Authentication for Mobile Agents *

S. Berkovits**, J. D. Guttman, and V. Swarup
{shim,guttman,swarup}@mitre.org

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

Abstract. In mobile agent systems, program code together with some process state can autonomously migrate to new hosts. Despite its many practical benefits, mobile agent technology results in significant new security threats from malicious agents and hosts. In this paper, we propose a security architecture to achieve three goals: certification that a server has the authority to execute an agent on behalf of its sender; flexible selection of privileges, so that an agent arriving at a server may be given the privileges necessary to carry out the task for which it has come to the server; and state appraisal, to ensure that an agent has not become malicious as a consequence of alterations to its state. The architecture models the trust relations between the principals of mobile agent systems and includes authentication and authorization mechanisms.

1 Introduction

Currently, distributed systems employ models in which processes are statically attached to hosts and communicate by asynchronous messages or synchronous remote procedure calls. Mobile agent technology extends this model by including mobile processes, i.e., processes which can autonomously migrate to new hosts. Numerous benefits are expected; they include dynamic customization both at servers and at clients, as well as robust remote interaction over unreliable networks and intermittent connections [7, 15, 25].

Despite its many practical benefits, mobile agent technology results in significant new security threats from malicious agents and hosts. In fact, several previous uses of mobile agents have been malicious, e.g., the Internet worm. Security issues are recognized as critical to the acceptability of distributed systems based on mobile agents. An important added complication is that, as an agent traverses multiple machines that are trusted to different degrees, its state can change in ways that adversely impact its functionality.

Threats, vulnerabilities, and countermeasures for the currently predominating static distributed systems have been studied extensively; sophisticated distributed system security architectures have been designed and implemented [13,

* This work was supported by the MITRE-Sponsored Research Program. Appeared in *Mobile Agents and Security*, G. Vigna (Ed.), LNCS 1419, Springer Verlag, 1998.

** S. Berkovits is also affiliated with the Department of Mathematical Sciences, University of Massachusetts–Lowell.

21]. These architectures use the access control model, which provides a basis for secrecy and integrity security policies. In this model, objects are resources such as files, devices, processes, and the like; principals are entities that make requests to perform operations on objects. A reference monitor is a guard that decides whether or not to grant each request based on the principal making the request, the operation requested, and the access rules for the object.

The process of deducing which principal made a request is called *authentication*. In a distributed system, authentication is complicated by the fact that a request may originate on a distant host and may traverse multiple machines and network channels that are secured in different ways and are not equally trusted [13]. Because of the complexity of distributed authentication, a formal theory is desirable: The formal theory shows how authentication decisions may be made safely and uniformly using a small number of basic principles.

The process of deciding whether or not to grant a request—once its principal has been authenticated—is called *authorization*. The authentication mechanism underlies the authorization mechanism in the sense that authorization can only perform its function based on the information provided by authentication, while conversely authentication requires no information from the authorization mechanism.

In this paper, we examine a few different ways of using mobile agents, with the aim of identifying many of the threats and security issues which a meaningful mobile agent security infrastructure must handle. We identify three security goals for mobile agent systems and propose an abstract architecture to achieve those goals. This architecture is based on four distinct trust relationships between the principals of mobile agent systems. We present and prove conditions necessary to establish each trust relation and then create an architecture that establishes the conditions. We use existing theory—the distributed authentication theory of Lampson et al. [13]—to clarify the architecture and to show that it meets its objectives. Finally, we describe a set of practical mechanisms that implement the abstract architecture.

This paper draws heavily from two papers that we have published [6, 5]. For related work on mobile agent security, see [3, 4, 16, 22–24, 11].

2 Mobile Agents

A mobile agent is a program that can migrate from one networked computer to another while executing. This contrasts with the client/server model where non-executable messages traverse the network, but the executable code remains permanently on the computer it was installed on. Mobile agents have numerous potential benefits. For instance, if one needs to perform a specialized search of a large free-text database, it may be more efficient to move the program to the database server rather than move large amounts of data to the client program.

In recent years, several programming languages for mobile agents have been designed. These languages make different design choices as to which components of a program's state can migrate from machine to machine. For instance,

Java [15] permits objects to migrate. In Obliq [1], first-class function values (closures) can migrate; closures consist of program code together with an environment that binds variables to values or memory locations. In Kali Scheme [2], again, closures can migrate; however, since continuations [10, 8] are first-class values, Kali Scheme permits threads to migrate autonomously to new hosts. In Telescript [25], functions are not first-class values; however, Telescript provides special operations that permit processes to migrate autonomously.

The languages also differ in their approach to transporting objects other than agents. When a closure or process migrates, it can either carry along all the objects (mutable data) that it references or leave the objects behind and carry along network references to the objects. Java lets the programmer control object marshalling. Object migration uses copy semantics which results in multiple copies of the same object; data consistency needs to be programmed explicitly if it is desired. In Obliq, objects remain on the node on which they were created and mobile closures contain network references to these objects; if object migration is desired, it needs to be programmed explicitly by cloning objects remotely and then deleting the originals. In Kali Scheme, objects are copied upon migration as in Java. In Telescript, objects can either migrate or stay behind when an agent that owns them migrates. However, if other agents hold references to an object that migrates, those references become invalid.

In this paper, we adopt a fairly general model of mobile agents. Agent servers are abstract processors, e.g., individual networked computers, interpreters that run on computers, etc. Agent servers communicate among themselves using host-to-host communication services. An agent consists of code together with execution state. The state includes a program counter, registers, local environment, control stack, and store.

Agents execute on agent servers within the context of global environments (called places) provided by the servers. The places provide agents with (restricted) access to services such as communication services or access to computational or data resources of the underlying server. Agents communicate among themselves by message passing. In addition, agents can invoke a special asynchronous "remote apply" operation that applies a closure to arguments on a specified remote server. Remote procedure calls can be implemented with this primitive operation and message passing. Agent migration and cloning can also be implemented with this primitive operation, using first-class continuation values.

3 Example: Travel Agents

In this section, we will study an example that is typical of many—though not of all—of the ways that mobile agents can be used effectively. We will try to draw out the most important security issues that they raise, as a concrete illustration of the problems of secure mobile agents.

Consider a mobile agent that visits the Web sites of several airlines searching for a flight plan that meets a customer's requirements. We focus on four servers:

a customer server, a travel agency server, and two servers owned by competing airlines, for instance United Airlines and American Airlines, which we assume for the sake of this example do not share a common reservation system. The mobile agent is programmed by a travel agency. A customer dispatches the agent to the United Airlines server where the agent queries the flight database. With the results stored in its environment, the agent then migrates to the American Airline server where again it queries the flight database. The agent compares flight and fare information, decides on a flight plan, migrates to the appropriate airline server, and reserves the desired flights. Finally, the agent returns to the customer with the results.

The customer can expect that the individual airlines will provide true information on flight schedules and fares in an attempt to win her business, just as we assume nowadays that the reservation information the airlines provide over the telephone is accurate, although it is not always complete.

However, the airline servers are in a competitive relation with each other. The airline servers illustrates a crucial principle: *For many of the most natural and important applications of mobile agents, we cannot expect the participants to trust one another.*

There are a number of attacks they may attempt. For instance, the second airline server may be able to corrupt the flight schedule information of the first airline, as stored in the environment of the agent. It could surreptitiously raise its competitor's fares, or it could advance the agent's program counter into the preferred branch of conditional code. Current cryptographic techniques can protect against some but not all such attacks. Thus, the mobile agent cannot decide its flight plan on an airline server since the server has the ability to manipulate the decision. Instead, the agent would have to migrate to a neutral server such as the customer's server or a travel agency server, make its flight plan decision on that server, and then migrate to the selected airline to complete the transaction. This attack illustrates a principle: *An agent's critical decisions should be made on neutral (trusted) servers.*

A second kind of attack is also possible: the first airline may hoodwink the second airline, for instance when the second airline has a cheaper fare available. The first airline's server surreptitiously increases the number of reservations to be requested, say from 2 to 100. The agent will then proceed to reserve 100 seats at the second airline's cheap fare. Later, legitimate customers will have to book their tickets on the first airline, as the second believes that its flight is full. This attack suggests two additional principles: *A migrating agent can become malicious by virtue of its state getting corrupted;* and *unchanging components of the state should be sealed cryptographically.*

4 Security Goals

Security is a fundamental concern for a mobile agent system. Harrison et al. [7] identified security as a "severe concern" and regarded it as the primary obstacle to adopting mobile agent systems.

The operation of a mobile agent system will normally be subject to various agreements, whether declared or tacit. These agreements may be violated, accidentally or intentionally, by the parties they are intended to serve. A mobile agent system can also be threatened by parties outside of the agreements: they may create rogue agents; they may hijack existing agents; or they may commandeer servers.

There are a variety of desirable security goals for a mobile agent system. Most of these concern the interaction between agents and servers. The user on behalf of whom an agent operates wants it to be protected—to the extent possible—from malicious or inept servers and from the intermediate hosts which are involved in its transmission. Conversely, a server, and the site at which it operates, needs to be protected from malicious or harmful behavior by an agent.

Not all attractive goals can be achieved, however, except in special circumstances. In the case of mobile agents, one of the primary motivations is that they allow a broad range of users access to a broad range of services offered by different—frequently competing—organizations. Thus, in many of the most natural applications, many of the parties do not trust each other. In our opinion, some previous work (for instance [23]) is vitiated by this fact: It assumes a degree of trust among the participants which will not exist in many applications of primary interest.

Nevertheless, the special cases may be of interest to some organizations. A large organization like the United States Department of Defense might set up a mobile agent system for inter-service use; administrative and technical constraints might ensure that the different parties can trust each other in ways that commercial organizations do not. In this paper, however, we will focus on the more generic case, in which there will be mistrust and attempts to cheat.

We assume that different parties will have different degrees of trust for each other, and in fact some parties may be in a competitive or even hostile relation to one another. As a consequence, we may infer that one party cannot be certain that another party is running an untampered server. An agent that reaches that party may not be allowed to run correctly, or it may be discarded. The server may forge messages purporting to be from the agent. Moreover, the server may inspect the state of the agent to ferret out its secrets. For this reason, we assume that agents do not carry keys.

Existing approaches for distributed security [12] allow us to achieve several basic goals. These include authenticating an agent's endorser and its sender, checking the integrity of its code, and offering it privacy during transmission, at least between servers willing to engage in symmetric encryption.

However, at least three crucial security goals remain:

(1) *Certification that a server has the authority to execute an agent on behalf of its sender.* If executing an agent involves contacting other servers, then a server may have to authenticate that it is a legitimate representative of the agent. The sender of an agent may want to control which servers will be allowed to authenticate themselves in this role.

(2) *Flexible selection of privileges, so that an agent arriving at a server may be given the privileges necessary to carry out the task for which it has come to the server.* There are some applications in which a sender wants his agent to run with restricted authority most of the time, but with greater authority in certain situations. For instance, in the travel agent example of Section 3, a data-collection agent collecting flight information on an airline server needs only ordinary privilege. However, when it returns to its home server or a travel agency server, the agent must request privilege so that it can select a flight plan and purchase a ticket. Thus, there must be a mechanism to allow an agent to request different levels of privilege depending on its state (including its program counter).

(3) *State appraisal, to ensure that an agent has not become malicious as a consequence of alterations to its state.* Because a migrating agent can become malicious if its state is corrupted, as in the case of the travel agent of Section 3, a server may want to execute a procedure to test whether an agent is in a harmful state. However, the test must be application-specific, which suggests that reputable manufacturers of mobile agents may want to provide each one with an appropriate state appraisal function to be used each time a server starts an agent. The code to check the agent's state may be shipped under the same cryptographic signature that protects the rest of the agent's code, so that a malicious intermediary cannot surreptitiously modify the state appraisal function.

In the remainder of this paper, we will focus our attention on achieving these three goals.

5 Security for Mobile Agents: Theory

In this section, we will describe a security architecture for mobile agent systems that is designed to achieve the security goals listed in Section 4. The architecture consists of two levels. The first is the *authentication* level. The mechanisms at this level combine to meet the first of the above security goals. The other two goals are achieved via a pair of *state appraisal functions* together with the mechanisms of the *authorization* layer of the architecture which determine with what authorizations the agent is to run.

5.1 Authentication

Authentication is the process of deducing which principal has made a specific request. In a distributed system, authentication is complicated by the fact that a request may originate on a distant host and may traverse multiple machines and network channels that are secured in different ways and are not equally trusted. For this reason, Lampson and his colleagues [13] developed a logic of authentication that can be used to derive one or more principals who are responsible for a request.

Elements of a Theory of Authentication The theory—which is too rich to summarize here—involves three primary ingredients. The first is the notion of *principal*. Atomic principals include persons, machines, and keys; groups of principals may also be introduced as principals; and in addition principals may be constructed from simpler principals by operators. The resulting compound principals have distinctive trust relationships with their component principals. Second, principals make *statements*, which include assertions, requests, and performatives.[1] Third, principals may stand in the *"speaks for"* relation; one principal P_1 speaks for a second principal P_2 if, when P_1 **says** s, it follows that P_2 **says** s. This does not mean that P_1 is prevented from uttering phrases not already uttered by P_2; on the contrary, it means that if P_1 makes a statement, P_2 will be committed to it also. For instance, granting a power of attorney creates this sort of relation (usually for a clearly delimited class of statements) in current legal practice. When P_1 speaks for P_2, we write $P_1 \Rightarrow P_2$. One of the axioms of the theory allows one principal to pass the authority to speak for him to a second principal, simply by saying that it is so:

$$(P_2 \textbf{ says } P_1 \Rightarrow P_2) \supset P_1 \Rightarrow P_2$$

This is called the *handoff* axiom; it says that a principal can hand his authority off to a second principal. It requires a high degree of trust.

Three operators will be needed for building compound principals, namely the **as**, **for**, and quoting operators. If P_1 and P_2 are principals, then P_1 **as** P_2 is a compound principal whose authority is more limited than that of P_1. P_2 is in effect a *role* that P_1 adopts. In our case, the programs (or rather, their names or digests) will be regarded as roles. Quoting, written $P \mid Q$ is defined straightforwardly: $(P \mid Q)$ **says** s abbreviates P **says** Q **says** s.

The **for** operator expresses *delegation*. P_1 **for** P_2 expresses that P_1 is acting on behalf of P_2. In this case P_2 must delegate some authority to P_1; however, P_1 may also draw on his own authority. For instance, to take a traditional example, if a database management system makes a request on behalf of some user, the request may be granted based on two ingredients, namely the user's identity supplemented by the knowledge that the database system is enforcing some constraints on the request. Because P_1 is combining his authority with P_2's, to authenticate a statement as coming from P_1 **for** P_2, we need evidence that P_1 has consented to this arrangement, as well as P_2.

Mobile agents require no additions to the theory presented in [13]; the theory as it exists is an adequate tool for characterizing the different sorts of trust relationships that mobile agents may require.

[1] A statement is a *performative* if the speaker performs an action by means of uttering it, at least in the right circumstances. The words "I do" in the marriage ceremony are a familiar example of a performative. Similarly, "I hereby authorize my attorneys, Dewey, Cheatham and Howe, jointly or severally, to execute bills of sale on my behalf." Semantically it is important that requests and performatives should have truth values, although it is not particularly important how those truth values are assigned.

Atomic Principals for Mobile Agents Five categories of basic principals are specifically relevant to reasoning about mobile agents:

- The *authors* (whether people or organizations) that write programs to execute as agents. Authors are denoted by C, C', etc.
- The *programs* they create, which, together with supplemental information, are signed by the author. Programs and digests of programs are denoted by D, D', etc.
- The *senders* (whether people or other entities) that send agents to act on their behalf. A sender may need a trusted device to sign and transmit agents. Senders are denoted by S, S', etc.
- The *agents* themselves, consisting of a program together with data added by the sender on whose behalf it executes, signed by the sender. Agents and digests of agents are denoted by A, A', etc.
- The *places* where agents are executed. Each place consists of an execution environment on some server. Places may transfer agents to other places, and may eventually return results to the sender. Places are denoted by I, I', etc.

Each author, sender, and place is assumed to have its own public/private key pair. Programs and agents are not allowed to have keys since they are handled by places that may be untrustworthy.

In addition to these atomic principals, the theory also requires:

- Public keys; and
- Compound principals built from keys and the five kinds of atomic principals given above, using the operators of the theory of authentication.

Three functions associate other principals with any agent A:

- The agent's program denoted as $\mathsf{program}(A)$.
- The agent's author (i.e., the author of the agent's program) denoted as $\mathsf{author}(A)$.
- The agent's sender denoted as $\mathsf{sender}(A)$.

Naturally, an implementation also requires certification authorities; the (standard) role they play is described in Section 7.

The Natural History of an Agent There are three crucial types of events in the life history of an agent. They are the creation of the underlying program; the creation of the agent; and migration of the agent from one execution site to another. These events introduce compound principals built from the atomic principals given above.

Program Creation. The author of a program prepares source code and a state appraisal function (denoted by max) for the program. The function max will calculate, as a function of the agent's current state, the maximum set of permissions to be accorded an agent running the program. Should max detect that the

agent state has been corrupted, it will set the maximum set of permissions at a reduced level, possibly allowing no permissions at all.

In addition, a *sender permission list (SPL)* may be included for determining which users are permitted to send the resulting agent. In the event that the entire SPL is not known at the time the program is created, another mechanism such as a *sender permission certificate (SPC)* can be used.

After compiling the source code for the program and its state appraisal function, the author C then combines these compiled pieces of code with the SPL and her name, constructs a message digest D for the result, and signs that with her private key. D is regarded as a *name* of the program of which it is a digest. C's signature on D certifies that C is the one who created the program named by D. With this certification, any entity can later verify that the C did indeed create the program and that the program's code, state appraisal function, and SPL have not changed, either accidentally or maliciously. Should C wish to add a sender to the permission list, she creates and signs an SPC certificate containing the program name D and the sender's name S.

By signing D, the author is effectively making a statement about agents A whose programs are D and about senders S who appear on the SPL of D: The author C is declaring that the sender S of a signed agent (A **for** S) speaks for the agent. Formally, this is the statement

$$C \,|\, A \,|\, S \textbf{ says } [S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$$

for all C, A, and S such that $C = \mathsf{author}(A)$, $S = \mathsf{sender}(A)$, and S is on the SPL of $\mathsf{program}(A)$. The author's signature on an SPC makes a similar statement about the sender named in the certificate.

We assume as an axiomatic principle that the author of an agent speaks for the agent. Formally, this is the statement

$$C \,|\, A \Rightarrow A$$

for all C and A such that $C = \mathsf{author}(A)$.

Agent Creation. To prepare a program for sending, the sender attaches a second state appraisal function (denoted by req), called the *request* function. req will calculate the set of permissions the sender wants an agent running the program to have, as a function of the agent's current state. For some states Σ, $\mathsf{req}(\Sigma)$ may be a proper subset of $\mathsf{max}(\Sigma)$; for instance, the sender may not be certain how D will behave, and she may want to ensure she is not liable for some actions. The sender may also include a *place permission list (PPL)* for determining which places are allowed to run the resulting agent on the sender's behalf, either via agent delegation or agent handoff (see below under Agent Migration). One can also consider *place permission certificates (PPCs)* whereby the sender can essentially add such acceptable places to the PPL even after the agent has been launched.

The sender S computes a message digest A for the following items: the program, its digest D, the function req, the PPL, S's name, and a counter S increments for each agent she sends. A is regarded as a *name* of the agent of which

it is a digest. She then signs the message digest A with her private key. S's signature on A certifies that S created the agent named by A to act on her behalf. The signed agent is identified with principal A **for** S.

By signing A, the sender S is effectively saying that it speaks for the signed agent (A **for** S). Formally, this is the statement

$$S \textbf{ says } [S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$$

for all A and S such that $S = \mathsf{sender}(A)$.

By signing the PPL within A, the sender S is saying that places I that appear on the PPL with an Agent Handoff tag can execute A as the principal (A **for** S), while places I that appear on the PPL with an Agent Delegation tag can execute A as the principal (I **for** A **for** S). Formally, these are the statements:

$$S \,|\, (A \textbf{ for } S) \textbf{ says } [I \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)] \quad \text{(Agent Handoff)}$$
$$S \,|\, (A \textbf{ for } S) \textbf{ says } [I \,|\, (I \textbf{ for } A \textbf{ for } S) \Rightarrow (I \textbf{ for } A \textbf{ for } S)] \quad \text{(Agent Delegation)}$$

for all A, S, and I such that $S = \mathsf{sender}(A)$ and I is on the PPL of A. The sender's signature on a PPC makes a similar statement about the place named in the certificate.

The act of creating the agent establishes the trust relationship embodied in the following theorem.

Theorem 1. *Let A be an agent such that $C = \mathsf{author}(A)$, $S = \mathsf{sender}(A)$, and S is on the SPL of $\mathsf{program}(A)$ or S holds an SPC for $\mathsf{program}(A)$. Then:*

$$S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)$$

Proof. The following assumptions hold:

(a) $C \,|\, A \Rightarrow A$ (axiom).
(b) $C \mid A \mid S$ **says** $[S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$ (derived from C's signature on $\mathsf{program}(A)$ and the SPL or SPC of $\mathsf{program}(A)$).
(c) S **says** $[S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$ (derived from S's signature on A).

Applying (a) to (b) yields $A \mid S$ **says** $[S \mid (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$ (d). The delegation axiom $X \ \wedge \ (Y \mid X) \Rightarrow (Y \textbf{ for } X)$ applied to (c) and (d) yields (A **for** S) **says** $[S \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)]$ (e). The result of the theorem then follows from (e) using the handoff axiom.

Before the sender dispatches A, she also attaches a list of parameters, which are in effect the initial state Σ_0 for the agent. The state is not included under any cryptographic seal, because it must change as the agent carries out its computation. However, S's request function req may impose invariants on the state.

Agent Migration. When an agent is ready to migrate from one place to the next, the current place must construct a request containing the agent A, its current state Σ, the current place I_1, the principal P_1 on behalf of whom I_1 is executing the agent, and a description of the principal P_2 on behalf of whom the next place I_2 should execute the agent starting in state Σ.

The statement $I_2 \mid P_2 \Rightarrow P_2$ asserts the expected trust relationship between I_2 and P_2, namely, that, whenever I_2 says P_2 makes a statement s, P_2 is committed to s. The authentication machinery can be construed as providing a proof of this statement. Depending on whether I_2 is trusted by I_1 or by the agent A, four different values of P_2 are possible, expressing four different trust relationships.

(1) *Place Handoff.* I_1 can hand the agent off to I_2. I_2 will then execute the agent on behalf of P_1. In this case, P_2 is P_1, and the migration request by I_1 is assumed to say $I_1 \mid P_1$ **says** $I_2 \mid P_2 \Rightarrow P_2$.
(2) *Place Delegation.* I_1 can delegate the agent to I_2. I_2 will combine its authority with that of P_1 while executing the agent.[2] In this case, P_2 is $(I_2$ **for** $P_1)$, and the migration request by I_1 is assumed to say $I_1 \mid P_1$ **says** $I_2 \mid P_2 \Rightarrow P_2$. The response by I_2 to accept the delegation is assumed to say $I_2 \mid P_1$ **says** $I_2 \mid P_2 \Rightarrow P_2$.
(3) *Agent Handoff.* The agent can directly hand itself off to I_2. I_2 will execute A on behalf of the agent. In this case, P_2 is $(A$ **for** $S)$, and A's PPL or a PPC must imply $S \mid (A$ **for** $S)$ **says** $(I_2 \mid P_2 \Rightarrow P_2)$. The migration request by I_1 does not assert anything and can be unsigned.
(4) *Agent Delegation.* The agent can delegate itself to I_2. I_2 will combine its authority with that of the agent while executing A. In this case, P_2 is $(I_2$ **for** A **for** $S)$, and A's PPL or a PPC must imply $S \mid (A$ **for** $S)$ **says** $(I_2 \mid P_2 \Rightarrow P_2)$. The response by I_2 to accept the delegation is assumed to say $I_2 \mid (A$ **for** $S)$ **says** $(I_2 \mid P_2 \Rightarrow P_2)$. The migration request by I_1 does not assert anything and can be unsigned.

In the first case, I_2 does not appear in the resulting compound principal. This requires I_1 to trust I_2 not to do anything I_1 would not be willing to do. In the third and fourth cases, because the agent itself is explicitly expressing trust in I_2, the resulting compound principal does not involve I_1. The agent trusts I_2 to appraise the state before execution. Assuming that the result of the appraisal is accepted, I_1 has discharged its responsibility. Place handoff and place delegation will usually be initiated directly by the server of the place, while agent handoff and agent delegation will usually be initiated by agent's code.

Agent launch may be regarded as place handoff where the sender's home place plays the role of I_2 and the sender herself acts as I_1.

Each time an agent migrates to a new place, the authentication machinery must verify that the statement $I_2 \mid P_2 \Rightarrow P_2$ is true. How this is done depends on which of the four cases of migration is involved. In each case, however, the verification is performed simply by checking to see if a small number of statements

[2] After such delegation, where the agent travels after I_2 and what privileges it will be given thereafter may depend on input from I_2 or trust in I_2.

are true. The following four theorems show what these statements are in each of the four respective cases.

Let A be an agent such that $S = \mathsf{sender}(A)$, and assume that A migrates from place I_1 as principal P_1 to place I_2 as principal P_2.

Theorem 2 (Place Handoff). *Let $P_2 = P_1$. Then $I_2 \,|\, P_2 \Rightarrow P_2$ follows from the following assumptions:*

(a) $I_1 \,|\, P_1 \Rightarrow P_1$ *(derived from A's certificates).*
(b) $I_1 \,|\, P_1$ **says** $I_2 \,|\, P_2 \Rightarrow P_2$ *(derived from I_1's request).*

Proof. Applying (a) to (b) yields P_1 **says** $I_2 \mid P_2 \Rightarrow P_2$ (c). The result of the theorem follows from (c) using $P_1 = P_2$ and the handoff axiom.

Theorem 3 (Place Delegation). *Let $P_2 = I_2$* **for** *P_1. Then $I_2 \mid P_2 \Rightarrow P_2$ follows from the following assumptions:*

(a) $I_1 \,|\, P_1 \Rightarrow P_1$ *(derived from A's certificates).*
(b) $I_1 \,|\, P_1$ **says** $I_2 \,|\, P_2 \Rightarrow P_2$ *(derived from I_1's request).*
(c) $I_2 \,|\, P_1$ **says** $I_2 \,|\, P_2 \Rightarrow P_2$ *(derived from I_2's response).*

Proof. Applying (a) to (b) yields P_1 **says** $I_2 \,|\, P_2 \Rightarrow P_2$ (d). The delegation axiom $X \wedge (Y \,|\, X) \Rightarrow Y$ **for** X applied to (d) and (c) yields P_2 **says** $I_2 \,|\, P_2 \Rightarrow P_2$ (e). The result of the theorem then follows from (e) using the handoff axiom.

Theorem 4 (Agent Handoff). *Let $P_2 = A$* **for** *S. Then $I_2 \,|\, P_2 \Rightarrow P_2$ follows from the following assumption:*

(a) $S \,|\, (A$ **for** $S) \Rightarrow (A$ **for** $S)$ *(derived by Theorem 1).*
(b) $S \,|\, (A$ **for** $S)$ **says** $[I_2 \,|\, P_2 \Rightarrow P_2]$ *(derived from A's PPL or accompanying PPC).*

Proof. The result of the theorem follows from (a) and (b) using $P_2 = (A$ **for** $S)$ and the handoff axiom.

Theorem 5 (Agent Delegation). *Let $P_2 = I_2$* **for** *A* **for** *S. Then $I_2 \,|\, P_2 \Rightarrow P_2$ follows from the following assumptions:*

(a) $S \,|\, (A$ **for** $S) \Rightarrow (A$ **for** $S)$ *(derived by Theorem 1).*
(b) $S \,|\, (A$ **for** $S)$ **says** $[I_2 \,|\, P_2 \Rightarrow P_2]$ *(derived from A's PPL or accompanying PPC).*
(c) $I_2 \,|\, (A$ **for** $S)$ **says** $[I_2 \,|\, P_2 \Rightarrow P_2]$ *(derived from I_2's response).*

Proof. Applying (a) to (b) yields $(A$ **for** $S)$ **says** $[I_2 \mid P_2 \Rightarrow P_2]$ (d). The delegation axiom $X \wedge (Y \mid X) \Rightarrow Y$ **for** X applied to (c) and (d) yield P_2 **says** $[I_2 \,|\, P_2 \Rightarrow P_2]$. The result of the theorem then follows using the handoff axiom.

We can now describe what happens when a place I_2 receives a request to execute an agent A with a state Σ on behalf of a principle P_2. First, I_2 will check the author's signature on the program of A and the sender's signature on A itself. This would be done using standard, well-understood, public key certification mechanisms [13]. Second, I_2 will authenticate P_2 by verifying that $I_2 \mid P_2 \Rightarrow P_2$ is true. This would be done by checking to see that the assumptions (given by the theorems above) which imply $I_2 \mid P_2 \Rightarrow P_2$ follow from A's PPL and PPCs, the certificates carried by A, and the certificates held by certification authorities. We have now met the first of the security goals proposed in Section 4, namely certification that a place has the authority to execute an agent, ultimately on behalf of its sender.

Admissible Agent Principals. Let an *admissible agent principal* be defined inductively by:

(1) A **for** S is an admissible agent principal if A is an agent and S is a sender.
(2) I **for** P is an admissible agent principal if I is a place and P is an admissible agent principal.

If we assume that an agent can be created and can migrate only in the ways described above, then an agent can only be executed on behalf of an admissible agent principal.

5.2 Authorization

The result of the *authentication* layer is a principal P_2 on behalf of whom I_2 has been asked to execute the agent. The purpose of the *authorization* layer is to determine what level of privilege to provide to the agent for its work. The authorization layer has two ingredients. First, the agent's state appraisal functions `max` and `req` are executed; their result is to determine what privileges ("permits") the agent would like to *request* given its current state. Second, the server has access control lists associated with these permits; the access control lists determine which of the requested permits it is willing to *grant*.

We will assume that the request is for a set α of permits; thus, a request is a statement of the form *please grant* α. In our approach, agents are programmed to make this request when they arrive at a site of execution; the permits are then treated as capabilities during execution: no further checking is required. We distinguish one special permit `run`. By convention, a server will run an agent only if it grants the permit `run` as a member of α.

The request is made by means of the two state appraisal functions. The author-supplied function `max` is applied to Σ returning a maximum safe set of permits. The sender-supplied appraisal function `req` specifies a desired set of permits; this may be a proper subset of the maximum judged safe by the author. However, it should not contain any other, unsafe permits. Thus, we consider P_2 to be making the conditional statement:

$$\textit{if } \texttt{req}(\Sigma) \subseteq \texttt{max}(\Sigma) \textit{ then please grant } \texttt{req}(\Sigma) \textit{ else please grant } \emptyset$$

I_2 evaluates $\texttt{req}(\Sigma)$ and $\texttt{max}(\Sigma)$. If either `req` or `max` detects dangerous tampering to Σ, then that function will request $\emptyset$. Likewise, if `req` makes an excessive request, then the conditional ensures that the result will be $\emptyset$. Since $\texttt{run} \notin \emptyset$, the agent will then not be run by I_2. Otherwise, P_2 has requested some set α_0 of permits.

In the logic of authentication presented in [13], authorization—the *granting* of permits—is carried out using access control lists. Logically, an access control list is a set of formulas of the form $(Q \textbf{ says } s) \supset s$, where the statements s are requests for access to resources and Q is some (possibly compound) principal. If a principal P **says** s_0, then I_2 tries to match P and s_0 against the access control list. For any entry $(Q \textbf{ says } s) \supset s$, if $P \Rightarrow Q$ and $s_0 \supset s$, then I_2 may infer s, thus effectively granting the request. This matching may be made efficient if P, Q, and s take certain restricted syntactic forms.

Since we are concerned with requests for sets of permits, if $\alpha \subseteq \alpha_0$ then *please grant* $\alpha_0 \supset$ *please grant* α. Hence, a particular access control list entry may allow only a subset of the permits requested. The permits granted will be the union of those allowed by each individual access control list entry

$$(Q \textbf{ says } \textit{please grant } \alpha) \supset \textit{please grant } \alpha$$

that matches in the sense that $P_2 \Rightarrow Q$ and $\alpha \subseteq \alpha_0$.

6 Example Revisited: Secure Travel Agents

We now return to our travel agents example (Section 3) and describe how the various trust relationships of that example can be expressed in our security architecture, and how state appraisal functions may be used to achieve their security goals.

6.1 Trust Relationships

In the example, a travel agency purchases a travel reservation program containing a state appraisal function from a software house. The state appraisal function determines when and how the agent will have *write privileges* to enter actual reservations in the databases of an airline, a hotel, or a car rental firm. Otherwise, it requests only *read privileges* to obtain pricing and availability information from those databases.

When a customer submits a tentative itinerary for a business trip or a vacation (via an HTML form, for example), the travel agency prepares to launch the travel reservation agent. It adds a permit request function. The agency has special relationships with certain airlines, hotels, car rental companies, and other travel agencies. The agency provides a PPL or PPCs to hand off or delegate authority to servers. For instance, the travel agency may be willing to hand off authority to its own server and to a neutral, trusted travel agency server, but it may wish only to delegate authority to Airline 1 and Airline 2 (since they have

vested interests). Alternatively, the agency may get special commissions from Airline 2 and may be eager to accept anything that airline suggests. As a result, it may be willing to hand off to Airline 2. The travel agency launches the agent at its server, with an initial state containing the customer's desired travel plans.

As its first task, the agent migrates to the Airline 1 server I_1. The migration request is for place delegation to Airline 1, giving I_1 the authority to speak on the agent's behalf. Airline 1 accepts this delegation and runs the agent as I_1 **for** A **for** S. This ensures that Airline 1 takes responsibility while speaking for the agent, for instance, while deciding that it is to the customer's advantage to visit a hotel that Airline 1 owns before moving to Airline 2. This is an example of the *agent delegating its authority* to Airline 1 (Theorem 5).

Airline 1 owns a hotel chain and has strong trust in its hotels such as Hotel 1. It sends the agent to the Hotel 1 server I_2 and gives Hotel 1 whatever authority it has over the agent. Hotel 1 runs the agent as I_1 **for** A **for** S, which is the principal that I_1 hands it. This kind of trust relationship is an example of Airline 1's *server handing off its authority* to Hotel 1 (Theorem 2). As a consequence of this trust, I_2 may grant the agent access to a database of preferred room rates.

Next, the agent migrates to Airline 1's preferred car rental agency Car Rental 1, whose server is I_3. Since Airline 1 does not own Car Rental 1, it delegates its authority to Car Rental 1. Car Rental 1 runs the agent as I_3 **for** I_1 **for** A **for** S. This causes Car Rental 1 to take responsibility while speaking on Airline 1's behalf. It also gives the agent combined authority from I_1 and I_3; for instance, the agent can obtain access to rental rates negotiated for travelers on Airline 1. Airline 1's *server has delegated its authority* to Car Rental 1 (Theorem 3).

The agent now migrates to the Airline 2 server I_4. The agent's PPL includes Airline 2 or the agent holds a PPC that directly delegates to Airline 2 the authority to speak on the agent's behalf. Airline 2 accepts this delegation and runs the agent as I_4 **for** A **for** S, again agent delegation (Theorem 5). Airline 1's server I_1 has now discharged its responsibility; it is no longer an ingredient in the compound principal. Except that the agent is carrying the results of its inquiries at Airline 1, Hotel 1 and Car Rental 1, it is as if the travel agency had just delegated the agent to Airline 2.

Once the agent has collected all the information it needs, it migrates to the customer's trusted travel agency (Travel Agency 1) server I_5 to compare information and decide on an itinerary. The agent's PPL or a PPC permits directly handing Travel Agency 1 the authority to speak on its behalf. Travel Agency 1 can thus run the agent as A **for** S. This permits Travel Agency 1 to make critical decisions for the agent, for instance, to make reservations or purchase a ticket. This kind of trust relationship is an example of the *agent handing off its authority* to Travel Agency 1 (Theorem 4).

6.2 Authorization

We next illustrate how state appraisal functions may be used to achieve their security goals. In particular, we will stress the goals 2 and 3 of Section 4, namely

flexible selection of permits and the use of state appraisal functions to detect malicious alterations.

In our first example, we will illustrate how the flexible selection of privilege may benefit the servers, in this case the airlines. Before any airline will grant write privileges for entering an actual reservation and payment information, it wants assurance that the agent has visited an acceptably neutral server to decide which reservation it should make. Since the author of the program knows best how these facts are stored in the agent's state, he knows how to perform this test. It may thus be incorporated into the state appraisal function `max`. Since the author is a disinterested, knowledgeable party, the server can safely grant the requested write privilege whenever it is within $\texttt{max}(\Sigma)$. The airline is reasonably sure that it is not being tricked into making bogus reservations, thus tying up seats that actually could be and should be sold.

The flexible selection of privilege may also benefit the sender. For instance, when the agent returns to the travel agency's trusted server, it may attempt to book a ticket provided sufficient information has already been retrieved. Suppose it is the travel agency's policy not to make a booking unless at least four airlines have been consulted. Thus, the agency writes—or selects—a permit request function `req` that does not allow the write privilege unless four alternate bookings are available from which to choose the best. `max` may not allow write privileges to make a booking unless the agent has visited a neutral server to make a decision on which booking to make. It is `req`, however, which further limits the booking privilege if there were not four alternate itineraries from which the choice was made. An exception-handling mechanism will send the agent out for more information if it tries to make a booking without this permit.

State appraisal may also be used to disarm a maliciously altered agent. Let us alter our example slightly so that the agent maintains—in its state—a linked list of records, each of which represents a desired flight. The code that generates this list ensures that it is finite (free of cycles); the code that manipulates the list preserves the invariant. Thus, there is never a need for the program to check that this list is finite. However, when an agent is received, it is prudent for the state appraisal function to check that the list has not been altered in transit. For if it were, then when the agent began to make its reservations, it would exhaust the available seats, causing legitimate travelers to choose a different carrier.

7 Security for Mobile Agents: Practice

We next describe keys, certificates, and protocols that can be used to implement the abstract security architecture of Section 5.

7.1 Principals and Keys

In Section 5.1, we introduced atomic principals (which include persons, machines, and agents), keys, and compound principals constructed from other principals by operators. Of these, keys are the only principals that can make statements

directly; they are the *simple* principals in the sense of Rivest and Lampson [20] that can certify statements made by other principals.

All other principals make statements indirectly. Their statements are validated via proof derivations in the theory of authentication (see Section 5.1). For example, a creator C of an agent A and its sender S, with their respective keys, can create the statements necessary to authorize another principal to speak for the compound principal A **for** S.

When an atomic principal signs a statement, the public key certificate of that principal serves to prove the identity of the principal making the statement. Similar proofs are needed when a compound principal makes a statement. The recipient of the statement must be able to determine who made the statement and whether that principal can be trusted with the statement. That is, the recipient must determine whether to act on the statement or not.

The proof authenticating the identity of an agent's principal may be complex, and its structure corresponds to the structure of the principal itself. The proof requires establishing the sequence of steps executed in building up the agent's compound principal. It examines how the agent's principal was modified at each step in the agent's migration from its launch to its current circumstance.

We assume the existence of a public key infrastructure with certification authorities (CAs) which issue certificates that bind two principals in a speaks-for relationship. Thus, a certification authority CA with key pair K_{CA}, K_{CA^-1} might issue a certificate of the form *Cert(P, Q, CA, validity period)* where P and Q are principals. An identity certificate is a special case of this where P is a public key and Q is (the identity of) the owner of the key. These certificates can be stored in appropriate directory servers. We take this certificate to mean that K_{CA} **says** $P \Rightarrow Q$. If a principal trusts K_{CA}, then it can conclude that $P \Rightarrow Q$. Otherwise, a certification chain of CA certificates must be built. Each certificate in the chain attests to the identity of the owner of the public key in the preceding certificate. The chain extends trust in the local CA's signature to trust in the signature of the distant CA which signed the principal certificate being verified. Trust metrics for certificate chains have been studied in [19].

It is apparent from the previous section that certain other certifications are required to prove the authority of a server to speak for a principal. Basic (non-key) certificates include the creator's signature on the agent code, her signature on the SPL of acceptable senders, and sender SPC certificates. Similarly, the sender's certificates include her signature on the creator's signature of the code, and on Agent Handoff and Agent Delegation PPLs or PPC certificates. Finally, a place's certificates include Place Handoff and Place Delegation certificates. All these certificates are signed by the CA-certified, long-term key of the creator, sender, or place. Verifying signatures in a chain of authentication certificates, of course, requires validity checks for the long-term public key certificates used. This presupposes an efficient validity/revocation infrastructure. The final word on building such an infrastructure is not yet in, but see [9, 14, 17, 18].

The remainder of this section describes in some detail how sequences of these certificates are developed.

7.2 Authentication

We examine the mechanisms necessary to support handoff and delegation certificates when the only key pairs belong to atomic principals. We first describe in detail the certificates required for Place Handoff, Place Delegation, and Agent Launch. Then we look at what is needed for Agent Handoff and Agent Delegation.

We assume that I_j holds a certificate or a sequence of certificates that prove that $(I_j \mid P_k) \Rightarrow P_k$, where P_k is some principal. This is assumption (a) of both Theorem 2 and Theorem 3. All signatures on any certificates involved in this proof have been created by atomic principals using those principals' private keys. We represent the certificate or sequence of certificates that prove that $(I_j \mid P_k) \Rightarrow P_k$ as $Cert((I_j \mid P_k) \Rightarrow P_k)$. We use the symbol $\parallel$ to denote concatenation of data strings.

Form of a Migration Request All migration requests have the same form. It is *Migrate(requesting server's ID, agent ID, principal ID, target server's ID, flag, validity period)*. A request $Migrate(I_1, A, P, I_2, f, t)$ denotes that server I_1, which has the authority to run agent A as principal P, requests server I_2 to run agent A as a new principal which is determined by the flag f. The flag f consists of two bits and is used to indicate the type of migration being requested. We shall use HO for Place Handoff, Del for Place Delegation, AH for Agent Handoff, and AD for Agent Delegation. The validity period t indicates the intended lifespan of this handoff or delegation. At the end of the validity period, the target server no longer speaks for the agent's principal. Other servers should ignore any statements the target server has made quoting the principal, if those statements were made after the expiration of the validity period.

Place Handoff (HO) Suppose that a server I_1 possesses an agent A and a set of certificates $Cert((I_1 \mid P_1) \Rightarrow P_1)$ for A. I_1 can handoff the agent A to a server I_2 by sending I_2 those certificates together with a signed migration request $Migrate(I_1, A, P_1, I_2, HO, t)$. By this request, $(I_1 \mid P_1)$ **says** $[(I_2 \mid P_1) \Rightarrow P_1]$. Then, by Theorem 2,

$$Cert((I_2 \mid P_1) \Rightarrow P_1) = Migrate(I_1, A, P_1, I_2, HO, t) \parallel Cert((I_1 \mid P_1) \Rightarrow P_1)$$

I_2, and any server that handles the agent after I_2, must check the signatures on the migration request by obtaining I_1's public key from a CA-signed certificate. As one possible implementation, the CA-signed certificate for I_1 can be included as part of the migration request and can be incorporated into $Cert((I_2 \mid P_1) \Rightarrow P_1)$. Otherwise, I_2 will have to obtain that key certificate from the CA or from some directory server.

Note that any server that checks the sequence of certificates can first examine the most recently added certificate and then work backwards in history. That way, if it encounters a server that it trusts to have checked all the preceding certificates, it can immediately cease its own verifications.

Place Delegation (Del) Suppose again that a server I_1 possesses an agent A and a set of certificates $Cert((I_1 \mid P_1) \Rightarrow P_1)$ for A. I_1 can delegate its authority over the agent A to a server I_2 by sending I_2 those certificates together with a signed migration request $Migrate(I_1, A, P_1, I_2, Del, t_1)$. By this request, $(I_1 \mid P_1)$ **says** $[(I_2 \,|\, (I_2 \textbf{ for } P_1)) \Rightarrow (I_2 \textbf{ for } P_1)]$.

If I_2 is willing to accept this delegation, it returns a signed response to I_1. The response, $Response(I_2, A, P_1, I_1, t_2)$, is signed by I_2's private key. The validity period t_2 in the response need not be the same as the period t_1 in the request. It is the period during which I_2 is willing to accept the delegation and must be a subset of t_1. I_1 checks I_2's signature on the response before sending the agent to I_2. This signature indicates that $(I_2 \,|\, P_1)$ **says** $[(I_2 \,|\, (I_2 \textbf{ for } P_1)) \Rightarrow (I_2 \textbf{ for } P_1)]$.

Then, by Theorem 3,

$$\begin{aligned} Cert((I_2 \,|\, (I_2 \textbf{ for } P_1)) \Rightarrow (I_2 \textbf{ for } P_1)) = \; & Response(I_2, A, P_1, I_1, t_2) \\ \| \; & Migrate(I_1, A, P_1, I_2, Del, t_1) \\ \| \; & Cert((I_1 \,|\, P_1) \Rightarrow P_1) \end{aligned}$$

along with CA-signed certificates in the case that the implementation requires their being sent from place to place.

I_2 verifies the signatures in $Cert((I_1 \mid P_1) \Rightarrow P_1)$ and the migration request just as in Place Handoff, above. Any other server that receives the agent after I_2 must also verify I_2's signature on the response as well as any other requests and responses of servers between I_2 and itself. Once again, if it encounters a server that it trusts, it may be able to stop checking certificates at that point.

Agent Launch In the above, we have discussed the certificates necessary for an agent's migration from one place to another using Place Handoff or Place Delegation. We now examine the certifications needed to launch an agent and note that they include precisely the same certifications required to support Place Handoff and Place Delegation.

Code Creation and Signature A creator of an agent's program prepares the source code for that agent and a state appraisal function called `max`. The function `max` is used to calculate the maximum permissions that are safe to afford an agent running the program. The creator then signs the combination of program and `max`. This insures the integrity of the code and of `max` and proves that the creator speaks for the program.

The creator may create a list of users, the sender permission list (SPL), who are permitted to send the resulting agent. That SPL can be included with the program and `max` as part of the object being signed. Subjects on the SPL may be individual users or groups of users.

Sender Permission Certificates As the creator may not know, a priori, all the users or groups of users who will want to employ the program, a mechanism for sender permission certificates (SPCs) is also needed. An SPC has the same

syntax as an element of an SPL. However, it is signed as a separate entity with data to show to which program it refers and who the signer is. The signer is again the program creator.

The creator can, for example, sign and deliver an SPC to any buyer of his program who executes a licensing agreement. The SPC can even include a validity period to correspond to the lifetime of the license being issued. For a group license, the creator can specify in the SPC what determines group membership or who decides such membership.

Sender's Customization The sender may augment the signed code and sender SPL she has received from the creator. She may add a state appraisal function called `req`. The function `req` is used to calculate the minimum permissions that are required to run the agent program. The sender may include a list of places that are permitted to run the agent A with principal (A **for** S). She may also include a list of places I that are permitted to run the agent with principal (I **for** A **for** S). The two lists comprise the place permission list (PPL). To insure the integrity of the code and of `req` and to prove that the sender, quoting A **for** S, speaks for A **for** S, she signs the combination of the creator's signature on the program and `max`, the function `req`, and any PPL. Conceivably the sender can make separate place permission certificates (PPCs), one for each place that can accept Agent Handoff and one for each place that can accept Agent Delegation. This latter alternative, however, is quite unlikely. The sender should know at the time of launch which places are acceptable for Agent Handoff or Agent Delegation. If new places become available after the agent's launch, it is unlikely that the sender will be available to sign an appropriate PPC.

The creator's signature and the sender's signature, provided that the sender is on the creator list of authorized senders, together imply that $(S \,|\, (A \textbf{ for } S)) \Rightarrow (A \textbf{ for } S)$ (see Theorem 1 for details of the proof of this statement). S is the initial place where the agent, with principal $P = A$ **for** S, exists. These signatures together form $Cert(S \,|\, P \Rightarrow P)$. The agent can now be sent to its first place by either Place Handoff or Place Delegation with S signing the appropriate migration request. For Place Delegation, the initial server creates and signs a response; whether or not that response is returned to the sender and verified is not important for this discussion.

Agent Handoff and Agent Delegation For both Agent Handoff and Agent Delegation, I_1 first checks that I_2 is on the PPL or is mentioned in a PPC. If it is, I_1 creates a request $Migrate(I_1, A, P_1, I_2, f, t_1)$ where the flag f is either AH or AD depending on the type of migration. Although this request takes the same standard form as the others, the agent principal P_1 at I_1 actually has no bearing on the migration.

If desired, I_1 may sign the request, and I_2 may verify the sequence of certificates $Cert((I_1 \mid P_1) \Rightarrow P_1)$ that allow I_1 to speak for P_1 and to make the migration request. I_2 may also recheck that it is on the proper sender-signed PPL or PPC. Ultimately, I_2 must decide if it wishes to run the agent with prin-

cipal (A **for** S) if Agent Handoff is requested or with principal (I_2 **for** A **for** S) if Agent Delegation is the request.

How the agent reached I_2 is really of no consequence. In fact, the new certificate for Agent Handoff is

$$Cert(I_2 \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)) = \text{the sender-signed PPL or PPC}$$

and for Agent Delegation is

$$\begin{aligned} Cert(I_2 \,|\, (I_2 \textbf{ for } A \textbf{ for } S) \Rightarrow (I_2 \textbf{ for } A \textbf{ for } S)) = \\ Response(I_2, A, (A \textbf{ for } S), S, t_2) \\ \| \text{ the sender-signed PPL or PPC} \end{aligned}$$

The response is signed by I_2, of course, and can be sent to I_1 or to S for verification.

8 Conclusion

Many of the most important applications of mobile agents will occur in fairly uncontrolled, heterogeneous environments. As a consequence, we cannot expect that the participants will trust each other. Moreover, servers may disclose the secrets of visiting agents, and may attempt to manipulate their state.

Existing techniques, intended for distributed systems in general, certainly allow substantial protection within the broad outlines of these constraints. However, substantial investment in mobile agent systems may await further work on new security techniques specifically oriented toward mobile agents. These new techniques focus on three areas. The first is programming language and system support to improve the safety of mobile code. The second is support for tracking the state carried by mobile agents. The third is trusted execution environments within which agents can make critical decisions safely. With advances in these areas, we believe that mobile agents will be an important ingredient in producing secure, flexible distributed systems.

In this paper, we have described a framework for authenticating and authorizing mobile agents, building on existing theory. Our approach models a variety of trust relations, and allows a mobile agent system to be used effectively even when some of the parties stand in a competitive relation to others. We have introduced the idea of packaging state appraisal functions with an agent. The state-appraisal functions provide a flexible way for an agent to request permits, when it arrives at a new server, depending on its current state, and depending on the task that it needs to do there. The same mechanism allows the agent and server to protect themselves against some attacks in which the state of the agent is modified at an untrustworthy server or in transit. We believe that this is a primary security challenge for mobile agents, beyond those implicit in other kinds of distributed systems.

References

1. L. Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 286–298, 1995. `http://www.research.digital.com/SRC/Obliq/Obliq.html`.
2. H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995. `http://www.neci.nj.nec.com:80/PLS/Kali.html`.
3. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. IEEE *Personal Communications Magazine*, 2(5):34–49, October 1995. `http://www.research.ibm.com/massive`.
4. D. Chess et al. Things that go bump in the net. Web page at `http://www.research.ibm.com/massive`, IBM Corporation, 1995.
5. W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS), LNCS 1146*, pages 118–130, September 1996.
6. W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *National Information Systems Security Conference*. National Institute of Standards and Technology, October 1996.
7. C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM Research Report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, March 1995. `http://www.research.ibm.com/massive`.
8. C. Haynes and D. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9:582–598, 1987.
9. R. Housley, W. Ford, W. Polk, and D. Solo. Internet public key infrastructure X.509 certificate and CRL profile. Internet Draft <draft-ietf-pkix-ipki-part1-06.txt>, Work in Progress, October 1997.
10. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
11. G. Karjoth, D. B. Lange, and M. Oshima. A security model for Aglets. In *IEEE Internet Computing*, pages 68–77, July/August 1997.
12. C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 1995.
13. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, November 1992.
14. S. Micali. Efficient certificate revocation. Technical Memo MIT/LCS/TM-542b, MIT, September 1997. See also US Patent 5666416.
15. Sun Microsystems. Java: Programming for the Internet. Web page available at `http://java.sun.com/`.
16. Sun Microsystems. HotJava: The security story. Web page available at `http://java.sun.com/doc/overviews.html`, 1995.
17. M. Myers. Internet public key infrastructure online certificate status protocol–OCSP. Internet Draft <draft-ietf-pkix-opp-ocsp-01.txt>, Work in Progress, November 1997.
18. M. Naor and K. Nissim. Certificate revocation and certificate update. In *7th USENIX Security Symposium*, San Antonio, CA, January 1998.

19. M. K. Reiter and S. G. Stubblebine. Toward acceptable metrics of authentication. In *IEEE Symposium on Security and Privacy*, pages 3–18, 1997.
20. R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. `http://theory.lcs.mit.edu/~rivest/publications.html`.
21. J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Winter Conference*, pages 191–202, 1988.
22. J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon*, 1996. `http://www.cs.umbc.edu/agents/security.html`.
23. C. Thirunavukkarasu, T. Finin, and J. Mayfield. Secret agents — a security architecture for KQML. In *CIKM Workshop on Intelligent Information Agents*, Baltimore, December 1995.
24. G. Vigna. Protecting mobile agents through tracing. In *Proceedings of the Third Workshop on Mobile Object Systems*, Finland, June 1997.
25. J. E. White. Telescript technology: Mobile agents. In *General Magic White Paper*, 1996. Will appear as a chapter of the book Software Agents, Jeffrey Bradshaw (ed.), AAAI Press/The MIT Press, Menlo Park, CA.