View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Universität Stuttgart
Fakultät Informatik

A Framework to Protect Mobile Agents by Using Reference States

Fritz Hohl

Email: `Fritz.Hohl@informatik.uni-stuttgart.de`

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

A Framework to Protect Mobile Agents by Using Reference States

Fritz Hohl

Bericht Nr. 2000/03
March 2000

A Framework to Protect Mobile Agents by Using Reference States[1]

Fritz Hohl
Institute of Parallel and Distributed High-Performance Systems
University of Stuttgart, D-70565 Stuttgart, Germany
Fritz.Hohl@informatik.uni-stuttgart.de

Abstract

To protect mobile agents from attacks by their execution environments, or hosts, one class of protection mechanisms uses "reference states" to detect modification attacks. Reference states are agent states that have been produced by non-attacking, or reference hosts. This paper examines this class of mechanisms and present the bandwidth of the achieved protection. First, a new general definition of attacks against mobile agents is presented. As this general definition does not lead to a practicable protection scheme, the notion of reference states is introduced. This notion allows to define a protection scheme that can be used to practically realize a whole number of mechanisms to protect mobile agents. Therefore, after an initial analysis of already existing approaches, the abstract features of these approaches are extracted. A discussion examines the strengths and weaknesses of the general protection scheme, and a framework is presented that allows an agent programmer to choose a level of protection using the reference states scheme. An example illustrates the usage of the framework, measurements present the overhead of the framework for the case of the example mechanism.

1. Introduction

Mobile agents are program instances that are able to migrate from one agent platform to another, thus fulfilling tasks on behalf of a user or another entity. They consist of three parts: code, a data state (e.g. instance variables), and an execution state that allows them to continue their program on the next platform. For the area of mobile agents, security is a very important aspect since neither the provider of an agent platform or an agent-based service, nor the owner of an agent wants to be harmed by employing this technology. This is a non-trivial requirement since for the first time, in mobile agent systems, not only the executing party has no vital interest to execute a program correctly, but also the employer of a program has to give away the control over the execution.

While the mechanisms that allow the executing party to protect its system seem to be feasible today, the protection of the agent, and, in turn its owner, is still subject of ongoing research.

One way to protect agents is to follow an organizational approach, i.e. to make sure that only trustworthy parties execute an agent. This can be realized either by maintaining the whole agent platform by only one institution, or by disallowing migration to unknown agent platforms. Currently, the first approach does not seem to be feasible since such a system would require a large number of vendors and clients in order to be worthwhile. The problem of the second approach is twofold: on one hand, trust is not a immutable attribute, but may change depending e.g. on the tasks an agent has to fulfil (although an airline as a big company is trustworthy, one does not want to depend on the goodwill of the company's host when comparing different flight prizes). On the other hand, not all resources needed for the execution of a certain agent may be available on trusted hosts.

Another way to protect agents is to use special, trusted, tamper-free hardware (see e.g. [11]). To use them in the near future, at least two things are necessary: the need for such devices by platform providers and a manufacturer who builds these devices. Again, currently, no commercial application fosters this need. Therefore, today, there is no manufacturer who produces these devices. Another problem is whether trusted hardware allows the cost-effective execution of agents, but this aspect will not be discussed here.

If neither organizational mechanisms nor special hardware can be used, mobile agents have to be protected by software means only. Currently, there are two approaches that try to protect an agent from all major attacks. The first approach, which is called Mobile Cryptography [9], aims at

1. This work was funded by the Deutsche Forschungsgemeinschaft (DFG).

converting agents into programs that work on encrypted data (i.e. the operations use encrypted parameters and return encrypted results without the need to decrypt these data during execution). There are three problems which have to be solved before this approach can be used. First, currently, only rational polynomial functions can be used as input "agents" (recently, there are plans to remove this restriction). Second, the used agent model does not allow agents that are protected by this approach to return plaintext data to untrusted hosts (as this could lead to security problems). Third, the efficiency of this approach is unknown (there is no implementation yet). The second approach based completely on software is called Time-limited Blackbox Protection [3]. Here, the agent code is obfuscated using techniques that are hard to analyse by programs. Since such an obfuscation can be broken by a human attacker given enough time, the agent bears an expiration date, after which the agent gets invalid. Successful attacks before this expiration date are impossible. In this approach, the input may be any agent, but there are problems that seem to prevent the employment in the near future. First, it is not known yet whether there are obfuscation techniques that are "hard" enough. Second, it is unclear whether the expiration date can be computed. Third, the efficiency of this approach is currently unknown (but is seems sure that at least the size of the agent increases significantly).

As we have seen, complete software protection of mobile agents is far from being mature enough to be used today. Therefore, other protection mechanisms have to be examined in meantime. These mechanisms will not be able to prevent every attack, but provide at least protection from certain attack classes. As we will see, one important class of protection mechanisms uses "reference states", i.e. agent states that have been produced by non-attacking, or reference hosts to detect modification attacks of malicious hosts.

This paper will examine this class of mechanisms and present the bandwidth of the achieved protection. For that purpose, a new general definition of attacks against mobile agents is presented in Section 2. Since this general definition in itself does not lead to a practicable protection scheme, the notion of reference states is introduced. This notion allows to define a protection scheme that can be used to practically realize a whole number of mechanisms to protect mobile agents. After an initial analysis of already existing approaches in Section 3, an extraction of the abstract features of these approaches. A discussion of the strengths and weaknesses of the general protection scheme is given in Section 4, In Section 5, a framework is presented that allows an agent programmer to choose a specific level of protection using the reference states scheme. An example illustrates the advantages of the framework in Section 6. After having measured the example in terms of overhead, Section 7 concludes this paper.

2. Attacks, Reference behaviour, and Reference States

In this section, we will examine the question of what an attack against a mobile agent is, and whether and how the answer leads to protection schemes. First, the used agent model is defined.

2.1 Agent execution model

In this paper, the following model of the execution of a mobile agent will be used (see also Figure 1).

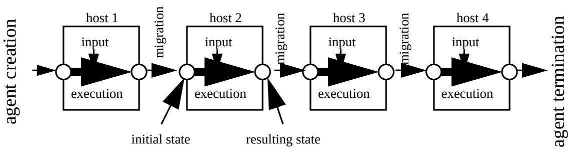

Fig. 1: Agent execution model

The agent is a construct consisting of code, data state, and execution state. The aim of an agent is to fulfil a task in behalf of its owner. For this purpose, the agent migrates along a sequence of hosts. The host takes the initial agent state, i.e. data and execution state, and starts an execution session. In this session, the host processes the agent using the code and some input, producing a resulting agent state. The input includes all the data injected from the outside of the agent, i.e. both communication with partners residing on other hosts and data received directly by or via the current host. The latter e.g. includes results from system calls like random numbers or the current system time. When the agent migrates to another host or dies, the execution session is finished on this host. The resulting state produced by one host is used as the initial state on the next host.

2.2 Attacks and reference behaviour

In the following, a malicious host is an execution environment of an agent that tries to attack (or at least plans to attack) an agent.

If we examine possible attacks by malicious hosts against mobile agents, we find lists like this (see Figure 2). These lists do not state all possible attacks, but areas, in which attacks can be categorized. An exhausting list of attacks themselves does not exist, since this set is open due to the problem area (see [3] for a discussion of this aspect).

1. spying out code
2. spying out data
3. spying out control flow
4. manipulation of code
5. manipulation of data
6. manipulation of control flow
7. incorrect execution of code
8. masquerading of the host
9. denial of execution
10. spying out interaction with other agents
11. manipulation of interaction with other agents
12. returning wrong results of system calls issued by the agent

Fig. 2: A list of possible attack areas

As shown by [3], this list can be reduced to the areas 2 and 4 - 7 (which can be called “blackbox set”), as other attacks seem either not to be preventable (areas 9, 12) or can be prevented given the prevention of the blackbox set.

The term “attack” related to mobile agent protection is rarely defined explicitly, but most often used in an intuitive manner. Since the term is normally understood as a violation of the expectations of the agent programmer or owner we can define attack as follows:

Definition: *An attack is a difference in behaviour between the attacking host and a non-attacking or reference host, i.e. one that acts as expected (*“reference behaviour”*) given the same state and resources (and unambiguous, complete specifications).*

In this definition, attacks include different behaviour due to (unintentional) errors, caused by a misinterpretation of the specifications or by technical faults.

Although this definition seems to be intuitively understandable, the term “reference behaviour” needs more explanation. One can argue that first, no two implementations of a specification behave equally, and second, the behaviour of even the same implementation may differ, depending on external factors, such as the actual state of thread scheduling, of the random number generator and so on. This may be true for a number of systems, but not on the level our notion of behaviour is situated on. We denote with “behaviour” the level of expectation of the agent programmer, i.e. the way the system has to behave in order to execute an agent. If this behaviour differs from the specification, the system acts in a way the programmer did not expect, so it is likely the agent will fail to run. This expectation of the programmer, based on the specification, will probably not determine the behaviour of the system in every detail (e.g. the implementation of integers at the bit level), but is, at an overall level, an adequate model of the system. Therefore, the difference in behaviour cannot be measured on a low level automatically, but by using the knowledge of the programmer to compare two executions instead.

The attack definition above leads to a protection scheme where the difference in behaviour is measured to prove or at least detect misbehaviour. There are two problems that restrict the practicability of this solution. First, some of the behaviour of the host cannot be observed from the outside of the host. In principle, either all malicious behaviour results sooner or later in perceptible actions, or - as the malicious behaviour does not result in a perceptible action - this behaviour does not matter since it has no consequences. Practically, it is too difficult to control all perceptible actions. Imagine e.g. that if the host reads a secret key of the agent and uses this key to decrypt some communication of the agent, this knowledge may result in an action that harms the agent owner. But first, you have to see that there was a breach in security, then you have to find out which host can be blamed for this and finally you have to prove it.

Second, it is at least difficult to provide the reference host with the state and resources of the untrusted host. As a host may e.g. offer a whole database, such a provision would require the transfer of possibly very much data (apart from the aspect that the host may find it unacceptable to transfer important data to a third party). Additionally, if these data have to be transported from the untrusted host, no one can check the equivalence of this data set to the one stored in the untrusted host. One could think of the possibility to run the reference host in a “hot stand-by” mode, providing it with the same input data as the untrusted host (i.e. if the database receives new data, this data would also be sent to the reference database). But, also this scheme fails as soon as new data is created inside the untrusted host.

2.3 Reference states

What can be done practically is to measure not the difference in behaviour between an untrusted and a reference host, but the difference in the variable parts of an agent computed from the untrusted host on the one hand and a reference host on the other hand, given the complete input during the computation. This leads us to:

Definition: *A reference state is the combination of the variable parts (i.e. the state) of a mobile agent executed by a host showing reference behaviour.*

This input includes all the data injected from the outside of the agent, i.e. both communication with partners residing on other hosts and data received directly by or via the current host. The latter includes e.g. results from system calls like random numbers or the current system time. It does not include e.g. results from procedures inside the agent as these can be recomputed using the agent code.

If we are able to measure the difference in state, we are able to detect attacks, that differ in the resulting state from a reference state. These attacks include write or modification attacks of the variable parts of the agent and attacks, where the agent code is not executed according to the specifications. The advantage of this approach is that even not every write, modification and incorrect execution attack is detected, but only those who indeed result in an incorrect state of the agent. This means e.g. that the host may modify the agent code temporarily due to optimization reasons without being blamed to attack the agent. What cannot be detected by this approach are read attacks and attacks where the party that compiles the input modifies or suppresses input.

3. Analysis of Existing Approaches

In this section, we will analyse the four existing approaches that fall into the area of mechanisms that use a kind of reference state to detect attacks by the host. First, we will describe the mechanisms and state the level of protection they offer. Afterwards we will classify them according to criteria like used moment of checking and the used reference data.

3.1 State appraisal

Farmer, Guttman and Swarup present in [2] a “state appraisal” mechanism that checks the validity of the state of an agent as the first step of executing an agent arrived at a host. This checking mechanism only considers the current state of the arrived agent. It can consist e.g. of a set of conditions that have to be fulfilled after the execution session. In this case, the reference data is structured as a set of rules. These rules are formulated by the programmer who stated relations between certain elements of the state. The check is done by the host that received an agent, and it is in the interest of this host to do so as it wants to execute only valid, i.e. untampered agents (which else might attack him). If the host does not check the agent (e.g. because the host collaborates with the attacking host), an attack against an agent cannot be detected.

The question of which further attacks cannot be detected depends partly on the powerfulness of the used checking mechanism. If e.g. for the conditions, only boolean and numerical operators are used (i.e. constructs that are not turing complete), there are computations that can be done by programs, but not by conditions. Therefore, there may be computations that cannot be checked by this kind of rules. The lack of the input to the agent also leads to attacks that cannot be detected. Imagine e.g. an agent that remotely receives prices for a good from different shops. Then a lowest price is computed and the other prices are removed. The host may modify the execution and/or the prices at its will without being detected as it is impossible to find an inconsistency in the resulting state without the used prices.

3.2 Server replication

In [8], Minsky et al. propose to use a fault tolerance mechanism to also detect attacks by malicious hosts. The authors assume for every stage, i.e. an execution session on one host, a set of independent, replicated hosts, i.e. hosts that offer the same set of resources (e.g. the same data), but do not share the same interest in attacking a host (e.g. because they are operated by different organizations). Every execution step is processed in parallel by all replicated hosts. After the execution, the hosts vote about the result of the step. At all hosts of the next step, the votes (i.e. the resulting agent states) are collected. The executions with the most votes wins, and the next step is executed. Obviously, even (n/2 - 1) malicious hosts can be tolerated. From our point of view, this means that an execution is checked by using a set of other executions, and by counting the number of equal results. Since the hosts work in parallel, the input to the agent has to be shared and one host must not be able to hold back input to the other hosts.

The server replication approach can detect all attacks that result in a different agent state. Collaboration attacks of less than n/2 malicious hosts of the same step can be detected. Additionally, even collaboration attacks between hosts of different steps can be found as long as the above condition holds.

3.3 Execution traces

Apart from checking the inherent integrity of agents or comparing agent states resulting from parallel execution, the third major idea to check the execution of an agent is to let the executing host produce an execution protocol or *trace*. In [10], Vigna presents an approach that uses this idea to allow an agent owner to check the execution sessions at different hosts when a fraud is suspected. For this purpose, every host records a trace that looks like the one in Figure 3b.

`10 read(x)`	`10 x=5`
`11 y=x+z`	`11`
`12 m=y+1`	`12`
`13 k=cryptInput`	`13 k=2`
`14 m=m+k`	`14`
Fig. 3a: Code fragment	**Fig. 3b: Trace of the code fragment**

A trace consists of pairs `(n,s)` where `n` denotes the identifier of the executed code statement. In case this statement modifies the state of the agent using information from the outside of the agent (i.e. “input” in our terms), `s` denotes the list of variable-value pairs that state the content of these variables after executing this statement.

After the execution, the host creates a hash of the trace and a hash of the resulting agent state. Theses hashes are signed by the host and are sent to the next host, together with the code and state of the agent. The trace itself has to be stored by the host. The agent continues to fulfil its task and returns to its home host afterwards. Now, the agent owner can decide whether he/she wants to check the agent or not. In case of a suspicion, he/she requests the traces from the corresponding hosts starting at the first host. First, he/she computes a hash of the received trace and compares this hash with the one stored at the next host. If these hashes are identical, the host commits on this trace. Then the agent with its initial state is re-executed. In case of statements that used input from the outside, the values recorded in the trace are used. If a hash of the resulting state of the agent on this host is equal to the one signed by this host (which can be provided also by the next host), this host did no cheat, and the checking process continues. The case that the following host pretends to have received a different initial agent state, is prevented by sending back a signed message that commits this state back to the sending host.

Obviously, the length of a trace increases with every execution step. Due to performance reasons, Vigna proposes therefore to use a modified trace without statement identifiers. But there is another reason why we are able to do without these identifiers. First, it can be argued that it is more important to check whether an execution yields the correct final agent state than that the execution followed a certain way. Second, a list of executed statement identifiers does not prove anything since an attacker can create a correct list and augment it with correct or incorrect input data. In this case, the attack is detected only if the resulting state is checked, not the statement identifiers. Therefore, identifiers are not needed from a security point of view.

This approach detects all attacks that result in a different state as long as the host does not lie about the input to the agent. Note that the owner can only determine which host played wrong, but not the difference in the agent state as only hashes of the final states exist.

3.4 Proof Verification

In [12], an approach is presented that uses the notion of proofs of correct execution. As before, these proofs consist of some execution information and the final result. The idea now is that there exists a more efficient way to check the computation by checking the proof than by recomputing the execution of the agent. These holographic proofs can be used to prove the existence of an execution trace that leads to the final state of an agent by checking only constantly many bits of the proof. A protocol that uses proof verification is described in [1]. Here, all proofs are sent to the agent originator, which checks the proofs after the agent finishes with its task. At first, it was only known that this holographic proof has a length l^d $(d>0)$ where l is the size of the execution trace. Biehl, Meyer and Wetzel proved in [1] that there exist proofs that are even sublinear or polylogarithmic in the size of the agent’s running time.

The problem of proof verification is, that currently, only NP-hard algorithms are known to construct holographic proofs. Therefore, this approach does currently not lead to practical mechanisms and will be, therefore, not considered here further.

3.5 Analysis

To obtain a better understanding of the protection bandwidth of the class of mechanisms that use a reference state, we have to extract the generic attributes out of the presented mechanisms and the relations between these attributes. These attributes are:

moment of checking

The reference state can be checked either
a) after every execution session on one host
b) after the agent has finished its task

Since the overall aim is to identify the host(s) that attacked an agent during its journey, and since malicious hosts may occur anywhere along the route, choosing b) also means that first, the route, i.e. the list of visited hosts has to be stored somewhere in a secure way. This can happen either by dynamically recording the stations, appending this information digitally signed to the agent data, or by sending this information to the owner upon every migration, or by having an a-priori, signed itinerary. Second, the used reference data has to be stored for every of the execution sessions, since, without this precaution, the malicious host cannot be identified.

In principle, one could think of checking in smaller time intervals, e.g. on the level of single statements. In reality though, you have to wait until an agent left a host since a host can always run two agents, a correct executed one and a manipulated one. Then, the agent that was executed correctly can be used to produce the (correct) checking output while using the manipulated agent in reality. Therefore, using a smaller time interval would not say anything, except that the host ran one agent correctly.

used reference data

Depending on the moment of checking, the reference data used by the algorithm might differ. If the execution is checked after an execution session or after the agent fulfilled its task, a combination of the initial state, the resulting state, the input to the session, the execution log and the replicated resources can be used.

used checking algorithm

Independent from the moment of checking, any of the following checking algorithms can be used (note that the presented algorithms mark only some points in the continuous bandwidth of possible algorithms):

rules

This term subsumes simple (i.e. non turing complete) rule mechanisms that allow to check e.g. postconditions in form of first order logic (e.g. moneySpent + moneyRest = moneyInitial). As has been argued in Section 3.1, such mechanisms may not detect all attacks, but often rules are easy to state and to check. Rules may use any of the presented reference data.

proofs

The term proofs denotes in this context representations of execution traces that are easier to check than execution traces themselves. Proofs do not need reference data as parameters, as they include all relevant data.

re-execution

Re-executions aims at executing an agent according to the reference specification given the same set of conditions (i.e. input) as the execution to check. As for rules and proofs, the whole checking process can be automated, i.e. supported by system mechanisms. After having re-executed the specified amount of statements (i.e. one, or a session, or a task), both executions are compared. This can be done either by comparing the “execution logs” that can contain e.g. changes in data and execution state, or by comparing the resulting agent states (without finding differences in the execution itself). Therefore, re-execution needs input, initial agent state, and execution log or resulting agent state as reference data. The power of the approach depends on the level of detail of the execution log. In case of using only the resulting state, the host can lie about the messages sent to communication partners (such as “send $100 to the host”). Even if the log contains such messages, it is not possible to check whether such a message was sent by just looking at the logs.

It can be argued that it is impossible to restore the conditions of the original executions for checking as these conditions may include e.g. racing conditions in case of parallel threads (this is no problem for agent systems that allow only one thread per agent).

Imagine e.g. that an agent computes a list out of an input, where the ordering of the elements depends on the timing of two threads the agent uses. Then the list cannot compared simply with the list of another execution as the other list may contain the same elements, but in different order. To solve this problem and the problem that input should be authenticated, a more powerful algorithm is needed.

arbitrary program

This is the most powerful algorithm as it includes the presented ones and allows for more, e.g. a certain compare method for resulting states or the possibility to ask a communication partner about received messages. Since this algorithm is not known in advance, the system can offer only basic support, i.e. the possibility to execute the program at the checking moments. Therefore, any of the reference data may be used by this checking mechanism.

The combination of these attributes opens a space of potential mechanisms that is much larger than the four approaches we have seen in this section. If we want to allow the programmer to choose a protection mechanism that is appropriate for his/her specific application, we have to offer him/her a framework instead of a single mechanism.

4. Strengths and Weaknesses of Mechanisms Using Reference States

As mentioned before, mechanisms using reference states cannot detect all possible attacks by malicious hosts. In this section, we will analyse the bandwidth of the resulting protection, identify applications that cannot be protected and discuss possible extensions.

4.1 Resulting protection bandwidth

The protection bandwidth depends on the used attributes. i.e. the moment of checking, the used reference data, and the checking algorithm. A mechanism at the lower end of the protection scale uses only the weakest attributes, i.e. it checks after the execution task, uses only the resulting agent state, and employs rules to check the execution. Since the check takes place after the agent fulfilled its task, a compromised agent (i.e. one that has been attacked) continue to work on other hosts. Unwanted actions the agent may have done as a result of the attack in interactions with honest partners can be blamed to the attacker, but it may be difficult to compensate them. Checking only the resulting states by using rules allows to detect only attacks that differ in these states, and that are detectable by the rules. If a rule e.g. checks whether the initial money equals the spent sum plus the remaining amount, an attack that led to an unwanted purchase cannot be uncovered. Although this mechanism can be performed very efficiently and does not delay the execution on the different hosts, it is not very powerful from a security point of view.

A mechanism at the higher end of the protection scale checks after every execution session, uses all possible reference data and allows for an arbitrary checking algorithm. If the next host checks the execution of the former host, it can be sure to execute an uncompromised agent in case of a successful check. Since the mechanism allows for re-executing the agent, the computation of a former host is comprehensible. If the checking mechanism additionally allows to ask the communication partners whether a certain input was issued by them, this aspect can be protected also. Obviously, this mechanism is more powerful than the simple one above. But its disadvantage is its computational and communication overhead: first, the computation is roughly doubled, and second, the system has to transport one more agent state plus the input at a host.

In case of the detection of a fraud, the question of the consequences remains. In a setting where an attacker can harm a party without consequences, just detecting attacks is useless. Only if legal, organizational or social steps can be taken, schemes like the presented one make sense. Although these considerations affects the overall security, they are outside the scope of this paper. Nevertheless, they deserve future examination.

4.2 Applications that cannot be protected

Attacks that do not result in a different agent state cannot be detected by using the presented protection scheme. Especially read attacks, i.e. attacks that aim solely at the knowledge of agent data, lie outside the scope, as these attacks do not leave traces in the agent state. If the goal is to achieve an complete agent protection, other mechanisms have to be developed for this purpose. Other attacks that cannot be detected are attacks where the executing host lies about the input an agent receives, and finally attacks, where the host forces the agent to do something (like buying a good), and, subsequently, migrates another, not compromised version of the agent. It can be argued that the latter attack is rather equal to a read attack, where the host learns about some agent data, and then uses it to harm an agent owner, but such a read attack may require more knowledge about the inner structure of the agent than one that just misuses an agent.

4.3 Possible extensions

To prevent the pretention of false input data, input can be used, that is signed by the party that produces the input, and which can be verified by the checking party using cryptographic means. Another possibility is to use a trusted third party that is used as a relay for input to the agent, so the input data is no longer controlled by the host.

If the attack that misuses the agent has to be prevented, again the idea of using a trusted third party can be used to establish a kind of proxy object for the agent, situated on another host. Parties wanting to interact with the agent have then to use this proxy object which is, therefore, able to log the interactions of the agent.

5. A Checking Framework for Mobile-Agents-Systems

In this section, a framework is presented that supports the implementation of a wide range of checking mechanisms using reference states. It provides functionality for employing the generic attributes found in the last section. The idea is to let the agent programmer decide about the check mechanism a host has to execute and to offer basic functionality like signing by the framework. Although it is implemented for the mobile agents system Mole, the presented scheme can be used for nearly every mobile agent platform implemented in Java that uses a weak migration scheme (i.e. that lets the programmer encode the execution state of an agent manually into variables that are transported automatically, and that executes a start procedure after every migration), and offers callback methods in agents called by the host. This is the case for most systems (see [6] for characteristics of most mobile agent systems). Since we want to support the generic attributes, we explain the framework in relation to these attributes:

moment of checking

Here we need callbacks for the different moments (see Fig. 4), i.e. after an execution session on one host, and after the agent fulfilled its task. The callback for the check moment after an execution session is called `checkAfterSession`. It is called as the first action on the next host, as it would be useless to check a session on the same host since then the host could also manipulate the check. The callback for the moment after the agent finishes its task is called `checkAfterTask`. It is called by the last host that executes the agent, often the home host of the agent.

used reference data
Here we have to do only two things: First, we have to make sure that, at the end of an execution session, we have the needed data in a form that allows to check the execution. Second, we have to transport this data. For mobile agents, the latter is trivial. All we have to do is to include the data in the data part of the agent as this part is transported automatically. For the former, we have to do more. The initial and resulting states are no problem since it is exactly this portion of data that has to be transported to and from the executing host. Replicated resources are simply objects that are appended to the agent (although this part may be large). To create an input list or an execution log, two ways can be followed. Either this information is collected by a modified Virtual Machine (which has easy access e.g. to the line numbers that may be included in the execution log), or written to special containers by code that is instrumented either automatically or manually. Using manually instrumented code has the advantage that the programmer can specify the type and format of the data, which can be more efficient if the checking algorithm is also provided by him/her.

Callbacks in the agent

`checkAfterSession()`
This method is called by the host as the first action when arriving

`checkAfterTask()`
This method is called by the last host

Interfaces implemented by agent

`InitalStateRequester`
declares need for initial state

`ResultingStateRequester`
declares need for resulting state

`InputRequester`
declares need for input

`ExecutionLogRequester`
declares need for execution log

`ResourceRequester`
declares need for host resources

Fig. 4: Framework methods agent

Finally, we want to choose which reference data we will use for checking. In case of creating reference data by manually instrumented code, this is done by the programmer in the routines that create this data, but if we have automatic support for creating reference data, this has to be pointed out to the framework. This can be done by declaring the implementation of interfaces named `InitalStateRequester`, `ResultingStateRequester`, `InputRequester`, `ExecutionLogRequester`, and `ResourceRequester`, similar to the usage of `Clonable` in Java.

Methods offered by host

`Object getInitalState()`
returns the intial state

`Object getResultingState()`
returns the resulting state

`Object getInput()`
returns the input

`Object getExecutionLog()`
returns the execution log

`Object getResource()`
returns the host resources

Fig. 5: Framework methods host

used checking algorithm
As the “arbitrary program” alternative is, on the one hand, the most powerful approach and, on the other hand, includes all other alternatives, it is enough to execute code written by the agent programmer when we want to check an execution. If we want support the other approaches, we can either choose to let the programmer include supporting code or we can offer this code in the system. Rules can be supported either by using a rule mechanism evaluating the desired formalism or we can encode the rules manually as program statements. For supporting proofs, we have to know the structure of the proof. If it consists of data, the proof can be transported as a part of the agent. Then we would have to include only the routines to check the proof. If the proof consists also of code, it can be encoded as an arbitrary program. Support for re-execution may happen on different levels. The problem is the question of how the original code can be used for re-execution. First, the code has to be executed a second time using the input taken from the reference input data. Second, output actions can be suppressed as they are not needed for checking the execution. Third, the resulting state has to be compared with the one of the original execution in a manner that can be specified by the agent programmer (due to the problems discussed in the last section). Solutions to this problem include a modified execution environment (i.e. a Java Virtual Machine) that is able to use the reference input set instead (in this case the unmodified code can be used), a copy of the orig-

inal code, automatically instrumented by statements that do the needed actions (i.e. second execution, output suppression, and state comparison), and finally, a copy of the original code that is instrumented manually by the programmer. To explore this aspect, the last solution was examined for the example application (see next section).An Example for Using the Framework

To illustrate the framework we choose a mechanism that is powerful, and that is not covered by the existing approaches. Using this mechanism we protected a generic example agent, and measured the overhead caused by using the protection mechanism. We used as a first step a complete manual approach, i.e. one where the programmer manually instruments the code to create the required reference data.

5.1 Used checking mechanism

A new checking mechanism was chosen to demonstrate that not only the existing approaches described in Section 3 can be based on the framework, but also other algorithms that are based on the idea of resulting states. The mechanism is described in detail in [4] and can be sketched here only roughly.

The mechanism is based on the “Traces” approach by Giovanni Vigna [10], but uses another moment of checking. In the Vigna approach, the owner needs a suspicion to start checking. In contrast to that, we decided to check an execution session in every case instead of the whole task if needed. For performance reasons, we decided to use the next host to check the execution session of the current host regardless of whether this next host is a trusted one (like the home host) or an untrusted one. This decision has the disadvantage that collaboration attacks of two and more consecutive hosts cannot be detected, but allows on the other hand to check the execution more timely and allows to prevent attacks due to the fact that checking happens regardless of whether the owner has a suspicion or not. As reference data, the initial and the resulting state of an execution session are used as well as the input to this session. The mechanism uses digital signatures and secure hash algorithms to authenticate the data a host produces. To prevent an attack by the checking host, initial states have to be signed by both the checking host and the checked host. The mechanism is optimized in the sense that execution sessions on trusted hosts are not checked (trusted hosts will not attack by definition). Finally the mechanism is able to present the complete state of an attacked agent instead of only hashes of the state, so the owner is able to prove his/ her damage in case of a fraud. The checking mechanism puts an overhead to the execution of the agent that can be expected to roughly double the costs of the execution of the unprotected agent (see [4] for a more detailed analysis).

5.2 Generic example agent

To demonstrate the framework and the used checking mechanism, a generic agent was implemented. After that, a second agent was created based on the first one, but protected using the mechanism described in Section 6.1. This agent migrates along a path of three hosts, where the first and the last host are trusted, the second one is untrusted. The agent can be parametrized by two values. The first parameter determines a “cycle” value, where every cycle means an integer summation of 1000 values. This summation cycle emulates the computational parts of an agent. In the measurement, a cycle value of either 1 or 10000 was used. The second parameters determines the number of input elements to the agent. Each input element consisted of a 10 byte string. In the measurement, a value of either 1 or 100 was used. Using these values, four different agent instances were generated and measured: 1 very small one, 1 with almost no input, but some computation, 1 with almost no computation, but 100 input elements and 1 agent that both computed some time and received 100 input elements. These four agents were executed two times: “plain”, without using the protocol (but being signed and verified as a whole) and “protected”, using the protocol.

5.3 Measurements

The measurement was implemented for the mobile agents system Mole [7], which uses Java as the agent programming language. As a security package, IAIK-JCE 2.0 [5] was used, which offers a pure Java implementation of different cryptographic algorithms. For signing purposes, DSA using a key length of 512 bits was chosen.

Table 1: Measured times for plain agents in [ms]

	sign & verify	cycle	remainder	**overall**
1 input, 1 cycle	209	2	93	**304**
100 inputs, 1 cycle	409	3	153	**564**
1 input, 10000 cycles	217	27158	93	**27468**
100 inputs, 10000 cycles	400	27235	155	**27789**

Table 2: Measured times for protected agents in [ms]

	sign & verify	cycle	remainder	**overall**
1 input, 1 cycle	237 (1.1)	3 (1.7)	345 (3.7)	**584 (1.9)**
100 inputs, 1 cycle	560 (1.4)	4 (1.5)	670 (4.4)	**1234 (2.2)**
1 input, 10000 cycles	235 (1.1)	36353 (1.3)	341 (3.7)	**36929 (1.3)**
100 inputs, 10000 cycles	472 (1.2)	36272 (1.3)	1983 (12.8)	**38727 (1.4)**

Table 1 shows the measured times for the four plain agents, Table 2 shows the corresponding times for the protected agents. The numbers in brackets in Table 2 denote the overhead factor compared to the values in Table 1. The last column shows the measured overall times, i.e. from starting the execution on the first host to the end of the execution on the last one. The times in the “sign&verify” column denote the time needed to compute and verify the complete message signature. The “cycle” column denotes the time needed for the summation cycles. The “remainder” column finally determines the times for all actions that do not fall into the other two categories.

In the configuration used for the measurement, a plain agent executes its main routine three times, a protected agent four times since one check is required. Therefore, the factors of the “cycle” column range about the value 1.3. The values in the “sign&verify” column change only moderately when using the protocol since signing more data needs not much more time compared to the time needed to start the signature. In the remainder column the protocol has to compare, sign and verify single states. Therefore, this value is much higher (by a factor of about 4) for a protected agent.

For the overall values, the factors range from 1.3 and 1.4 for the two agents with an overwhelming portion of computation (i.e. cycle) of over 95% to 1.9 and 2.2 for the two agents without much computation. Since in the measurements only migration in one address space was used, no code transfer was needed. The code that may be transferred can be divided in to parts: one part that includes the functionality of the plain agent and one part that contains the protection mechanism. The first part is the same for both agents as the protected agent offers the same functionality. For often used agent classes, this part may be already stored in a cache of the host. The second part may be unique for an agent, i.e. it may have to be transported over the net.

Please note that the times were measured without using a just-in-time compiler. By using such a compiler, the times are reduced by a factor of 0.6 for the first two agents and by about 50 for the last two agents.

6. Conclusion and future work

Security is an important aspect of using open mobile agent systems, especially in the area of electronic commerce. While other problems seems to be soluble today, the protection of mobile agents from attacks by their executing environments is still not completely solved if only software means can be used. One important area of protection mechanisms employs “reference states”, i.e. agent states that have been produced by non-attacking, or reference hosts. To allow the programmer to select a mechanism out of this area that is adequate for his/her application, a framework is needed that is able to offer support for these mechanisms. Such a framework has been presented in this paper after having extracted the abstract attributes of four existing approaches that use reference states. To illustrate the framework, an example mechanism has been described that uses an approach different to the existing ones. Using this mechanism, the overhead of using the protection algorithm has been measured for a generic mobile agent. It showed that the example mechanism roughly doubles the costs of the execution while offering a good level of protection in case of non-collaborating attackers.

The presented class of mechanism is not able to prevent every attack, but protects a mobile agent from modification attacks that result in a state different from the reference state. To complete the protection level, another mechanism has to be found that prevents read attacks, i.e. attacks that aim at reading data values contained in the agent. If such a mechanism exists and if it can be combined with the framework introduced in this paper, the goal of protecting mobile agents from malicious hosts may be in range of practical usability.

References

[1] Biehl, Ingrid; Meyer, Bernd; Wetzel, Susanne: Ensuring the Integrity of Agent-Based Computations by Short Proofs, in: Kurt Rothermel, Fritz Hohl (Eds.): Mobile Agents, Proceedings of the Second International Workshop, MA’98. pp 183-194. Springer-Verlag, Germany, 1998

[2] Farmer, William; Guttmann, Joshua; Swarup, Vipin: Security for Mobile Agents: Authentication and State Appraisal, in: Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS), Springer Verlag, pages 118-130, September 1996

[3] Hohl, Fritz: Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts, in: Giovanni Vigna

(Ed.): Mobile Agents and Security. pp 92-113. Springer-Verlag, 1998

[4] Hohl, Fritz: A New Protocol Protecting Mobile Agents From Some Modification Attacks. Technical Report Nr. 09/99, Faculty of Informatics, University of Stuttgart, Germany, 1999. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/tr9909.ps

[5] The IAIK JCE project page. http://jcewww.iaik.tu-graz.ac.at/

[6] The Mobile Agents List. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html

[7] Baumann, Joachim; Hohl, Fritz; Rothermel, Kurt; Straßer, Markus: Mole - Concepts of a Mobile Agent System, World Wide Web, Vol. 1, Nr. 3, pp. 123-137, 1998

[8] Minsky, Yaron; van Renesse, Robbert; Schneider, Fred; Stoller, Scott: Cryptographic support for fault-tolerant distributed computing, in: Proceedings of the Seventh ACM SIGOPS European Workshop, pages 109-114, Connemara, Ireland, September 1996. http://www.tacoma.cs.uit.no/papers/SIGOPS.ft-agents.ps

[9] Sander, Tomas; Tschudin, Christian F.: Protecting Mobile Agents Against Malicious Hosts, in: Giovanni Vigna (Ed.): Mobile Agents and Security. pp 44-60. Springer-Verlag, 1998

[10] Vigna, Giovanni: Cryptographic Traces for Mobile Agents, in: Giovanni Vigna (Ed.): Mobile Agents and Security, pages 137-153. Springer-Verlag, 1998

[11] Wilhelm, U.G.; Staamann, S.; Buttyàn, L.: Introducing trusted third parties to the mobile agent paradigm, in: J. Vitek and C. Jensen, editors, Secure Internet Programming: Security Issues for Mobile and Distributed Objects, Lecture Notes in Computer Science, pages 471-491. Springer-Verlag, 1999.

[12] Yee, Bennet:A Sanctuary for Mobile Agents. Technical Report CS97-537. Computer Science Department, University of California in San Diego, USA, 1997