Identifying collusions: Co-operating malicious hosts in mobile agent itineraries

E.C. Vijil
School of Information Technology
Indian Institute of Technology,
Bombay, India - 400076
vijil@it.iitb.ac.in

Sridhar Iyer
School of Information Technology
Indian Institute of Technology,
Bombay, India - 400076
sri@it.iitb.ac.in

ABSTRACT

A mobile agent is vulnerable to attacks by malicious hosts executing it. One of these attacks is the tampering of the data being carried by the agent. This is particularly relevant in comparison shopping scenarios where the mobile agent collects price quotes from various hosts. A malicious host in the itinerary could modify/delete the prices quoted by previous hosts, and/or insert spurious prices on behalf of latter hosts and modify the itinerary.

While there is some work on detecting tampering by individual hosts, there is not much work on identifying malicious hosts that may *collude* with each other to tamper with the mobile agent's data. In this paper[1] we present mechanisms that enable the detection of tampering by individual as well as colluding hosts, and also the identification of such hosts.

1. INTRODUCTION

Mobile agents are software programs that may move from one host to another to perform computations on behalf of their owner. The set of hosts visited by the agent (termed as the *itinerary*), may be *static*, i.e., pre-determined by the agent owner, or *dynamic*, i.e., decided by the agent as it moves in the network. The reader may refer to [2], for a survey of the mobile agent design paradigm, different agent frameworks and possible applications. One interesting application for mobile agents is comparison shopping in e-commerce [3]. Here, the agent owner launches the agent with a description of the goods he wishes to purchase and the agent visits several hosts in the e-market, collects their price quotes and returns to the owner.

At any host, the execution environment of the agent is controlled by the host. Hence mobile agents are vulnerable to attacks by malicious hosts [1, 4]. One of these attacks is the tampering of data being carried by the agent. For example, in the above comparison shopping scenario, a malicious host could modify/delete the prices quoted by other hosts, and/or modify the itinerary itself.

Several schemes have been proposed for detecting the modification of an agent's data by individual malicious hosts [3, 5]. However, these schemes do not address the problem of two or more malicious hosts *colluding* with each other to delete the data of other hosts. For example, suppose an agent visits hosts $H_1,\cdots,H_i$, $H_{i+1},\cdots,H_j$, $H_{j+1},\cdots,H_n$. Now if hosts H_i and H_{j+1} are both malicious and collude with each other, they may delete the data of $H_{i+1},\cdots,H_j$ without being detected.

In [5], Karnik et. al, propose the notion of *AppendOnlyContainer* for detecting the tampering of an agent's data by individual malicious hosts. However the mechanism does not indicate the identity of the malicious hosts. Also, two hosts *colluding* to delete the data of intermediate hosts may escape detection in this mechanism.

In this paper, we incorporate and extend the notion of the *AppendOnlyContainer* to include not only the detection of tampering but also the identification of the malicious host. Subsequently, we introduce the notion of Expected Number of Deletions (END), that helps us to detect deletion of data by colluding malicious hosts in static as well as dynamic itineraries.

The paper is organized as follows: In Section 2 we discuss the *AppendOnlyContainer* and in Section 3 we describe our extension to this mechanism that enables identification of the malicious host. In Sections 4 and 5, we describe our schemes for detecting deletion of data by colluding malicious hosts. Section 6 concludes with a discussion of related work.

We use the following notation throughout this paper:

$E_A(X)$	Encryption of data X using public key of A.
$D_A(X)$	Decryption of data X using private key of A.
$hash(X)$	A one way hash on the data X.
$Sig_A(X)$	Signing of $hash(X)$ using private key of A.

2. AOC: APPEND ONLY CONTAINERS

In this section, we briefly discuss the *AppendOnlyContainer* (AOC) mechanism proposed in [5]. Given an agent that visits several hosts to collect data, the agent owner may use the AOC mechanism to detect any modification/deletion of data by individual malicious hosts. The basic idea in AOC is: For each visited host C, record

1. the data collected at the host, X,
2. the digital signature of the host on that data, $Sig_C(X)$, and

[1]This project was funded by the Ministry of Information Technology, India.

3. the identity of the signing host, C.

A *checksum* is used to detect modification/deletion of data from the AOC. The computation of the *checksum* and its verification is discussed below.

When an agent starts its itinerary, its AOC is empty. The checksum is initialized by encrypting a random nonce with the agent owner's public key:

$$checkSum = E_{owner}(N_a) \quad (1)$$

This nonce N_a is kept secret and is not carried by the agent.

When a host C wants to insert data X, C first signs X using its own private key, D_C. Then the data item X, its signature $Sig_C(X)$ and the identity of the host C, are inserted into the appropriate arrays in the AOC. The checksum is then updated as follows:

$$checkSum = E_{owner}(checkSum + Sig_C(X) + C) \quad (2)$$

where + denotes *concatenation* of the corresponding values.

When the agent returns to the owner, the integrity of the data is verified by decrypting the *checksum* and verifying the signature, in an iterative manner. Each step of the iteration is:

$$D_{owner}(checkSum) \rightarrow checkSum + Sig_C(X) + C \quad (3)$$

followed by verifying if:

$$E_C(Sig_C(X)) == hash(X) \quad (4)$$

If in the last iteration, the agent owner recovers the original random nonce N_a, it can be inferred that the AOC has not been tampered with.

If the verification procedure fails in any iteration, the agent owner can infer that the AOC has been tampered with. It also implies that the values extracted up to this iteration are valid, while other values whose signatures are still nested within the checksum cannot be relied upon.

In [6, 7], Roth describes an attack that will allow a malicious host to forge the AOC. The problem arises because a malicious host can abuse other hosts as oracles for signing and checksum computation. Assume that a malicious host M receives an agent A_O created by host O. Let the *checksum* carried by that agent be $checksum_O$. M creates another agent A_M and initializes its checksum with $checksum_O$. M can successfully add all the data added to A_M's AOC to A_O's AOC without risking detection.

A solution to this problem, also suggested in [7], is as follows: For each agent instance, a unique identifier is constructed by the owner as $id = hash\,(D_{owner}\,(code, timestamp))$, where *code* is the agent's code. The *timestamp* is used to distinguish between multiple instances of the same agent. This unique identifier is carried by the agent while it visits the hosts in its itinerary. Whenever a host C signs a data item X, it actually computes $D_C(id, X)$. This means that X is valid only in the context of the agent instance *id*. During verification, the host owner needs to check the context of the agent instance and verify that each data item was added in the context of the correct agent instance.

Since the above problem does not bear directly upon the focus of this paper, in the interest of simplicity, we continue to use the basic AOC mechanism for our discussion. Our proposal can be easily extended for an AOC mechanism augmented with the above modification.

Now, we observe that while the AOC mechanism ensures the detection of: (i) any modification/deletion by a host to the data collected from earlier hosts, and (ii) any modification by a host to its own data after it has been added to the AOC, it does not indicate the identity of the malicious host. Also, as explained subsequently (section 4.1), two hosts *colluding* to delete the data of intermediate hosts may escape detection in the AOC mechanism. In the next sections, we incorporate and extend the notion of AOC in order to provide solutions to these problems.

3. X-AOC: IDENTIFYING THE MALICIOUS HOST

In the AOC mechanism a host C upon adding data X into the AOC, re-computes the *checksum* using $Sig_C(X)$ and C. While this is sufficient for the agent owner to detect tampering of data by a malicious host, it does not indicate the identity of the malicious host.

We propose *X-AOC*, an extension to the AOC mechanism that enables the identification of the malicious host. The main idea is: A host C upon adding data X to a *container*, should append $Sig_C(container)$ and C, to the *container*. $Sig_C(container)$ is used to re-compute the *checksum*. In other words, a host, instead of merely signing the data item added by it, is required to sign the entire contents of the container at that point.

X-AOC assumes that all hosts in the itinerary add some data to the container. In case of multiple malicious hosts, X-AOC indicates the identity of the *last* malicious host that added data to the container. The detailed algorithm for X-AOC is shown in Figure 1.

The checksum is initialized as in equation 1. To insert an item X, any given host C first adds X to the *container* and signs the *container* using its private key, D_C. The checksum is then updated as follows:

$$checkSum = E_{owner}(checkSum + Sig_C(container) + C) \quad (5)$$

When the agent returns to the owner, the integrity of the data is verified by decrypting the checkSum and verifying the signature, in an iterative manner. Each step of the iteration is:

$$D_{owner}(checkSum) \rightarrow checkSum + Sig_C(container) + C \quad (6)$$

followed by verifying if:

$$E_C(Sig_C(container)) == hash(container) \quad (7)$$

If in the last iteration, the agent owner recovers the original random nonce N_a, it can be inferred that the *container* has not been tampered with.

If the verification procedure fails in any iteration, the agent

```
class X-AOC{
    Vector dataContainer;
    byte[] checkSum;
    X-AOC (PublicKey k, int nonce){
        dataContainer = new Vector();
        checkSum = encrypt (nonce); // with key k
    }
    public void checkIn(Object X) {
        dataContainer.addElement(X);
        sign = host.sign(datacontainer);
        checkSum = encrypt (checkSum + sign + hostId);
    }
    public boolean verify (PrivateKey k, int nonce){
        previoushost = null;
        loop{
            checksum = decrypt (checkSum);
            sign = extract.sign (checksum);
            currenthost = extract.hostId (checksum);
            checkSum = extract.checkSum (checksum);
            If (Verify(sign) == TRUE), then
                previoushost = currenthost
            Else
                throw security exception.
            // Either currenthost or previoushost is malicious.
        }until (checkSum == nonce);
    }
}
```

Figure 1: ***The X-AOC mechanism***

owner can infer that the *container* has been tampered *by either the current host or the previous host.* For example, suppose hosts $H_1, \cdots, H_i, H_j, \cdots, H_n$ are visited by the agent, in that order. Without loss of generality, let the verification fail while verifying $Sig_{H_i}(container)$. This implies that the tampering was done either by H_i after generating the checksum, or by H_j before generating the checksum. Hence, the malicious host can be identified to be one of either H_i or H_j.

Thus, the X-AOC mechanism not only detects modification/deletion of data by malicious hosts, but also indicates the identity of the malicious host. In the next section we discuss mechanisms to detect collusions among malicious hosts, in static as well as dynamic itineraries.

4. COLLUDING MALICIOUS HOSTS

In this section we address the problem of collusion among malicious hosts and propose some solutions to the same. We use the AOC mechanism to demonstrate collusions. However, our solution can be easily extended to any protocol that uses other variants of signature chaining to protect data items.

4.1 Attack

Suppose an agent visits hosts H_1, $\cdots$, H_i, H_{i+1}, $\cdots$, H_j, H_{j+1}, $\cdots$, H_n, in that order. Further, assume that hosts H_i and H_{j+1} are both malicious and collude with each other.

Host H_i on receiving the agent, does the following:

1. It adds its own data D_i and signature $Sig_{H_i}(D_i)$ to the AOC.
2. It re-computes the checksum as given in equation (2). We shall denote this checksum by $checkSum_i$.
3. It forwards the agent to H_{i+1} and sends $checkSum_i$ to H_{j+1}.

H_{j+1} on receiving the agent does the following:

1. It removes data items $D_{i+1}, \cdots, D_j$ from the AOC.
2. It adds its own data D_{j+1} and signature $Sig_{H_{j+1}}(D_{j+1})$ to the AOC.
3. It re-computes the checksum as given in equation (2). However, it uses $checkSum_i$ instead of $checkSum_j$.
4. It forwards the mobile agent to the next host in the itinerary.

In the conventional AOC mechanism, the agent owner would be unable to detect that items $D_i, \cdots, D_j$ have been removed from the AOC.

We now propose solutions for detecting collusions among malicious hosts, in static as well as dynamic itineraries.

4.2 Detecting collusions in static itineraries

In the case of a *static itinerary*, the agent owner apriori knows the identities of the hosts to be visited by the agent. The order in which these hosts are visited may also be static or may be dynamically decided by the agent.

The X-AOC mechanism can be easily extended for detecting collusions in a static itinerary, as follows:

1. Each visited host is required to compulsorily add some data to the *container*, and
2. The agent owner while verifying the integrity of the *container*, checks if there is data corresponding to each host in the itinerary.

If the data corresponding to a host H is missing, then it indicates that the data was deleted by colluding malicious hosts. If the order in which various hosts are visited is also known, then the missing data also gives an indication of the identities of the malicious hosts.

In the next section we present our scheme for detecting collusion in dynamic itineraries.

5. DETECTING COLLUSIONS: DYNAMIC ITINERARIES

In the case of a *dynamic itinerary*, the agent owner does not apriori know the identities of the hosts to be visited by the agent. The set of hosts to be visited (and the order in which they are to be visited) is dynamically decided by the agent.

A simple extension to the X-AOC mechanism for detecting collusions in a dynamic itinerary would be:

1. Any given host C, adds data X to the *container*, and updates the *checksum* as in equation 5.

2. C sends a *notification* M_{CX}, to the agent owner, indicating that C was visited.

3. The agent owner while verifying the integrity of the *container*, as in equations 6, 7, also checks if there is data corresponding to each host that sent a *notification*.

For any *notification*, if the corresponding data is missing from the *container*, then it implies that the data was deleted by colluding malicious hosts.

However, the above solution is expensive in terms of the number of notifications required. We argue that all hosts need not participate in the collusion detection process. We use a notion called *Expected Number of Deletions* to reduce the number of notifications, without significantly reducing the ability of the owner to detect collusions.

5.1 Expected Number of Deletions

Let k and n be the owner's estimate of the number of *malicious* and *honest* hosts respectively. Now the malicious hosts may collude with each other to delete the data of one or more honest hosts. We assume that malicious hosts do not act against each other, and only the data of honest hosts may get deleted.

The Expected Number of Deletions (END) is the average number of honest hosts whose data may get deleted, assuming that all permutations of the itinerary are equally likely. If $P(m)$ is the probability that the data of exactly m honest hosts are deleted, then,

$$END = \sum_{m=0}^{n} mP(m) \tag{8}$$

We now have the following theorems.

THEOREM 1. $P(m) = (n+1-m)\frac{\binom{k-2+m}{k-2}}{\binom{k+n}{k}}$.

PROOF. Given k malicious hosts and n honest hosts, there are $k+n$ hosts in the itinerary and $\binom{k+n}{k}$ possible configurations of the itinerary. For a given configuration, let $x_1, \cdots, x_k$ be the positions of the malicious hosts. Without loss of generality, we assume that x_1 is the position of the first malicious host, x_k is the position of the last malicious host, and that there are m honest hosts between x_1 to x_k.

Let the positions in the itinerary be numbered in the range $[1, 2, \cdots, k+n]$. Now the highest value that x_k can take is $k+n$. Since there are m honest hosts and $k-2$ malicious hosts in between positions x_1 and x_k, the highest value that x_1 can take is $(k+n)-(k-2+m)-1$, i.e., $(n+1-m)$. Similarly, the lowest value that x_1 can take is 1 and the lowest value that x_k can take is $1+(k-2+m)+1$, i.e., $(k+m)$. Thus $1 \le x_1 \le (n+1-m)$ and $(k+m) \le x_k \le (k+n)$.

Now for a given m, the pair x_1, x_k can be chosen in $(n+1-m)$ ways. Also, there are $\binom{k-2+m}{k-2}$ configurations of the $(k-2)$ malicious hosts and m honest hosts in between x_1 and x_k.

Thus the probability that the data added by exactly m honest hosts will be deleted is given by
$P(m) = (n+1-m)\frac{\binom{k-2+m}{k-2}}{\binom{k+n}{k}}$. □

THEOREM 2. $END = \frac{n(k-1)}{k+1}$.

PROOF. We note that

$$\sum_{m=0}^{n} P(m) = 1$$

Hence from Theorem 1 we have

$$\sum_{m=0}^{n} (n+1-m)\binom{k-2+m}{k-2} = \binom{k+n}{k}. \tag{9}$$

Now

$$END = \sum_{m=0}^{n} mP(m)$$

Since, the $m=0$ term does not contribute to END, we have

$$\begin{aligned}
END &= \frac{1}{\binom{k+n}{k}} \sum_{m=1}^{n} (n+1-m)m\binom{k-2+m}{k-2} \\
&= \frac{1}{\binom{k+n}{k}} \sum_{m=1}^{n} (n+1-m)m\frac{(k-2+m)!}{(m)!(k-2)!} \\
&= \frac{1}{\binom{k+n}{k}} \sum_{m=1}^{n} (n+1-m)\frac{(k-2+m)!}{(m-1)!(k-2)!} \\
&\quad \text{Let } m' = m-1 \\
&= \frac{1}{\binom{k+n}{k}} \sum_{m'=0}^{n-1} (n-m')\frac{(k+m'-1)!}{(m'!)(k-2)!} \\
&= \frac{1}{\binom{k+n}{k}} \sum_{m'=0}^{n-1} (n-m')\frac{(k-1)(k+m'-1)!}{(L!)(k-1)!} \\
&= \frac{k-1}{\binom{k+n}{k}} \sum_{m'=0}^{n-1} (n-m')\binom{k+m'-1}{k-1} \\
&\quad \text{Let } k' = k+1 \text{ and } n' = n-1 \\
&= \frac{k-1}{\binom{k+n}{k}} \sum_{m'=0}^{n'} (n'+1-m')\binom{k'+m'-2}{k'-2} \\
&\quad \text{Using equation 9} \\
&= \frac{k-1}{\binom{k+n}{k}}\binom{k'+n'}{k'} \\
&= \frac{k-1}{\binom{k+n}{k}}\binom{k+n}{k+1} \\
&= \frac{n(k-1)}{(k+1)}
\end{aligned}$$

□

Table 1: END: Experimentation results

Malicious Hosts (k)	Honest Hosts (n)	Notifications Sent (t)	Deletions Detected	Reduction in Notifications
5	20	3	96 %	77 %
5	20	5	99 %	63 %
5	50	3	95 %	91 %
5	50	5	99 %	85 %
15	50	3	96 %	93 %
15	50	5	99 %	90 %
20	100	5	100 %	95 %
30	100	5	100 %	94 %

It is interesting to note that END is directly proportional to n, the number of honest hosts and, its value approaches n as the number of malicious hosts, k, increases.

5.2 Reducing notifications

As mentioned earlier, detecting collusions by requiring each host to send notification to the agent owner is expensive in terms of the number of notifications. Let each host in the itinerary send notifications with a probability λ. Then, the average number of notifications sent by hosts whose data are deleted is given by $\lambda \cdot END$. We need just one notification from such a host to detect a collusion. However, to take into account the variance in the actual number of deletions from the estimate END, especially for small k, we require that at least 't' messages are sent, where $t \geq 1$. In other words, $\lambda \cdot END = t$ or $\lambda = \frac{t}{END}$. In practice, we found that a value of $t = 5$ works well.

While it may seem intuitive that a host needs to send a notification with a greater probability as k increases, this is not the case. This is because as k increases more honest hosts are likely to have positions in between the malicious hosts and hence an individual honest host now may reduce the probability for its sending a notification.

If k is large, then λ becomes close to $\frac{t}{n}$. If we have no idea about the number of malicious hosts, then we can be conservative and assume that $k = 2$. That is each host will send notifications with a probability slightly greater than $\frac{3t}{n}$.

We use END to reduce the number of notifications as follows:

1. The agent owner calculates END using Theorem 2.
2. The agent owner now calculates the value $\lambda = \frac{t}{END}$.
3. Each host on receiving the agent does the following:
   (a) It adds its data to the *container* and updates the *checksum* as given in equation 5.
   (b) It generates a random number, $r, (0.0 \leq r \leq 1.0)$.
   (c) If $r \leq \lambda$, it sends a *notification* to the owner.
   (d) It forwards the mobile agent to the next host in the itinerary.
4. The agent owner while verifying the integrity of the *container*, as in equations 6, 7, also checks if there is data corresponding to each host that sent a *notification*.
5. For any *notification*, if the corresponding data is missing from the *container*, then the agent owner infers that the data was deleted by colluding malicious hosts.

This probabilistic notification leads to a tremendous decrease in the number of notifications sent, without compromising much on the ability to detect collusions, since a single notification from any one of the affected hosts is sufficient to determine that a deletion due to collusion has occurred.

5.3 Experimentation

We performed experiments to determine the efficacy of our solution. In each run, the number of honest hosts and the number of malicious hosts were varied. An itinerary was chosen by a random permutation of hosts. The agent owner and the hosts follow the scheme in section 5.2. A collusion is detected by the agent owner if it receives a message from a host whose data would have been deleted by the colluding hosts. The actual number of deletions detected due to notifications were noted. The simulations show that our method can assure a high degree of confidence in detecting deletions while significantly reducing the number of notifications. The results are summarized in Table 1.

6. CONCLUSIONS

In a related work [3], Karjoth et. al., propose protocols to detect modification/deletion of data collected by free roaming agents. While some of these protocols are able to detect *modification* to the data by colluding hosts, they do not tackle the problem of hosts colluding with each other to *delete* the data of intermediate hosts in the itinerary. Some other problems regarding the robustness of these protocols are reported in [6].

We have extended the *AppendOnlyContainer* mechanism proposed in [5] to not only detect modification/deletion of data by individual malicious hosts, but also to identify the malicious host.

We have also proposed further mechanisms to detect malicious hosts that collude with each other to delete the data of other hosts, for static as well as dynamic itineraries. We have shown that the notion of Expected Number of Deletions helps us to significantly reduce the number of notifications that must be sent while offering a reasonable degree of confidence in detection of deletions.

7. ACKNOWLEDGMENTS

We would like to thank Volker Roth for his helpful suggestions and the email discussions that we had on the subject.

This project was funded by the Ministry of Information Technology, India, under the project "Mobile agents for collaborative distributed applications", 2001-2002. We would like to thank all the members of the project review committee for their encouragement and support in this project.

8. REFERENCES

[1] D. M. Chess. Security Issues in Mobile Code Systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 1–14. Springer-Verlag, June 1998.

[2] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[3] G.Karjoth, N.Asokan, and G. Gulcu. Protecting the computation results of free-roaming agents. In *Proceedings of the second International workshop on Mobile agents (MA '98), LNCS 1477*, pages 195–207, Berlin Heidelberg, 1998. Springer Verlag.

[4] W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000. National Institute of Standards Technology.

[5] N. Karnik and A. Tripathi. Security in the ajanta mobile agent system. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1999.

[6] V.Roth. On the Robustness of some Cryptographic Protocols for Mobile Agent Protection. In *Proceedings of fifth International workshop on Mobile agents (MA 2001)*, Atlanta, USA, 2001. Springer Verlag.

[7] V.Roth. Programming Satan's agents. In *Ist International workshop on Secure Mobile Multi-Agent Systems*, Montreal, Canada, 2001.