Universität Stuttgart
Fakultät Informatik

A Protocol
to Detect Malicious Hosts Attacks
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 Protocol
to Detect Malicious Hosts Attacks
by Using Reference States**

Fritz Hohl

Bericht Nr. 1999/09
August 1999

A Protocol to Detect Malicious Hosts Attacks by Using Reference States

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 presents a new protocol using reference states by modifying an existing approach, called "traces". In contrast to the original approach, this new protocol offers a model, where the execution on one host is checked unconditionally and immediately on the next host, regardless of whether this host is trusted or untrusted. This modification preserves the qualitative advantages like asynchronous execution, but also introduces two new problems: input to the execution session on one host cannot be held secret to a second host, and collaboration attacks of two consecutive hosts are possible. The overhead needed for the protocol roughly doubles the cost of the mobile agent execution.

1 Introduction

Mobile agents are program instances that are able to migrate from one agent host 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 host. Code can be considered to be constant during the lifetime of the agent whereas the data and execution state contain constant and variable parts. Mobile agents and other mobile code entities extend the potential of (stationary) distributed systems by the possibility of programs being executed on computers that are often not maintained by the employer of that program. As then two parties are involved in running a program, guarantees have to be given that one party will not harm the other.

One aspect of this problem is the fear of the computer owners of inviting viruses, worms and trojan horses to damage their system. As the owners do not know the arriving program in advance in every case, there have to be technical means that prevent mobile programs from becoming a nightmare. Fortunately, with the advent of mobile code systems like Java applets, this aspect has been investigated already to an extend where it does not seem a principal problem to prevent mobile agent from attacking the hosts they are executed on (although, of course, further research is needed to obtain adequate solutions for mobile agent based applications).

The other aspect of a program not being executed by its employer anymore is the protection need of a mobile agent against potential attacks by the executing party. This need exists even in modest applications, especially of those in the electronic commerce domain. Here, the problem is not already solved by mobile code systems since it is mainly the difference between mobile code entities and mobile agents that has to be protected: the state of the single mobile agent instance that persists while the agent migrates to the different execution nodes. Furthermore, the problem of protecting running programs from their runtime environments itself seems to be principally difficult.

One approach to protect mobile agents, i.e. to prevent attacks by malicious host, is the employment of reference states. The execution of a mobile agent can be seen as a list of resulting agent states, i.e. the combination of data and execution state at the end of an execution on a single host. The reference state is a resulting agent state produced by an agent executed on a reference host, i.e. one that did not attack the agent. If a reference state can be computed somehow, attacks that differ in the resulting state from a reference state, can be detected.

This paper presents a new protocol using reference states. This protocol differs from previous ones mainly by the decision to check the execution of a mobile agent on a previous host in every case on the next host.

This paper is structured as follows. First, the problem of malicious hosts is presented in Section 2. Section 3 describes the idea of reference states starting from a general definition of the term "attack". In Section 4, approaches are presented that use the idea of reference states, an analysis discusses problems related to these approaches. Starting from this analysis, Section 5 presents the idea of a new protocol that recombines ideas from the previous approaches. This protocol is described in Section 6. After a discussion of the strengths and weaknesses of the new protocol, in Section 7 measurements are presented that indicate the practical usability of the approach in terms of overhead costs. Section 8 concludes the paper and presents future work.

2 The Problem of Malicious Hosts

In this section, the problem of malicious host is introduced. Further, approaches that tackle this problems are sketched roughly. Two more detailed analysis of the problem of malicious hosts can be found e.g. in [ST98] and [Hoh98].

The fact that the runtime environment (the host) may attack the program (the agent), hardly plays a role in existing computer systems. Normally, the party that maintains the hosts also employs the program. But in the area of open mobile agents systems, an agent is in most cases operated by another party, the agent owner. This environment leads to a problem that is vital for the usage of mobile agents in open systems: the *problem of malicious hosts*. A malicious host can be defined informally as a party that is able to execute an agent that belongs to another party and that tries to attack that agent in some way (a better definition is presented in the next section).

To illustrate this problem we will use a small purchase agent as an example. The central procedure `startAgent`, which is called after the agent is initialized, could look like this:

```
public void startAgent() {

  if (shoplist == null) {
    shoplist = getTrader().
     getProvidersOf(“BuyFlowers”);
    go(shoplist[1]);
    break;
  }
  if (shoplist[shoplistindex].
   askprice(flowers) < bestprice) {
    bestprice = shoplist[shoplistindex].
                  askprice(flowers);
    bestshop = shoplist[shoplistindex];
  }
  if (shoplistindex >= (shoplist.length - 1)) {
    // remote buy
    buy(bestshop,flowers,wallet);
    // go home and deliver wallet
    go(home);
    if (location.getAddress() = home) {
      location.put(wallet);
    }
  }
  go(shoplist[++shoplistindex]);
}}
```

Out of the number of possible attacks of a malicious host, two will be shortly described:

“Stealing” the electronic money

If the value of the electronic money “coin” in variable `wallet` is defined by the knowledge of the bitstring representation of it, the attacker can simply copy the bitstring and spend the money. The agent cannot see the attack when it happens; it may notice it when the agent tries to spend the money some migrations later.

Suppress other offers
There are two possible ways to reach this attack goal. First, the attacker can simply alter the value of `bestshop` to one that is preferred by the attacker. When it now also alters the value of `shoplistindex`, the agent will buy at the preferred shop the next time `startAgent()` is called. The second way is to manipulate the way the statements of the program are executed. If the `if`-statements of lines 9 and 15 always yield true, then the same effect is reached. Note that this manipulation does not leave traces as the code is not altered, just the way it is executed.

2.1 Existing approaches

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. [WSB99]). 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. 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 [ST98], aims at 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 [Hoh98]. 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.

3 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.

3.1 Agent execution model

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

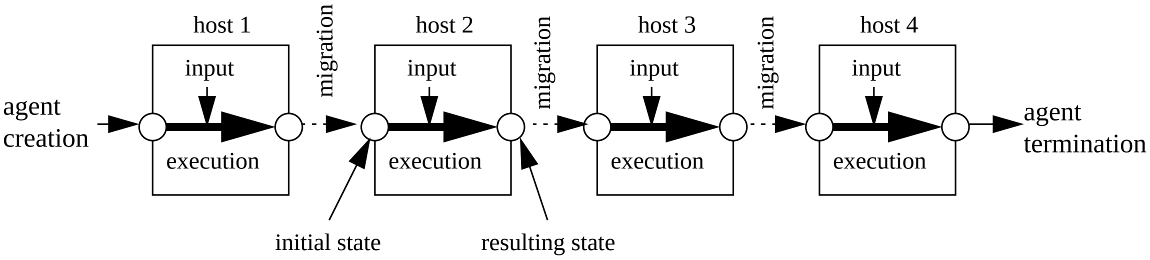

Fig. 1: Agent execution model

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.

3.2 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.

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:

Def: *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.

3.3 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:

Def: *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.

3.3.1 Attacks that cannot be detected using reference states

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 Approaches that Use Reference States

In this section, we will present the four existing approaches that fall into the area of mechanisms that use a kind of reference state to detect attacks by the host. Afterwards we will analyse them and discuss problems related to these approaches.

4.1 State appraisal

Farmer, Guttman and Swarup present in [FGS96] 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).

4.2 Server replication

In [MRS96], 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.

4.3 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 [Vig98], 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 2b.

```
10  read(x)
11  y=x+z
12  m=y+1
13  k=cryptInput
14  m=m+k
```

Figure 2a: Code fragment

```
10  x=5
11
12
13  k=2
14
```

Figure 2b: 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. These 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.

Due to performance reasons, Vigna proposes 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.

4.4 Proof Verification

In [Yee97], 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 [BMW98]. Here, all proofs are sent to the agent originator, which checks the proofs after the agent finishes with its task. 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.

4.5 Discussion

We will now discuss the advantages, disadvantages and costs of the approaches (except the proof verification mechanism) presented before. Then we will analyse whether these approaches already offer the characteristics needed for an every day´s employment. As we will find that a slightly different attack detection model is needed, we will discuss how to achieve such a model using elements of the presented approaches.

4.5.1 Advantages and disadvantages of the approaches

State appraisal

State appraisal is a very simple approach: easy to use, and easy to implement. On the other hand, it may be too simple for some purposes. The lack of the input to the agent e.g. 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. Another problem is 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. Finally, this approach may under certain circumstances not be able to detect collaboration attacks of consecutive hosts. If a rule e.g. checks whether the lower of the last two prices is stored as the lowest price so far, the last two host could agree on another, higher lowest price. The additional costs of the approach are: the transport of the checking mechanism, and one execution of the checking mechanism per visited host.

Server replication

The server replication approach has one big advantage: it tolerates collaboration attacks of even (n/2 - 1) collaborating malicious hosts. The big disadvantage is that it is unrealistic to assume an environment where a number of replicated hosts fail independently (see [Yee97]). If hosts are replicated, normally either all replicated servers are under the same administrative control (as at least commercial companies seem not willing to give their data to other parties) or run on identical systems, thus making external attacks at all servers possible if the attacker is able to attack one host. The additional costs of the approach are: the transport of one entire agent and its input, and the execution of this agent per additional host. As the hosts of a stage share the same resources, input in this case means only data from outside the executing hosts. For collecting the “votes”, the transport of one signature per additional host is required.

Traces

The traces approach detects attacks even in case of (n - 1) malicious hosts. In contrast to the other two approaches, the execution of the agent is not done in every case (but only in case of a suspicion of the owner), and even if so, the check takes happen after the agent returned home. The additional costs of the approach can be distinguished into the costs that occur in every case, and the costs that occur in case of a check. In the first case, only some signatures have to be transported to the next host after an execution. In the second case, one agent state and the input to an execution has to be transported to the home host per host visited. Additionally, the agent has to be executed a second time (even if the input data has not to be computed by a host).

4.5.2 What do we need?

As the state appraisal approach is not powerful enough, and the server replication approach does not fit into a realistic scenario of an already existing organizational structure, the traces approach offers the best choice of the existing approaches. But the two disadvantages of the approach, checking only in case of a suspicion and doing it after the agent returned home, first, allow an attacker to have a chance not to be detected, and second, allow an attacked agent to be executed on a consecutive honest host. Both problems weaken the protection of the mobile agent, although the usage of a reference state would allow to do better. Therefore, what we need, is a modified traces approach that do not have these two disadvantages.

5 Towards A New Protocol to Detect Result Inequivalent Attacks

Our aim is to modify the traces approach in a way that every execution on a host is checked after this execution unconditionally, i.e. without the need of having a suspicion. At first, this seems simple: the home host has to check the execution after it took place on a host. After a successful check, the home host allows the agent to migrate to the next host. The problem is that then the usage of mobile agents for the application is useless. The home host executes the whole agent from start to return, getting the input to the agent remotely from the hosts it occurred. This model is widely known as client-server, and there is no need for migrating mobile agents then. Additionally, this modified approach costs more since a set of other host execute the agent a second time, and the agent has to be shipped from host to host. Finally, this simple modification abrogates one of the advantages of using mobile agents: asynchronous execution. As mobile agents offer the possibility to run autonomously, i.e. in this case, independent of a connection of the home host to the agent system, the mobile agent paradigm can be used e.g. for applications based on mobile user devices. Normally, these devices are connected to the Internet by expensive connections that offer only limited bandwidth and reliability. A model, where the executing host had to wait on a connection and computation of the mobile devices, which has often also very limited resources, would be not suitable for that kind of environment.

What we really need is a variant of the traces approach that uses the idea of the state appraisal mechanism to use the following host to check the execution of an agent at the current host. This variant will have the disadvantage that collaboration attacks of two ore more consecutive hosts cannot be detected. If this flaw is a problem for a given environment or an application, the idea of the server replication approach of using a parallel set of hosts executing the same agent can be used to detect collaboration attacks of (in this case) n-1 hosts.

5.1 The main idea

The main idea of the protocol presented in the following is to use the checking mechanism of the traces approach, i.e. re-computing the execution session of one agent on a host using the input to the agent during this session. In contrast to the traces approach, this re-execution is not done on the home host, but, like in state appraisal approach, on the next host (see Figure 3). Again, the input contains all of the data that comes from the outside of the agent. This includes communication with third parties such as messages, and data that was produced by the host using its non-shared resources like databases, random generators or a system clock.

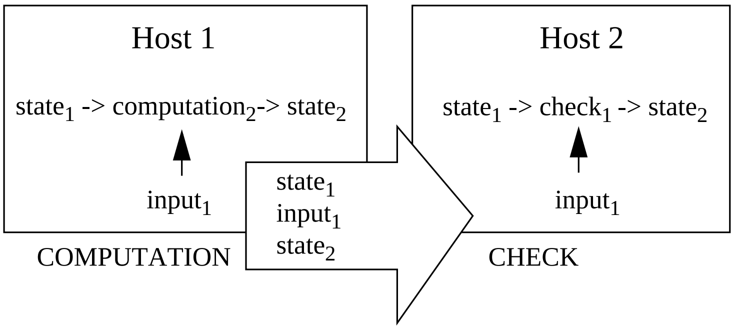
Host 1
state$_1$ -> computation$_2$-> state$_2$
input$_1$
COMPUTATION
state$_1$
input$_1$
state$_2$
Host 2
state$_1$ -> check$_1$ -> state$_2$
input$_1$
CHECK

Fig. 3: Re-execution mechanism

Without using a special protocol, a host can modify, suppress or insert third party communication at its will.

To allow an immediate checking of an execution session, the re-execution takes place on the next host regardless of whether this host is trustworthy or not.

In the following, a *trusted* host is one that is known in advance to not attack the agent. Additionally, a trusted host is willing to execute checks and the like for an agent. Trusted hosts are visited either because the agent wants to be executed there to fulfil its main task, or because it wants the host to check something. If we assume that the migration starts and ends at the "home" host, then the first and last host can be also assumed as being trusted.

An *untrusted* host is one from which it is not known whether it will attack or not. Untrusted hosts are visited since the agent wants to have something executed there that relates to the agent's task. Also untrusted hosts are willing to execute checks if this work is needed to make sure that they cannot be made responsible for a harm caused by an attack against an agent. However, untrusted agents may try to attack an agent even while checking.

6 The New Protocol

Now, we will present a protocol that allows to check a computation on the next host regardless of whether these hosts are trusted or untrusted. We will start by having a look on the protocols of each host for an example configuration. This configuration consists of a list of hosts an agent wants to visit one after another. To be as general as possible we will consider that only the first and the last host are trusted, all other hosts are untrusted.

6.1 Non-optimized protocol

Note that in the following, $\text{sign}_x(y)$ means only the signature of message y by host x, i.e. $\text{sign}_x(y)$ does not include message y.

```
1.1   compute $\text{state}_2$
1.2   transfer agent code to next host(s) if needed
1.3   add $\text{sign}_1(\text{state}_2)$ to message
1.4   add $\text{state}_2$ to message
1.5   sign message and send it to next host
```

Fig. 4: Protocol for first node

The first node (i.e. the one that starts the agent) has simply to compute and sign the initial state, transfer the code, the signature and the initial state of the agent to the next node. Like in all protocols of this section, the message containing all transferred data is signed by the host.

The second node first gets the message from the previous host and checks the signature. If the signature cannot be verified, the previous host is informed. In this case, the previous host can either resend the message or stop the agent. As this aspect does not concern the question of whether a previous host attacked an agent, this aspect will not discussed further in this paper.

```
2.1    get message from previous host and check signature
2.2       if not true, COMPLAIN and STOP
2.3    get $state_2$, $sign_1(state_2)$ from message
2.4    check $sign_1(state_2)$
2.5       if not true, COMPLAIN and STOP
2.6    compute $state_3$ from $state_2$ using $input_2$
2.7    transfer agent code to next host(s) if needed
2.8    add $sign_1(state_2)$, $sign_2(state_3)$ to message
2.9    add $input_2$, $state_2$, $state_3$ to message
2.10   sign message and send it to next host
```

Fig. 5: Protocol for second node

If everything is ok, the second node starts executing the agent using some input (the aspect of checking the signature in line 2.4 is discussed in the next paragraph). The resulting state is signed by the host, this signature and the one received by the first host, the states, the input and -if needed- the code is sent to the next host.

```
3.1    get message from previous host and check signature
3.2       if not true, COMPLAIN and STOP
3.3    get $state_{i+1}$, $sign_i(state_{i+1})$ from message
3.4    check $sign_i(state_{i+1})$
3.5       if not true, COMPLAIN and STOP
3.6    get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
3.7    check $sign_{i-1}(state_i)$
3.8       if not true, COMPLAIN and STOP
3.9    compute $state_{i+1}$ from $state_i$ using $input_i$
3.10   check if $state_{i+1}$ from line 3.9 and $state_{i+1}$ received from predecessor differ
3.11      if true, COMPLAIN and STOP
3.12   compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
3.13   transfer agent code to next host(s) if needed
3.14   add $sign_i(state_{i+1})$, $sign_{i+1}(state_{i+2})$ to message
3.15   add $input_{i+1}$, $state_{i+1}$, $state_{i+2}$ to message
3.16   sign message and send it to next host
```

Fig. 6: Protocol for untrusted node i+1

From the third node until the node before the last one, the protocol described in Fig. 6 is used. Again, first, the signature of the whole message is checked. Then, the resulting state $state_{i+1}$ from the previous host is taken from the message. Additionally, the signature of the previous host signing this resulting state is taken and checked. This seem to be double work as the whole message, also containing $state_{i+1}$, was already signed by the previous host and checked by this one. The reason for that construct is the fact, that it might be easier to transfer this signature to the next host (see line 3.14) without the need to transport the whole message.

The check phase encompasses lines 3.6 to 3.11. The host first verifies the signature of the host before the previous host on the initial state of the original computation on the previous host ($state_i$). After line 3.8 the current host can be sure that the host before the previous host produced this state and that the previous host also gained this state by either checking the computation (in case that the host before the previous host is untrusted) or by the knowledge that this host is trusted. This agreement is stated by the inclusion of the signature $sign_{i-1}(state_i)$ in the message and by the signature of the whole message by the previous host. The current host then checks the computation of the previous host by re-executing the agent using $input_i$, and by comparing the resulting state with the received one. If the resulting states differ, the last node attacked the agent. In this case, the third node complains, i.e. sends the agent and all other data to the owner of the agent. Then the owner is able to examine the case and to initiate legal measurements if needed. As the owner is then able to present both the signed final state from the second node and a reference state (either from the third node or from re-executing the agent itself), the difference, i.e. the modification can be proved.

Then, the host executes it's normal computation, thus gaining the final state $state_{i+2}$. If needed, code is transferred to the next host. Finally, the current host signs the final state and adds it to the message to the next host. The signature of the previous host signing the initial state of the computation was added, too, so the next host can check the integrity of this state in order to check the computation of the current host itself. The message to the next host is finished by adding the other ingredients needed for checking, i.e. $input_{i+1}$, $state_{i+1}$, and $state_{i+2}$. Then the message is signed and transferred to the next host.

```
4.1   get message from previous host and check signature
4.2      if not true, COMPLAIN and STOP
4.3   get $state_{i+1}$, $sign_i(state_{i+1})$ from message
4.4   check $sign_i(state_{i+1})$
4.5      if not true, COMPLAIN and STOP
4.6   get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
4.7   check $sign_{i-1}(state_i)$
4.8      if not true, COMPLAIN and STOP
4.9   compute $state_{i+1}$ from $state_i$ using $input_i$
4.10  check if $state_{i+1}$ from line 4.9 and $state_{i+1}$ received from predecessor differ
4.11     if true, COMPLAIN and STOP
4.12  compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
```

Fig. 7: Protocol for last node i+1

Finally, the last node simply has to check the computations of its predecessor (see Fig. 7) and to do final computations (if any).

6.2 Optimized Protocol

A problem of the described protocol is that also the computations on a trusted node are checked on consecutive nodes. This is not optimal since a trusted node does not attack. Therefore, if an agent knows about the trustworthiness of nodes, the protocol can be specified in a way that regards this case.

```
5.1   get message from previous host and check signature
5.2      if not true, COMPLAIN and STOP
5.3   get $state_{i+1}$, $sign_i(state_{i+1})$ from message
5.4   check $sign_i(state_{i+1})$
5.5      if not true, COMPLAIN and STOP
5.6   if (predecessor is untrusted) then
5.7      get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
5.8      check $sign_{i-1}(state_i)$
5.9         if not true, COMPLAIN and STOP
5.10     compute $state_{i+1}$ from $state_i$ using $input_i$
5.11     check if $state_{i+1}$ from line 5.10 and $state_{i+1}$ received from predecessor differ
5.12        if true, COMPLAIN and STOP
5.13  compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
5.14  transfer agent code to next host(s) if needed
5.15  add $sign_i(state_{i+1})$, $sign_{i+1}(state_{i+2})$ to message
5.16  add $input_{i+1}$, $state_{i+1}$, $state_{i+2}$ to message
5.17  sign message and send it to next host
```

Fig. 8: Protocol for untrusted node i+1

The protocol for the first node remains the same. According to the trustworthiness of a node, one of the following protocols is used for the following nodes. Fig. 8 describes the protocol for an untrusted node i+1. Compared to the protocol of Fig. 6, the check part (lines 5.7 - 5.12) is executed only in case that the previous host is untrusted.

Compared to the protocol for an untrusted node, the one for a trusted node (see Fig. 9) just lacks the transfer of $sign_i(state_{i+1})$, $input_{i+1}$, and $state_{i+1}$ to the next host. These are exactly the elements for checking a computation. As node i+1 is trusted, they are therefore not needed at the next host.

```
6.1   get message from previous host and check signature
6.2       if not true, COMPLAIN and STOP
6.3   get $state_{i+1}$, $sign_i(state_{i+1})$ from message
6.4   check $sign_i(state_{i+1})$
6.5       if not true, COMPLAIN and STOP
6.6   if (predecessor is untrusted) then
6.7       get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
6.8       check $sign_{i-1}(state_i)$
6.9           if not true, COMPLAIN and STOP
6.10      compute $state_{i+1}$ from $state_i$ using $input_i$
6.11      check if $state_{i+1}$ from line 6.10 and $state_{i+1}$ received from predecessor differ
6.12          if true, COMPLAIN and STOP
6.13  compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
6.14  transfer agent code to next host(s) if needed
6.15  add $sign_{i+1}(state_{i+2})$ to message
6.16  add $state_{i+2}$ to message
6.17  sign message and send it to next host
```

Fig. 9: Protocol for trusted node i+1

Finally, the protocol for the last node needs only to check previous computations by untrusted hosts. Therefore, it uses the protocol for a trusted node without the last four lines.

6.3 Costs of the protocol

As the protocol acts per session, i.e. per execution of an agent on one host, the costs of the protocol will also be stated per session.

The maximum additional costs required by the protocol per session can be specified as follows:

- 2 signature checks
- 1 signature generation
- 1 execution
- 1 state comparison
- 1 transport of input
- 1 transport of state
- 2 transports of signatures
- additionally, the overall signature is extended by the transported data

if the agent code is stable over the agent's life. The costs for an unprotected session are: 1 execution, 1 transport of state, 1 transport of code. Additionally, the complete transport of an agent to the next should always be signed by the sending host and verified by the receiving one. This requirement results from general security needs, and can, therefore, not counted as overhead costs of the protocol.

If all states of an agent have the same size, and if the executions need the same time, and if the transport of code needs as much time as the generation, the check, and the transport of the signatures plus the comparison of the states, then the protocol roughly doubles the costs for a session on an untrusted host following another untrusted host.

In reality, the costs of the protocol can be less. First, the checking execution does not need to wait for input (which may be the result of a lengthy computation), second, the checking execution can skip output operations (real interaction is not needed). Third, state can be transported as a difference to the previous state.

With a large input (e.g. when an agent searches a database), the costs of the protocol can be much higher. In Section 7, the overhead will be measured for different cases.

6.4 Further optimization

Apart from exploiting the difference between trusted and untrusted hosts, there are several possibilities to further optimize the protocol. One optimization already mentioned is the transport of a state as a difference to the last transported state. As in the protocol always two consecutive states are migrated, the difference is smaller than a whole state if there are common elements. Another optimization tries to shorten the transported input: As we have seen in Section 3, the traces approach does not store the real input to an agent, but the result of the first statement that used that input. Although this method may be questionable in terms of security, it may be useful in terms of cutting down the input size. Imagine a first statement that finds the URLs in an HTML document. Certainly the size of the URLs is smaller than the size of the whole document. Therefore, it is better in this case to store the result of this statement than storing the document. Note that this does not apply in every case. If the first statement is e.g. a decompression function, it is obvious that it is better to store the input data than the output data. Therefore, an optimized protocol could decide about whether to transport the input or the result of the first statement by using these considerations.

Further optimization can result from the fact that the computation and check phases do neither need to be executed totally sequential nor on a single host. The degree of time dependence of the phases depends on the question of whether it is possible to execute an agent without knowing whether the previous host was malicious or not. At least two answers are possible. Either the effects of the execution can be made undone (using a rollback or compensation). Then the host waits with issuing a “commit” until the check and compare phases signal success. Or the effects cannot be rolled back or compensated. Then, the host could await the result of the check and compare phase when the first statement is about to be executed that contains a non-compensative interaction.

Finally, re-execution may be replaced partially by faster methods partially, at least in some cases. Techniques like invariant and state checking does not detect all possible attacks, but, depending on the code of the agent, they may be used to replace parts of the execution. The problem of this optimization technique is, that it is not known yet whether re-execution can be replaced automatically. A first approach to allow faster methods is therefore to let the agent programmer decide what to do, i.e. using a callback method provided by the programmer instead of re-executing code.

6.5 Extension of the protocol tolerating n malicious hosts

If it is possible to use more than one host for re-execution, the protocol can be extended to tolerate n malicious hosts. For that purpose, we need n hosts that check the same computation (i.e. we need n (checking) + 1 (computing) host per session). If now n hosts out of the n+1 hosts collaborate and lie about the resulting state, still one host reports a different state (Either the computing host creates the correct state or one checking host detects a difference in states). Since our goal is to detect fraud, this difference is enough to stop the execution and to allow the agent owner to handle the situation. The described approach is similar to the server replication mechanism presented in Section 3. The difference lies in the fact that a detection approach needs only n+1 hosts instead of 2*n for a fault-tolerance-style mechanism.

6.6 Discussion of the protocol

The protocol presented in this section offers the requested features, i.e. unconditional and immediate checking of an execution session while using the normal mobile agent model, thus conserving the possible advantages of mobile agents. By modifying the traces approach, two disadvantages resulted from this modification compared to the original approach.

The most severe disadvantage is the problem that even attacks are not detected that differ in the resulting state, if two or more consecutive hosts collaborate (it is not enough that any two hosts collaborate). If the second host signs a resulting state that was modified by the previous host, a third host cannot detect this fraud. There are several possibilities to overcome this problem. First, if it is acceptable to wait with the checks, then checks can be done by the next trusted host, which may be the last one in the worst case.

The other problem is the limitation that input cannot be held secret from the checking host(s). This may or may not be a problem depending on the question of whether there are additional mechanisms that protect agent data from being read by the executing host or not. If there is no such mechanism (and currently there is only one approach [ST98] that gives a first clue for the existence of such mechanisms), every host can read all agent data. In this case, there is a privacy problem only for such input data that is normally not transported to the next host. If -in the other case- there are read protection mechanisms, either then the protocol reveals every input to the checking host, or it can use a "protected form" of the input for re-execution at the next host (in case of [ST98], input have to be "encrypted" before transport).

7 Measurements

To evaluate the cost estimation of Section 6.3, the protocol was implemented prototypically for a generic mobile agent using a framework for mechanisms using reference states (see [Hoh00]). 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 bytes 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", with using the protocol.

The measurement was implemented for an mobile agents system that uses Java as the agent programming language. As a security package, IAIK-JCE 2.0 [IAI99] 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 of 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 denote 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 as 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 the signature algorithm has to process only more data. In the “remainder” column the protocol has to do things like comparing, signing and verifying 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. As in the measurements only migration in one address space was used, no code transfer was needed. If such a transfer needs to be done, it would reduce the overhead factors a little bit since code transfer is the same regardless of using the protocol or not.

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.

8 Conclusion & Future Work

Mobile agents not only offer qualitative and quantitative advantages like asynchronous execution and less network usage, but also create new security problems. One of these problems result from the introduction of a new party, the code executor, and the need to protect the data state of the mobile agent: the malicious host problem. A malicious host is one that attacks the agent that is executed on this host. Apart from organizational and hardware-based approaches, a pure software-based solution would pose the least restrictions on the execution environment.

One class of software-based approaches uses the idea of reference states, i.e. agent states at the end of an execution session on one host, generated by a honest host. This class is able to prevent attacks that differ in these resulting states, e.g. modification attacks, but not e.g. read attacks. There are four approaches that use the idea of a reference state. None of them combines the advantages of offering a powerful protection and a model where an execution session is checked unconditionally and immediately, preserving the mobile agent model.

Therefore, a new variant is needed that overcomes these problems. For that purpose a new checking protocol was presented that allows to check a computation that happened on one host. This check is done on the next host by re-executing the computation using the input that occurred during the original execution. The check takes place on the next host regardless of whether this host is trusted or not. If we estimate the additional costs, we find that the protocol roughly doubles the costs of the execution. As the protocol uses a checking scheme that is done on the next host, two problems occur: First, collaboration attacks of two consecutive hosts cannot be detected, and second, input to an original execution cannot be held secret from the checking host without further techniques. For solving the first problem, a protocol variant was sketched that uses n+1 hosts per execution to tolerate n malicious hosts. The second problem has to be handled by future mechanism that allow the prevention of read attacks. To evaluate the protocol it was prototypically implemented for a generic mobile agent using a framework to implement mechanisms using reference states [Hoh00]. The measurements based on this implementation verified the initial estimation of the costs.

For the future, it is planned to examine the question of how input to and output from the agent by a communication partner outside the host can be protected from attacks by the host.

References

[BMW98] 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

[FGS96] 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, pp. 118-130, 1996

[Hoh98] 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

[Hoh00] Hohl, Fritz: A Framework to Protect Mobile Agents by Using Reference States. In: Proceedings of ICDCS 2000, IEEE Press, to appear.

[IAI99] The IAIK JCE project page. http://jcewww.iaik.tu-graz.ac.at/

[MRS96] 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, pp. 109-114, Connemara, Ireland, September 1996. http://www.tacoma.cs.uit.no/papers/SIGOPS.ft-agents.ps

[ST98] 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

[Vig98] Vigna, Giovanni: Cryptographic Traces for Mobile Agents, in: Giovanni Vigna (Ed.): Mobile Agents and Security, pp. 137-153. Springer-Verlag, 1998

[WSB99] 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, pp. 471-491. Springer-Verlag, 1999.

[Yee97] Yee, Bennet:A Sanctuary for Mobile Agents. Technical Report CS97-537. Computer Science Department, University of California in San Diego, USA.