Cryptographic Traces for Mobile Agents

Giovanni Vigna

Dip. Elettronica e Informazione, Politecnico di Milano
P.za L. Da Vinci 23, 20133 Milano, Italy
vigna@elet.polimi.it

Abstract. Mobile code systems are technologies that allow applications to move their code, and possibly the corresponding state, among the nodes of a wide-area network. Code mobility is a flexible and powerful mechanism that can be exploited to build distributed applications in an Internet scale. At the same time, the ability to move code to and from remote hosts introduces serious security issues. These issues include authentication of the parties involved and protection of the hosts from malicious code. However, the most difficult task is to protect mobile code against attacks coming from hosts. This paper presents a mechanism based on execution tracing and cryptography that allows one to detect attacks against code, state, and execution flow of mobile software components.

1 Introduction

Mobile code technologies are languages and systems that exploit some form of code mobility in an Internet-scale setting. In this framework, the network is populated by several loosely coupled *computational environments*, or *sites*, that provide support for the execution of *executing units*, or *agents*. Agents represent sequential flows of computation which are characterized by a *code segment*, providing the static description of the behavior of a computation, and an *execution state*, containing control information related to the state of the computation, such as the call stack and the instruction pointer.

Mobile code technologies can be divided in two sets [6]. *Weakly mobile* technologies allow an application to send code to a remote site in order to have it executed there, or to dynamically link code retrieved from a remote site in order to execute it locally. The transferred code may be accompanied by some initialization data but no migration of execution state is involved. Examples of weakly mobile technologies are Java [19] and the Aglets system [16]. *Strongly mobile* technologies allow an executing unit that is running at a particular site to move to a different computational environment. In this case, the executing unit is stopped and its code and execution state are marshaled into a message that is sent to the remote site. The destination site restarts the unit from the statement that follows the invocation of the migration primitive. Examples of strongly mobile languages are Telescript [30] and Agent-Tcl [12]. A survey of several mobile code languages and systems can be found in [10].

While the mobile code approach to designing and implementing distributed applications provides a greater degree of flexibility and customizability with respect to the traditional client-server approach [4], it raises some serious security issues. Agents travel across the network on behalf of users, visiting sites that may be managed by different authorities (e.g., a university or a company) with different and possibly conflicting objectives [20]. Therefore, mobile code systems must provide mechanisms to protect hosts and execution environments from misbehaviors or attacks coming from roaming agents as well as mechanisms to protect agents from malicious sites. In addition, agents should be protected against eavesdropping or tampering during migration from site to site.

Protection of agents while traveling over an untrusted network can be achieved using well-known cryptographic protocols (e.g., the Secure Socket Layer [9]). Mechanisms and policies to protect a site from code coming from an untrusted source have been the focus of recent research [28,21,32,17]. Using suitable access control and sandboxing mechanisms, it is possible to protect execution environments effectively against a wide range of attacks.

By far, the hardest security problem is represented by the protection of agents from attacks coming from the computational environments that are responsible for their execution. In fact, execution environments must access agents' code and execution state to be able to execute them. As a consequence, it is very difficult to prevent disclosure, tampering, or incorrect execution of agents.

We propose a mechanism that allows for detecting possible misbehavior of a site with respect to a roaming agent by using cryptographic traces. Traces are logs of the operations performed by an agent during its lifetime. The proposed mechanism allows an agent owner to check, after agent termination, if the execution history of the agent conforms to a correct execution.

The paper is organized as follows. Section 2 presents some related work on mobile code and security. Section 3 introduces some concepts and assumptions underlying cryptographic tracing. In sections 4 and 5 we describe the mechanism and its applications. Section 6 discusses the applicability of the proposed approach and some of its limits. In Section 7 we describe a mobile code language that implements the tracing mechanism. The language is then used in a simple electronic commerce application. Section 8 draws some conclusions and illustrates future work.

2 Related Work

Protecting programs against attacks coming from the interpreter responsible for their execution is a challenging problem[1]. Some efforts have been devoted to determining which goals are achievable and which are not [5,8]. For example, it is not possible to guarantee that an environment will execute an agent correctly and to its completion, or to achieve total protection of agent data from disclosure.

[1] Presently, most mobile code systems consider the site as a trusted entity and therefore they do not provide any mechanism to protect agent execution.

Presently, solutions to the problem of protecting mobile agents against attacks coming from their execution environment are aimed at *prevention* or *detection.*

Prevention mechanisms try to make it impossible (or very difficult) to access or modify agents' code and state in a meaningful way. One possible approach is to adopt *tamper-proof devices* [31]. These devices are processors that execute agents in a physically sealed environment. The system internals are not accessible even by its owner without disrupting the system itself. While these systems can provide a high level of protection, they require dedicated (expensive) hardware. Therefore, they are not easily deployed on a large scale. A software-based approach is followed by *code scrambling* [14,25] mechanisms. In this case, the mobile code is "rearranged" before it is moved to a remote site. The technique used to modify the code makes it difficult to re-engineer the code but preserves its original behavior. A simpler solution performs *partial encryption* of the agent's components. Using this mechanism, an agent protects data that must be used at a particular site by encrypting them with the site's public-key. This way, data are accessible only when the agent reaches the intended execution environment. Obviously, this approach requires that (at least part of) the route that will be followed by the agent is known in advance. A new and promising approach exploits *cryptographed functions* [23]. In this case, the mobile code performs an algorithm that, given some external inputs, computes a cryptographed value. The site has no clue about which is the function actually computed and therefore cannot meaningfully tamper with algorithm execution and computation results. Presently, this mechanism has been applied to the evaluation of polynomial and rational functions.

Detection mechanisms aim at detecting illegal modification of code, state, and execution flow of a mobile agent. While static code can be easily protected by using digital signatures, state and execution flow are dynamic components and therefore other mechanisms must be devised. For example, the *state appraisal* mechanism [7] associates a mobile agent with a state appraisal function. When a roaming agent reaches a new execution environment, the appraisal function is evaluated passing as a parameter the agent's current state. The appraisal function checks if some invariants on the agent's state hold (e.g., relationships among variables). This way, some malicious attempts to tamper with the agent's state can be detected.

We introduce a mechanism that aims at detecting *any* possible illegal modification of agent code, state, and execution flow. The mechanism is based on post-mortem analysis of data —called *traces*— that are collected during agent execution. Traces are used as a basis for program execution verification, i.e., for checking the agent program against a supposed history of its execution. This way, in case of tampering, the agent's owner can prove that the claimed operations could have never been performed by the agent.

3 Tracing Execution

The proposed mechanism assumes that all the involved principals, namely users and site owners, own a public and a secret key that can be used for encryption and digital signatures [22]. The public key of a principal A is denoted by A_p, while A_s is used for the corresponding secret key. Principals are users of a *public key infrastructure* [15] that guarantees the association of a principal identity with the corresponding public key by means of certificates. We assume that, at any moment, any principal can retrieve the certificate of any other principal and verify the integrity and the validity of the associated public key.

The process of encrypting[2] a message m with a key K is expressed by $K(m)$. In addition, we will use one-way hash functions in order to produce cryptographically secure compact representations of messages. The hash value obtained by application of the one-way hash function H to the message m is denoted by $H(m)$. The process of signing[3] a message with a secret key is denoted by $X_s(m)$, where X is the signing principal. Several examples of cryptosystems and one-way hash functions can be found in [24].

A moving agent is composed of a code segment p and the associated execution state S^i, which has been determined, at some specified point i, by code execution. The state includes global data structures, the call stack, and the program counter. We assume that the code is *static* with respect to the lifetime of the agent, that is, the agent cannot change its own code segment as the result of its execution. This constraint will be removed in Section 6. The code segment is composed of a sequence of *statements*, that can be *white* or *black*. A white statement is an operation that modifies the agent's execution state on the basis of the value of the agent's internal variables only. For example, the statement $x := y + z$, where x, y, and z are variables contained in the agent's execution state, is a white statement. A black statement modifies the state of the program using information received from the external execution environment. For example, the statement $read(x)$, that assigns to the variable x a value read from the terminal, is a black statement. This is not a new concept. For example, the Perl [18] language implements a security mechanism, called *tainting*, that allows the programmer to keep track of the variables whose value has been determined on the basis of information retrieved from the external environment (e.g., terminal input or environment variables).

A trace T^p of the execution of program p is composed of a sequence of pairs $\langle n, s \rangle$, where n represents a unique identifier of a statement, and s is a *signature*. When associated with a black statement, the signature contains the new values assumed by internal variables as a consequence of the statement

[2] Public-key cryptography is slow when compared to symmetric cryptography. In the sequel, when we will need bulk encryption using public keys we will assume that the message has been encrypted with a randomly generated secret key and that the key has been protected with the original public key. For example, if the application of a symmetric encryption process parameterized by a key K to a message m is represented by $K(m)$, then $A_p(m)$ is equivalent to $A_p(K), K(m)$.

[3] If not explicitly noted, $X_s(m)$ is considered equivalent to $m, X_s(H(m))$.

execution. For example, if the $read(x)$ instruction gets the value 3 from the terminal, the associated signature will be $x := 3$. The signature is empty for white statements.

We make some assumptions about the execution infrastructure. First of all, we assume that all the interpreter implementations are certified correct and respectful of the semantics of the language. For example, the interpreter owner must provide some kind of third-party certification of correct implementation of the language to be able to charge agents for the services used. This way, a site owner cannot claim that, in his/her interpreter implementation, the execution of a *while* statements means "buy two hundred shares of Microzooft in the stock market". Second, we assume that all the principals participating in the infrastructure respond to some trusted party that will be involved in case of (claimed) misbehaviors.

The following two sections address problems of increasing complexity, firstly considering *remote code execution*, and then *mobile agents*. In describing the protocols we employ the following notation:

$$X \xrightarrow{m_i} Y : F_1, F_2, \ldots, F_n.$$

The expression above means that principal X sends message m_i to principal Y. The contents of the message are a sequence of fields $F_1, F_2, \ldots, F_n$.

4 Remote Code Execution

Remote code execution, also known as *remote evaluation*, is a mechanism that allows an application to have code sent to a remote host and executed there. Remote execution is a well-known mechanism that dates back to the 70s, when it was used for remote job submissions [1], and has always been available to UNIX users by means of the `rsh` facility. Remote code execution represents the basis for several mobile code systems like Obliq [3] and M0 [29]. A formal definition of the remote evaluation mechanism is presented in [26].

Suppose now that principal A wants to execute program p on site B. Therefore, A sends B the following signed message:

$$A \xrightarrow{m_1} B : A_s(A, B, i_A, t_A, K_A(p), TTP).$$

The first two fields of the message specify that the message is coming from A and it is directed to B. When B receives m_1, it uses A's public key to verify the message signature. This way, B is assured that the message was actually sent by A and that the message was intended for itself. The third field (i_A) is a unique identifier used to mark all the messages that are involved in this execution request and to protect from replay attacks. The following field (t_A) is a timestamp to guarantee freshness[4]. The next field is the code to be executed,

[4] The timestamp may be associated to a time interval to limit execution time or the validity of the request.

encrypted using a random secret key K_A, chosen by A. The last field is the identifier of a trusted third party (TTP) that will be involved if A or B claim that the other principal is not playing fair.

B can reject or accept the request on the basis of the available information (the identity of the sender, the unique identifier, the timestamp, and the trusted third party chosen by A). In either case B replies with a signed message containing the outcomes of its decision, say M. If B rejects the request and refuses to execute the code, M contains the error message that motivates the rejection. If B accepts the request, M contains an acceptance statement that represents B's commitment to the execution of p and implicitly requests the decryption key K_A:

$$B \xrightarrow{m_2} A : B_s(B, A, i_A, H(m_1), M).$$

A receives m_2 and validates the message. Thus, A is assured that the message was sent by B, that it was intended for itself, and that the message refers to the execution request identified by i_A. If M is a rejection message the protocol ends. If the request has been accepted then A sends a signed message containing the key K_A protected using B's public key:

$$A \xrightarrow{m_3} B : A_s(A, B, i_A, B_p(K_A)).$$

When B receives the message, it checks message validity and then extracts the key K_A by using its own secret key. Then, B decrypts $K_A(p)$, retrieves the code to be executed, and sends A an acknowledgment message:

$$B \xrightarrow{m_4} A : B_s(B, A, i_A, H(m_3)).$$

Then, B executes code p and, during execution, it produces the associated trace T_B^p. The trace contains the identifiers of the executed statements and the signatures associated with black statements.

When the program terminates, B sends A a signed message containing the program's final state S_B encrypted using a random key K_B, chosen by B, a checksum of the execution trace T_B^p (whose extended form is stored —for a limited amount of time— by B), and a timestamp t_B:

$$B \xrightarrow{m_5} A : B_s(B, A, i_A, K_B(S_B), H(T_B^p), t_B).$$

When A receives the message, it replies with a signed acknowledgment message implicitly requesting the key to access the computation results:

$$A \xrightarrow{m_6} B : A_s(A, B, i_A, H(m_5)).$$

This message is A's commitment to pay for the services used by the mobile code if the execution was correct. B replies to m_6 with a signed message containing the key K_B, protected with A's public key:

$$B \xrightarrow{m_7} A : B_s(B, A, i_A, A_p(K_B)).$$

A verifies the signature on the message and then extracts K_B by using its own secret key. Then, A decrypts $K_B(S_B)$ and accesses the results of the computation.

After having accessed the results, if, for some reason, A suspects that B cheated while executing p, it can ask B to produce the trace. B cannot refuse because of its signed message m_5. After B delivers the complete trace T_B^p, A checks whether the trace is the one actually referenced in message m_5 by computing $H(T_B^p)$ and comparing it with the value contained in B's message. Finally, A validates the execution of p with respect to the trace T_B^p. That is, the code is re-executed step by step, and at each step the identifier of the current statement is compared with the one contained in the trace at the corresponding step. Every time a statement that involves some input from the outside environment must be executed, the input value is extracted from the corresponding signature. Note that the complexity of the validation process is linear with the size of the execution trace.

If, at some point, a discrepancy between the simulated execution and the trace is found or if the final state of the simulator does not match the value S_B provided by B then B cheated by modifying the code, by modifying some program variables, or by tampering with the code execution flow. If no difference between the simulated execution and the trace is found but B charged A for actions that the code did not perform during the simulation, then B cheated by overcharging A for services or resources that have not been used.

In both cases, A can prove B's misbehavior to the trusted third party. In fact, using message m_2, A can prove that B received code p and committed to its execution. Then, using message m_5, A can prove that B claimed to have executed p following trace T_B^p to obtain the final state S_B. In fact, B signed $H(T_B^p)$ and $K_B(S_B)$ and cannot provide a different trace or change the computation results. The proposed mechanism also provides a means to protect a well-behaving site against a cheating user. In fact, B can prove to the trusted third party that A requested the execution of code p by providing m_1 and that A accepted to pay for the resources consumed by showing message m_6.

The protocol described so far assumes that the participants are playing fair by following the protocol as expected. If A and/or B does not play fair, the trusted party must be involved to force the misbehaving participant to a correct behavior. In the following we analyze what these misbehaviors can be and how they are solved.

After A sends B message m_1, B should send message m_2. At this step there are two possible misbehaviors: (i) A could *claim* to have sent m_1 without actually having sent it; (ii) B omits producing message m_2 claiming that it has never received message m_1. In both cases, A —the party that is interested in having the code executed— contacts the trusted party TTP providing message m_1 and requesting its delivery to B. Note that TTP has no means of determining who is not playing fair and therefore cannot apply any sanction. Its role is simply to guarantee (and certify) that message m_1 was delivered to B.

After this step B must send a response message. Again, there are two possible misbehaviors: (i) B could *claim* to have sent m_2 without actually having sent it; (ii) A could omit sending message m_3 claiming that m_2 was never sent by B. In both cases, B —the party that is interested in charging A for the execution of p— contacts the trusted party and asks it to deliver message m_2 to A. Now A is committed to send message m_3. If A fails to send message m_3, B can contact the trusted party and ask it to force A to behave correctly. If A sent message m_3 but B claims it has never received it and for this reason it did not produce message m_4, A can ask the trusted party to deliver message m_3 and force B to provide a receipt.

After B executed the code, it should send message m_5 with the encrypted results. B can play unfair by pretending to have sent the message, while A can misbehave by pretending not to have received it. In both cases, B is in charge of contacting the trusted party and force the delivery, because B needs message m_6 to be able to charge A. If B omits sending message m_7 claiming it has never received m_6, A —the party that is interested in obtaining the results— can contact the trusted party, produce m_5 and m_6 as evidence, and force B to deliver message m_7.

The protocol described so far addresses the problem of detecting tampering in case of remote execution of code. Code execution involves two principals: the owner of the code and the remote site responsible for code execution. In the next section we will extend the protocol to take into account a computation that involves several sites and the transfer of intermediate execution states.

5 Mobile Agent

In the following we consider a scenario in which a mobile agent starts from an initial home site and then jumps from site to site to perform a particular task. Eventually the agent will terminate and the results will be delivered to its owner at its home site.

Let us suppose that the agent starts at site A (its "home" site) and, at some point, it requests to migrate to site B. As a consequence, the code p of the agent and its current execution state S_A are transferred to site B. That is, A sends B the following signed message:

$$A \xrightarrow{m_1} B : A_s(A, B, K_A(p, S_A), A_s(A, i_A, t_A, H(p), TTP)).$$

The first two fields state that the message is sent from A to B. The following field contains the agent code (p) and its initial state (S_A) encrypted with a random key K_A chosen by A. The fourth field is the *agent token* that contains some static data about the agent that will be used during further hops in the agent trip. The agent token ($agent_A$ for short) contains A's identity, the agent's identifier i_A, a timestamp t_A indicating the time of agent dispatching, the hash value of the agent's code $H(p)$, and the identity of the trusted party (TTP) that will be involved in possible dispute resolution, similarly to the procedure described in the previous section. The token is signed by A.

When B receives m_1, it uses A's public key to check the signature on both the message and the agent token. As in the case of remote code execution, at this point B can refuse or accept agent execution on the basis of the information contained in the message. In both cases, B sends A the following signed message:

$$B \xrightarrow{m_2} A : B_s(B, A, i_A, H(m_1), M).$$

A validates the message and examines M. If M represents a rejection then the protocol ends. Otherwise, M is B's commitment to execute the agent and is an implicit request for the key K_A. In this case, A sends the key to B, protected with B's public key:

$$A \xrightarrow{m_3} B : A_s(A, B, i_A, B_p(K_A)).$$

B checks the message validity, extracts the key using its own secret key, and decrypts the agent's code and state. Then, B sends A a signed acknowledgment message:

$$B \xrightarrow{m_4} A : B_s(B, A, i_A, H(m_3)),$$

and begins agent execution.

B executes the agent until the agent requests to migrate to another site, say C. As a consequence, B stops the execution of the agent and sends C two consecutive signed messages:

$$B \xrightarrow{m_5} C : B_s(B, C, agent_A, H(T_B^p), H(S_B), t_B),$$

$$B \xrightarrow{m_5'} C : B_s(K_B(p, S_B), H(m_5)).$$

The first message contains the names of both the sender and the receiver, the agent token, a hash value of the trace T_B^p produced by agent execution on B, a hash value of the current state S_B, and a timestamp t_B. The next message contains the agent's code and the current state S_B, encrypted with a random key K_B chosen by B, followed by a hash of the previous message[5]. After having received both messages, C checks the corresponding signatures. In addition, C computes $H(m_5)$ and compares the result with the hash value contained in m_5'. Then C checks the signature on $agent_A$ using A's public key and verifies that the agent was sent originally by A at time t_A. C accepts or rejects the migration request with a signed message that replies to m_5 and m_5':

$$C \xrightarrow{m_6} B : C_s(C, B, i_A, H(m_5, m_5'), M).$$

If M represents a rejection, then B restarts the agent by returning an error message as the result of the statement that requested the migration. If M is an

[5] The two messages m_5 and m_5' are kept distinct because the verification procedure uses the data contained in m_5 only. This way it is possible to avoid retransmission of the whole code and state of the agent during trace validation.

acceptance message, then C commits to execute the agent and implicitly requests the decryption key K_B. Therefore, B replies with a signed message containing the requested key protected using C's public key:

$$B \xrightarrow{m_7} C : B_s(B, C, i_A, C_p(K_B)).$$

C receives the message and uses K_B to access the agent's state and code. Then it checks that the code has not been modified by B computing $H(p)$ and comparing the resulting value with the hash contained in the agent token. In addition, C checks if the hash of S_B matches the value contained in m_5. If both the code and the state of the agent have been sent correctly, C sends a signed acknowledgment that terminates the transfer:

$$C \xrightarrow{m_8} B : C_s(C, B, i_A, H(m_7)).$$

This protocol is repeated for every subsequent hop until the agent terminates. Upon termination, the final site, say Z, retrieves from the agent token the name of the "home site" of the agent (i.e., A) and contacts the home site to deliver the final state of the agent. Therefore Z sends A the following signed message:

$$Z \xrightarrow{m_n} A : Z_s(Z, A, agent_A, H(T_Z^p), K_Z(S_Z), t_Z).$$

A checks the message validity and requests K_Z with a signed acknowledgment message:

$$A \xrightarrow{m_{n+1}} Z : A_s(A, Z, i_A, H(m_n)).$$

Z provides the key K_Z —protected using A's public key— in a signed message that terminates the protocol:

$$Z \xrightarrow{m_{n+2}} A : Z_s(Z, A, i_A, A_p(K_Z), H(m_{n+1}))$$

After having examined the results of the computation or after having received the charges for the resources used by the agent during its execution, if A thinks that one or more of the sites involved cheated, it starts the verification procedure. Thus, A asks B (the first site in the agent route) to provide its trace T_B^p. B cannot deny receipt of the agent and having committed to its execution because of message m_2 and must provide the trace. A starts the simulation following the provided trace. The simulation eventually reaches the instruction that requests the migration to site C. The agent's simulated state, at that point, is S'_B. Then, A asks C to provide its trace and a copy of message m_5 (that is signed by B). C cannot deny acceptance of the migration request, because B has messages m_7 that states the commitment of C to agent execution. A extracts from message m_5 the hash values of S_B and T_B^p. Therefore, A can compare $H(S_B)$ to $H(S'_B)$. In addition A can verify the integrity of the trace previously provided by B. If both checks succeed, B did not cheat. As a consequence, A restarts the simulation of the agent's execution following the trace provided by C.

This process continues until it reaches agent termination. At the end of the simulation if state S'_Z and S_Z coincide and the trace provided by Z produces the same hash value $H(T_Z^p)$ contained in message m_n, then the agent has been executed correctly. Alternatively, if some discrepancy is found during the verification of the trace provided by site X, then X cheated.

6 Discussion

Cryptographic tracing is a mechanism for detection of illegal tampering with agent execution. By relying on this mechanism, the agent owner can verify with a high degree of certainty that his/her agent was executed conforming to its specification, that is, its code. Therefore, the agent's owner is protected against service overcharging. In addition, if the agent behaves incorrectly because its code and/or state has been modified in a malicious way against its original specification, the principal responsible for its execution can relinquish responsibility and determine who "brainwashed" the agent. Obviously, if there are ways to induce the agent to attack other sites by providing carefully crafted inputs, the agent's owner may be held responsible for the damage caused by the agent. In this respect, the developer of the agent's code must use the same caution required when developing privileged programs that could receive parameters by untrusted principals (e.g., SUID programs in the UNIX operating system [11] or CGI scripts that process inputs received from browsers [27]). Greater care must be used if the language being traced allows for dynamic evaluation of code, that is, if there exists some kind of *evaluate* statement that takes as a parameter a string and interprets it as code. This is a feature of most scripting language (e.g., Perl or Tcl) and represents both a danger and a blessing. In fact, while dynamic evaluation may allow a malicious site to drive an agent to execute arbitrary pieces of code, it makes it possible to remove the static constraint on code. If code can be managed as data, the code fragments determined by interacting with the computational environment during execution will be recorded in the execution trace and the corresponding statements will be traceable and verifiable as well.

Being able to detect mobile agent tampering is an asset. Yet there are some issues that limit the applicability of the tracing mechanism. Some limitations stem from the scope of the mechanism. First of all, cryptographic tracing is a mechanism that allows detection of tampering *after* agent execution. Therefore if timely detection of tampering is needed a different mechanism must be devised. In addition, the mechanism does not provide any means to determine *a priori* if tampering occurred. The decision to perform trace validation must rely on the evaluation of the outcomes of agent execution and on the charges received for agent operations[6]. The proposed protocol could be extended to include in the messages carrying the agent a list of signed hash values of the traces produced during execution at previously visited sites; this way, a site could refuse to execute the agent if it has visited certain sites, or perform trace validation

[6] This is somewhat similar to reconciling credit card statements.

of the agent's execution from its start to the current state before resuming the agent. Another issue is that the proposed mechanism does not offer any protection against disclosure. Differently from mechanisms based on code scrambling, partial encryption, or cryptographed functions, the code and the state of an agent are accessible by the site responsible for agent execution. Therefore if the mobile agent's code and state must be kept secret, then the tracing mechanism must be extended with the aforementioned mechanisms.

Some limits come from the assumptions made in Section 3. In particular, the mechanism requires that all the sites participate in some kind of infrastructure that allows for key distribution and management, interpreter certification, service billing, and sanctioning of the principals. In fact, since cryptographic tracing is a *detection* mechanism, it is useless unless there is a way to sanction cheating sites or the principals responsible for misbehaving agents. Therefore the involved parties must be liable in case of misbehavior or must be subject to some kind of social control. This constraint poses some limitation to the scalability of the proposed mechanism.

An obvious limit comes from the quantity of resources that are needed to enforce the mechanism. The protocols described in sections 4 and 5 make extensive use of public-key cryptography. These cryptographic algorithms are considerably slow when compared to secret-key cryptography. Yet, they are necessary to provide authentication and non-repudiation between untrusted parties. Another issue is posed by the size of the traces collected during execution. In fact, the size of the traces may be large, even if compressed. Several mechanisms can be used in order to reduce the size of the traces. For example, instead of a complete execution trace it could be possible to log just the signature of black statements and the points in execution where control flow is non-deterministic. Another extension to the mechanism could be devised that allows the programmer to define a *range* of statements to be traced. The programmer may require that the values of some critical variables must satisfy a set of constraints before entering that particular group of statements. In a similar way, a mechanism could allow a programmer to specify that a group of statements must *not* be traced and that just the final values of the modified variables must be included in the trace. This mechanism could be useful in search procedures when, during a loop, several values are retrieved from the external environment and only a subset is saved in the agent's state.

Another set of issues stem from the nature of the language adopted for agents. If the language is too low-level, the traces produced during execution would be really large. If the language allows management of very complicated data structures, the modifications to the agent's internal state could be difficult to represent and could require a lot of space. In addition, we have made the assumption that agents cannot share memory and are single threaded. If this is not the case, an extension to the tracing mechanism is required. In order to check the execution of an agent, a user would need the trace of all the agents or threads that shared some memory portion with the agent thread under examination. In addition, traces should be extended with some timing information that allows for deter-

mining the order of the statements executed by the different threads. As one can easily understand, this mechanism would be practically infeasible.

7 An Electronic Commerce Agent

In order to give an example of the operations involved in agent execution, tracing, and verification, we introduce a simple mobile code language, called SALTA (*Secure Agent Language with Tracing of Actions*)[7], that implements the tracing mechanism. The language is then used to develop an electronic commerce application.

SALTA is a modified version of the Safe-Tcl language [2,21]. The Safe-Tcl language has been restricted further, and two new instructions have been added, namely: `request` and `go`.

The `request` command allows an agent to access the services provided by a site. The `request` command takes as arguments the service name and a list of strings representing the service parameters. The `go` command allows an agent to migrate to a remote site specified as the command argument. When an agent executes the `go` command, it is suspended and its code and execution state (values of variables, call stack, and program counter) are packed into a message. The message is delivered to the destination site following the protocol described in Section 5. Upon arrival, the agent is unpacked, its execution state is restored, and its execution is restarted from the command following the `go`. From this point of view this language is similar to Agent-Tcl [13].

We use this minimal language to develop a simple agent application. A user, at site `home.sweet-home.com` wants to buy a home video of Tarantino's *Pulp Fiction* movie. Therefore, he dispatches an agent to a site called `agents.virtualmall.com` dedicated to maintain a directory of electronic shops. Once there, the agent performs a directory query for sites offering home videos. Then, the agent visits the provided sites. At each site the agent contacts the local catalog service to determine the current price of the *Pulp Fiction* home video. When all prices have been collected, the agent identifies the best offer and, if the best price is less then a specified amount—say, twenty dollars—the agent goes to the selected site and buys the home video. The transaction identifier is stored in the variable `result`. Upon completion of its task, the agent terminates and its state is returned to its home site. Figure 1 shows the agent's code.

Now suppose that the directory service at `agents.virtualmall.com` has suggested two sites, namely `agents.brockbuster.com` and `agents.towelrecords.com`. The prices of the *Pulp Fiction* home video are fifteen dollars at `agents.brockbuster.com` and seventeen dollars at `agents.towelrecords.com`. The execution trace will be the one shown in Figure 2.

Let us suppose that the Towel Records site wants to modify Brockbuster's offer so that Towel Record's offer appears to be the most convenient. Before the agent computes the best price, the Towel Records site modifies the home video

[7] "Salta" is also an Italian verb meaning "jump".

```
1    : go agents.virtualmall.com
2    : set shoplist [request directory query homevideo]
3    : foreach shop $shoplist {
3.1  :     go $shop
3.2  :     set price($shop) [request catalog movies "Pulp Fiction"]
     : }
4    : set best_price 20
5    : set best_shop none
6    : foreach p [array names price] {
6.1  :     if {$price($p) < $best_price} {
6.1.1:         set best_price $price($p)
6.1.2:         set best_shop $p
     :     }
     : }
7    : if {$best_price > 20} {
7.1  :     set result "No offers below \$20!"
7.2  :     exit
     : }
8    : go $best_shop
9    : set result [request buymovie "Pulp Fiction" $best_price]
10   : exit
```

Fig. 1. The Pulp Fiction Agent.

home	virtualmall	brock	towel
1	2,shoplist = brock towel	3.2,price(brock) = 15	3.2,price(towel) = 17
	3	3	4
	3.1	3.1	5
		9,result = PF11	6
		10	6.1
			6.1.1
			6.1.2
			6
			6.1
			6.1.1
			6.1.2
			7
			8

We made the following substitutions:

```
home        = home.sweet-home.com
virtualmall = agents.virtualmall.com
brock       = agents.brockbuster.com
towel       = agents.towelrecords.com
```

Fig. 2. Pulp Fiction Agent execution traces.

price associated with Brockbuster, raising its value to twenty-two dollars. As a consequence, the agent buys the home video at Towel Records. The trace, in this case will be the one shown in Figure 3.

home	virtualmall	brock	towel
1	2,shoplist = brock towel	3.2,price(brock) = 15	3.2,price(towel) = 17
	3	3	*price(brock) = 22*
	3.1	3.1	4
			5
			6
			6.1
			6
			6.1
			6.1.1
			6.1.2
			7
			8
			9,result = PF11
			10

The illegal tampering with the agent's state is showed in italics.

Fig. 3. Execution traces resulting from tampering.

After having received the agent's final state, the agent owner may decide to verify agent execution. In this case, he/she retrieves the trace produced by the agent at the first site, namely `agents.virtualmall.com`. Then the owner of the agent simulates the agent's execution following the provided trace, until it comes to the request to migrate to another site, i.e., `agents.brockbuster.com`. As a consequence, the agent's owner asks Brockbuster's site to provide the agent's trace and the signed checksum of the agent's state received from `agents.virtualmall.com`. If the checksum computed over the simulator's state does not match the value provided by the Brockbuster site, then Virtuall Mall's site cheated. Otherwise simulation proceeds. When the verification process reaches instruction 8 (see figures 1 and 3) an inconsistency is flagged. In fact, the simulator holds value `agents.brockbuster.com` for variable `best_shop` and therefore the `go` command should have migrated the agent to the corresponding site. Since `agents.brockbuster.com` sent a signed checksum of the agent state just before the agent left, and `agents.towelrecords.com` has signed a receipt for that message, A can determine that Towel Records' site cheated. In addition, using the messages produced during the protocol, A can prove to a third party that `agents.towelrecords.com` tampered with state in an illegal way. The same procedure can be used to flag out tampering with code, inconsistencies in state transmission, computational flow diversion, and service overcharging.

8 Conclusions and Future Work

The interest in code mobility has been raised by the availability of a new breed of technologies featuring the ability to move portions of application code and possibly the corresponding state among the nodes of a wide-area network. Being able to move computations from host to host, mobile code systems raise serious security issues. One of the most difficult problems to solve is the protection of roaming agents from computational environments. We presented a mechanism, based on execution tracing, that allows an agent owner to detect illegal tampering with agent data, code, and execution flow, under certain assumptions. The proposed system does not require dedicated tamper-proof hardware or trust between parties. A language that implements the system concepts has been described, together with a simple electronic commerce example.

Presently, the Software Engineering group at Politecnico di Milano is designing and implementing a first prototype of the SALTA language. The prototype is written in Java, while cryptographic functionalities are implemented using PGP [33]. Aside from the tracing mechanism described in this paper, the SALTA language includes mechanisms to protect sites against malicious agents. We plan to use SALTA to implement electronic commerce applications that require accountability and a high degree of protection against frauds.

References

1. J.K. Boggs. IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility. IBM Technical Disclosure Bulletin 752, IBM, August 1973.
2. N. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, First Virtual Holdings, Inc, 1994.
3. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
4. A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proc. of the 19^{th} Int. Conf. on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.
5. D.M. Chess, B. Grosof, C.G. Harrison, D. Levine, C. Paris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communication*, October 1995. Also available as IBM Technical Report.
6. G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 93–111. Springer, April 1997.
7. W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Authentication and State Appraisal. In Springer, editor, *Proc. of the 4^{th} European Symp. on Research in Computer Security*, volume 1146 of *LNCS*, pages 118–130, Rome, Italy, September 1996.
8. W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. of the 19^{th} National Information Systems Security Conf.*, pages 591–597, Baltimore, MD, USA, October 1996.
9. A. Freier, P. Karlton, and P. Kocher. The SSL Protocol — Version 3.0. Internet Draft, March 1996.

10. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 1998. to appear.
11. S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security*, chapter Tips on Writing SUID/SGID Programs, pages 716–718. O'Reilly, 1996.
12. R.S. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., December 1995.
13. R.S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proc. of the 4^{th} Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.
14. F. Hohl. An approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems, 1997.
15. ITU-T. Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. ITU-T Recommendation X.509, November 1993.
16. D.B. Lange and D.T. Chang. IBM Aglets Workbench—Programming Mobile Agents in Java. IBM Corp. White Paper, September 1996.
17. S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. In *Proc. of the 4^{th} Int. World Wide Web Conf.*, Boston, Massachusetts, USA, December 1995.
18. L.Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly, 2^{nd} edition, 1996.
19. Sun Microsystems. The Java Language: An Overview. Technical report, Sun Microsystems, 1994.
20. J. Ordille. When agents roam, who can you trust? In *Proc. of the First Conf. on Emerging Technologies and Applications in Communications*, May 1996.
21. J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. Technical report, Sun Microsystems, November 1996. Printed in this volume.
22. R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Comm. of the ACM*, 21(2):120–126, February 1978.
23. T. Sander and C. Tschudin. Towards Mobile Cryptography. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.
24. B. Schneier. *Applied Cryptography – Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., 2^{nd} edition, 1996.
25. K.B. Sriram. Hashjava - A Java Applet Obfuscator. `http://sbketch.org/hash-java.html`, July 1997.
26. J.W. Stamos and D.K. Gifford. Implementing Remote Evaluation. *IEEE Trans. on Software Engineering*, 16(7):710–722, July 1990.
27. L.D. Stein. The World Wide Web Security FAQ. `http://www.w3.org/Security/-Faq/`, January 1998.
28. J. Tardo and L. Valente. Mobile agent security and Telescript. In *Proc. of IEEE COMPCON'96*, February 1996.
29. C. Tschudin. *An Introduction to the* `MO` *Messenger Language.* Univ. of Geneva, Switzerland, 1994.
30. J.E. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents.* AAAI Press/MIT Press, 1996.
31. U.G. Wilhelm. Cryptographically Protected Objects. Technical report, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1997.
32. F. Yellin. Low Level Security in Java. Technical report, Sun Microsystems, 1995.
33. P. Zimmerman. *PGP User's Guide*, March 1993.