# Protecting Mobile Agents through Tracing

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 provide a flexible and powerful platform to build distributed applications in an Internet scale, but they rise strong requirements from the security point of view. Security issues include authentication of the different remote parties involved and protection of the execution environments from malicious agents. Nonetheless, the most difficult task is to protect roaming agents from execution environments. This paper presents a new mechanism based on execution tracing and cryptography that allows an agent owner to determine if some site in the route followed by the agent tried to tamper with the agent state or code.
**Keywords:** mobile code, mobile agents, security, auditing.

## 1 Introduction

Mobile code systems are receiving interest by both the industry and the academia. The benefits provided by a more flexible approach with respect to the well-known client-server paradigm and the corresponding message-based technologies, are understood and a consistent research effort is invested in developing technologies and methodologies.

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*. Executing units are threads of execution that own a *state* and some *code* that defines their behavior. Mobile code technologies can be roughly divided into two sets. *Weakly mobile* technologies allow an executing unit to send some code to a remote site in order to have it executed there or to dynamically link code retrieved from a remote site. Examples of weakly mobile technologies are Java [9] and Mole [21]. *Strongly mobile* technologies allow an executing unit that is running on a particular site to move to a different computational environment. In this case, the executing unit is stopped, its code and execution state are marshaled into a message that is sent to the remote site, and its execution is resumed there, starting from the statement that follows the invocation of the migration primitive. Examples of strongly mobile languages are Telescript [24] and Agent-Tcl [10]. A survey of several mobile code languages and systems can be found in [6].

While the mobile code approach to designing and implementing distributed application has proved valuable [4], it raises some serious security issues: protection of hosts and execution environments from misbehaviors or attacks coming

from agents, protection of executing units from each other, protection of agents information while traveling through untrusted networks, protection of agents from malicious computational environments.

Protection of agents information while traveling over an untrusted network can be achieved using well-known cryptographic protocols. Mechanisms and policies to protect a site from code coming from an untrusted source have been the focus of recent research [22, 17, 25, 14]. Using suitable access control and sandboxing mechanisms, it is possible to protect effectively the execution environment from attacks.

The hardest security problem is represented by the protection of agents from malicious actions of the computational environments they are running in.

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

The paper is organized as follows. In Section 2 we present some related work on mobile code and security. In Section 3 we describe the proposed security mechanism and its applications. In Section 4 we describe a mobile code language that implements the tracing mechanism and in Section 5 we use such language in an example. In Section 6 we examine some limitations of our approach. Section 7 draws some conclusions and illustrates future work.

## 2 Agents and security

The problem of protecting an agent from the hosting computational environment has been addressed in different ways.

Most mobile code systems consider the site as a trusted entity and therefore they do not provide any mechanism to protect agent execution. Others [5, 8] have tried to identify which goals are impossible or very difficult to achieve. Tasks being considered impossible are: prevention of tampering with agents state (unless some kind of hardware trusted platform is adopted), guarantee that an environment will execute an agent correctly and to its completion, guarantee that the agent is moved to the requested sites, and total protection of agent data from disclosure[1].

In [7] a protection mechanism against sites trying to tamper with agents state is presented. The mechanism is based on state appraisal functions. These functions express invariants that the agent state must satisfy. This way, some malicious attempt to tamper with the agent state can be detected.

Our approach is stronger: we want to be able to detect any possible unauthorized modification of agent code and state. Obviously, we cannot foresee the information that an agent will receive from the site it is running on or from other agents. From this point of view, the agent developer must use the same caution

[1] Partially encrypted agents are a possible partial solution to the problem.

required when developing privileged programs that could receive parameters by untrusted principals (e.g., CGI scripts).

## 3 Cryptographic traces

As stated in [5], it is impossible to *prevent* malicious or faulty sites from tampering with agents. Sites must have access to an agent code and state in order to support its execution. Still, there are means to detect abnormal behaviors or unauthorized modifications of agents components.

We propose a mechanism for tracing the execution of a migrating agent in an unforgeable way. Such traces can be used as a basis for program execution verification, i.e., for checking the agent program against a supposed history of its execution. This way, cryptographic tracing allows the agent owner to prove, in case of tampering, that the operations the agent is accounted for could have never been performed.

The proposed protocol 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 [18]. The public key of a principal $A$ is denoted by $A_p$, while $A_s$ is used for the corresponding secret key. The principals are users of a *public key infrastructure* [12] that guarantees the association of a principal with the corresponding public key. We assume that, at any moment, any principal can retrieve the public key of any other principal and verify its integrity. For the sake of simplicity, both the public key of a user and the associated certificate will be denoted by $A_p$. The process of encrypting a message $m$ with a key is expressed by $A_x(m)$. 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 function $H$ to the message $m$ is denoted by $H(m)$. Several examples of public key cryptosystems[2] and one-way hash functions can be found in [19].

A moving agent is composed by some code $C$ and some state $S^i$ that has been determined, at some specified point $i$, by code execution. The state includes global data structures, the call stack and the program counter. Note that in some cases state can be in a degenerate form, for example if the agent has not yet started its execution.

We will assume that the code is *static* with respect to the lifetime of the agent and that it is composed of a sequence of statements. A *white* statement is an operation that modifies the content of the program state only on the basis of variables internal to the program. For example, the statement $x := y + z$, where $x$, $y$, and $z$ are variables of the program, is a white statement. A *black* 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. I.e., if the application of a symmetric encryption process, parameterized by a key $K$ to a message $m$ is represented by $K(m)$, then $A_x(m)$ is equivalent to $A_x(K), K(m)$.

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.

We assume that all the interpreter implementations are certified to be correct and to respect the semantics of the language[3].

A trace $T_C$ of the execution of program $C$ 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 to black statements, the signature contains the new values assumed by internal variables as a consequence of statement execution. For example, if the $read(x)$ instruction gets the integer 3 from the terminal, the associated signature will be $x := 3$. The signature is empty for white statements.

In the following sections we will address problems of increasing complexity. In particular, we will follow the distinction made by [15], firstly tackling the problem of protecting *remote code execution*, and then analyzing *boomerang agents* and *multi-hop* agents.

### 3.1 Remote code execution

Remote code execution, also known as *remote evaluation*, is a mechanism that allows a program to have some code, together with some initialization data, 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 that has always been available to UNIX users by means of the `rsh` facility. A formal definition of the remote evaluation mechanism is presented in [20]. Remote code execution represents the basis for several mobile code systems like TACOMA [13], Obliq [3], and M0 [23].

Suppose now that user $A$ wants to execute code $C$ on site $B$. Then, $A$ sends $B$ the following message :

$$A \xrightarrow{m_0} B : A, B_p(C), A_s(H(C), B, A', t_A, i_A).$$

Since the code is protected using the public key of $B$, it is accessible by $B$ only. When $B$ receives the messages, it decrypts the second part of the message using its secret key $B_s$ to obtain the code. Then it uses the principal name specified in the first part of the message ($A$) to retrieve the corresponding public key $A_p$. The key is used in order to decrypt the third part of the message. $B$ computes $H(C)$ and compares the result with the hash value contained in the third part of the message. If the two values match, $B$ is assured that the message was sent by $A$, and that the code is unmodified. Since the third part of the message contains $B$, $B$ can be sure that the message was intended for himself. The principal identified by $A'$ is the recipient of every receipt token that will be produced by the protocol. $A'$ and $A$ may coincide, or may denote a different recipient. This option can be used in case $A$ has disconnected after sending

[3] For example, the interpreter owner must provide some kind of third-party certification of correct implementation of the language in order to be able to charge agents for the services they use.

$m_0$ or if it requires that some different trusted third party collects the receipts (e.g., a digital notary). $t_A$ represents a timestamp to guarantee freshness[4] and $i_A$ represents a unique identifier to protect from replay attacks.

After authentication has been performed, $B$ sends $A'$ a receipt of the received message, containing the identity of the sender and the third part of message $m_0$, signed with his own secret key:

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), B, A', t_A, i_A)).$$

This way, $A'$ can verify that $B$ received a code execution request from $A$ at time $t_A$ and that $A'$ was actually indicated as the recipient of receipt messages. At any moment, $A$ can prove that $B$ received $C$ by showing to $A'$ that the computation of the hash $H(C)$ produces a result corresponding to the value contained in $m_1$.

Once the initial handshake has been performed, $B$ executes $C$, and produces a corresponding trace $T_C$. When the execution terminates, $B$ sends a signed message to $A'$ containing a checksum of the program final state $S^1$, a checksum of the execution trace $T_C$ (whose extended form is stored (for a limited amount of time) by $B$), and the unique identifier $i_A$:

$$B \xrightarrow{m_2} A' : B, B_s(H(S^1), H(T_C), i_A).$$

Then, $B$ sends a signed message to $A$ containing the program final state $S^1$ (i.e., the results of the computation[5]) protected using $A$'s public key together with the unique identifier $i_A$:

$$B \xrightarrow{m_3} A : B, B_s(A_p(S^1), i_A).$$

Since both the messages are signed, $B$ cannot send a modified state to $A$ or a modified checksum to $B$ without being detected.

If $A$ suspects that $B$ cheated while executing $C$, he can ask $B$ to produce the trace and $A'$ to produce the receipt messages. Then, $A$ checks whether the trace is the supposed one by using the value $H(T_C)$ contained in $m_2$. Finally, $A$ replicates the execution of the program $C$ following the trace $T_C$. The validation process *should* produce the final state $S^1$. If it is not so, $B$ cheated, by modifying the code or by modifying some program variables, and $B$'s misbehavior can be proved to third parties. Note that the complexity of the validation process is linear with the size of the execution trace.

### 3.2 Boomerang agents

A *boomerang agent* is an agent that at a certain point migrates to a remote site $B$ in order to perform some task, and when the task is accomplished gets back to its home site.

[4] The timestamp may be associated to a time interval in order to limit agent execution.

[5] The state that is returned is the *final* state containing just the data that represent the outcomes of the computation.

In this case, there is some state $S^0$ associated to the code $C$. Therefore, the first message will be :

$$A \xrightarrow{m_0} B : A, B_p(C, S^0), A_s(H(C), H(S^0), B, A', t_A, i_A).$$

From the contents of the message $B$ can be sure that it was the intended recipient and that both the code and state were actually sent by $A$ and are untampered. Therefore, $B$ sends $A'$ the following receipt :

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), H(S^0), B, A', t_A, i_A)),$$

so that $A'$ can verify that $B$ received an agent to be executed from $A$ at time $t_A$, and that $A'$ was actually indicated by $A$ as the recipient of receipt messages. At any moment, $A$ can prove that the code and the state received by $B$ were those specified in $m_1$ by a process analogous to the one performed in the previous case.

Then, $B$ creates a agent image initialized with the code and the state provided by $A$ and it resumes its execution. The execution goes on, generating the corresponding trace, until the agent decides to migrate back home. Then, $B$ sends $A'$ a signed message containing an hash of the agent state just after the migration statement ($S^1$), a hash of the execution trace, and the unique identifier $i_A$:

$$B \xrightarrow{m_2} A' : B, B_s(H(S^1), H(T_C), i_A),$$

and, in addition, a signed message to $A$ containing the program final state $S^1$ protected using $A$'s public key together with the unique identifier $i_A$.

$$B \xrightarrow{m_3} A : B, B_s(A_p(S^1), i_A),$$

Since $A$ owns the agent's code $C$ and the current state, he can restore the computation of the agent.

Like in the previous case, if $A$ suspects that $B$ cheated, he can ask $B$ to produce the trace. The validation process should produce the final state $S^1$ starting from state $S^0$.

### 3.3 Multi-hop agent

A *multi-hop agent* is an agent that, in order to perform a particular task starts from a initial home site and then jumps from site to site, to get back home when the task has been accomplished. There are several problems related to the authentication of an agent that has visited a possibly large list of sites with different levels of trust. Here, we will tackle the problem of assuring that none of the visited sites can tamper with the roaming agent execution without being detected.

The first hop of the agent produces two messages similar to the ones produced in the previous case:

$$A \xrightarrow{m_0} B : A, B_p(C, S^0), A_s(H(C), H(S^0), B, A', t_A, i_A),$$

and:

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), H(S^0), B, A', t_A, i_A)).$$

Then $B$ executes the agent until it requires to move to another site $D$. As a consequence, $B$ stops the agent execution and sends $A'$ a signed message containing the name of the site that will represent the next hop of the agent route, a hash value of the final state $S^1$, a hash value of the execution trace, and the unique identifier $i_A$:

$$B \xrightarrow{m_2} A' : B, B_s(D, H(S^1), H(T_C^1), i_A).$$

Then, $B$ sends $D$ a message composed as follows. Firstly there are the name of the sender and the name of the principal that is responsible for agent execution. The following part contains the unit code and the current state, encrypted with $D$'s public key in order to protect the code and the state from unauthorized disclosure. Then there is a hash value of the current state and the name of the intended recipient signed with the sender secret key. The last part of the message contains a copy of the third part of $m_0$:

$$B \xrightarrow{m_3} D : B, A, D_p(C, S^1), B_s(H(S^1), D), A_s(H(C), H(S^0), B, A', t_A, i_A).$$

$D$ decrypts the third part using its secret key $D_s$. Afterwards, $D$ retrieves the public keys associated to principals $A$ and $B$, and uses them to decrypt the remaining parts of the message. From the contents of the part signed by $B$, $D$ can argue that it was the intended recipient of the message, that the message was actually sent by $B$ and that the state $S^1$ included in the third part was correctly transmitted. From the part of the message signed by $A$, $D$ can deduce that the code contained in the third part is correct and that the agent was initially dispatched by $A$ . In addition, $D$ can be sure that $A$ meant to use $A'$ as recipient of the receipt messages.

Then, $D$ sends $A'$ a signed message containing a signed receipt of the last two parts of the message:

$$D \xrightarrow{m_4} A' : D, D_s(B_s(H(S^1), D), A_s(H(C), H(S^0), B, A', t_A, i_A)).$$

Now $A'$ can be sure that $D$ received $C$ and $S^1$ correctly, i.e., $B$ did not send $D$ code or state that are different from those used in order to compute the hash values that have been delivered to $A'$ in message $m_2$.

This protocol is repeated until the agent decides to terminate. In this case, the final site, say $Z$, sends $A'$ a signed message containing the hash value of both the final state $S^n$ and the execution trace $T_C^n$, and the unique identifier $i_A$:

$$Z \xrightarrow{m_n} A' : Z, Z_s(H(S^n), H(T_C^n), i_A).$$

Then, $Z$ sends $A$ a signed message containing the program final state $S^n$ (i.e., the results of the computation) protected using $A$'s public key, together with the unique identifier $i_A$:

$$Z \xrightarrow{m_{n+1}} A : Z, Z_s(A_p(S^n), i_A).$$

| | | | | |
|---|---|---|---|---|
| agent | append | array | break | case |
| catch | clock | concat | continue | error |
| expr | for | foreach | format | global |
| go | if | incr | join | lappend |
| lindex | linsert | list | llength | lower |
| lrange | lreplace | lsearch | lsort | pid |
| proc | reply | request | return | scan |
| service | set | split | string | subst |
| switch | unset | uplevel | upvar | while |

**Table 1.** SALTA command set.

After having examined the computation results, if $A$ thinks that one or more of the site involved cheated, he can ask $A'$ to provide the receipts generated by the agent trip. In addition, $A$ asks each site to produce the corresponding traces.

$A$ simulates the agent execution starting from $S^0$ and following, in the right order, the execution traces. At each step $i$, the partial state $S^i$ should produce a hash value equal to the one contained in the corresponding receipt message, otherwise the site at step $i$ cheated.

## 4 The SALTA language

The SALTA language (*Secure Agent Language with Tracing of Actions*[6]) is a modified version of the Safe-Tcl language [2, 17]. The available Tcl [16] command set is showed in Table 1. The Safe-Tcl language has been restricted further, and some new instructions have been added: `request`, `service`, `reply` and `go`.

The `request` command takes as a parameter an agent name, a command name and possible parameters. The request to execute the command is dispatched to the specified agent (if it exists). The server agent will accept the request by using the `service` command which returns the command string that represents the command invocation. Upon completion, the agent returns the service results by using the `reply` command which delivers the results to the agent that made the service request.

Agents are identified by using an URL-like scheme, in which an agent is identified by the protocol modifier `agent`, followed by the name of the host (possibly extended with a port number) running the computational environment in which the agent resides, and the agent name. For example an agent named `luke` at host `agents.starwars.com` is referenced by `agent://agents.starwars.com/luke`.

The `go` command causes the suspension of the program, the packing of its code and execution state (values of variables, call stack, and program counter),

[6] "Salta" is also an italian verb meaning "jump".

```
1    : set home home.sweet-home.com
2    : go agents.virtualmall.com
3    : set shoplist [request directory query homevideo]
4    : foreach shop $shoplist {
4.1  :     go $shop
4.2  :     set price($shop) [request catalog movies "Pulp Fiction"]
     : }
5    : set best_price 21
6    : set best_shop none
7    : foreach p [array names price] {
7.1  :     if {$price($p) < $best_price} {
7.1.1:         set best_price $price($p)
7.1.2:         set best_shop $p
     :     }
     : }
8    : if {$best_price > 20} {
8.1  :     go home.sweet-home.com
8.2  :     request terminal print "No offers below \$20!"
8.3  :     exit
     : }
9    : go $best_shop
10   : set trans_id [request buymovie "Pulp Fiction" $best_price]
11   : go home.sweet-home.com
12   : request terminal print "Transaction id: $trans_id"
13   : exit
     : }
```

**Figure 1.** The Pulp Fiction Agent.

| home | virtualmall | brock | towel |
|---|---|---|---|
| 1 | 3,shoplist = brock towel | 4.2,price(brock) = 15 | 4.2,price(towel) = 17 |
| 2 | 4 | 4 | 4 |
| 12 | 4.1 | 4.1 | 5 |
| 13 | | 10,trans_id = PF11 | 6 |
| | | 11 | 7 |
| | | | 7.1 |
| | | | 7.1.1 |
| | | | 7.1.2 |
| | | | 7 |
| | | | 7.1 |
| | | | 7.1.1 |
| | | | 7.1.2 |
| | | | 7 |
| | | | 8 |
| | | | 9 |

We made the following substitutions:
`home` = `home.sweet-home.com`
`virtualmall` = `agents.virtualmall.com`
`brock` = `agents.brockbuster.com`
`towel` = `agents.towelrecords.com`

**Figure 2.** Agent execution traces.

and the shipping of the packed executing unit to a remote computational environment specified as a parameter of the `go` command. When the packed executing unit reaches its destination, it is unpacked, its state is restored and its execution is restarted from the command following the `go`. From this point of view SALTA is similar to Agent-Tcl [11].

SALTA code is *static*, i.e., it is not possible to evaluate code that is produced dynamically. In particular, it is not possible to invoke a command whose name is determined by the value of a variable.

## 5 A Pulp Fiction Agent

In order to give a precise idea of the operations involved in agents execution tracing and verification, we describe a simple electronic commerce 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 having home video offers. Then, the agent moves to each of the provided sites, where it contacts the local catalog agent in order to determine the current price of the *Pulp Fiction* home video. When all prices have been collected, the agent identifies the best offer and then, if the best price is less then a specified amount (say, twenty

dollars), the agent goes to the selected site and contacts a selling agent to buy the home video. Finally, the agent goes back home and reports the transaction identifier associated to the purchase. Figure 1 shows the agent code.

Now, let us suppose that the directory service at `agents.virtualmall.com` has suggested two sites, namely `agents.brockbuster.com` and `agents.towelrecords.com`, and 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 the Brockbuster offer pretending to be the cheapest offer for the movie. Then, before the agent computes the best price, the Towel Records site modifies the home video price associated to Brockbuster, raising its value to twenty-two dollars. As a consequence, the agent buys the home video at Towel Records and gets back home.

In order to verify agent execution, $A$ retrieves the traces associated to the different sites visited by the agent, and the receipt messages associated to each hop. Then, $A$ repeats the agent execution following the provided traces. When the verification process reaches instruction 7.1 (see figures 1 and 2) an inconsistency is flagged. Since `agents.brockbuster.com` sent a signed checksum of the agent state just before the agent left the site and `agents.towelrecords.com` has provided an identical checksum, it means that the modification must have happened at the Towel Records site. Thus, since there are no instructions that assign a value to the variable containing Brockbuster's price in those listed in the trace of the operations performed at Towel Records, $A$ can prove that Towel Records cheated.

The same procedure can be used to flag out tampering with code, inconsistencies in state transmission, computational flow diversion, and so on.

## 6 Limits of the approach

The proposed approach has some limits.

First of all, the size of the traces may be huge, 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 points in execution where control flow changes (e.g., conditional statements and loops) and the signature of black statements[7]. Another 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 a small set of them is saved in the agent state.

[7] This technique has been suggested by a reviewer.

Second, we 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 determining the order of the statements executed by the different threads. As one can easily understand, this mechanism would be practically unfeasible.

Third, the mechanism makes the assumption that the code is static. This forbids the use of optimization techniques like just-in-time compilation. In order to overcome this limit a just-in-time compiler should produce additional information (similar to the information used for debugging purposes) in the compiled module. This information should allow to map the execution of the compiled module on the source code of the agent.

## 7 Conclusions and Future Work

Mobile code systems rise serious security issues. One of the most difficult problem to solve is the protection of roaming agents from computational environments. We presented a new mechanism, based on execution tracing, that allows an agent owner to detect any possible attempt to tamper 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.

Future work will refine the mechanism to include some transaction protocol that may be used to commit actions performed by agents. The transaction protocol could allow a site that represents an intermediate step in the agent route to verify code execution since agent start-up. We are working also on an extension of the protocol that guarantees privacy of the identities of the principals involved. In addition, the SALTA language is under development and much of our work will be dedicated to design a mobile code system that protects *both* agents and computational environments.

## 8 Acknowledgments

We would like to thank the anonymous reviewers that provided comments and suggestions that did not just flag out erroneous and unclear points, but proposed useful ideas and improvements.

## 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.S. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, First Virtual Holdings, Inc, 1994.
3. Luca 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 A. Fuggetta and A. Wolf, editors, *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, 1997. To appear.
5. David Chess, , Benjamin Grosof, Colin Harrison, David Levine, Colin Paris, and Gene Tsudik. Itinerant Agents for Mobile Computing. Technical report, IBM Research Division - T.J. Watson Research Center, 1995.
6. G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*. Lecture Notes on Computer Science, April 1997.
7. William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Authentication and State Appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, September 1996. Lecture Notes in Computer Science No. 1146.
8. William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, Md., October 1996.
9. James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, October 1995.
10. Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., December 1995.
11. Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.
12. ITU-T. Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. ITU-T Recommendation X.509, November 1993.
13. Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23, Department of Computer Science, University of Tromsø and Cornell University, Tromsø, Norway, June 1995.
14. S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, USA, December 1995.
15. J. Ordille. When agents roam, who can you trust? Technical report, Bell Labs, Computing Science Research Center, 1996.
16. J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1995.
17. J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. Technical report, Sun Microsystems, November 1996.
18. R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120 – 126, February 1978.
19. Bruce Schneier. *Applied Cryptography – Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1994.
20. J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

21. M. Strasser, J. Baumann, and F. Hohl. Mole—A Java Based Mobile Agent System. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, pages 327–334, Linz, Austria, July 1997.
22. Joseph Tardo and Luis Valenta. Mobile agent security and Telescript. In *Proceedings of IEEE COMPCON '96*, February 1996.
23. C. F. Tschudin. *An Introduction to the* `MO` *Messenger Language*. University of Geneva, Switzerland, 1994.
24. James E. White. Telescript Technology: Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.
25. F. Yellin. Low Level Security in Java. Technical report, Sun Microsystems, 1995.