generatekey.initialize(1024, random);     //Generate the keypair  KeyPair getkeys =
generatekey.generateKeyPair();  PrivateKey private_key = getkeys.getPrivate();  PublicKey
public_key = getkeys.getPublic();     //Sign the data   Signature algorithm =
Signature.getInstance("SHAwithDSA", "SUN");     //Initialise the signature object
algorithm.initSign(private_key);     //Supply the signature object to the data to be signed
FileInputStream pathHistory = new FileInputStream(Itinerary);  BufferedInputStream bufferin =
new BufferedInputStream(pathHistory);  byte[] buffer = new byte[1024];  int length;  while
(bufferin.available() != 0) {   length = bufferin.read(buffer);   algorithm.update(buffer,0,length);  };
bufferin.close();     //Generate the signature  byte[] createsignature = algorithm.sign();     //Save
the Signature & Public key in files  signaturefile.write(createsignature);  signaturefile.close();
//Save public key in file  byte[] key = public_key.getEncoded();  keyfile.write(key);  keyfile.close
();          } catch (Exception e) {             System.err.println("Caught exception " + e.toString());
}     }
```

```
 //Remote host verifies previous host void VerifySignature() {      try {      //Read in the encoded
public key bytes.                FileInputStream keyfile= new FileInputStream(keys);
//The byte-array encryptionkey contains the encoded public key bytes             byte[]
encryptionkey = new byte[keyfile.available()];                keyfile.read(encryptionkey);
keyfile.close();                   //Key specification  X509EncodedKeySpec publicspec = new
X509EncodedKeySpec(encryptionkey);  KeyFactory keyFactory = KeyFactory.getInstance
("DSA", "SUN");    //Generate a public key  PublicKey public_key = keyFactory.generatePublic
(publicspec);    //Input signature bytes  FileInputStream signaturefile = new FileInputStream
(signature);             byte[] verifysignature = new byte[signaturefile.available()];
signaturefile.read(verifysignature);           signaturefile.close();             //Initialise the signature
object for verification.   Signature signature = Signature.getInstance("SHA1withDSA", "SUN");
signature.initVerify(public_key);    //Signature verification  FileInputStream datafile = new
FileInputStream(Itinerary);  BufferedInputStream bufferin = new BufferedInputStream(datafile);
byte[] buffer = new byte[1024];  int length;  while (bufferin.available() != 0) {      length =
bufferin.read(buffer);      signature.update(buffer, 0, length);  };    boolean verifies =
signature.verify(verifysignature);   System.out.println("signature verifies: " + verifies);
bufferin.close();          } catch (Exception e) {             System.err.println("Caught exception " +
e.toString());         }     }
```

ADDENDUM E

Source code for partial result encapsulation

```
public void EncryptFile() {  try {  switch (finish) {  case 1: Cipher des_encrypt1;   //Create a
DES key   KeyGenerator generateKey1 = KeyGenerator.getInstance("DES");
//SecretKey   key_for_des1 = generateKey1.generateKey();    //Create the cipher
des_encrypt1 = Cipher.getInstance("DES");      //Initialise cipher for encryption
des_encrypt1.init(Cipher.ENCRYPT_MODE,key_for_des1);     //Create cipher stream   File
ResultsFile1 = new File("C:/data/ResultsFile.dat");   FileInputStream fileInput1 = new
FileInputStream(ResultsFile1);   CipherInputStream cis1 = new CipherInputStream(fileInput1,
des_encrypt1);   FileOutputStream writefile1 = new FileOutputStream(EncryptedResults, true);
//encrypt data   byte[] buffer1 = new byte[8];   int length1 = cis1.read(buffer1);   while (length1 !=
-1) {    writefile1.write(buffer1, 0, length1);    length1 = cis1.read(buffer1);   };       writefile1.close
();    fileInput1.close();         case 2: Cipher des_encrypt2;   //Create a DES key   KeyGenerator
generateKey2 = KeyGenerator.getInstance("DES");     //SecretKey   key_for_des2 =
generateKey2.generateKey();      //Create the cipher   des_encrypt2 = Cipher.getInstance
("DES");      //Initialise cipher for encryption   des_encrypt2.init
(Cipher.ENCRYPT_MODE,key_for_des2);      //Create cipher stream   File ResultsFile2 = new
File("C:/data/ResultsFile.dat");   FileInputStream fileInput2 = new FileInputStream(ResultsFile2);
CipherInputStream cis2 = new CipherInputStream(fileInput2, des_encrypt2);   FileOutputStream
writefile2 = new FileOutputStream(EncryptedResults, true);     //encrypt data   byte[] buffer2 =
new byte[8];   int length2 = cis2.read(buffer2);   while (length2 != -1) {   writefile2.write(buffer2, 0,
length2);   length2 = cis2.read(buffer2);   };    writefile2.close();    fileInput2.close();  }  } catch
(Exception e) {             System.out.println(e.getMessage());        }  }
```

```
void DecryptFile() {      Cipher des_decrypt1;      Cipher des_decrypt2;           try {
des_decrypt1 = Cipher.getInstance("DES");  des_decrypt1.init
(Cipher.DECRYPT_MODE,key_for_des1);             //Create decrypted file  FileInputStream
encryptedFile1 = new FileInputStream(EncryptedResults);  FileOutputStream writefile1 = new
FileOutputStream(DecryptedResults);  CipherInputStream cis1 = new CipherInputStream
(encryptedFile1, des_decrypt1);     int total;   //decrypt data  byte[] buffer1 = new byte[8];  int
length1 = cis1.read(buffer1);  total = length1;  while (length1 != -1) {   writefile1.write
(buffer1,0,length1);   length1 = cis1.read(buffer1);   total = total + length1;      };
encryptedFile1.close();               des_decrypt2 = Cipher.getInstance("DES");  des_decrypt2.init
(Cipher.DECRYPT_MODE,key_for_des2);             //Create decrypted file  FileInputStream
encryptedFile2 = new FileInputStream(EncryptedResults);  CipherInputStream cis2 = new
CipherInputStream(encryptedFile2, des_decrypt2);     //decrypt data  byte[] buffer2 = new byte
[8];  int length2 = cis2.read(buffer2);  while (length2 != -1) {   writefile1.write(buffer2,0,length2);
length2 = cis2.read(buffer2);      };  writefile1.close();  encryptedFile2.close();           } catch
(Exception e) {             System.out.println(e.getMessage());        } }
```