M.Tech. Dissertation

Security Issues in Mobile Agents

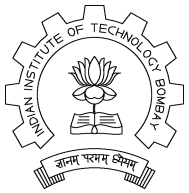
Submitted in partial fulfillment of requirements
for the degree of

Master of Technology

By

E. C. Vijil
Roll No. : 00329014

Under the guidance of
Prof. Sridhar Iyer



Kanwal Rekhi School of Information Technology
Indian Institute of Technology, Bombay
Mumbai, 400 076

Abstract

An autonomous mobile agent is an executing program that can migrate from machine to machine in a heterogeneous network under its own control. An agent can either follow a pre-assigned path on the network or determine its itinerary based on the data collected from the network. Facilities for highly dynamic movement of code and data enables a program to take advantage of the locality of data. It also allows one to optimize between the requirements of low bandwidth, high latency and disconnected network connections.

This computing paradigm which exploits code, data and state mobility raises many new security issues, which are quite different from conventional client/server systems. Agent servers which provide an execution environment for the agents to execute can be attacked by malicious agents. Similarly agents could be carrying sensitive information about their owners and should be protected from tampering by malicious hosts. Also, the data collected by the agent from one host should be protected from tampering by another host in the itinerary.

In this report, we examine the various security issues that arise in mobile agents in general with special reference to data collection agents. We propose an algorithm to identify the malicious host modifying the data in data collection agents. Multiple hosts can collude to remove the data collected by the agent from previous hosts. We give a probabilistic collusion detection algorithm to detect deletion of data by colluding malicious hosts.

Acknowledgments

I would like to express my gratitude to Prof. Sridhar Iyer for his continuing guidance, encouraging words and support. Thanks are also due to Vikram Jamwal, Pradnesh Rane and Aneesh V for the fruitful discussions that we had on the subject.

Contents

Chapter 1

Introduction

1.1 The Mobile Agent Paradigm

Over the years, computers have evolved from huge monolithic devices with very little memory to client–server environments that allow complex and varied forms of distributed computing. From remote job–entry terminals to Java applets, from magnetic tapes to distributed databases, various limited forms of code and data mobility have always existed. Mobile agent is a recent computing paradigm which allows complete mobility of cooperating applications to supporting platforms to form a loosely-coupled distributed system.

A mobile agent can be thought of as a software program, which can travel from one place to another. The agent initiates the trip by executing a "go" instruction which takes as an argument the name or address of the destination. The next instruction in the agent's program is executed in the destination machine. In other words, a mobile agent is not bound to the system where it begins execution. It 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 the object with which the agent wants to interact, and then to take advantage of being in the same host or network as the object [13, 14, 2]. A good survey of the mobile agent design paradigm, different agent frameworks and possible applications can be found in [6]. [27] gives common pitfalls in agent-oriented software development.

Various forms of mobility exist in mobile code. *Strong mobility* is the ability of a mobile code system to allow migration of both the code and the execution state of an execution unit to a different computation environment.

On the other hand, *weak mobility* is the ability of a mobile code system to allow code transfers across different computing environments. The code can contain some initialization data, but no execution state is transferred.

1.2 Mobile Agent Applications

In this section, we give some applications of mobile agents and show how they offer a convenient and efficient methodology for designing distributed applications.

Almost everything that can be done using mobile agents can also be done using the conventional client/server programming model. In short, there is no 'killer application' for mobile agents. However, advocates of the mobile agent paradigm, look at mobile agents as a technology that can solve a lot of problems in a uniform and efficient manner rather than as a technology that enables new things that weren't possible in any other way [12].

The three main domains where agents can be put to use are described in this section [10].

Data Intensive Operations

Mobile Agents can be effectively used when a user with specialized needs wants to use large amounts of data located at a remote site. As an example, a user might want to count the occurrences of the word, 'tamil', in all the postings of soc.culture.indian newsgroup. This requirement is very specific and it is unlikely that any service provider will have a ready-made application for this purpose. Also, transferring all the postings of the particular newsgroup, to the local site may not be practical. In this scenario, an agent programmed for the specific task can be deployed. Significant savings in bandwidth can be achieved with very little application support from the server.

Disconnected Operations

Mobile agents are very useful when support for disconnected operations are required. Appliances like cellphones and PDAs could be connected to the network through wireless links. They can release an agent programmed to do a particular task into the network and disconnect themselves from the network. The agent can complete its task autonomously and wait for the device to connect again to deliver the results.

Dynamic Deployment of Software

An organization might have hundreds of PCs or PDAs which need to be configured with a new version of software. A mobile agent can be deployed with the configuration logic to all these devices. No special support (except that of an agent execution environment) is required on these devices. Another possibility is to have these devices send agents that wait for interesting events to happen at the remote server. As an example, an agent can be deployed to notify a user through his cellphone whenever there is a stock market crash.

1.3 Security Issues

Mobile agent systems provide a computing infrastructure upon which mobile agents belonging to different and potentially untrusted users can execute. The communication medium is inherently insecure and the different agents and agent systems may have conflicting objectives. In this scenario, a variety of attacks can be conceived. Unauthorized users may eavesdrop network traffic and observe agents in action, or worse an active intruder may modify the code, data or state of an agent in transit. Agents may attack the agent platform (host) supporting their execution to gain unauthorized access to resources. Note that some of these problems are typical in conventional distributed (client/server) systems and are solved by cryptographic techniques. But these techniques, should be adopted for use in mobile code systems.

A new security issue is introduced in mobile code systems: Protection of mobile agents from malicious sites. Sites could tamper with agents' code or state, forcing them to disclose or change private information, run a program with different initial states multiple times and observe the results, "brainwash" them to attack other agents/hosts, delay access to specific and critical resources or deny services to enable other hosts or agents to gain unfair advantage. It is also possible for a group of agents and hosts to act together toward the realization of the above goals.

Mobile code system security is very difficult to achieve because it violates many fundamental assumptions that exist in most existing computer security measures [3]. For example, an important assumption is that a user will only execute programs which came from trusted, identifiable sources. And whenever, the program attempts to perform an operation, it is assumed that the user running the operation intends to perform that operation. Trojan horses are a notable exception to this assumption. But there is no general solution to the menace of Trojan horses, except that the user should be

careful as to run only programs which come from trusted sources. This is no longer possible to ensure in mobile code systems. It may not be possible or desirable for a host to monitor the actions performed by an agent arriving in its execution environment.

Another important problem is that of authentication. Methods based on cryptography exist to ensure that a given person really sent a given message and that the message was not altered in transit (see for instance [20]). But solutions that work well in systems, with a small number of users do not scale well to mobile code systems, where transactions involving strangers occur frequently. If we are using public key cryptography for digital signatures, then we should have reliable public key distribution mechanisms. But no general and scalable solutions for the key distribution problem are known. Besides, for a process which is migrating, determining the components in the program which should be signed, and who needs to sign what is a major semantic issue.

1.4 Scope of the Project

In this project, we look into the general issues in mobile agent security, paying special attention to the problem of malicious hosts. A classification of threats is given and some suggested solutions are examined in detail.

Then, we discuss the various security issues that arise in 'data collection agents' – agents which visit various hosts to collect data from them. In particular, the principle of AppendOnlyContainer, which can be used to detect malicious modifications and deletions of data collected by the agent, is described. The AppendOnlyContainer can only detect that a tampering has been made. It cannot detect the host which tampered with the data. We propose an extension to the AppendOnlyContainer mechanism, which can identify the malicious host that tampered with the data.

Multiple hosts can collude against an agent to delete data items added by other hosts. The AppendOnlyContainer mechanism will fail to detect these deletions. We examine the problem of collusions in a variety of scenarios, in sufficient detail. A trivial solution is available to detect such deletions if the agent uses a static itinerary. However, the situation is complicated if the agent decides on the itinerary dynamically during the course of its travel. Deletion of data can be detected if we allow each host to communicate directly with the agent initiator. But, this increases the message overhead and makes disconnected operation of the agent initiator difficult. No general solution is known to detect collusion in dynamic itineraries.

We introduce a probabilistic scheme to handle this problem. Depending on the agent initiator's prior knowledge of the itinerary and threat perceptions, two methods to detect malicious deletions are given. One of these methods can be used to detect collusions without sacrificing the ability of the agent initiator to disconnect from the network while the agent is in motion.

1.5 Organization of this Report

Chapter 2 classifies the threats in mobile code systems and illustrates why conventional cryptographic techniques are insufficient. Some countermeasures suggested in the literature are also explained. The security issues in data collection agents and our extension to the *AppendOnlyContainer* mechanism are given in chapter 3. Chapter 4 explains the problem of colluding malicious hosts and our algorithm to detect such collusions. Our simulation of this algorithm is explained in Chapter 5. Chapter 6 concludes this report and suggests future work.

Chapter 2

Security Threats and Counter Measures

2.1 Classification of Security Threats

Threats in mobile code systems can be broadly classified as: threats emanating from an agent attacking an agent platform, an agent platform attacking an agent,and an agent attacking another agent on the agent platform [9]. Various aspects and implications of these threats are explored in detail in the following sections.

2.1.1 Platform-to-agent

This category represents the class of threats where hosts compromise the agents. The set of threats include masquerading, denial of service, eavesdropping, and alteration. These attacks are most difficult to detect and prevent, since the host has full control of the agents' code and data.

Masquerading

An agent platform can masquerade as another agent platform in an attempt to deceive a mobile agent as to its true destination. As an example, a mobile agent entrusted with the task of finding the "lowest price" of a commodity by visiting various virtual shops, can be fooled by a malicious masquerading platform, by making it believe that all other shops have quoted a higher price. Thus, the masquerading platform can harm both the visiting agent and other agent platforms.

Denial of Service

A malicious agent platform may ignore service requests, introduce unacceptable delays during the execution of time critical tasks or even terminate the agent without notification. Agents on other platforms waiting for the results of a non-responsive agent can become deadlocked. An agent can also become livelocked if more work is continuously generated for the agent by the malicious platform.

Eavesdropping

The classical threat of eavesdropping in electronic communication is more serious in mobile agent systems because an agent platform can, not only monitor communications, but also every instruction executed by the agent, all unencrypted or public data it brings to the platform and all data generated on the platform. An agent may be exposing proprietary algorithms, trade secrets or other sensitive information. Even if the platform is unable to automatically extract the secret information, it may be able to infer the meaning from the types of services requested. For example, an agent may be communicating with a travel agent. Though the exact details of the communication is unknown to the platform, the communication may indicate that the person on whose behalf the agent is acting is planning a trip and will be away from home in the near future. The platform may sell this information to a suitcase manufacturer who may begin sending unsolicited advertisements, or even worse, the platform may share this information with thieves who may target the home of the traveler [9].

Alteration

Alteration includes modification of data, state and code. Modification cannot be prevented but it should be possible for another agent or platform to detect unauthorized modifications. Modification is typically prevented by using digital signatures. But digital signatures are useful only for signing code and static data. The original author can digitally sign the agent's code and read–only data. Signatures cannot be used to detect malicious modifications to dynamic data modified at different hosts. A host can change the data generated by other hosts in the agent's itinerary. Such changes, if not immediately detected, will be impossible to track down after the agent has visited other platforms and undergone countless changes in state and data. Check–pointing and rollback will be extremely difficult because an

agent's final state and data might be dependent on the behavior of countless autonomous agents whose behavior cannot be recreated. In other words, re-execution will not always yield same results. Also, one cannot depend too much on the virtual machine of the malicious platform because the platform could be running a modified version of the virtual machine.

2.1.2 Agent-to-platform

This category represents the set of threats in which agents exploit security weaknesses of an agent platform or launches attacks against an agent platform. Often, an agent platform is required to execute programs from potentially untrusted sources. This means that we can no longer say, "don't download and run untrusted programs", any more. These threats include masquerading, denial of service and unauthorized access.

Masquerading

An agent can take the identity of another agent to gain unauthorized access to resources or to shift the blame for any actions for which it does not want to be held accountable. The trust of an agent or the owner of an agent can be destroyed by a masquerading agent.

Denial Of Service

Mobile agents can launch a denial of service attack by consuming excessive amounts of the platform's computing resources. There is also the possibility of a program consuming excessive resources due to bugs in the program. Practices to help mitigate these risks like configuration management, design reviews and testing are not immediately applicable to mobile code systems because the mobile computing paradigm requires a platform to accept and execute an agent whose code may have been developed outside its organization and has not been subject to any a priori review.

Unauthorized access

Access control mechanisms are used to prevent unauthorized users from accessing resources. Resource allocation should be done in accordance with the platform's security policy. Authentication is used to identify an agent by the agent platform. How to authenticate and trust an agent which might have visited many untrusted hosts in its itinerary is a serious issue.

2.1.3 Agent-to-Agent

This category represents the set of attacks in which agents exploit security weaknesses of other agents or launch attacks against other agents. These threats also include masquerading, denial of service, unauthorized access and repudiation.

Masquerading

An agent can pose as a platform offering some services to another agent. For example, an agent can pose as a "virtual shop" offering various goods and services, and fool another agent into revealing credit card numbers. As usual, masquerading harms both the agent that is deceived and whose identity has been assumed.

Denial of Service

It is possible for a malicious agent to launch a denial of service attack against other agents. For example, repeatedly sending messages to another agent will cause undue burden on the message handling routines of the recipient. If the agent is charged based on its utilization of CPU resources, this could lead to a potential monetary loss to the victim.

Unauthorized access

If the access control mechanism of the platform is poor, then an agent can directly invoke the public methods of other agents, or modify its code and data. Eavesdropping by agents is also an issue.

2.2 Counter Measures

2.2.1 Conventional Approaches

Conventional techniques for achieving security in distributed systems can also be put to use in mobile code systems. Public key cryptography, digital signatures and session keys can be used for achieving various security goals. As mentioned earlier, these methods cannot be directly applied but must be adapted for use in mobile code systems.

To illustrate this point, difficulties in using traditional digital signatures for authenticating mobile agents is explained in detail. A user can digitally sign an agent on its home platform before it moves into a second platform.

The second platform on receiving the agent can use this signature to verify the integrity of the agent's code, data and state. On the agent's subsequent hop to the next platform, the initial signature from the first platform remains valid for the original code, data and state information. But nothing can be said about the state or data generated on the second platform. The second platform can optionally sign the state and data generated on the second platform. But if the third platform does not trust the second platform, then this signature does not mean much because the second host could have modified the state and data arbitrarily before signing. Hence, even if the third platform trusts the original agent, it cannot allow the agent to execute with its complete privileges because the agent has passed through an untrusted platform.

2.2.2 Protecting the Agent from Malicious hosts

Since platforms and hosts have complete control over the agents using their executing environments, counter measures for protecting agents rely mainly on detection measures to act as a deterrent. Some general techniques for protecting an agent are explained below.

Computing with Encrypted Functions

Computing with Encrypted Functions [19] explores a method to ensure computation privacy of mobile code in an untrusted host. It also explains a method by which a mobile agent remotely signs a document without disclosing the user's private key.

The problem of computation privacy arises because of the inability of the agent to prevent disclosure of the program it wants to have executed. If we have a method by which we can encrypt a program, and let a platform execute it without the need for decryption, this problem is essentially solved. In particular, we have code privacy and code integrity in the sense that specific tampering is not possible. The problem is stated by Sander and Tschudin as follows:

> Alice has an algorithm to compute a function f. Bob has an input x and is willing to compute $f(x)$ for her, but Alice wants Bob to learn nothing substantial about f. Moreover, Bob should not need to interact with Alice during the computation of $f(x)$.

We can distinguish between a *function* and the *program* that implements it. Functions can be encrypted such that their transformation can again be

implemented as programs. The resulting program will consist of cleartext instructions that a processor understands. But the processor will not understand the "program's function".

Suppose that we can transform the function f to some other function $E(f)$. Let $P(f)$ denote the program which implements the function f. A protocol for non-interactive computing with encrypted functions is given below.

1. Alice encrypts f.
2. Alice creates a program $P(E(f))$ which implements $E(f)$.
3. Alice sends $P(E(f))$ to Bob.
4. Bob executes $P(E(f))$ at x.
5. Bob sends $P(E(f))(x)$ to Alice.
6. Alice decrypts $P(E(f))(x)$ and obtains $f(x)$.

A general scheme based on the above idea is described here. We encrypt a polynomial f by composing it with another function s. Assume f is a rational function (the quotient of two polynomials) and s is also a rational function that is easily invertible. Now, let $E(f) := s \circ f$. For decrypting, Alice inverts s and calculates $s^{-1} \circ E(f)$.

Decomposition Problem: Given a multivariate rational function h that is known to be decomposable, find s and f such that $h = s \circ f$.

No polynomial time algorithm for decomposing multivariate rational functions is known. And, ways to generate rational functions s for encrypting f that are easy to invert have been proposed by Shamir [21].

If f is a signature algorithm with an embedded key, the agent can sign a document without the platform discovering the key. Similarly, if f is an encryption algorithm containing an embedded key, the agent can encrypt information at the platform.

The idea is elegant and the challenge is to find appropriate encryption schemes that can transform arbitrary functions. Algebraic homomorphic encryption schemes have been proposed as a possible candidate. A lot of work needs to be done in this field to explore various possibilities.

Environmental Key Generation

Environmental Key Generation [17] allows an agent to take a particular course of action when an environmental condition becomes true. The idea is similar to the notion of ephemeral keys. The key is randomly created at the time of use and destroyed immediately afterward. In this setup, the agent will have encrypted data/code and a method to search through the environment for the data needed to generate the decryption key. The data can be located at Usenet news groups, web pages etc. When the proper environmental information is located, the key is generated and the encrypted code is decrypted and acted upon. Without the environmentally supplied input, the agent cannot decrypt its own message. In other words, it is clueless as to its own function and is thus resistant to analysis aimed at determining its function.

Let N be an integer corresponding to an environmental observation, H a one–way hash function, and M the hash of the observation N needed for activation. Note that, N is not known to the agent and only $H(N)$ is known. We can have a simple condition like

$$\text{if } H(N) = \text{M then } K := N$$

to generate the key. This technique ensures that a platform or any eavesdropper cannot uncover the triggering message or response action by directly reading the agent's code.

This approach has several weaknesses. A platform can easily modify the agent to delay the execution of code after it has been decrypted and the code can be thoroughly analyzed before execution. Also, some platforms limit the capability of an agent to execute dynamically created code for security reasons.

Partial Result Encapsulation

An approach used to detect tampering by malicious hosts is to encapsulate the results of an agent's action, at each platform visited for subsequent verification. The verification can be done either when the agent returns to the point of origin or at intermediate points. Encapsulation could use different mechanisms for different objectives. Digital signatures can be used for integrity and accountability and encryption for ensuring confidentiality. The encapsulation can be done by the agent, platform or by a trusted third party. Though the solution is not general, it allows certain types of tampering to be detected.

Mutual Itinerary Recording

An agent's itinerary can be recorded and tracked by another cooperating agent and vice versa, in a mutually supportive arrangement [18]. When an agent moves to another platform, an agent conveys the last platform, current platform and next platform information to the cooperating peer through an authenticated channel.

Hosts are given a *color*. Given a particular host, a *white* host is completely trusted. Hosts which might be malicious are *grey*. Those hosts which might collaborate with at least one other host in an attack against a mobile agent is *red*. The security of mobile agents against attacks by malicious hosts can be improved by distributing critical operations of a mobile agent between two cooperating agents, each of which operates in one of two disjunct nonempty sets of hosts H_a and H_b which holds the following condition:

> No red host of either set is willing to cooperate with a red host of the other set.

The idea is to split or secretly share data that might be stolen by a single host.

Itinerary Recording with Replication and Voting

Rather than a single copy of an agent performing a computation, multiple copies are used. Even if a malicious agent corrupts a few copies, enough replicates survive to successfully complete the operation. This approach is suitable where agents can be duplicated without problems and survivability is the major concern.

Obfuscated Code

Most mobile agent frameworks [7, 26, 22, 24], use the Java programming language because of its portability and platform independence. The Java source code is converted into platform independent bytecode (class files) which retains most of the information in the source code [16]. Standard techniques in compiler optimization like dataflow analysis and control flow analysis (see [1]) can be used to get valuable information about the structure and semantics of programs. Mocha [25] was one of the first decompilers for Java. Disassemblers and decompilers have even been constructed for purely binary programs [4].

An agent is a blackbox if its code and data cannot be read or modified at any time [8]. The main problem is that there is no known method or

algorithm for providing blackbox protection. Computing with encrypted functions is cited as an example, but serious reservations about the limited range of input specifications that apply are raised. A time limited blackbox implies that code or data of an agent cannot be read or modified within a known time interval and that after the interval, *the attacks do not have effects*. For example we can declare that *electronic coins* become invalid after an expiry date. If such a quality can be ensured, then strong encryption algorithms are not required and obfuscation algorithms will suffice.

The idea is to transform a program into another program which is difficult to understand, but functionally identical. A taxonomy of obfuscating transformations can be found in [5]. The implementation of a Java obfuscator is explained in [15].

A main drawback of this kind of solution is the lack of an approach for quantifying the protection interval provided by the obfuscation algorithm. Further, no techniques are known for establishing the lower bounds on the complexity for an attacker to reverse engineer an agent's code.

2.2.3 Protecting the Agent Platform

An agent should not be allowed to harm the agent platform or other agents. Techniques for protecting the Agent Platform from a malicious agent is explained in this section.

Software-Based Fault Isolation

This is a method of isolating application modules into distinct fault domains enforced by software. A module can do whatever it wants in its fault domain but, it may not write or call (or optionally read) an address outside its fault domain except through an explicit cross-fault-domain call. This technique is known as sandboxing. Fault isolation can also be done in hardware by putting each module in a different address space but this is very expensive because there is a lot of runtime overhead for each context switch.

Safe Code Interpretation

Mobile code systems are often developed using interpreted script or programming language because this can support agent platforms on heterogeneous computer systems. Java and Tcl are the most common examples. Security features can be built into the interpreter to prevent possible misuse.

The Java programming language has security features built into its interpreter. It uses the sandbox security model to isolate memory and method

access. Security is ensured through a variety of mechanisms. Static type checking in the form of byte code verification is done before execution. Dynamic checking is done at runtime. A separate namespace is maintained for untrusted downloaded code. A security manager mediates all accesses to system resources. Support for dynamic code downloading, digitally signed code, remote method invocation, object serialization and platform heterogeneity make it an ideal platform for agent development.

Safe Tcl is a scripting language used in the early development of the Agent Tcl System. In Safe Tcl, a second "safe" interpreter pre-scans any harmful commands from being executed by the main Tcl interpreter. Different safe interpreters can be used to enforce varying security policies.

Signed code

A digital signature can be used as a means of confirming the authenticity of an object, its origin and its integrity. The semantics of a signature varies widely. For example, a signature can indicate the author of the code, but not that the code will perform without fault or error.

We examine the security issues in data collection agents and some solutions in the next chapter.

Chapter 3

Security in Data Collection Agents

Data collection agents are mobile agents which visit various hosts to collect data from them. For example, a 'shopping agent' could visit various vendors of a product to collect quotations for the product. Mobile agents are especially suitable for such applications because the initiator of the agent need not remain connected to the network after sending the agent.

The Ajanta Mobile Agent System [24, 23] has extensive support for Data Collection Agents. They provide three containers - *ReadOnlyContainer*, *AppendOnlyContainer* and *TargetedState* for secure transmission of data [11]. In this chapter we review the security issues in data collection agents, with special reference to the features offered by the Ajanta System.

3.1 Modification of Data by Malicious Hosts

Often, a mobile agent has to visit a sequence of hosts and malicious modifications to data generated by one host, by another host needs to be detected. As an example, a shopping agent may migrate to various vendor's servers, collecting quotations for goods to be purchased. Any modifications done by a vendor to the quotations collected from an earlier vendor must be detected.

The *ReadOnlyContainer* mechanism in the Ajanta System can be used to detect such modifications.

3.1.1 ReadOnlyContainer

Any modifications made to data in the *ReadOnlyContainer* can be detected by subsequent hosts. A simple digital signature mechanism is used to achieve this. The data to be protected is signed by the host adding it. Any other host can verify the signature.

Note that declaring the data as a constant in the programming language used to code agents is not sufficient. For example an object declare as *final* (in the Java programming language), could be modified by another host running a non-standard version of the Java Virtual Machine.

3.2 Deletion of Data by Malicious Hosts

Often, an agent needs to collect data from the sites it visits and wishes to detect any modification or deletion of data by subsequent sites in the itinerary. The *AppendOnlyContainer*, of the Ajanta System uses a combination of digital signatures and incremental signing to detect such modifications and deletions.

3.2.1 AppendOnlyContainer

This section examines the working of AppendOnlyContainer in the Ajanta Mobile Agent System. The following notation will be used throughout this report.

E_A : Encryption using *public* key of A.
D_A : Encryption using *private* key of A.
$Sig_A(X)$: Signing of data X using private key of A.

Initialization and Insertion

The AppendOnlyContainer object contains a vector of objects to be protected, along with a vector of their corresponding digital signatures and the identities of the signers. It also contains a *checksum* which is used to detect tampering. When an agent object is created, its AOC is empty. The checksum is initialized by encrypting a *nonce* with the agent owner's public key:

$$checkSum = E_{owner}(N_a) \tag{3.1}$$

This nonce N_a is kept secret and not carried by the agent. In the shopping agent example, at any stage during its travels, the agent program can

collect quotations from various vendors and insert the value in AOC. The check-in procedure requests the current vendor's agent server C to sign the data object to be inserted, X, using its own private key, D_C. The data object, its signature and the identity of the signer are inserted into the corresponding vectors in the AppendOnlyContainer. Then, the checksum is updated as follows:

$$checkSum = E_{owner}(checkSum + Sig_C(X) + C) \tag{3.2}$$

The signature and the signer's identity are concatenated to the current value of the checksum. This is then encrypted using the agent's public key, rendering it unreadable by anyone other than the agents's owner.

Verification

When the agent returns home, the owner can verify the integrity of the data. The verification process works backwards, decrypting the nested encryptions of the checksum, and verifying the signature corresponding to each item stored.

$$D_A(checkSum) \Rightarrow checkSum + Sig_C(X) + C \tag{3.3}$$

The *verify* procedure checks for:

$$E_C(Sig_C(X)) == hash(X) \tag{3.4}$$

If verification succeeds for all iterations, we will be able to recover the original random nonce N_A.

If any mismatches are found, the agent owner knows that the corresponding object has been tampered with, and needs to be discarded. It also implies that the objects extracted up to this point were valid, while other objects whose signatures are nested deeper within the checksum cannot be relied upon.

3.3 Identifying the malicious host

A malicious host can modify/delete the data collected from previous hosts visited by the agent. When the agent returns to the agent owner's site the verification mechanism tests the integrity of the data. The tampered data can be detected by the invalid checksum value. The malicious host however is not detected and can continue to tamper with the agent's data.

We suggest a modification to the AppendOnlyContainer mechanism which is shown in Figure 3.1. The principal idea is to sign the entire data carried by the agent, when the host needs to insert a data item.

Assumptions

We assume that all hosts in the itinerary adds some data to the container, the link is free of active intruders and that there is only one malicious host.

3.3.1 Initialization and Insertion

The checksum is initialized by encrypting a nonce N_a with the agent A's public key E_A as before. This nonce N_a is kept secret and not carried by the agent. At any stage during its travels, the agent program can collect quotations from various vendors and check the value into the container. The check-in procedure requests the current vendor's agent server C to sign the entire vector of objects using its own private key D_C. The object is inserted in the *datacontainer* vector and the checksum is updated as follows:

$$checkSum = E_A(checkSum + Sig_C(datacontainer) + C) \tag{3.5}$$

The current agent server signs the entire data $Sig_C(datacontainer)$. The signature $Sig_C(datacontainer)$ and the signer's identity C are concatenated to the current value of the checksum. This is then encrypted using the agent's public key, rendering it unreadable by anyone other than the agents's owner. Note that the entire contents of the data collected by the agent so far is signed by the host while adding its data.

3.3.2 Verification

Verification of the data is similar to the AOC scheme. The verification process works backwards, decrypting the nested encryptions of the checksum, and verifying the signature corresponding to the data stored.

$$D_A(checkSum) \Rightarrow checkSum + Sig_C(data) + C \tag{3.6}$$

where C is the current signer, and **data** is the data object in the objects vector. The D_A is the private key of A. The *verify* procedure checks for:

$$E_C(Sig_C(data)) == hash(data) \tag{3.7}$$

```
class AppendOnlyContainerExtn{
    Vector dataContainer; // the objects to be protected
    byte[] checksum;
    AppendOnlyContainerExtn(Publickey k, int nonce){
        dataContainer = new Vector();
        checksum = encrypt (nonce); // with key k
    }

    public void checkIn(Object X) {
        dataContainer.addElement(X);
        sig = host.sign(dataContainer);
        checksum = encrypt (checksum + sig + currenthost);
    }

    public boolean verify (PrivateKey k, int nonce){
        loop{
            checksum = decrypt(checksum);
            // Separate signature and hostname from checksum
            and verify this signature.
            // If verification fails, either hostname
            or previoushost is the malicious host.
            previoushost = hostname;
        }until what's left is the initial nonce;
    }
}
```

Figure 3.1: *The Extended AppendOnly Container*

If verification succeeds for all iterations, we will be able to recover the original random nonce N_A.

If any mismatches are found, the agent owner knows that the corresponding object has been tampered with. The objects extracted up to this point are valid, but subsequent objects whose signatures are nested deeper within the checksum cannot be relied upon.

Let us assume that hosts $H_1, H_2, \cdots, H_i, H_j, \cdots, H_n$ were visited by the agent, in that order. Without loss of generality, let the verification fail while verifying the data added by H_i. This will happen only if the tampering was

done by H_i after generating the checksum or by H_j before generating the checksum. Thus, either H_i or H_j is the malicious host.

In this chapter, we examined mechanisms to detect modifications and deletions to data collected by data collection agents. We proposed an extension to the *AppendOnlyContainer* mechanism which can identify the malicious host modifying or deleting the data. In the next chapter, we see how two or more hosts can act together to delete data items from a mobile agent without getting detected. We also give a probabilistic collusion detection algorithm to detect such modifications.

Chapter 4

Collusion in Data Collection Agents

AppendOnlyContainer can detect modifications and deletions to data, if they are carried out by individual hosts without any help from other hosts in the itinerary. However, two or more hosts can collude to delete data items added by other hosts without getting detected. For example, in a shopping agent, two vendors can collude to remove the quotations added by a third vendor. In this chapter we examine the problem of collusions in detail. We also propose a probabilistic collusion detection scheme to detect collusions.

4.1 Attacks by Colluding Malicious Hosts

Let us suppose that an agent wishes to collect data from hosts H_1, H_2, H_3, $\cdots$, H_i, H_{i+1}, $\cdots$, H_j, H_{j+1}, $\cdots$, H_n, in that order. Further, assume that hosts H_i and H_{j+1} are both malicious and willing to collude. Host H_i on receiving the agent, does the following:

1. It adds its own data D_i, to the *AppendOnlyContainer*.
2. It recomputes the checksum as given in equation (3.2). We shall denote this checksum by $checkSum_i$.
3. It sends $checkSum_i$ to H_{j+1}.

H_{j+1} on receiving the agent does the following:

1. It adds its own data D_{j+1}, to the *AppendOnlyContainer*.

2. It recomputes the checksum as given in equation (3.2). But, instead of using the current value of checksum carried by the agent, it uses $checkSum_i$.

3. It removes data items $D_i, \cdots, D_j$ from the *AppendOnlyContainer*

Note that the agent owner or subsequent hosts in the itinerary has no way of detecting that items $D_i, \cdots, D_j$ have been removed from the container. In short, colluding hosts hosts are capable of removing data items, escaping detection. However, they are incapable of making modifications to data items added by other hosts. We give some solutions to tackle this problem in the next section.

4.2 Detecting Collusions

The problem of colluding malicious hosts was explained in Section 4.1. If the agent owner knows the hosts visited by the agent in advance, then this problem is trivially solved. The agent owner can make all hosts add data to the container (null data, if the host has none). When the agent returns, the owner can check if all hosts visited by the agent have added data to it. If the data item corresponding to a host H is missing, then the data has been removed as a result of collusion, unless H itself is a malicious host. The particular data items which are missing also give an indication of the malicious hosts. However, a malicious host can deliberately keep away from adding its dataitem to the container with the aim of implicating its neighbors of collusion.

4.2.1 Overview

Depending on the requirements of the agent initiator, two approaches can be taken to detect collusions:

- All the hosts that added data to the agent can notify the agent initiator. If the initiator receives notification from a host and its data is missing from the container, then it could imply deletion of data by colluding malicious hosts. But, this prevents disconnected operations and involves significant message overhead.

- The agent initiator can query various hosts which it believes was visited by the agent. This allows disconnected operations but involves a greater message overhead.

We argue that all hosts need not participate in the collusion detection process. The idea is to reduce the number of messages exchanged, while not significantly reducing the ability of the initiator to detect collusions. We introduce the notion 'Expected Number of Deleted Hosts' to achieve this objective.

4.2.2 Expected Number of Deleted Hosts

Let n be the number of hosts visited by the agent. If n is not known to the agent initiator in advance, then n is estimated by the owner depending on the application. Also, let k be the owner's estimate of the number of malicious hosts and $m = n - k$. For simplicity, let us assume that $k = 2$. The argument can be easily extended for $k > 2$.

Assuming that the actual number of malicious hosts is also 2 (our estimate was very accurate), the number of hosts whose data items get deleted can be anywhere from 1 to $n - 2$. More hosts are deleted if the malicious hosts are placed far apart and vice versa. The Expected Number of Deleted Hosts (ENDH) is the average number of hosts that are deleted, assuming all permutations of hosts are equally likely.

If $P(i)$ is the probability that exactly i hosts are deleted, then,

$$ENDH = \sum_{i=0}^{n-2} i.P(i) \tag{4.1}$$

For n hosts, there are $n!$ different itineraries. We assign a probability of $\frac{1}{n!}$ to each of them.

For $k = 2$, let us calculate the number of permutations in which at least one host will be deleted. As an analogy, consider a linear arrangement of m honest hosts. Such an arrangement creates a total of $m + 1$ gaps between the hosts. If we denote the hosts by H_i and the gaps by G_i, the arrangement can be represented as $G_1, H_1, G_2, H_2, \cdots, G_m, H_m, G_{m+1}$. The two malicious hosts can fit into the $m + 1$ gaps created by the m hosts. This can be done in $\binom{m+1}{2}$ ways. Further, the m honest hosts can be permuted in $m!$ ways and the two malicious hosts can be permuted in two ways. Hence the total number of permutations is given by $2.m!.\binom{m+1}{2}$.

Similarly, $P(i)$, the probability that exactly i hosts are deleted, can be calculated. To delete i hosts, the two malicious hosts should occupy the gaps $\{G_k, G_{k+i}\}, 1 \leq k \leq (m - i + 1)$. For example, a single host is deleted when the two malicious hosts occupy the first and third positions, second and fourth positions, $(m-2)^{nd}$ and m^{th} positions and so on. Therefore, to

delete a single host, the two malicious hosts can occupy m different positions. To delete two hosts, the malicious hosts can occupy $m-1$ different positions. And to delete i hosts, the malicious hosts can occupy $m - i + 1$ different positions. Since the two malicious hosts can be permuted in 2 different ways and the m honest hosts can be permuted in $m!$ ways, $P(i)$, the probability that exactly i hosts are deleted is given by,

$$P(i) = \frac{2.m!}{n!}.(m - i + 1) \tag{4.2}$$

Hence,

$$ENDH = \frac{2.m!}{n!}.\sum_{i=1}^{m} i.(m - i + 1) \tag{4.3}$$

4.2.3 Detecting Collusions : Notification by Proactive Hosts

The basic idea is to allow the hosts which add data to the agent to notify the initiator that it has added some data. If we allow all hosts to notify the initiator, then the problem is essentially solved. However, this creates a lot of message overhead and makes disconnected operations difficult. The idea is to reduce the number of messages, while not sacrificing too much on the ability of the initiator to detect collusions.

The actual number of ways in which a host can be deleted, depends upon its position in the itinerary. But a host H has no way of determining its exact position in the itinerary because the deletion of few hosts could have happened before the agent reaches H. In this scenario, ENDH is a useful measure, which can help H decide whether or not it should send a message to the initiator. For a fixed n, H should send a message with a higher probability if the value of ENDH is high and vice versa. Since, a host in the itinerary (other than the initiator) has very little idea about its exact position, n and k, it cannot reliably calculate ENDH. Moreover, a single message from one of the deleted hosts is sufficient to determine that a deletion due to collusion has occurred.

Hence, we require the initiator to calculate ENDH. The fraction $\frac{ENDH}{n}$, is a good measure of whether or not a host should a message. After adding its data to the container, the host will determine whether or not to send a message by throwing a biased coin. It sends a message with probability $\frac{ENDH}{n}$. This leads to a tremendous decrease in the number of messages sent, while not losing too much on the ability to detect collusions.

Messages cannot be received by the agent initiator if it has disconnected itself from the network after releasing the agent. In this case, another host in the itinerary which is known to be honest or any other secure host can be configured to receive the notification messages.

4.2.4 Detecting Collusions : Querying by the Agent Initiator

If the hosts need to disconnect after sending the agent, then an alternative scheme can be used to detect collusions. Instead of making the host send notification messages to the initiator, the initiator can query the various hosts. The ENDH parameter can be used here too. The host will send query messages to various hosts with probability $\frac{ENDH}{n}$. If a host responding the query message has added data to the container, it will respond with a positive response. Otherwise, it responds in negative. If a host responds positively and its data item is absent in the container, it possibly implies deletion of data through collusion and the initiator can take affirmative action. Note that the initiator could be sending queries to hosts which never received the agent. This cannot be avoided when the initiator has incomplete knowledge of the hosts which actually added the data.

A problem with this approach is that a malicious host can deliberately send a positive response to the query even if it has not added any data. This would make the agent initiator believe that a collusion had taken place when no collusion exists.

If $k > 2$, various scenarios are possible. Malicious hosts can collude in many different ways. This makes a general analysis impossible. However, if the collision model is decided upon, the ENDH can be calculated, using the same principle.

A summary of various security threats and the solutions that can be applied is given in Table 4.1.

Itinerary	**Type of Thre**
Static	Modfication
Static	Modification
Static	Modification
Static	Modification
Static	Modification
Static	Modification
Static	Deletion
Static	Deletion
Static	Deletion
Static	Deletion
Static	Deletion
Static	Deletion
Dynamic	Modfication
Dynamic	Modification
Dynamic	Modification
Dynamic	Modification
Dynamic	Modification
Dynamic	Modification
Dynamic	Deletion
Dynamic	Deletion
Dynamic	Deletion
Dynamic	Deletion
Dynamic	Deletion
Dynamic	Deletion

Chapter 5

Experimentation

In this chapter, we describe the setup that was used to simulate the probabilistic collusion detection technique and significant results. The simulations were done in Java without depending on any particular mobile agent framework.

The total number of hosts in the itinerary can be varied. Two hosts are designated as malicious. Then, an itinerary is chosen by a random permutation of the hosts. The agent initiator calculates the ENDH using equation 4.3. The actual number of hosts that are deleted depends on the position of the two malicious hosts. The subsequent actions depend on the actual method (Notification by pro-active hosts or Querying by the Agent Initiator) used for detecting collusions.

Notification by Proactive Hosts

The agent caries the probability, $prob = \left(\frac{ENDH}{n}\right)$ with it, where n is the owner's estimate of the number of honest hosts. Each host on receiving the agent does the following.

1. It adds its data to the AppendOnlyContainer.
2. It generates a random number, $r, (0.0 \leq r \leq 1.0)$.
3. If $r \leq prob$, sends a message to the initiator, indicating it has added data to the container.

A collusion is detected by the agent initiator if it receives a message from a host that would have been removed by two colluding hosts. The

experiment was repeated a number of times, each time using a different itinerary. We found that collusions were detected in more than 90% of the cases. The reduction in the number of messages was around 67%.

Querying by the Agent Initiator

The agent initiator has a list of m hosts which are likely to be visited by the agent. The initiator calculates the probability, $prob = \left(\frac{ENDH}{m}\right)$. After the agent returns, the initiator does the following:

1: **for all** i such that $1 \leq i \leq m$ **do**
2: It generates a random number, $r, (0.0 \leq r \leq 1.0)$.
3: **if** $r \leq prob$ **then**
4: Sends a query to host i, asking whether it added any to the agent
5: **if** Host i gives a positive reply and the data is absent from the AOC **then**
6: Declare Collusion
7: **end if**
8: **end if**
9: **end for**

We assumed that the host will query 20% more hosts than in the actual itinerary. The collusions were detected in more than 90% of the cases. However, the reduction in the number of messages was around 25%. This is expected because the host could be querying a lot of hosts which are not in the actual itinerary.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Mobile agents offer a convenient programming paradigm whose benefits cannot be exploited fully if various security issues are not mitigated. This project examined the various security issues that arise in mobile agents. In particular, the threats posed by malicious hosts to data collection agents was examined in detail. The *AppendOnlyContainer* mechanism can be used to detect modifications and deletions to the data collected by data collection agents. However it fails to detect the malicious host which tampered with the data. We proposed an extension to *AppendOnlyContainer* that can detect the malicious host.

The problem of colluding malicious hosts was studied. However, two or more hosts can collude to delete data added by other hosts in the itinerary. The extent of damage depends on the position of malicious hosts in the itinerary. The AppendOnlyContainer mechanism will fail to detect collusions. We proposed a probabilistic method to detect collusions. A summary of various security threats and the solutions that can be applied is given in Table 4.1.

6.2 Future Work

Our extension to the AppendOnlyContainer mechanism can identify only the last malicious host modifying the data. A general method to identify all the malicious hosts will prove very useful in the deployment of mobile agents

for mission critical applications. Our probabilistic method for detecting collusions requires the hosts to contact the owner(or vice versa), which might not be always possible for some applications. A new approach which does not rely on messaging is highly desirable.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.

[2] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), T. J. Watson Research Center, Yorktown Heights, Yorktown Heights, New York, 1994.

[3] D. M. Chess. Security Issues in Mobile Code Systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 1–14. Springer-Verlag, June 1998.

[4] C. Cifuentes and K. J. Gough. Decompilation of binary programs. Technical Report FIT-TR-1994-03, School of Computing Science, Queensland University of Technology, 19, 1994.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.

[6] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[7] G. Glass. Objectspace voyager core package technical overview, 1999. Object Space White Paper, Available from: http://www.objectspace.com/developers/voyager/white/index.html.

[8] F. Hohl. Time Limited Blackbox Security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 92–113. Springer-Verlag, June 1998.

[9] W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000. National Institute of Standards Technology.

[10] D. Johansen. Interview in, D. Milojicic, Trend Wars: Mobile Agent Applications, IEEE Concurrency, pp 80-90, July-September, 1999.

[11] N. Karnik and A. Tripathi. Security in the ajanta mobile agent system. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1999.

[12] D. Kotz. Interview in, D. Milojicic, Trend Wars: Mobile Agent Applications, IEEE Concurrency, pp 80-90, July-September, 1999.

[13] D. B. Lange. Mobile objects and mobile agents: The future of distributed computing? In *Proceedings of the European conference on Object-Oriented Programming*, pages 1–12, 1998.

[14] D. B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, 1999.

[15] D. Low. Java control flow obfuscation. Master's thesis, University of Auckland, 1998.

[16] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, 1997.

[17] J. Riordan and B. Schneier. Environmental key generation towards clueless agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 15–24. Springer-Verlag, June 1998.

[18] V. Roth. 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*, pages 147–154, France, 1998.

[19] T. Sander and C. F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 44–60. Springer-Verlag, June 1998.

[20] B. Schneier. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1994.

[21] A. Shamir. Efficient signature schemes based on birational permutations. In D. R. Stinson, editor, *Proc. CRYPTO 93*, pages 1–12. Springer, 1994. Lecture Notes in Computer Science No. 773.

[22] Tokyo Research Laboratory, IBM. *Aglets Specification 1.1 Draft.* Available from http://www.trl.ibm.co.jp/aglets/spec11.html.

[23] A. Tripathi, N. Karnik, M. Vora, and T. Ahmed. Ajanta – A System for Mobile-Agent Programming. Technical Report TR98-016, Department of Computer Science, University of Minnesota, 1998.

[24] A. R. Tripathi, N. M. Karnik, M. K. Vora, T. Ahmed, and R. D. Singh. Mobile agent programming in ajanta. In *International Conference on Distributed Computing Systems*, pages 190–197, 1999.

[25] H. V. Vliet. Mocha, the java decompiler – available from http://www.brouhaha.com/ eric/computers/mocha.html, 1996.

[26] D. Wong, N. Paciorek, T. Walsh, J. Dicelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, pages 86–97, Berlin, Germany, April 1997. Springer Verlag.

[27] M. Wooldridge and N. R. Jennings. Pitfalls of agent-oriented development. In K. P. Sycara and M. Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 385–391, New York, 1998. ACM Press.