DESIGN, IMPLEMENTATION AND TESTING OF MOBILE AGENT PROTECTION MECHANISM FOR MANETS

Khaled E. A. Negm
Etisalat College of Engineering, Sharjah, POB 980, UAE

Keywords: Secure Mobile Transactions, Mobile Agent Protection, Business to Business Secure Transaction.

Abstract: In the current research, we present an operation framework and protection mechanism to facilitate secure environment to protect mobile agents against tampering. The system depends on the presence of an authentication authority. The advantage of the proposed system is that security measures is an integral part of the design, thus common security retrofitting problems do not arise. This is due to the presence of AlGamal encryption mechanism to protect its confidential content and any collected data by the agent from the visited host. So that eavesdropping on information from the agent is no longer possible to reveal any confidential information. Also the inherent security constraints within the framework allow the system to operate as an intrusion detection system for any mobile agent environment. The mechanism is tested for most of the well known severe attacks against agents and networked systems. The scheme proved a promising performance that makes it very recommended for the types of transactions that needs highly secure environments, e. g., business to business, stock market updates, and any real time data synchronization..

1 INTRODUCTION

In a broad sense, a software agent is any program that acts on the behalf of a user, just as different types of agents (e.g., travel agent and insurance agents) that represent other people in day-to-day transactions in real world. Applications can inject mobile agents into a network, allowing them to roam the network on either a predetermined path, or agents themselves determine their paths based on dynamically gathered information. Having accomplished their goals, the agents return to their “hosts” in order to report their results to the user.

However; the mobile agent paradigm also adds significant problems in the area of security and robustness. Malicious agents are similar to viruses and trojans, they can expose hosts, they visit, to the risk of system penetration. While in transient, the agent’s state becomes vulnerable to attacks in different ways. An agent is likely to carry-as part of its state-sensitive information about the user identity, e.g., credit card information, personal confidential preferences, or any other form of electronic credentials. Such data must not be reveled to any unauthorized hosts or modified by unauthorized users. Unless some countermeasures are taken, such agents can potentially leak or destroy sensitive data and disrupt the normal functioning of the host.

In the current research we present a protection scheme for the mobile agents that incorporate standard cryptographic mechanisms into the agent transfer protocol functions. The use of the one-way-hashing and digital signatures is two fold; *first* detect active, passive and tampering attacks, and *second* to establish the identity of the servers participating in the anti-tampering program (ATP) (Vincenzetti 1993 and Sielken 1999). Also encryption is used to prevent passive attacks on the agent's state while it is in transient (Roth 2000 and Gary 1998).

2 MOBILE AGENT SECURITY ANALYSIS

Mobility allows an agent to move among hosts seeking computational environment in which an agent can operate. The host from which an agent originates is referred to as the home host that normally is the most trusted environment for an agent (Fuggetta 1998, FIPA 1998, and OMG-TC 1997).

In the mobile agent environment, security problem stems from the inability to effectively

E. A. Negm K. (2005).
DESIGN, IMPLEMENTATION AND TESTING OF MOBILE AGENT PROTECTION MECHANISM FOR MANETS.
In *Proceedings of the First International Conference on Web Information Systems and Technologies*, pages 389-396
DOI: 10.5220/0001236403890396

extend the trusted environment of an agent’s home host to other hosts. The user may digitally sign an agent on its home host before it moves onto a second platform, but this resembles a limited protection. The next host receiving the agent can rely on this signature to verify the source and integrity of the agent’s code, data, and state information provided that the private key of the user has not been compromised. For some applications, such minimal protection may be adequate through which agents do not accumulate state. For other applications, these simple schemes may prove inadequate. For example; the Jumping Beans agent system addresses some security issues by implementing a client- server architecture, whereby an agent always returns to a secure central host first before moving to any other platform (Ad Astra 1998, Negm 2003, and Negm 2004).

Some other category of attacks on the agent involves tampering by its executing visited hosts. As such, if that server is corrupted or becomes malicious, the agent's state is vulnerable to modification (Farmer 1996). Although a lot of research has been done in this area, one of the remaining problems is the presence of a nontrusted malicious host that attacks mobile agents, for example; a travel agency’s agent system might modify the best offer the agent has collected, so that its own offer appears to be the cheapest one. Also, the travel agency might change the list of travel agencies that the agent is going to visit to increase its chances to propose a better offer and/or get the prices of other travel agencies before making its offer to the agent. All of these attacks involve eavesdropping and tampering and yet all the published schemes represent a simple mechanism of protection that can not guarantee secure transactions for the agents.

3 PROTECTION MECHANISM AND ITS IMPLEMENTATION

Several In the current research we implement a mechanism by which tampering of sensitive parts of the state can be detected, stopped, and reported to the Master Agent (MA). The framework is composed of different modules.

First the initialization module, this module includes two coordinating entities MA and Slave Agents (SAs). The user resides on its own platform and/or on a server to create the MA acquiring only that MA must exclusively reside on a secure trusted host. Then the MA creates SAs on another host (or the same MA host) in which being created on a secure host is not a must. Next MA defines tasks and subtasks to the SAs to achieve based on the user preferences. Then the SAs move from host to host to finish the tasks (and/or subtasks) given from the MA (that includes a central knowledge-base and a central management components.).

The *second* module is the Constraints Module that contains conditions and rules for each agent to follow. This module presents the first line of defense in which the characteristic details and operational parameters of the visited host are listed. The *third* Module is the Encryption Module, presenting the second line of defense to afford the security for the agents’ states. The encryption module contains two parts. The *startup* part, allows the user to declare which part of the agent as a read-only in which any tampering with the read-only objects can be detected. The *second* part is a secure storage container, that allows the agent to create an append-only container by which the agent can check in data (when executed) and store it in the container, so no one can delete or modify it without detection.

3.1 The Initialization Module

The concept of MA-SA was first introduced by Buschmann in 1996 to support fault tolerance, parallel computation and computational accuracy (Buschmann 1996). Also Lange demonstrated in 1997 that it is also applicable to support tasks at remote destinations and extended it to fit mobile agents (White 1997). The MA-SA concept is interacting as follows: the MA creates SAs, then the master delegates the subtasks to the SAs, and finally after the slaves have returned the results, the master combines the results. The master can assign more than one task at a time and the slaves can execute them concurrently. A major benefit of this abstraction is the exchangeability and the extensibility in which decoupling the SA from the MA and creating an abstract slave class allows to exchange the slaves’ implementation without changes in the master’s code.

Depending on the MA-SA concept, we built up a system to facilitate a solution to the mobile agent security problem. To achieve this, confidential data is contained in a secure place that is the MA host (or heavily protected if carried by the SAs). Then the SA must carry essential data to fulfill the task assigned by the MA (Tripathi 1999).

Tables 1 and 2 present the two listings of the pseudo code implementation of MA and SA. First, the doTask() method is called so the MA moves to the first host where it uses its strategies to split the tasks into subtasks. Then the MA assigns subtasks to the SAs. Afterwards it waits for the results which will be returned by the SAs.

Table 1: MA pseudo code

```
Public class MA extends Agent {
 private ConstarintManager cm;
 private Vector Tasks;
private vector sentSAIds;
 protected void doTask() {
 do {
  getCurrentHost().transfer(this object)
  splitTasks();
  waitForResults();
  mergeResults();
  } while (!supertask.finished());
  sendResultsMAHome();
 }
 private void splitTask() {
  // 1. apply strategy to divide the task
  // 2. refine constraints for the subtasks
  for (int i=0; i < tasks.size();++){
  SA = new SA (subtask, constraints);
  sentWorkIds.add(w.getId());
  w.doTask();
  }
  }
}
```

Table 2: SA pseudo code

```
Public class SA extends Agent {
 private ConstarintManager;
 private Vector Tasks;
 SA (Task t){task=t; }
 protected void doTask() {
  do {
 task.execute();
 addResult(task.getResults());
 getCurrentHost().transfer(this object)
 } while (!task.finished());
}
  private void addResult(Results=r){
   if (cm.checkConstarints(task,r))
    sendResulstToMA;
  }
}
```

3.2 The Constraint Module

After starting the initialization module, the constraints module starts running in a supervisory parallel fashion during the transactions. The constraints module is composed of three parts:

a. *Routing Constraints***:** which define variables for the agent's itinerary that lists hosts, operating systems' type and version number including hopes for travelling. This type has to be checked every time before an agent moves to another location.

b. *Execution Constraints*: which define requirements on the SA visited system's environment which contain a limitation list of hardware (the amount of memory storage) or software (for example a specific version of the database-access software or an LDAP-service) requirements.

c. *Merging Constraints*: which define the relations between subtasks that are generated by the strategies. In contrast to the other constraints, merging constraints are stored exclusively by the MA.

3.3 The Cryptography Module

The cryptography module provides a secure container for any credentials that the agent might carry and acts as an intrusion detection system to discover tampering. This protection mechanism contains two parts:

a. The read only-state: in which it function to assign part of the "agent's object" as read-only sub-object in which its credentials could not be modified by anyone, and thus are read-only during its travels. To protect such read-only state we have to declare the associated objects as constants and incorporate a cryptographic mechanism to protect these constants.

In Table 3 we list the pseudo code of this object. It contains a vector of objects of arbitrary type, along with the agent owner's digital signature on these objects. The digital signature is computed by first using a one-way hash function to digest the vector of objects down to a single 128-bit value, and then encrypt it using the private key of the agent's owner. The Digital Signature Algorithm (DSA) is used for this purpose (Bellare, M.1997).

$$sign = K_A^-(h(objs))$$

The `verify` method of the `ReadOnlyContainer` object allows any host on the SA's path to check whether the read-only state has been tampered via contacting the certifying authority to honor the user's signature (while it needs an access to the agent's public key.) It uses the public key to decrypt the signature, and compares the result with a recomputed one-way hash of the vector of objects. If these values match, the visited host can assume that none of the objects has been modified since the signature was computed. Thus, the condition it checks are:

$$h(objs) = K_A^+(sign) .$$

The read-only container mechanism is limited in utility to those parts of the state that remain constant throughout the agent's travels. But in real life, SAs collect data from the hosts it visits and need to prevent any subsequent modification of the data. This could be termed as write-once data.

b. Append-only logs: This object guarantees that the stored entries within it can not be deleted, modified or read by an unauthorized user. When data object needs to be nonmodifiable for the remainder of the agent's journey, it can be inserted into this append only log and to provide secrecy, the data is then encrypted with the MA's public key before it is stored in the log file. We used this module to preserve the results that the SA's had gathered. The pseudo code of this object is shown in Table 4.

The `AppendOnlyContainer` object contains vector of objects to be protect, along with their corresponding digital signatures and the identities of the signers (in case of MA only). It also contains a `checkSum` array to detect tampering. When an SA is created, its `AppendOnlyContainer` is empty. The checksum is initialized by encrypting a nonce with the agent's public key

$$checkSum = K_A^+(N_a)$$

This nonce N_a is not known to any host other than the MA's host, and must be kept secret. Therefore, it is not carried by the SA. The encryption is performed using the ElGamal cryptosystem (ElGamal 1984). At any stage during the SAs travel, the agent can use the `checkIn` method to insert an object X (of any type) into an `AppendOnlyContainer`. For example, after collecting a quotation from a travel agent, it can check the in-value, in order to protect it from any further modification. The `checkIn` procedure requests the current server "C" to sign the object using its own private key. The object, its signature and the identity of the signer are inserted into the corresponding vectors in the `AppendOnlyContainer`. Then, the checksum is updated as follows

$$checkSum = K_A^+(checkSum + Sig_C(X) + C) .$$

First, the signature and the signer's identity is concatenated to the current value of the checksum. This byte array is then encrypted further using the MA's `ElGamal` public key, rendering it to be unreadable by anyone other than the agent's owner. Then, the encrypted version of the object would be carried along and protected from tampering. When the agent returns, the user can use the verify method to ensure that the `AppendOnlyContainer` has not been tampered. As shown in Table 4, the verify process works backwards, unrolling the nested encryptions of the checksum, and verifying the signature corresponding to each item in the protected state. In each iteration of this loop, the following decryption is performed

Table 3: The `ReadOnlyContainer` pseudo code

```
class ReadOnlyContainer {
  Vector objs; // the read-only objects being carried along
  byte[] sign; // owner's signature on the above vector
  // Constructor
  ReadOnlyContainer(Vector o, PrivateKey k) {
    objs = o;
    sign = DSA—Signature (hash(objs), k);
  }
  public boolean verify(PublicKey k) {
    // Verify the agent owner's signature on the objects
    // using the owner's public key
  }
}
```

Table 4: The AppendOnlyContainer

```
class AppendOnlyContainer –{
  Vector objs; // the objects to be protected
  Vector signs; // corresponding signatures
  Vector signers; // corresponding signers' URNs
  byte[] checkSum; // a checksum to detect tampering
  // Constructor
  AppendOnlyContainer(PublicKey k, int nonce) {
  objs = new Vector(); // initially empty
  signs = new Vector(); // initially empty
  signers = new Vector(); // initially empty
  checkSum = encrypt (nonce); // with ElGamal key k
  }
  public void checkIn (Object X) {
  // Ask the current server to sign this object
  sig = host.sign (X);
  // Next, update the vectors
  objs.addElement (X);
  signs.addElement (sig);
  signers.addElement (current server);
  // Finally, update the checksum as follows
  checkSum = encrypt (checkSum + sig + current server);
  }
  public boolean verify (PrivateKey k, int nonce) {
  loop {
    checkSum = decrypt (checkSum); // using private key k
    // Now chop off the ''sig'' and server's URN at its end.
    // These should match the last elements of the signs and
    // signers vectors. Verify this signature.
  } until what ever is left is the initial nonce;
  }
}
```

$$K_A^-(checkSum) \Rightarrow checkSum + Sig_S(X) + S,$$

where S is the server in the current position of the `objs` vector. The verify procedure then ensures that

$$K_S^+(Sig_S(X)) == h(X).$$

If any mismatches are found, the agent's owner knows that the corresponding object has been tampered and then it can discard the value. The objects extracted up to this point can still be relied upon to be valid, but other objects whose signatures are nested deeper within the checksum can not be used. When the unrolling is complete, we are left with the random nonce that was used in the initialization of the checksum. This number is compared with the original random number N_a. If it does not match, a security exception can be thrown.

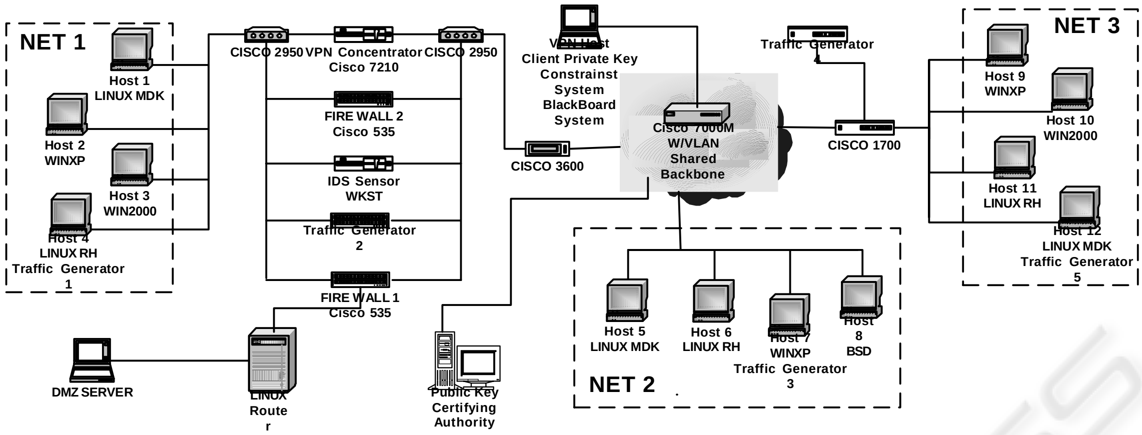

Figure 1: The testing network

Table 5: Testing Scenario Parameters

Scenario	Client	Master (I/O)	Slave Host: ports	Target Hosts
1	VPN host	VPN host: (4444/3333)	DMZ host: 3062 DMZ host: 3063 DMZ host: 3064	H1: 3155 H5: 3150 H10: 2774
2	VPN host	DMZ host (44444/60000)	H4: 3009 H11: 3010 H12: 3011	NET1 NET2 NET3

Table 6: Validity and Parallelizing Test Parameters

packets	Bytes	Source Ports	Destination Ports
20	3000	Any	3150
40	7050	Any	3155
14	2683	Any	2774
56	6388	Any	All the remaining
138	19121	All traffic	All traffic

4 TESTING

The basic goal of the testing is to monitor the system behavior against malicious attacks and measure the network utilization for different operational scenarios. We executed the most common well know attacks for agents, systems, and networks against the proposed system and collected the results to study the feasibility (CVE 2005). Five traffic generators are installed and distributed among its testing network to simulate the real world environment. Additional normal www traffic is generated while activating and running the system to introduce the normal competitive packet dynamics and latencies within the queuing buffers in each router (TG 2004).

The major role of the utilization testing is to evaluate the network resources usage while implementing the framework. Also we performed functionality testing of the framework in which *"Parallelizing"* scheme enables concurrent task execution. In every testing scenario, there is a list of hosts for the SAs to visit according to their respective predefined strategy.

4.1 Validity and Parallelizing Test

In this scenario, the client operates from the VPN host at which he creates the MA Then the MA creates three ASs on the DMZ host from which they start travelling to their designated hosts according to the predefined constraints.

Each SA queries its target host via the dedicated port for such a process. Then each SA will activate a security query to the CVE host requesting security clearance to communicate to the dedicated target hosts. On receiving the clearance it will proceed to collect and/or communicate to the target host. In case of successful transaction, the collected information is returned to MA. Then the MA prepares the final report and pass it to the user. Note that this is not a fully guaranteed security check, but it helps in some ways to eliminate some security risks especially for home users.

In here two of the SAs are targeting hosts 5 and 10 will stop execution due to the fact that the dedicated ports of communication assigned by these host match malicious attacks (according to the CVEs) on the SA itself, namely the deep throat, the Foreplay and the Mini BackLash attacks on port 3150 and the subseven, and subseven 2.1 Gold on port 2774. This is achieved through the confirmation channel between the SAs and the MA to approve communication via the designated port by the visited host. The MA confirms communication after checking the CVEs list.

4.1 DDoS Attack Test

In this scenario a malicious software is activated at Host 1 acting against the three networks in which host 6 and 9 are trojaned to be malicious to deny any execution to all arriving agents. In general the MA creates five the SAs at Host 5. Then each one moves to all hosts to collect the desired information. During this test, the MA enforces a new constraint that concerns retries in denial-of-service attacks as:

```
* if repeatedCreation() < 3 then
begin true end
else alarm_user(); false end.
```

The method `repeatedCreation()` returns the number of already done retries to create a SA for a certain task. So for example if one of the SA failes and the MA creates another one, then the return value of this method would be one. The constrains for the SAs are the same as in the previous scenario:

```
* if placename == "Host 2→12" then
begin true end.
* if ostype == "LINUX MDK or RH"
then begin true end.
```

In here the system information is not collected from hosts in NET1 because it suffers from DDoS and host 11 because it does not have the correct name and the last one because it is not the desired Linux machine. But the encryption module will detect this behavior, file it, and report it back to the user via the blackboard system.

The DDoS will not propagate from NET1 to the other networks because of the network intrusion detection systems (NIDS) and host based intrusion detection systems (HIDS) installed to filter out any traffic back and forth. The SA that moves to host 1 do not return any status report or result within the given deadline so the MA retried to send it several time. After retrying it twice the MA's constraint number one returns false. Thus, the MA stops trying to send an agent to these hosts and returns a special report to the user.

This shows that a malicious host can not trap or stop the overall process by a denial of service attack. When the SA does not return within a given deadline the MA could start another one or redefine the subtasks and then start a new one.

5 SUMMARY AND CONCLUSION

Mobile agents differ from other techniques in regard to security issues and security mechanisms, whose requirements are not met by classical security systems. Concerning security in traditional operating systems, the system is always trusted. This is not true for mobile agents, here the visited operating system can be the untrusted one and the agent is the trusted one. The problem arising is that the users have no chance to check the functionality of the operating system.

To eliminate some of the security risks we incorporate a sophisticated mechanism to be built in within the mobile agent design by which none would be able to retrofit into the application. This aim is fully accomplished. The framework limits the risks of leakage and tampering as the data stored in the Master Agent will never be accessible to potential malicious hosts, since it will only reside on trusted hosts. In addition to implementing the MA-SA system in an enhanced way to facilitate full optimizad operation and protection to the agent system.

Besides the main intent to make mobile agent technology more secure the Master Agent-Slave Agent Framework provides additional benefits and boosts some of the mobile agent's advantages due to its design and structure (e.g. flexibility, simplicity, separation of concerns, etc.). Its separation of code focusing on coordination and code focusing on computation make the pattern an ideal basis for the framework. This design allows easy integration of this framework in applications and eases porting to other mobile agent systems.

The framework consists of a coordinating entity (the MA) and several independent entities (the SAs). The MA holds all the current knowledge found by the Slave Agents and uses this knowledge to accomplish its task. The key difference to the client-server paradigm is that the MA component is mobile as well. So it can move to a host near the area its SAs scenarios will operate in. The only prerequisite is that the MA must exclusively visit secure trusted places. In the worst case this is the host where it has been initialized. We have demonstrated that this framework solves special aspects of mobile agent security, in addition to that eavesdropping information and tampering the agent is no longer possible or does not reveal any confidential information.

Every time the agent departs a host, its server inserts a log entry into the AppendOnlyContainer. This entry includes the current server's name, the name of the server from which the agent arrived, and the name of its intended destination. This travel log can be used by the agent's owner when the agent returns, to verify that it followed the itinerary prescribed when it was dispatched.

If the agent's itinerary is known in advance of its dispatch, we can insert a copy of the itinerary into the agent's `ReadOnlyContainer`. Thus, each host visited by the agent has access to the original itinerary, as intended by the agent's creator. The receiving host can check the current itinerary to ensure that the agent is following the specified path, and that the method to be executed is as specified originally.

This ensures that any tampering with the method's parameters by any host on the agent's path can be detected, before the agent is allowed to execute. In addition, an audit trail of the agent's migration path can be maintained using an instance of the `AppendOnlyContainer` class. One limitation of `AppendOnlyContainer` scheme is that the verification process requires the agent's private key, and can thus only be done by the agent's host.

6 FUTURE WORK

Currently we are working on enhancing the IDS feature of the system by adding a backboard system to the encryption module. But in this case we have to implement a rigorous reporting mechanism from the slave agents to the master agent.

ACKNOWLEDGEMENT

The author would like to thank Cisco systems in Dubai, UAE to support this research by the needed Cisco equipments. Also the author would like to acknowledge the Etisalat Academy in Dubai to facilitate the premises to run this research.

REFERENECS

Bellare, M. *et al* (1997). S. Goldwasser, and D. Micciancio, “Pseudo-Random Number Generation with Cryptographic Algorithms: the DSS Case, Crypto 97, LNCS 1294, pp. 1-12.

Buschmann, F., *et al* (1996). Pattern-Oriented Software Architecture: A System of Patterns, John Wiley, UK.

CVE (2005). Common Vulnerability Exposure http://cve.mitre.org/.

ElGamal, T. (1984). A public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, Proc. of Crypto ’84, LNCS 196, pp. 10-18.

Farmer, W. *et al* (1996). Security for Mobile Agents: Issues and Requirements. In Proc. of the 19th International Information Systems Security Conference, pp. 591-597.

FIPA (1998). Agent Management,. Specification, part 1, ver. 2.0, Foundation for Intelligent Physical Agents, 1998.

Fuggetta, G, and G. Vigna (1998). Understanding Code Mobility, IEEE Transactions on Software Engineering, 24, pp. 342-361.

Gray, R., (1998). D’Agents: Security in a Multiple Language, Mobile-Agent System, in Mobile Agents and Security, G. Vigna, ed., LNCS 1419 pp. 154-187, Springer.

Negm, K. A. E. (2003). Implementation of Secure Mobile Agent for Ad-Hoc Networks, WEAS Transactions on Communications, Vol. 2, pp. 519-526.

Negm, K. A. E., and W. Adi (2004). Secure Mobile Code Computing in Distributed Remote Environment, Proc. 2004 IEEE International Conference on Networking, Sensing and Control, pp. 270-275.

OMG-TC (1997). Mobile Agent System Interoperability Facilities Specification, OMG-TC-orbos/97, 1997.

Roth, V., (2000). Scalable and Secure Global Name Services for Mobile Agents,” 6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages.

Sielken, R. (1999). Application Intrusion Detection, Univ. of Virginia Computer Science Technical Report CS-99-17.

TG (2004).Traffic Generator, http://www.postel.org/services.html.

Tripathi, A. *et al* (1999), N. Karnik, N. Vora, T. Ahmed, R. Singh, Mobile Agent Programming in Ajanta, Proc. of 19th IEEE International Conference on Distributed Computing Systems, pp. 190-197.

Vincenzetti, D. and M. Cotrozzi 1993. ATP anti tampering program, in Edward DeHart, ed., Proc. of Security IV Conf.-USENIX Assoc., pp 79-90.

White, J. 1997. Mobile Agents,” in Software Agents (J. Bradshow, ed.) 437-472, MIT Press.