An infrastructure for detecting and punishing malicious hosts using Mobile Agent Watermarking

Oscar Esparza Jose L. Muñoz Joan Tomàs-Buliart Miguel Soriano

Abstract

Mobile agents are software entities consisting of code, data and state that can migrate autonomously from host to host executing their code. In such scenario there are some security issues that must be considered. In particular, this article deals with the protection of mobile agents against manipulation attacks performed by the host, which is one of the main security issues to solve in mobile agent systems. This article introduces an infrastructure for Mobile Agent Watermarking (MAW). MAW is a lightweight approach that can efficiently detect manipulation attacks performed by potentially malicious hosts that might seek to subvert the normal agent operation. MAW is the first proposal in the literature that adapts software watermarks to verify the execution integrity of an agent. The second contribution of this article is a technique to punish a malicious host that performed a manipulation attack by using a Third Trusted Party (TTP) called Host Revocation Authority (HoRA). A proof-of-concept has also been developed and we present some performance evaluation results that demonstrate the usability of the proposed mechanisms.

Keywords: mobile agent security, malicious hosts, software watermarking, host revocation

1 Introduction

Mobile agents are software entities that consist of code, data and state, and that can migrate from host to host performing actions on behalf of a user. For instance, mobile agents can be used to support wireless terminals, which usually are resource-constrained. In fact, mobile agents are especially useful to perform functions automatically in almost all electronic services, like distributed computing, e-commerce or data mining. Figure 1 shows a possible mobile agent scenario.

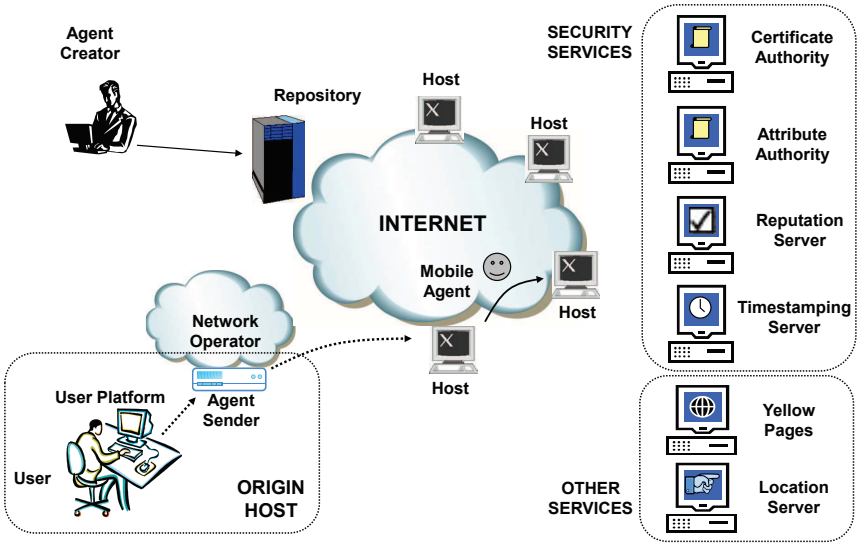


Figure 1: Mobile Agent Scenario

In this scenario, let us imagine that the user wants to arrange a meeting in which invitees are spread all over the country in different local offices. The date and place of the meeting may depend on several parameters such as room availability, room capacity, available commodities (projector, blackboard, etc) and, of course, on the invitees' agendas. Then, the user uses a mobile agent that travels all the hosts of the invitees. Finally, the mobile agent will decide the place and date of the meeting, and it will introduce a new entry in the agenda of each invitee. We consider that the user and the agent sender platform are the same entity. We call this entity "Origin Host" and we consider it trustworthy.

On the contrary, executing hosts cannot be considered trusted entities. For instance, these executing hosts might be competitors. Before sending the agent, the user can have information about the trustworthiness of the hosts, for instance obtained in previous transactions, or consulted to external reputation servers. Depending on the degree of trustworthiness of these hosts, the user can decide if protection techniques are needed or not, or it simply can avoid sending the agent to them. There are several reasons that can lead a host to become malicious and attack a mobile agent to obtain a certain profit. In the previous example, one of the invitees could try to manipulate the agent to impose the meeting to be in her local office because she is averse to travel, or simply to avoid spending money in a flight. Even, she can try to impose a date in which an adversary of another local office cannot assist to the meeting to damage the reputation of this adversary. In this article we assume a pessimistic view, considering the executing hosts as non trusted entities (they can try to attack an agent when they execute it), so protection

mechanisms are needed. Other principals can appear in this scenario, like entities that provide security services, or others that provide other services.

Two entities must be considered to study the security weaknesses of the scenario: the mobile agent and the executing host (or simply host). Protection is necessary when trustworthy relationships between these entities cannot be assured. Accordingly, these are the main cases that can be found [5, 27, 17]:

- **Host protection**: the system administrator's duty is protecting the hosts from attacks that try to exploit the weaknesses of the execution platform. The attacks performed by external entities (e.g. other hosts) are out of the scope of this article because they can appear irrespective of the use of mobile agents. However, there are new threats that must be taken into account. The hosts must be protected against the attacks that an agent can perform while it is executing its code. Continuing with the example of the arrangement of a meeting, a malicious mobile agent could try to delete entries that should not have permission to access in the agenda of a user. Fortunately, most of these attacks can be detected or avoided by using a proper access control and sandboxing techniques, which are techniques that control the execution environment. In addition, some other proposals try to determine if the agent's code is malicious before executing it [14, 26].

- **Communications security**: the agent can receive multiple attacks from any external entity while it is migrating from host to host. Typical attacks are eavesdropping or manipulation of data. Data eavesdropping attacks can be avoided by using encryption techniques, so any confidential information should be encrypted. Data manipulation attacks can be detected by using digital signature [19]. This includes any part of the mobile agent (the code, the itinerary, etc). In fact, before sending the agent the origin host signs the mobile agent to assure that it will not be modified during its journey by any host. So then, the hosts are sure that the agent's code has not been manipulated by any external entity (this includes any other previous host in the itinerary). In both cases, data eavesdropping and manipulation, the use of well-known cryptographic protocols is necessary.

- **Agent protection**: it is not so easy to protect the agents from the attacks of the execution platform. This is considered the most difficult security problem to solve in mobile agent systems for most of the authors [13, 18]. There are many reasons that can lead a host to attack the agents that is executing. The host can try to obtain an economical benefit or a favorable execution, or just it can try to damage the reputation of another principal. Following the previous example, a rival host could try to manipulate the agent to impose the meeting in another local office, but in a room without projector, so the

organizer of that local office will make a bad impression. It is difficult to prevent or detect eavesdropping and manipulation attacks performed by the host during the execution. The host has complete control of the execution and hence it can read or modify any part of the mobile agent: the code, the execution flow, the state, the itinerary, the communications, or even the results. This is the reason why there is not a published solution that protects the mobile agents completely from the attacks of an executing host. This kind of attacks is also known as the problem of the "*malicious hosts*".

In this article, we address the detection and punishment of manipulation attacks performed by malicious hosts. A malicious host performs a manipulation attack when, trying to achieve a certain purpose, modifies any part of the mobile agent to disrupt its proper execution. In the previous example, a possible manipulation attack could be that a malicious host modifies the execution of the agent to impose the date of the meeting without taking into account the agendas of the rest of invitees. As previously mentioned, we assume a pessimistic view, considering the execution of the agent in a non-trusted community of hosts. In case that the hosts can be considered trusted, no protection mechanism is needed against manipulation attacks. Our goal is to avoid manipulation attacks by dissuading hosts, because detection can lead to punishment. Taking into account these objectives, this article explains two mechanisms that work together to achieve an effective and usable protection mechanism against manipulation attacks.

In first place, we propose an infrastructure for Mobile Agent Watermarking (MAW). MAW is a lightweight approach to detect manipulation attacks. MAW is the first proposal in the literature that adapts software watermarks to verify the execution integrity. It must be clarified that the primary goal of MAW is not to develop a new software watermarking scheme but to use and adapt existing watermarking techniques. The novelty of our proposal is that we use and embed the watermarks in a different way and for a different purpose than traditional software watermarking systems. Indeed, different types of watermarking techniques can be used in our infrastructure, and these techniques might also be changed in the future according to advances in the watermark research area.

The second contribution of this article is a mechanism to punish the malicious hosts by using a Third Trusted Party (TTP), that is, a trusted entity for all the entities of the system. In our proposal, this TTP is called Host Revocation Authority (HoRA from here on). The HoRA stores in a database the information of those hosts that have been proven malicious in order to avoid new attacks from them. The punishment mechanism proposed is based on the idea of host revocation, which essentially consists in avoiding sending mobile agents to the hosts that previously attacked other agents. Both detection and punishment working together can achieve an effective protection mechanism against manipulation attacks.

Preliminary versions of the previous mechanisms were presented by the authors at conference papers [12] and [11]. In this article, we review and extend them, including guidelines about how to choose the watermarks to embed into the agent's code and also performance evaluation by means of a proof-of-concept implementation. We also think that the article introduces a new application of watermarking that may open a new research area. Finally, it is worth to mention that the performance results demonstrate the usability of the overall solution.

The rest of the article is organized as follows: Section 2 provides the reader a review of the required background; Section 3 explains how to detect manipulation attacks using MAW; Section 4 details the HoRA functionalities; Section 5 presents some performance evaluation by means of a proof-of-concept of the system; The conclusions of the article can be found in Section 6.

2 Background

2.1 Malicious Hosts

The attacks performed by malicious hosts are considered the most difficult to face within the mobile agent scenario regarding security [4]. There are two main attacks of this kind: (1) eavesdropping attacks, in which the host tries to extract information from the execution of the agent. The system must provide execution privacy to face these attacks, but this security service is difficult because eavesdropping attacks cannot be detected, only avoided; and (2) manipulation attacks, in which the executing host tries to modify the proper execution of the agent. Providing execution integrity is also a quite difficult security service because the executing hosts have complete control over the agent's execution.

The literature about countermeasures for malicious host attacks can be divided in two kinds of approaches: attack avoidance and attack detection approaches. Regarding attack avoidance approaches, they try to avoid attacks before they happen. Some authors introduced the idea of a tamper-proof hardware subsystem [32, 20] where agents can be executed in a secure way, but this forces each host to buy this hardware. Hohl presents obfuscation [15] as a mechanism to assure the execution integrity during a period of time, but this time depends on the capacity of analyzing the code of the malicious host. The use of encrypted programs [29] is proposed as the only way to give execution privacy and integrity to mobile code. The difficulty here is to find functions that can be executed in an encrypted way. Published attack avoidance techniques are difficult to implement or computationally expensive. For this reason, we consider attack detection techniques more promising because they are usually easier to implement. The objective of attack detection approaches is detecting manipulation attacks. In [21], the authors intro-

duce the idea of replication and voting, but this proposal can only be used as an attack detection approach if the hosts in the same stage are independent.

In [31], Vigna introduces the idea of the cryptographic traces, which are logs of the operations performed by the agent. The operations of the agent can be categorized in white statements, which modify the agent's state due to internal variable values; and black statements, which alter the agent's state due to external variables. These traces contain the changes performed to internal variables as a consequence of black statements. A re-execution of the agent can be performed with these traces. Instead of sending the traces, the hosts must store them to save network bandwidth. This is due to their size depends on the amount of input data, which can be huge. If the origin host suspects that a host modified the agent and wants to verify the execution, it asks for the traces and executes the agent again. If the new execution does not agree with the traces, the host is cheating. The approach not only detects manipulation attacks, but it also proves the malicious behavior of the host. However, this approach has two main drawbacks: (1) verification is only performed in case of suspicion, but the way in which a host becomes suspicious is not explained, and (2) for an indefinite period of time, each host must reserve enough capacity to the storage of traces of past transactions because the origin host can ask for them. These drawbacks can be relieved by controlling the agent's execution time in the hosts [10], but even with this complement, the use of traces might be still too expensive for all the entities involved.

2.2 Software Watermarking

Digital watermarking has been traditionally used to provide copyright protection for different kinds of digital objects. In the copyright protection scenario, a distributor embeds the watermark into the digital object, so its ownership can be proved later. Software watermarking is the term used when the digital object is a software application. Software watermarking has been used to detect software piracy (i.e. the illegal copying and resale of software applications). In addition, software watermarks have also been used in other scenarios such as tamperproofing or obfuscation [9]. In fact, in this article we use software watermarks for yet another purpose: detecting manipulation attacks in mobile agent systems.

According to [8] there are three parameters that essentially define the characteristics and security of a software watermarking scheme: (1) *the data rate* expresses the quantity of hidden data that can be embedded within the digital object; (2) *the stealth* expresses how imperceptible the embedded data is to an observer; and (3) *the resilience* expresses the hidden message's degree of immunity to attacks performed by an adversary. All watermarks exhibit a trade-off between these three parameters and the related cost.

2.2.1 Classification of Software Watermarks

Software watermarks are usually classified in two types:

- **Static watermarks.** The static watermark is embedded in the executable code of the program. The main drawback of static watermarks is that they can be detected even without running the program and thus, they are susceptible to attack by anyone of reasonable skill in software analysis. One of the most robust static watermarking techniques is presented in [30]. In that proposal, Venkatesan *et al*. treat the program as a control flow graph, in which a watermark graph is added to form the marked program.

- **Dynamic watermarks**. The watermark depends on conditions during the execution of the program. These conditions can be related with input data, user-interaction, a packet from network, a special file, the program state etc. This makes dynamic watermarks much more difficult to detect because in general the application must be run several times to detect the watermark. As dynamic watermarks are relatively new, there are still few published proposals of this kind [8, 24]. In the literature we can find three types of dynamic watermarks:

  - *Easter egg watermarks*, in which the application performs an action that is immediately perceptible for the user when a special input sequence is entered.
  - *Execution trace watermarks*, in which the watermark is embedded within the program trace (either instructions or addresses).
  - *Data structure watermarks*, in which the watermark is embedded within the state of the program (global, heap, stack data, etc.).

2.2.2 Threat Model for Copyright Protection

The objective of an illegal software redistributor is to make the watermark invalid without changing the behavior of the program. In this sense, three main attacks can be performed:

- **Subtraction attacks:** an attacker that knows the location of the watermark can try to delete it from the code, in the hope that the program after the extraction will be still useful.

- **Distortive attacks:** an attacker without knowledge about the location of the watermark can apply transformations that uniformly distort the code trying to make the watermark unrecognizable.

- **Additive attacks:** an attacker adds its own mark, in the hope that it will be impossible to detect that the real watermark precedes the new fake one.

Most distortive attacks are based on semantic preserving program transformations. These transformations only modify the program appearance. Some examples of classical semantic preserving program transformations are obfuscation, translation and optimization (compilation, decompilation or binary translation).

Regarding the strength of both types of software watermarks, in general, static watermarks are simpler than the dynamic ones, but also weaker against attacks [8, 9]. Static watermarks are usually easy to distort by using any semantic preserving program transformation. On the other hand, most published dynamic watermarking schemes are resilient to some of these transformations when applied individually, but not to combined attacks of some of them. For this reason, most watermarking schemes are designed to make it difficult to locate and change the watermark when semantic preserving program transformations are used.

However, in this article we present a mechanism that uses software watermarking techniques not to protect the copyright of programs, but to protect the execution of a mobile agent in an untrusted host, the Mobile Agent Watermarking (MAW) infrastructure. The objective of the agent's owner is to assure that the agent has been properly executed by embedding a software watermark in the agent's code. On her side, the objective of malicious hosts is also different, modifying the agent's execution to obtain a certain profit. As we will discuss later in Section 3.5, attacks based on semantic preserving program transformations against watermarked agents are useless for the attacker in this scenario. This is because these transformations only affect the code appearance, not code behavior. In this case, the modified code will be executed in the proper manner, so the execution integrity is assured.

In this article, we also present the implementation of our proposal MAW, which is based on a particular data structure watermark, the Collberg-Thomborson (CT) algorithm [8], which is also known as Dynamic Graph Watermarking. There is an implementation of this CT algorithm within the SandMark Project [7], which we have adapted to the mobile agent scenario. Next, we summarize how the CT algorithm works to better understand the implementation of our proposal (the concrete CT-based implementation of MAW is later presented in Section 3.6).

2.2.3 Dynamic Graph Watermarking

The CT algorithm (also called *Dynamic Graph Watermarking* [8]) is based on embedding watermarks within the topology of graphs built dynamically in memory during the execution of a program. The structure embedded is a *graph-watermark* (G). The *graph-watermark* contains in its topology a representation of a number

N, which is the product of two large primes p and q. A program called *recognizer* or R can retrieve this graph from memory. Then, N can be retrieved from G, and finally, the author can prove that she has embedded the corresponding graph into the code because she knows p and q. Graphs are a suitable mechanism to embed marks because as it is known [8], the analysis of large graphs involves significant complexity and requires an important computational effort. Furthermore, attacks based on semantic modifications of the source code are useless because they do not alter the execution of the marked code, and thus, the watermark can be recovered by means of the analysis of the memory during the execution.

Figure 2 illustrates the steps of this mechanism. The algorithm starts selecting two large primes (p and q) and calculating $N = p \times q$. Then, the algorithm continues as follows:

1. Embedding N into the topology of a given graph G.
2. Creating the code W which generates G.
3. Embedding W in the original code O to generate a O_0, which given an input I, the recognizer R is able to extract W (and hence N).
4. Using tamperproofing to avoid W being removed (generating O_1).
5. Using obfuscation to difficult analysis (generating O_2). In this case, the recognizer R and the watermark code W become R' and W' respectively because of obfuscation.
6. Extracting the recognizer R' and distributing the marked code O_3.
7. After distribution, an attacker can generate O_4 distorting the code O_3 to make the watermark invalid.
8. The author of the code can prove her authorship by applying the recognizer R' to O_4 using the special input I. This will generate the graph G in memory, so N can be found. As N is not a random number but it has been chosen deliberately as the product of two large primes, then the author demonstrates authorship just factoring N (publishing p and q).

One of the main difficulties of this algorithm is the embedding process of the mark within a graph. Collberg *et al*. mention in [8] some possible ways to do so. From these, we summarize here the Radix-k encoding, which is the one that we will use in our implementation of MAW. In Radix-k, the number used as watermark is embedded by means of a circular linked list. In fact, every number can be encoded as $n = \sum_{i=0}^{k-2} a_i k^i$. So, the base-$k$ digit is encoded by the length of the list and an extra pointer, which points to the first node. Every node encodes

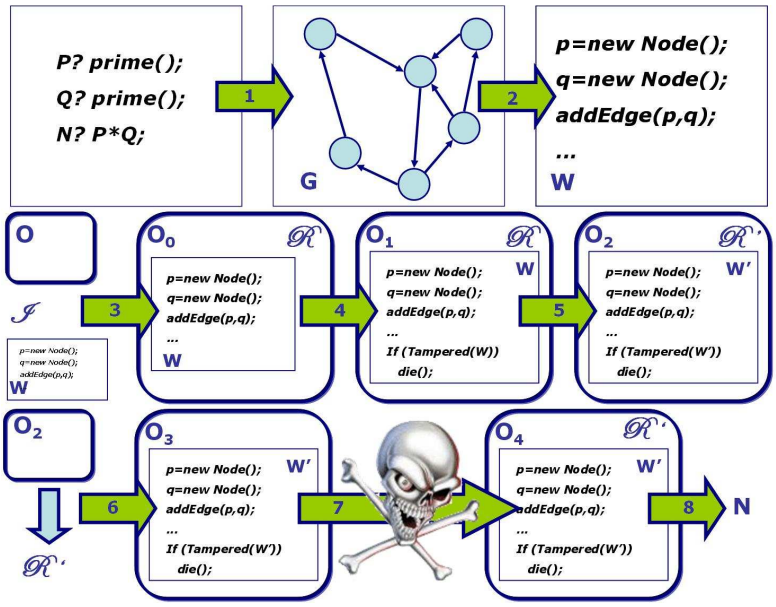


Figure 2: **Scheme of Dynamic Graph Watermarking:**

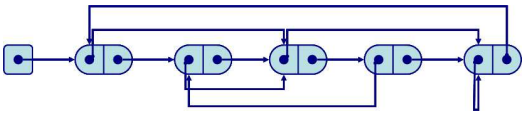


Figure 3: Example of embedding n=4453 by means of Radix-k

an a_i by pointers: if the pointer is null, then $a_i = 0$; if the pointer points itself, then $a_i = 1$; if it points the next element, then $a_i = 2$, and so on. Equation 1 and Figure 3 show an example of Radix-6 encoding, which codifies the number $n = 4453$ with these coefficients $a = \{a_0, a_1, a_2, a_3, a_4\} = \{1, 4, 3, 2, 3\}$.

$$n = 1 \times 6^0 + 4 \times 6^1 + 3 \times 6^2 + 2 \times 6^3 + 3 \times 6^4 = 4453 = 61 \times 73 \quad (1)$$

3 Mobile Agent Watermarking (MAW)

MAW is a lightweight approach to detect manipulation attacks. That is to say, with MAW one can verify whether an agent was or was not properly executed by a host. It must be stressed that our infrastructure is the first approach that adapts software watermarks to solve the problem of the malicious hosts.

The MAW infrastructure works as follows: the original agent is modified by introducing a dynamic software watermark. Watermarked agents generate output

data according to a set of rules. We name these rules "*integrity rules*". Integrity rules are secret, that is to say, they are only known by the origin host. The watermarked agent generates output data organized in what we call a "*data container*". The organization of the data container is performed according to the integrity rules. In this sense, we say that the watermark is transferred to the data container. For instance, let us illustrate the process with one very simple integrity rule. Let us assume that the agent generates an integer as a piece of output information. To introduce the integer in the container, it is multiplied by two if a certain global variable is even. The associated integrity rule is to check that this piece of data in the container is even if the global variable is even. In general, integrity rules are related with input data, internal values (heap, stack etc.), dummy data and data from external communications. Therefore, we can say that our infrastructure is based on dynamic software watermarking (since watermarks are created at runtime depending on the program's state).

Finally, the origin host receives the agent, which has been executed by all the hosts of the itinerary. Then, the origin host applies the set of integrity rules to all the data containers (there is one container for each host in the itinerary). These integrity rules are a set of logical properties that a container must fulfill (if it has not been tampered). These rules are responsible for demonstrating that the presence of the watermark is the result of deliberate actions. If a container does not fulfill the rules, this means that the corresponding watermark has been modified, and hence the corresponding host is malicious.

In summary, the process has three phases: (1) watermark embedding: modify the agent to embed the watermark generation code (see Section 3.1); watermark transference: create the container during the agent's execution to transfer the watermark and hide the results (see Section 3.2); and detecting manipulation: watermark verification using integrity rules (see Section 3.3).

3.1 Watermark Embedding

Current software watermarking techniques must be adapted to our scenario because they were not originally designed for creating execution integrity proofs (containers). As explained in Section 2.2, there are two main kinds of software watermarking techniques: static and dynamic watermarks [8]. The nature of static watermarks makes it impracticable to transfer the embedded watermark to the container. As static watermarks are embedded in the executable file itself (i.e. they are not related to the program state), they cannot be used to build our dynamic containers. Hence, we need to use a dynamic watermark approach to generate the containers at runtime taking into account the program state. Among the existing dynamic software watermarking approaches, the appropriate one for MAW is the "data structure watermark" because it is the one that takes into account the pro-

gram state to generate the watermark (see Section 2.2 and references [8, 6, 25]). In particular, in our implementation, we have used the Collberg-Thomborson (CT) algorithm [8], which was summarized in Section 2.2.3.

Regarding the size of the MAW watermark, it is important because it determines the probability of detecting manipulation attacks. In our scheme, the size of the marked code is determined before sending the agent. This size is limited and it is the same for all the hosts in the agent's itinerary. Moreover, the container (output data) generated by each execution has also a limited size. The maximum size of containers is not arbitrary, it can be decided by the programmer to control the accuracy to detect manipulation attacks. If we want to improve detection ability and prevent an attacker from modifying the agent's code without being detected, we must increase the size of containers. Obviously, increasing the size of the watermark is also more costly in terms of transmission resources consumption. This is because the size of the agent is increased since on one hand, the marked code is bigger than the original one, and on the other hand, the containers are also bigger because they contain more redundancy. This last effect is even more significant since the agent carries a container for each execution in a host. This implies that the agent's size grows as it traverses more hosts, and therefore this size depends on the length of the agent's itinerary.

3.2 Watermark Transference

To detect manipulation attacks, the mobile agent must create at runtime the proofs, and send them back to the origin host to assure the execution correctness. In MAW, these proofs are stored in a logically-structured data "*container*" that is created in each host during execution. The container is created using dummy data, input data, internal values (heap, stack etc.) and data from external communications. The agent can diffuse and confuse all this information into the container to hide the actually desired execution results. Diffusing values means repeating these values into several different places, and confusing values means modifying these values to different ones, for example by adding constant values.

Obviously, all these data are not organized into the container at random. The way this information is incorporated into the container is essential to extract the watermark when the agent returns to the origin host. As we said previously, the transferred watermark must be reliably located and extracted from the container and, it must let us demonstrate that its presence into the container is the result of deliberate actions. In short, each executing host creates a container, which is the digital cover where the agent transfers the watermark. Then, each host must digitally sign its container. When the agent finishes traveling its itinerary, it returns to the origin host. All containers arrive at the origin host together with the mobile agent. Next, the origin host uses them to verify the execution integrity and to

extract the desired execution results of each host.

3.3 Detecting Manipulations

The origin host must verify the execution integrity when the agent comes back with all the containers. To do so, the origin host uses its secret set of integrity rules related to the previously-embedded watermark. These integrity rules are a set of logical properties that a container must fulfill to demonstrate that it has not been tampered. They are also responsible for demonstrating that the presence of the watermark is the result of deliberate actions. These actions are inferred from the modifications performed over the original agent's code during the watermark embedding process. If a container does not fulfill the integrity rules, this means that the watermark has been modified, and the corresponding host is malicious. A tampered container can be used as a proof of the malicious behavior of a host. The host cannot repudiate this situation since it digitally signed the container.

It is worth to mention that the embedded watermark is the same for all the hosts, that is, all hosts execute the same marked code. In the same sense, the integrity rules are the same for all the hosts, because they are inferred directly from the agent's code. This means that the origin host uses the same integrity rules to demonstrate the presence of the watermark into the containers. However, this does not mean that all the containers have the same data. Each container is different because it depends on the execution in each host, and hence the data used to fill in the container is different (input data, internal data, data from communications, dummy data, etc.). This could lead us to think that our proposal uses fingerprinting instead of watermarking, because the data structure is different for each host (container). However, we consider that our approach uses watermarking because the embedded mark is the same for all the hosts despite the representation of this mark is different for each container.

Finally, our infrastructure also allows an origin host to prove, in front of an external third party, that a certain host of the itinerary performed a modification attack over the agent. However, the integrity rules cannot be treated as a proof directly. Instead, the origin host must send them together with the agent's code and the signed container of the accused host to the third party. Then, the third party executes the agent several times with random input data. As any honest execution of the agent (independently of the input data) will generate valid containers, the new containers created during these random executions should fulfill the integrity rules. This procedure assures that the integrity rules are valid. Then, the external entity can verify whether the host being accused is in fact malicious by applying the integrity rules to its container.

3.4 Advantages and Drawbacks of MAW

MAW is a lightweight attack detection approach if it is compared to the most widely known proposal of the cryptographic traces [31]. These are some of its advantages:

- **Size of the proofs:** in MAW, the size of the proofs to check the execution integrity is limited. The maximum size of the containers is determined by the programmer to control the accuracy to detect manipulation attacks. The containers can be little enough to let the agent carry them. In the cryptographic traces approach, the size of the traces depends on the amount of input data of the mobile agent, which can be quite big.

- **Proof storage:** in MAW, the executing hosts do not need to store any kind of proof. In the cryptographic traces approach, the hosts must store the traces for an indefinite period of time.

- **Hosts to verify:** in MAW, the origin host can verify the execution integrity of all the hosts of the itinerary. In the cryptographic traces approach the verification is performed in case of suspicion.

- **Verification tasks:** in MAW, the origin host has to apply the integrity rules to the containers to verify the execution integrity. In the cryptographic traces approach, the origin host must ask for the traces to the suspicious host and execute the agent again.

MAW has also some drawbacks, which affect mainly performance:

- **Watermark embedding:** the origin host must embed the watermark into the agent's code by using software watermarking techniques and must infer the integrity rules.

- **Code size:** there is an increase in the code size. Embedding a watermark means that some overhead is added to the original code. This enlargement will depend on the embedded watermark and therefore, creating, storing and sending marked agents consume more resources.

- **Execution time:** the execution of marked agents consumes more CPU.

- **Mobile agent size:** the mobile agent in MAW must carry the containers. This implies an additional load. This load grows up each time the mobile agent visits a host. The maximum size is reached when the agent returns to the origin host.

3.5 Design of the Watermarks for MAW

This Section discusses the motivations of malicious hosts, the attacks that can perform, and also the properties and requirements that the software watermarks should have to implement the MAW infrastructure.

3.5.1 Threat Model for MAW

As we mention in Section 2.2.2, the objective of an attacker in the copyright protection scenario is to make the watermark of a program invalid to illegally redistribute this program later. To do so, the main attacks against software watermarks are subtraction, addition and distortion. On the other hand, the motivations of an attacker in the mobile agent scenario are different. A malicious host may have several reasons to attack a mobile agent. For instance, it can attack the mobile agent to obtain some benefits from the execution, to damage the reputation of another host, or just for fun. There are several kinds of attacks in this scenario (denial of service, eavesdropping, impersonation, etc). However, we will focus on manipulation attacks because MAW has been designed to detect this particular kind of attacks. Just remind that manipulation attacks are those in which a malicious host tries to manipulate the proper execution of the agent to achieve a certain purpose. So then, the objective of an attacker will be not to make the watermark invalid, but to manipulate the execution without altering the transferred watermark, because any change in the transferred watermark will cause the detection of the attack.

The malicious host may try to manipulate the agent's code to obtain a certain benefit. However, all the attacks that are used in the copyright protection scenario to manipulate code are totally useless to attack a mobile agent protected with the MAW infrastructure. Distortive attacks, which are usually based on semantic preserving program transformations (translation, optimization, obfuscation, etc.), are useless to attack MAW because these transformations only affect the code appearance, and not the code behavior. Hence, the modified code will be executed in the proper manner (which is precisely our objective, assuring the execution integrity). On the other hand, if a malicious host tries to remove the embedded watermark or to add a new one to the agent's code, the changes in the transferred watermark produced by these attacks will reveal that the agent has been modified.

The host can also try to attack containers. However, a host cannot manipulate the containers of previous executing hosts because they are signed by their creators. Thus, a malicious host can only try to manipulate its own container. In this case, the objective of the attacker is manipulating the container to obtain a certain profit, but without altering the transferred watermark. However, this will be hard to achieve thanks to MAW because the host does not know which parts of the container are part of the watermark (this would be equivalent to know the integrity

rules, which are secret). Therefore, before changing any part of the container the host should infer where the watermark is.

To infer where the watermark is, a malicious host can also try to analyze the inputs and outputs to extract information from the mobile agent. Unfortunately, it is unfeasible to detect or prevent that a host changes its own input data, which are located in its internal database. In fact, this should be considered an eavesdropping attack because the host does not alter the proper execution of the agent. As a conclusion, MAW cannot detect this because it is not a manipulation attack. However, MAW can avoid the attacker to extract information from the execution. Let us suppose that a malicious host introduces fake input data and executes the mobile agent to analyze the generated container. Despite this can be done many times obtaining different containers, this does not mean that the malicious host can generate a container at its discretion (containers must fulfill the integrity rules if the agent's execution has not been modified). For this reason, a malicious host performing different executions cannot infer the integrity rules by comparing these containers because any change in the input data will cause that most data within the container will also change. In the same sense, colluding hosts that share their containers cannot infer where is the watermark. Even if a malicious host is successful obtaining some piece of information about how the containers are constructed, it would be unfeasible to construct a valid container that achieves the purposes of the attacker. This is due to the watermark being large and distributed within the whole container, and also because the attacker doesn't know all the integrity rules.

3.5.2 Watermark Properties

These are the most important properties of the watermarks to be embedded into the mobile agent: (1) the *stealth*. This is the most important property of the watermark, because a malicious host without knowledge about where the watermark is can only try random changes, which affect the transferred watermark; (2) the *data rate* is also quite important because it improves the security of the watermark. A bigger watermark makes manipulating the container without altering the transferred watermark more difficult. However, this affects adversely the cost of the watermark, especially in terms of transmission resources as containers are sent back to the origin host; finally, (3) the *resilience* is not as important as the previous properties, because the use of semantic preserving transformations does not affect the code behavior. As a consequence, watermarks with little resilience can be used in our scenario. So, we do not require maximizing all the properties. This allows us to use simpler and less costly watermarks.

3.6 Implementation of MAW using the CT Algorithm

This section describes the main guidelines about how we have implemented the MAW infrastructure. In particular, our implementation is based on an adaptation of the Collberg-Thomborson (CT) algorithm [8] which is available within the SandMark Project [7]. We would like also to point out that other different software watermarking algorithms could also be used to detect manipulation attacks. Obviously, the different peculiarities of each algorithm must be taken into account.

As previously elaborated in Section 2.2.3, the CT algorithm dynamically builds a graph in memory when the program is fed with a special input. The recognizer program is able to find this graph in memory, and to extract from this graph (for instance using the Radix-k encoding) a number N, which is the product of two large primes. As it is computationally unfeasible to factor a number which is the product of two large prime numbers, the creator demonstrates authorship by simply publishing the two factors. In our implementation of MAW, all the executions build a graph in memory, independently on the input data that we use to feed the agent. So, the agent just needs to transfer the graph which is placed in memory to the container. Our recognizer uses the container (instead of memory) to find the product of primes N and thus, to assure execution correctness. The recognizer should be also considered part of the integrity rules (in fact, the integrity rules are more general as they also describe some more relationships among data within the container).

For the sake of simplicity, we illustrate our implementation by means of an example. Let us suppose that we are executing the agent in the host n. The agent is fed with some input data that come from the previous execution host $n-1$. Let us suppose that we have six values of this kind $s^n = (s_0^n, \cdots, s_5^n)$, which are the initial state of the agent in this host. After the execution, we will obtain some output data. Let us suppose that we have five of these values $o^n = (o_0^n, \cdots, o_4^n)$, which should be included within the container together with the transferred watermark.

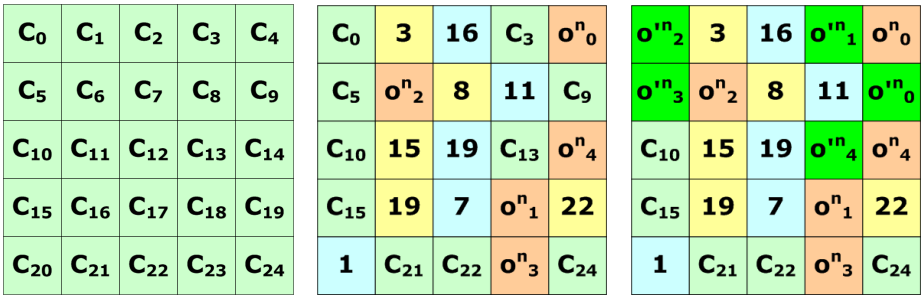
Then, we have to follow the following steps to construct the container:

1. The agent must calculate a binary initial sequence IS that will be used to establish the starting position of the watermark, and also to obscure the container. The IS should reflect that a particular execution has been performed in a certain host, that depends on a initial state, and that it is time dependent. Following the example, we calculate IS as the hash of the concatenation of the identifier of host n (ID_n), a subset of the initial state (some values in clear and some hashed), and a timestamp TS_n:

$$IS = hash(ID_n||s_2^n||s_3^n||hash(s_5^n)||TS_n) \quad (2)$$

This time stamp TS_n should also be sent to the origin host together with the container to make possible to re-calculate IS from these values.

2. After that, the agent fills out the cells of the container with random values. Figure 4(a) shows the container at that moment. c_j represents the random data stored in the position j of the container.



(a) Container with random data

(b) Container after embedding the output data

(c) Container before XOR

Figure 4: Container generation process.

3. In this step, the agent starts transferring the watermark to the container. The first thing is locating the initial cell, p_n, of the graph into the container. In our example, the agent calculates p_n by performing a modulus operation over the initial sequence IS taking as base the number of cells of the container (25 in this example):

$$p_n = IS \, mod \, 25 = 7. \quad (3)$$

In this case, the initial sequence IS is the extra pointer which provides us the position within the container of the first node within the circular linked graph (in the example, cell 7).

4. Next, we start transferring the rest of the graph from memory to the container. In this example, we use the same watermark of Section 2.2.3, i.e. we use the Equation (1), which codifies the number $N = 4453$ with the following Radix-6 coefficients:

$$a = \{a_0, a_1, a_2, a_3, a_4\} = \{1, 4, 3, 2, 3\} \quad (4)$$

We denote the graph with the following set of tuples:

$$w^n =< \{x_0^n, y_0^n\}, \{x_1^n, y_1^n\}, \{x_2^n, y_2^n\}, \{x_3^n, y_3^n\}, \{x_4^n, y_4^n\} > \quad (5)$$

As in Figure 3, each node is formed by two elements (we map each of these elements in one cell, so we will need two consecutive cells for each graph node). The first cell is used to store the x_i^n value in base k, and it will be used to obtain the a_i Radix-6 coefficients. In our example, as $IS = 7$, cell 7 is the first cell of the first node of the graph, and hence it contains x_0^n. The second cell stores y_i^n, which is the index of the cell that corresponds to the next node of the graph. Therefore, cell 8 contains $y_0^n = 11$, which is a pointer to the second node of the graph (located in cells 11 and 12). Except the first node (that depends on the IS), the agent can put the rest of nodes randomly in any place of the container, because we can reconstruct the complete graph using the pointers. In addition, the last node points to the first one (it is a circular linked graph). Finally, the agent does not directly store the values into the container, but it calculates them as the subtraction of the value of the cell and the index of the cell $a_i = x_j^n - j$ (being j the index of this cell). For instance, $a_0 = 8 - 7 = 1$, $a_1 = 15 - 11 = 4$, $a_2 = 22 - 19 = 3$, etc.

5. After the graph embedding, the agent must store the result of the execution $o^n = (o_0^n, \cdots, o_4^n)$. In our example, the agent chooses the positions to store these output values into the container using, once again, the value p_n (7, in this example). Basically, this value indicates the number of empty cells between two different output values. The last node of the graph (in the example, cell 18) is used as starting point. Thus, cell 4 stores the first output value o_0^n because it is 8th empty cell starting from cell 18. Figure 4(b) shows the container after transferring these values.

6. The next step is to generate an additional vector that will enhance the diffusion of the output data by storing them in some extra positions. In this example, we use a vector of 5 positions $o'^n = (o'^n_0, \cdots, o'^n_4)$, in which $o'^n_i = (o_i^n)^j \ mod \ p_n$. These values will be stored using the same rule than the o_i^n values (7 empty cells between different values), and starting from the last value added to the container (in this case o_4^n). Figure 4(c) shows the container after embedding the o'^n values. These relationships will also be part of the integrity rules.

7. Finally, all the container is XOR with IS to obscure all these data. Thus, only the entities that know IS can know the real contents of the container.

To detect manipulations, the origin host should have the container, the initial state values $s^n = (s_0^n, \cdots, s_5^n)$ (sent by the previous host and acknowledged by

host n), and the timestamp TS_n. Having all these data, the origin host can calculate IS. With that value, the origin host XOR the container using IS to recover the data within the container. Once the IS value is known, we can also know the initial cell of the watermark. If host n has been honest, then $IS = 7$, and applying the recognizer to the container we can obtain that the Radix-6 values are $a = \{1, 4, 3, 2, 3\}$, so $N = 4453$. Just emphasize that N is not a random number but the product of two primes, and that the only entity capable of factoring it is the origin host ($N = 61 \times 73$). The origin host also verifies that the output values o^n and $o'^{\,n}$ are properly located into the container and that the relationships among them are correct. If so, we can consider that host n is honest, and we can extract the results from the output values.

The security of this implementation lies on the impossibility of knowing which kind of data is storing each cell (watermark, output data, random data, etc). The executing host does not know how the initial sequence IS has been calculated, nor how the container has been constructed, nor the relationships among cells. As it has been shown, data within the container are very dynamic, and they change as it changes the executing host, the time, the agent's initial state or any other type of input data. The previous example illustrates the main steps that performs our MAW implementation. However, the real implementation performs some extra steps that have been deliberately omitted for the sake of clarity of the previous explanation. To achieve a practical level of robustness indeed, additional steps must be performed.

8. We have some order in the way to construct the container (first introduce the watermark, after that introduce the results, and at the end XOR the container). In a real scenario, all these steps should be mixed, so the attacker cannot infer how the container has been constructed. In addition, some other relationships among the output values could be added to enhance diffusion and robustness, $o''^{\,n}$, $o'''^{\,n}$ (e.g. using arithmetical operations among them).

9. We have constructed the watermark using a Radix-k encoding that needs pointers, so all the cells that compose the graph has values in the range $[0 - 25]$. This can help an attacker to locate the watermark. In a real scenario these values should be obscured for instance by using any kind of arithmetical operation.

10. A malicious host that provides the same initial state, the same host identifier, and the same timestamp to the agent will always obtain the same IS. Thus, the position of the watermark will always be the same. Obviously, for a real case, IS should change depending on other different parameters, so the attacker cannot infer where is the starting point of the watermark.

4 Punishing Attacks with the HoRA

This section introduces a punishment mechanism based on a new entity: the Host Revocation Authority (HoRA). The HoRA must be considered a Trusted Third Party (TTP) in the mobile agent system. In this sense, the HoRA must be considered a TTP in the mobile agent system like the Certification Authority (CA) is considered in the Public Key Infrastructure (PKI). In our opinion, attack detection approaches should be accompanied with some punishment policies. Little attention has been paid to punishment mechanisms in mobile agent systems. In fact, our proposal is the only punishment system that can be found in the literature. The HoRA uses a punishment mechanism based on host revocation. The aim of host revocation is to distinguish the malicious hosts from the honest ones. For this purpose, the HoRA stores a database with all the information related to past attacks. The main job of the HoRA is providing this information to the origin hosts to avoid new attacks from malicious hosts. In this article, we summarize the tasks that the HoRA has to carry out (status checking and host revocation), we present our implementation and, we present some performance results to evaluate the cost of the HoRA when using MAW as a detection mechanism.

4.1 Status Checking

Before sending the agent, the origin host must consult the revocation information in order to delete malicious hosts from the agent's itinerary. Assuming that the HoRA works in a similar way as the Certification Authority regarding certificate revocation, there are two possible ways of consulting the status of the hosts: online or offline. The decision of which of these policies must be used depends on multiple factors, like the available transmission resources, the number of origin hosts that may launch requests, or the computational capacity of the entities.

4.1.1 Offline Status Checking

In offline status checking, we consider that an origin host may lose the connectivity to the HoRA. If this happens, the origin host will not have any revocation information available. The idea behind the offline system is to make accessible the revocation information available in a given moment using a black list: the Host Revocation List (HRL). An HRL is a list, which is signed by the HoRA and, that contains all the identifiers of the hosts that have been revoked. The origin hosts can download the HRL and store it for some time. Then, the HRL can be used to remove revoked hosts from itineraries before sending agents. To take into account new malicious hosts, the origin hosts have to update the HRL periodically. In this sense, the HRL works in a similar way as the traditional Certificate Revocation

List (CRL) in the PKI [16]. The origin hosts can download the HRL directly from the HoRA, but this may cause a bottleneck in the system. To solve this problem the HoRA can put the HRL in repositories[1]. The repositories must also update the HRL periodically. The offline status checking mechanism does not avoid attacks completely. A host that is detected as malicious can attack agents until it is introduced in the HRL and the origin hosts update their lists. The less time between updates of the HRL, the less attacks can be performed by the new malicious hosts. However, frequent updates affect adversely the network bandwidth.

4.1.2 Online Status Checking

In the online status checking policy, origin hosts request revocation information directly from the HoRA. To do so, the origin hosts use the Online Host Status Protocol (OHSP). When a request arrives, the HoRA consults its internal database and sends a signed response pointing out the state of each host. This mechanism works in a similar way as the Online Certificate Status Protocol (OCSP) used in the PKI [23]. There are several reasons that can lead an origin host to use the online mechanism. For instance, origin hosts that send agents sporadically do not need to store and update the HRL periodically. Furthermore, the risk of suffering attacks from malicious hosts is minimized since the status is checked online which allows immediate detection and rejection of malicious hosts from itineraries. However, with the online policy, the HoRA may become a bottleneck in the system because it receives requests from all the origin hosts, and it must answer each request with a digitally signed response which is computationally expensive. For this reason, the HoRA can also delegate online checking to authorized entities called responders[2].

4.2 Host Revocation

The second task of the HoRA is managing the revocation information. As the revoked hosts are not removed from the database, this task consists mainly in adding new hosts. If the origin host has detected malicious hosts using MAW, it starts a protocol to revoke them. The objective of any revocation protocol is delivering in a reliable way all the proofs to the HoRA in order to demonstrate that an executing host is malicious. A proof is a piece of evidence that a TTP can use to verify that an attack was performed by a malicious host. In the case of MAW, the proofs are the containers, and the way to detect manipulation attacks are the set of integrity rules. Hence, the HoRA can only revoke a host in case there are proofs of its malicious behavior, that is, the HoRA needs evidences that demonstrate

[1] A repository is a non-trusted location in the network where it is possible to store contents to make them available to download.

[2] A responder is a trusted location in the network that can send signed responses.

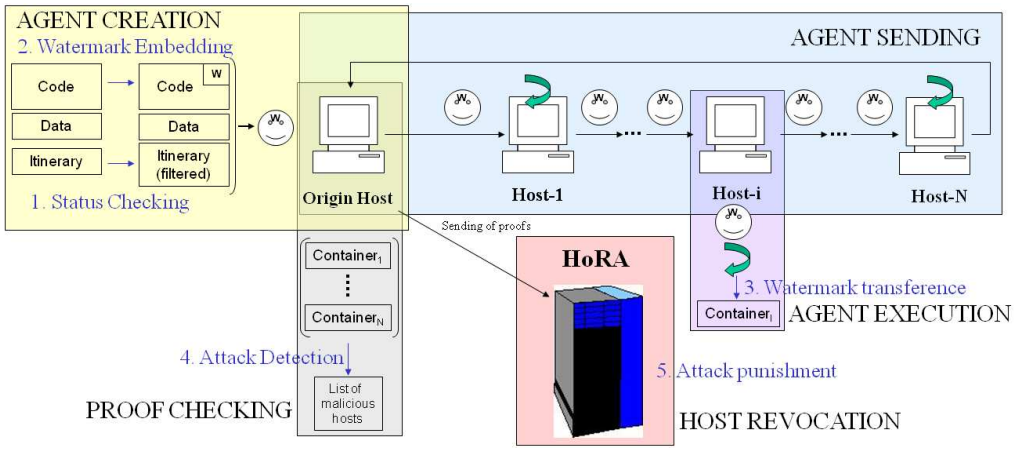


Figure 5: Working of our Proposal

unmistakably that this host was malicious. In this sense, the containers can be used to detect manipulation attacks because they have been signed by the hosts, so a malicious host cannot repudiate that it generated a certain container. The HoRA must verify that the set of integrity rules matches the agent's code. This can be done by executing the agent several times with random input data (as explained in Section 3.3). A revocation protocol for MAW can be found in [12].

4.3 Summarizing the Overall Process

We can divide the agent's lifetime in four phases (see Figure 5):

- **Agent Creation:** in this phase the origin host prepares the agent before sending it. This includes performing the status checking to filter the malicious hosts from the agent's itinerary, and also embedding the watermark into the agent's code.

- **Agent Sending:** in this phase the origin host sends the mobile agent to perform its tasks. The agent will migrate from host to host executing its code and performing the actions that the user has programmed (for instance, arranging a meeting). During the agent's execution in each host, the agent must create and store the containers, which are the proofs that will assure the execution integrity. The embedded watermark has been transferred to these containers during the execution. Hence, all the containers (one for each host) will return with the mobile agent to the origin host.

- **Proof Checking:** in this phase the origin host looks for malicious hosts. When the agent returns to the origin host, it extracts the containers of all

the hosts and it verifies the signatures to detect possible errors in communications. These errors must not be considered manipulation attacks because they could be produced by the communication channel. If the containers do not have communication errors, the origin host applies the integrity rules to them in order to verify the correctness of the transferred watermarks. If a container does not fulfill the rules, this host is malicious and hence the origin host can start a revocation protocol.

- **Host Revocation:** in this phase the origin host sends the proofs of the execution integrity to the HoRA using a revocation protocol. The HoRA verifies the validity of these proofs, that is, it verifies that the signature of the container is valid and that the integrity rules correspond to the agent's code by executing the agent again with random input data. If these proofs are valid, the HoRA revokes the malicious host, so this host will not receive agents any more.

5 Proof-of-concept and Performance Evaluation

In this section, we present some performance results of the prototype that we have developed to show the usability of our agent watermarking mechanism. We also present some performance results about a proof-of-concept of the HoRA.

5.1 Mobile Agent Watermarking

We have used the Aglet Software Development Kit 2.0.1 (ASDK) [3] as mobile agent platform to implement the agent watermarking prototype for MAW. ASDK provides an API to create aglets (AGent-appLETS), which are Java objects that can be used as mobile agents. ASDK provides the Tahiti server, where the aglets can be sent, received and executed. Tahiti is supported by a Java Virtual Machine (JVM) included in the Java Development Kit (JDK) 1.3.1 [2]. The Java Cryptography Extension (JCE 3.01) [1] includes some additional cryptographic Java libraries that have been used to protect the mobile agent. Finally, it is worth to mention that the results in the following sections have been taken in a laboratory equipped with four computers (one working as the HoRA, two as executing hosts, and one to send mobile agents). The four computers are Pentium IV at 2.4 GHz, 512 MB of RAM and Linux SuSE 9.0 as Operating System.

We have tested our watermarking mechanism using some sample agents available within ASDK, and the results show that the average size of the marked code is increased in an 11%, and the average execution time is increased in a 19%. These results are dependent on the environment that we have used to make the

test and on the watermark used (the CT algorithm in our case). However, we think that our results are a good estimation of the cost of MAW because they are coherent with the ones published by previous studies about software watermarks in [28]. In accordance to the previous values, we can conclude that our watermarking mechanism does not introduce too much overhead in the system.

5.2 HoRA

A preliminary proof-of-concept for the HoRA has also been implemented. The implementation has two software entities: the *revoker* and the *manager*. The *revoker* is devoted to host revocation. In other words, the *revoker* performs re-executions of the agent to verify the execution correctness. Thus, the *revoker* can be considered the element that decides if a host is revoked or not. On the other hand, the *manager* is charge of managing the interface with users. This interface must allow users to send their revocation requests. In addition, the *manager* must also implement at least one method (HRL or OHSP) for distributing revocation information. Remember that the distribution revocation mechanism allows a origin user to check whether a host is currently revoked or not. The goal of dividing the implementation of the HoRA in two parts is that this division allows us to reuse with minimal changes any generic revocation *manager*. Generally speaking, a revocation *manager* keeps updated the currently revoked items in a central revocation database and makes publicly available the information of this database to end users with some status checking protocol. In particular, to implement the *manager* of the HoRA, we have reused a previously developed revocation *manager* for a PKI revocation scenario called CERVANTES (CERtificate VAlidatioN TESt-bed)[3]. We present results about the two main time-and-bandwidth consuming tasks for the HoRA, which are host revocation and status checking.

5.3 Host Revocation

Table 1 shows a summary of results about the time spent for the origin host and the HoRA to perform a host revocation. An average between 20 executions has been calculated to every result presented. The agent's execution time has also been added in order to compare it with the previous times. We present the results for two revocation protocols, one that provides data privacy, and another that does not provide it. Both protocols provide integrity and authentication of all the exchanged data (for further information see the revocation protocol for MAW presented in [12]).

[3] http://sourceforge.net/projects/cervantes

Times	**Revocation protocol (without privacy)**	**Revocation protocol (with privacy)**
Agent's execution time	120 ms	147 ms
Revocation time at the origin host	27 ms	51 ms
Revocation time at the HoRA	153 ms	181 ms

Table 1: Agent Execution Time versus Revocation Times

From these results we can extract that starting a revocation request does not suppose a great cost for the origin host, even when privacy is required. In fact, the origin host only needs to generate the revocation request, signing and encrypting (if privacy is required) all the proofs, and finally sending them to the HoRA. However, the cost that supposes to process a revocation process for the HoRA is a little more expensive. In case of using the protocol without privacy, the HoRA needs a 27% more CPU time to process a revocation request than a host to execute the agent, and a 23% more in case of the protocol with privacy. This is due to the fact that the HoRA not only needs to execute the agent (at least once) to demonstrate that the integrity rules correspond to the agent's code, but it must also perform extra tasks like verifying the signature of the origin host or decrypting the results.

5.4 Status Checking

To evaluate the resources consumed by status checking we consider that the computer that sends mobile agents simulates the behavior of N independent origin hosts. These origin hosts always get the revocation information using the *pull* mode, that is, the origin hosts (acting as clients) send the revocation requests, and the HoRA (acting as server) receives them and sends responses. The elapsed time between a request and the next one sent by an origin hosts has been generated by using an exponential inter-arrival probability density function. We assume that each origin host has a certificate. We also assume that there is an average of 10% revoked hosts. This figure is also used in other revocation scenarios such as PKI and credit cards. When using HRL, each origin host has its own HRL stored in cache during its validity period, and it must update it when expires.

Figure 6 shows the bandwidth utilization in case of using HRL. The tested scenario has $N = 10000$ origin hosts. The HRL validity period is $VP = 6$ hours and the status request rate per origin host and hour $r = 2$ requests/hour, that is, each origin host sends an average of two agents per hour. Notice that the

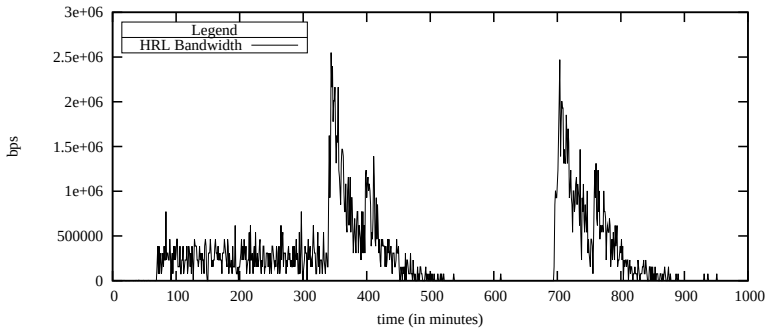


Figure 6: HRL bandwidth utilization

bandwidth utilization has peaks. This is because all the clients have the same HRL copy in their cache so the HRL expires at the same time for everybody. This is why the bandwidth peaks are localized around the expiration dates (every 6 hours in this scenario). This is a well-known drawback that can be mitigated with a mechanism called overissuation. Overissuation consists in allowing multiple HRLs to have overlapping validity periods. Put in another words, overissuing means issuing more than just one HRL during a validity period. The result is that the HRLs in the users' caches will expire at different times and thus requests to the HoRA for new HRLs will be more spread out.

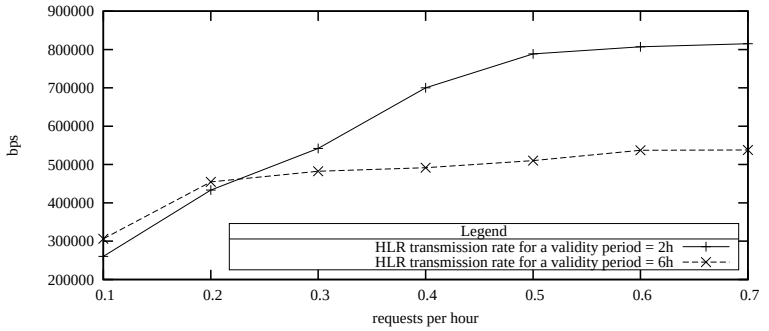
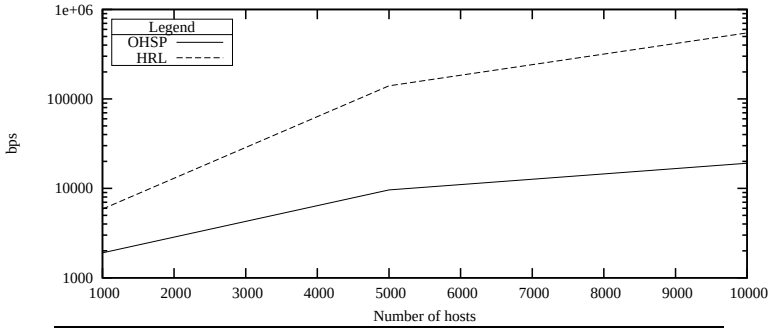


Figure 7: Effect of the HRL validity period in the bandwidth utilization

Figure 7 shows how decreasing the HRL validity period affects bandwidth. The tested scenario has also $N = 10000$ origin hosts. Validity periods of $VP = 2$ hours and $VP = 6$ hours are compared with regards the request rate r. There is a trade-off between risk and network bandwidth usage: in general smaller validity periods imply more bandwidth utilization. The bandwidth utilization tends to a threshold when increasing the status requests. The existence of this threshold is due to the fact that when r grows, users start benefiting from the cached HRLs. Then, the request rate towards the HoRA reaches a threshold, and thus

the bandwidth also reaches a threshold. Observe that the bandwidth threshold is reached around $r/VP \approx 1$. This is because when these magnitudes are similar, there is a high probability of having a cached HRL. We have used the HoRA proof-of-concept to find these results, but the bandwidth threshold has also been theoretically predicted for generic revocation scenarios [22].



Revocation System	**Processing capacity/request**
HRL	negligible
OHSP	$22ms$

Figure 8: OHSP and HRL scalability comparative.

Figure 8 shows how scalable are HRL and OHSP when the number of executing hosts M is increased. In Figure 8, for simplicity, the number of origin hosts N is equal to the number of executing hosts. Notice that the bandwidth is presented in a logarithmic scale. It can be observed that HRL is not bandwidth-scalable. This is because on one hand, HRLs become bigger with a bigger population of executing hosts, and on the other hand, HRL downloads are increased with a bigger population of origin hosts. As a conclusion, bandwidth grows with $N \times M = N^2$. The processing capacity requirements of HRL can be considered negligible. In OHSP, the bandwidth and the processing capacity both grow linearly with the number of origin hosts N because the processing capacity and the communication overhead of OHSP does not depend on the number of revoked hosts. The bandwidth figure has a reasonable value, but the processing time might be a bottleneck in the case of relatively large populations of origin hosts that have a high request rate or if the HoRA is attacked by a flood of queries.

6 Conclusions

In this article the authors introduce two techniques that work together to achieve an effective and usable protection mechanism for mobile agents against manipulation attacks performed by a malicious host during execution. On one hand, MAW

has been presented as an effective and lightweight attack detection mechanism. We have explained the main ideas behind MAW, and we have discussed about which are the most appropriate software watermarks to protect mobile agents. We have also introduced the guidelines to implement MAW using the CT algorithm. On the other hand, the HoRA has been presented as a generic TTP with punishment capabilities. The combined use of the two security mechanisms leads to a reliable environment for honest users and hosts, which is worth even at the expense of introducing some overhead. This article also includes some performance results to evaluate the cost of the proposed mechanisms, and the results show that the cost of the overall system is low enough to make it usable in practice.

References

[1] Java Cryptography Extension (JCE). Institute for Applied Information Processing and Communication of the Graf University of Technology. http://jce.iaik.tugraz.at/download/evaluation/index.php.

[2] Java Development Kit. http://java.sun.com/downloads/.

[3] SourceForge projects, Aglet Software Development Kit (ASDK). http://sourceforge.net/projects/aglets.

[4] D. Chess. Security considerations in agent-based systems. In *First IEEE Conference on Emerging Technologies and Applications in Communications (etaCOM)*, 1996.

[5] D. Chess. Security Issues in Mobile Code Systems. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

[6] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Conference on Programming Language Design and Implementation (SIG-PLAN'04)*, 2000.

[7] C. Collberg, G. Myles, and A. Huntwork. Sandmark - a tool for software protection research. *IEEE Security and Privacy*, 1(4), 2003.

[8] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages (POPL)*, 1999.

[9] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[10] O. Esparza, J.L. Muñoz, M. Soriano, and J. Forné. Punishing Malicious Hosts with the Cryptographic Traces Approach. *New Generation Computing*, 24(4):351–376, 2006.

[11] O. Esparza, M. Soriano, J.L. Muñoz, and J. Forné. Host Revocation Authority: a Way of Protecting Mobile Agents from Malicious Hosts. In *International Conference on Web Engineering (ICWE 2003)*, volume 2722 of *LNCS*. Springer-Verlag, 2003.

[12] O. Esparza, M. Soriano, J.L. Muñoz, and J. Forné. Punishing manipulation attacks in mobile agent systems. In *IEEE Global Telecommunications Conference (Globecom 2004)*, 2004.

[13] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for mobile agents: issues and requirements. In *19th National Information Systems Security Conference*, 1996.

[14] W.M. Farmer, J.D. Guttmann, and V. Swarup. Security for Mobile Agents: Authentication and State Appraisal. In *European Symposium on Research in Computer Security (ESORICS)*, volume 1146 of *LNCS*. Springer-Verlag, 1996.

[15] F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

[16] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile, 1999. RFC 2459.

[17] W. Jansen. Countermeasures for Mobile Agent Security. *Computer Communications, Special Issue on Advanced Security Techniques for Network Protection*, 2000.

[18] W. Jansen and T. Karygiannis. Mobile Agent Security. Special publication 800-19, National Institute of Standards and Technology (NIST), 1999.

[19] D. Kinny. Reliable agent communication - a pragmatic perspective. *New Generation Computing*, 19(2):139–156, 2001.

[20] A. Maña, J. Lopez, J.J. Ortega, E. Pimentel, and J.M. Troya. A framework for secure execution of software. *International Journal of Information Security*, 3(2):99–112, 2004.

[21] Y. Minsky, R. van Renesse, F. Schneider, and S.D. Stoller. Cryptographic Support for Fault-Tolerant Distributed Computing. In *Seventh ACM SIGOPS European Workshop*, 1996.

[22] J.L. Muñoz and J. Forné. Evaluation of Certificate Revocation Policies: OCSP vs. Overissued CRL. In *DEXA Workshops 2002. Workshop on Trust and Privacy in Digital Business (TrustBus02)*, pages 511–515. IEEE Computer Society, 2002.

[23] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP, 1999. RFC 2560.

[24] G. Myles and H. Jin. Self-validating branch-based software watermarking. In *Information Hiding (IH05)*, volume 3727 of *LNCS*. Springer-Verlag, 2005.

[25] J. Nagra and C. Thomborson. Threading software watermarks. In *6th International Information Hydind Workshop*, 2004.

[26] G. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

[27] R. Oppliger. Security issues related to mobile code and agent-based systems. *Computer Communications*, 22(12):1165–1170, 1999.

[28] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 308–316, 2000.

[29] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

[30] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, 2001.

[31] G. Vigna. Cryptographic traces for mobile agents. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

[32] B.S. Yee. A sanctuary for mobile agents. In *DARPA workshop on foundations for secure mobile code*, 1997.