A Mechanism to Avoid Collusion Attacks Based on Code Passing in Mobile Agent Systems*

Marc Jaimez, Oscar Esparza, Jose L. Muñoz,
Juan J. Alins-Delgado, and Jorge Mata-Díaz

Universitat Politècnica de Catalunya, Departament Enginyeria Telemàtica,
1-3 Jordi Girona, C3 08034 Barcelona, Spain
{marc.jaimez,oscar.esparza,jose.munoz,juanjo,jmata}@entel.upc.es

Abstract. Mobile agents are software entities consisting of code, data, state and itinerary that can migrate autonomously from host to host executing their code. Despite its benefits, security issues strongly restrict the use of code mobility. The protection of mobile agents against the attacks of malicious hosts is considered the most difficult security problem to solve in mobile agent systems. In particular, collusion attacks have been barely studied in the literature. This paper presents a mechanism that avoids collusion attacks based on code passing. Our proposal is based on a Multi-Code agent, which contains a different variant of the code for each host. A Trusted Third Party is responsible for providing the information to extract its own variant to the hosts, and for taking trusted timestamps that will be used to verify time coherence.

Keywords: Mobile agent security, malicious hosts, collusion attack, code passing.

1 Introduction

Mobile agents are software entities that move code and data to remote hosts. Mobile agents can migrate from host to host performing actions autonomously on behalf of a user. The use of mobile agents can save bandwidth and permits an off-line and autonomous execution in comparison to habitual distributed systems. Mobile agents are especially useful to perform functions automatically in almost all electronic services, like electronic commerce and network management. Despite their benefits, massive use of mobile agents is restricted by security issues [16,10]. In this scenario, two main entities are considered to study the security weaknesses: the mobile agent (or simply agent) and the executing host (or simply host). These are the main attacks: (1) the agent attacks the host: the protection of the host from malicious agent attacks can be achieved

* This work is funded by the Spanish Ministry of Science and Education under the projects CONSOLIDER-ARES (CSD2007-00004), SECCONET (TSI2005-07293-C02-01), ITACA (TSI2007-65393-C02-02), P2PSEC (TEC2008-06663-C03-01) and, by the Government of Catalonia under grant 2005 SGR 01015 to consolidated research groups.

O. Markowitch et al. (Eds.): WISTP 2009, LNCS 5746, pp. 12–27, 2009.
© IFIP International Federation for Information Processing 2009

by using sand-boxing techniques and a proper access control [7]; (2) attacks to the communications: the protection of the agent while it is migrating from host to host can be achieved by using well-known cryptographic protocols [12]; and (3) the host attacks the agent: there is no published solution to protect agents completely from these attacks. This later attack is known as the problem of the malicious hosts.

This paper introduces a new mechanism which avoids collusion attacks based on code passing. In a code passing attack, a host sends the agent's code to another host which is not the next proper destination host, but another further in the itinerary; this host can analyze the agent's code before arriving through the legal route. Collusion attacks are difficult to prevent or detect, and for this reason most of current approaches do not support them. Only few approaches try to limit them by using itinerary protection techniques [3,2,15]. Our mechanism involves using a different agent for each host to be resilient to these collusion attacks. To do so, we generate different *variants* of the code, which are merged to build up a *Multi-Code Agent* (MCA), which is the final entity that is sent to the hosts. When receiving the MCA, each host request to a Trusted Third Party (TTP) some *extracting instructions*, that is, some information that each host needs to extract its own variant from the MCA. This request is also used to obtain a trusted time reference to control the time coherence among all the hosts. This MCA has been constructed to avoid that two or more colluding hosts analyze their variants to locate possible vulnerabilities within the agent's code. Moreover, the time references are used to limit the time of execution in the hosts, so suspicious behaviors could be detected.

The rest of the paper is organized as follows: Section 2 describes the main published approaches to protect mobile agents; Section 3 details the proposed mechanism to avoid collusion attacks; finally, some conclusions can be found in Section 4.

2 Malicious Hosts

The attacks performed to a mobile agent by an executing host are considered the most difficult problem regarding mobile agent security [11,10]. It is difficult to detect or prevent the attacks performed by a malicious host during the agent's execution. Malicious hosts could try to get some profit of the agent reading or modifying any part of the agent due to their complete control on the execution. Moreover, if two or more malicious hosts collude to perform an attack, the task of protecting the agent becomes even more difficult to achieve. These are some published approaches that deal with the problem of the malicious hosts. Some authors think that the only solution to this problem is the use of a closed tamper-proof hardware subsystem where agents can be executed in a secure way [23,22,14], but this forces each host to buy a hardware equipment and to consider the hardware provider as trusted. The environmental key generation [18] makes the agent's code impossible to decrypt until the proper conditions happen on the environment. However, there is no protection once the code has been decrypted. Hohl presents obfuscation [8] as the mechanism to assure the execution

integrity during a period of time, but this time depends on the capacity of the malicious host to analyze the obfuscated code. The use of encrypted programs [20] is proposed as the only way to give privacy and integrity to mobile code. The difficulty here is to find functions that can be executed in an encrypted way. Vigna [21] introduces the idea of cryptographic traces that the agent takes during execution. The origin host asks for the traces if it wants to verify execution, but verification is only performed in case of suspicion. Suspicious hosts can be detected by controlling the agent's execution time in the hosts [6,4], but even with this complement the use of traces is still too expensive. Some other later approaches are also based on the use of Vigna's traces [24,13], but none of them solves the problem in a satisfactory way. In [5], the authors use software watermarking techniques to detect manipulation attacks, but not collusion attacks. Roth [19] presents the idea of cooperative agents that share secrets and decisions and have a disjunct itinerary. This fact makes collusion attacks difficult, but not impossible. In [9], Hohl defines the reference states as these that have been produced in reference (trusted) hosts. Some proposals based on reference executions [1,17] have been presented, but they add too much communication overhead.

3 Multi-Code Agent

In this paper, we present a new mechanism which makes mobile agents resilient to collusion attacks based on code passing. As we mentioned previously, collusion attacks have been barely studied in mobile agent systems. In a code passing attack, a host sends the agent's code to another host which is not the next proper destination host, but another further in the itinerary; thanks to that, this host can analyze the agent's code before it arrives through the legal route.

Figure 1.a shows a collusion attack based on code passing: the origin host sends the mobile agent to Host-1, which executes the agent properly and sends it to the next host (Host-2) using the appropriate itinerary. At the same time, Host-1 sends the agent's code to another host in the itinerary, say Host-4, so this will have more time to analyze it in order to locate possible vulnerabilities. To avoid this, we generate different *variants* of the code, which are merged to build up a *Multi-Code Agent* (MCA), which is the final entity that is sent to the hosts. When receiving the MCA, each host requests to a TTP some *extracting instructions*, that is, some information that each host needs to extract its own variant. This request is also used to obtain a trusted time reference to control the time coherence among all the hosts. This MCA has been constructed to avoid that colluding hosts analyze their variants to locate vulnerabilities in the code. The time references are used to limit the time of execution in the hosts, so suspicious behaviors could be detected.

Figure 1.b shows the main idea of this new mechanism. The origin host sends the MCA to the first agent in the itinerary, Host-1. This asks for the extracting instructions to the TTP in order to obtain its own variant. This request is also used to get a trusted time reference from the TTP. After that, Host-1 executes

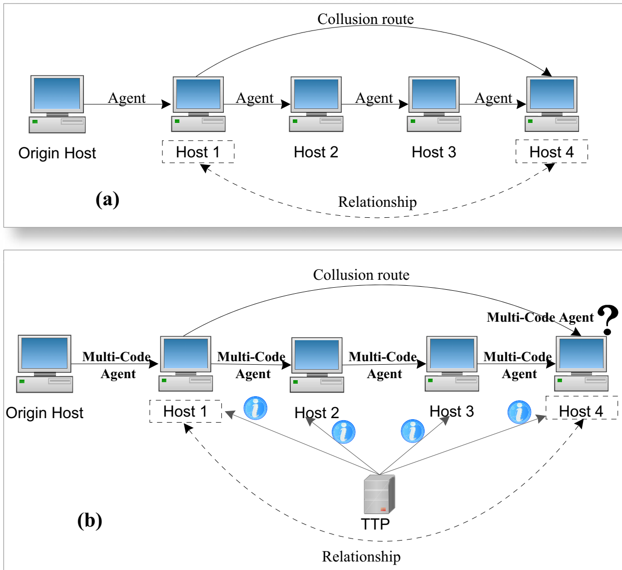

Fig. 1. The code passing problem

its variant, and sends the MCA to the next host in the itinerary, Host-2. Host-1 could send the MCA (or even its variant) to another host, say Host-4 trough the collusion route. However, Host-4 cannot extract its own variant because it needs its extracting instructions, which should be provided by the TTP. As well, Host-4 cannot ask for its extracting instructions before Host-3 because the TTP controls the time coherence.

3.1 Codification Part

The codification part begins with the modification of the original agent in order to obtain different variants of it (see Figure 2.a). As it is shown, the original agent has a Code section, which contains the whole Bytecode of the agent. The original agent also has a Results section, which refers to the data structure where the results of the execution will be stored. We need to slightly modify the Code and Results sections of the original agent to construct a variant for each host in the agent's itinerary. In this particular example, there are three variants of the original agent because the agent is going to be executed in three different hosts. These new variants of the agent have to accomplish two basic functions: providing a different Bytecode, and providing a different data structure for the results of the execution. After obtaining all the variants, the codifier takes the Bytecode of each of them and builds the MCA. This process is divided in three steps: Merging, Meshing up, and Compressing. The merging step (see Figure 2.b) involves taking the Bytecode of all agents and putting them all together in a single file. The meshing up step (see Figure 2.c) involves mixing the

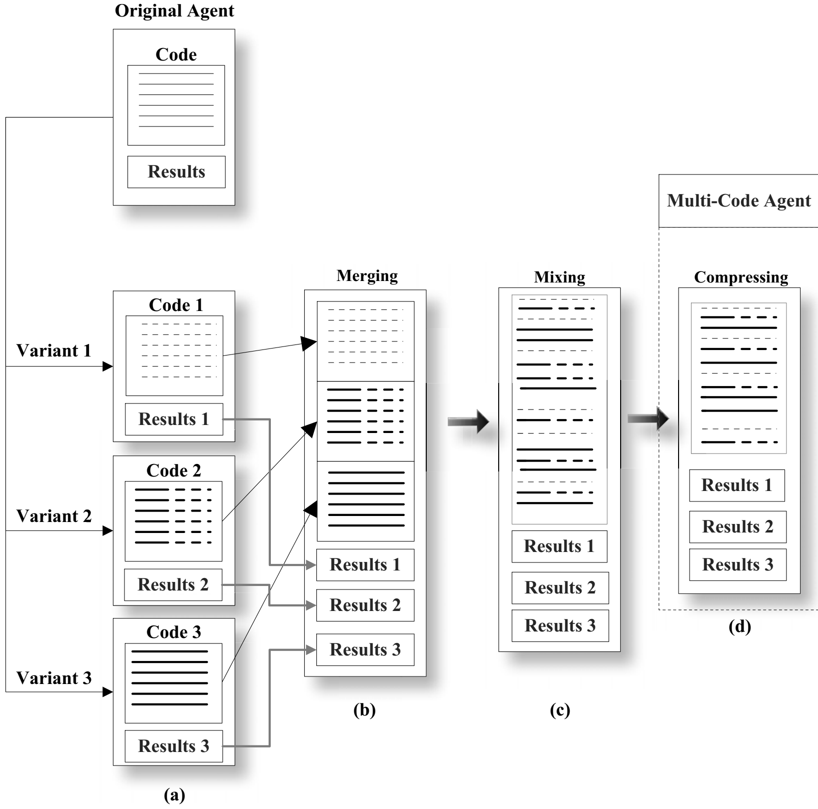

Fig. 2. Codification part

code instructions of the different agent variants. And finally, the compressing step (see Figure2.d) involves eliminating redundancy. At the end of the codification part, we have the MCA, which includes a Code section which contains the Bytecode of all the agent variants, and a Results section where the execution results of each host will be stored.

Creation of the new agents. The Multi-Code agent contains the code of all the variants of an original agent. These variants have to accomplish two basic functions: providing a different Bytecode (we want each host to execute a different agent), and providing a different data structure for the results of the execution (we want to verify that each host has executed the right agent). The process of creating the new agent variants is divided in two steps:

Step1. Modification of the code to obtain different data structures. In this step we modify the Results section of the original agent to obtain a Results section for each variant. Each of these must contain an *Identity Mark*, which is a data structure that will demonstrate that each host has executed its own variant.

The way that each host uses to put all this information (the values themselves, their order and their relationships) within the results section is how this host in particular implements its Identity Mark. This Results section is finally fitted within the MCA, and sent to the next host.

Step2. Modification of the code to obtain Bytecode variability. The goal of this step is to obtain a set of new agent variants that will perform the same tasks than the original agent, but using a different code. We need the agents to be different and difficult to analyze. However, we recommend the agents to have a similar structure (same classes, methods, names, variables, etc) and use the code of the methods to introduce differences among variants. If we take into account this restriction, we will be able to reduce significantly the length of the MCA and the length of the extraction instructions. Just notice that using the same structure for all the variants does not directly mean that we have to maintain the structure of the original agent.

To illustrate the importance of maintaining the basic structure of the classes in each agent variant, we will give a simple example with an original agent and two variants of it. Figure 3.a shows the code of the original agent, which has a single class named OriginalAgentCode. The class contains a single method named method1() that calculates a simple arithmetic operation $a = 10 + 20$ and returns the value of the result. In the example, we take the same structure of the original agent to create the new agent variants (one class and one method), but we add two new variables, b and c, to make the code of the method different. These new variables will allow us to perform the same arithmetic operation in different ways. Figure 3.b shows the code of the first variant, and Figure 3.c shows the code of the second variant.

We have measured the size of this example in terms of Bytecode, and both variants weight 3328 bits, but only 144 bits are different (only a 3.42% of the Bytecode is different from one variant to another). These similarities are due to the fact that we have used the same structure for the two variants. If there were

```
public class OriginalAgentCode {

  int method1( ) {

    int a = 10 + 20;
    return a;
  }
}
```

(a)

```
public class OriginalAgentCodeA {

  int method1( ) {

    int a;
    int b;
    int c;

    a = 10;
    b = 10 + a;
    c = a + b;
    return c;
  }
}
```

(b)

```
public class OriginalAgentCodeB {

  int method1( ) {

    int a;
    int b;
    int c;

    c = 10;
    a = c + 10;
    b = a + c;
    return b;
  }
}
```

(c)

Fig. 3. Original code and two agent variants

more than two hosts in the agent's itinerary, we would need more agent variants, and we should implement the same arithmetic operation in other ways.

Building an Identity Mark. The Identity Mark is a vector that contains n values. These values could correspond to execution results R_i or to intermediate values V_j. A simple example of the implementation of an Identity Mark with $n = 5$ is shown in Figure 4. The Host executes its variant and obtain two values corresponding to execution results (R_1, R_2) and three intermediate values (V_1, V_2, V_3). With all these values, the agent builds a data structure that implements its particular Identity Mark, and which also contains the execution results. Finally, this data structure is sent to the origin host using the MCA.

The strength of the Identity Mark is based on two main characteristics:

1. The order of the values: the Identity Mark is an ordered vector of n values. So then, $n!$ possible Identity Marks are possible by simply ordering the values in different ways. Hence, the probability of generating the particular Identity Mark of a certain variant will be $\frac{1}{n!}$. In the example of 4, the Identity Mark contains five values (R_1, R_2, V_1, V_2, V_3), which could be stored following one hundred and twenty different sequences.
2. The intermediate values: not only the order of the values is specific for each agent, but the values themselves are obtained in a different way depending on each agent. We assume that values corresponding to execution results (R_i) cannot be changed. However, intermediate values V_j can be computed in different ways. For instance in the example of Figure 4, the computation of V_2 in Agent A may depend on R_1, V_1 or some other input data only used in Agent A code. On the other hand, the computation of V_2 in Agent B may depend only on R_2. Thanks to that, a host cannot establish any direct relationship between V_2 from Agent A, and V_2 from Agent B.

As it can be directly deduced, increasing the number of values n of the the Identity Mark vector provides much more possibilities to protect the agent from collusion attacks. In Subsection 3.5, a detailed example of a collusion attack

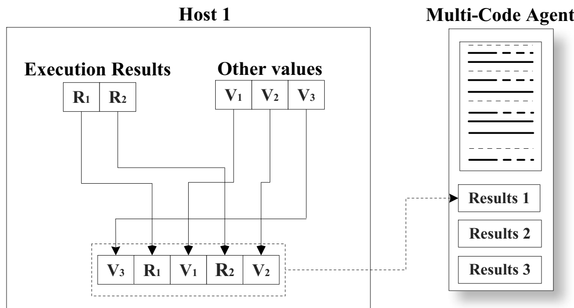

Fig. 4. Example of an Identity Mark

is provided, and it is stated that even if a host has analyzed the code of another agent variant, the probability of generating a valid Identity Mark for its designated variant is very low.

Merging. On this step, the java files are compiled, the class files are obtained, and finally these class files are merged to build the MCA. Following with the previous example proposed in Section 3.1, we compile the OriginalAgentCodeA.java file and the OriginalAgentCodeB.java file, and we obtain two new files: the OriginalAgentCodeA.class file and the OriginalAgentCodeB.class file.

A detailed example of the merging process concerning the Bytecode is given in Figure 5. Starting from the original Agent (Figure 5.a), the two new agents variants are created. Figure 5.b corresponds to the specific Bytecode of the Original Agent A, which is colored black. Figure 5.c corresponds to the specific Bytecode of the Original Agent B, which is Grey colored. Finally, Figure 5.d shows the resultant Bytecode of the MCA (without mixing and compressing). For simplicity, the rest of the Bytecode code, which is shared by both agents (variant A and variant B) and the MCA, is not shown in Figure 5.

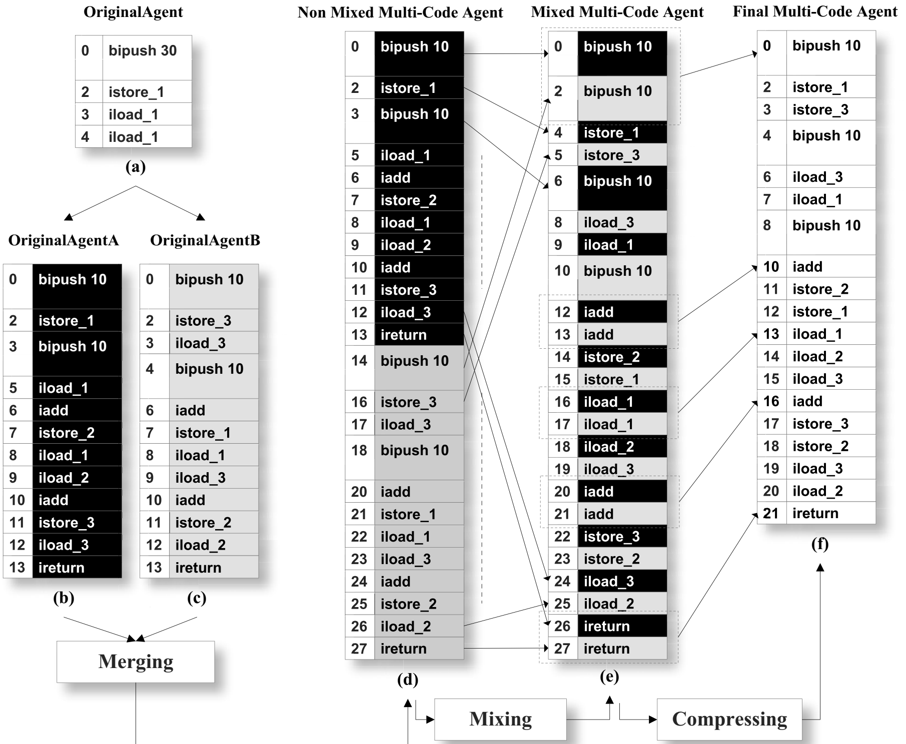

Fig. 5. A Bytecode view of the Multi-Code Agent construction

Mixing. We have started from an Original Agent, which has been modified to obtain two different implementations of it. After that, these new agents have been merged to build up a single MCA. Now, the code of the agents is going to be mixed up. Before the mixing, the black and grey Bytecode is separated in two areas, and it is easy to identify them by an attacker. After the mixing, an attacker will have difficulties to identify which Bytecode corresponds to each variant. A detailed example of the mixing process concerning the Bytecode is given in Figure 5.e, which shows a possible final distribution of the Bytecode instructions after the mixing process. It is important to stress that the new instruction distribution order, in the Mixed Multi-Code Agent, could be any. The more altered is the instruction order in the MCA, the more difficult is to understand the Bytecode. However, the complexity and length of the extracting instructions increases among the alteration of the instruction order. In our case, we are going to preserve the natural order of the instructions in order to simplify the extracting instructions.

Compressing. The last modification in the Multi-Code codification part is compressing the Bytecode instructions. As a result of the mixing process, some consecutive Bytecode instructions are equal, and every time this duplicity happens, we can erase one of the duplicated instructions (see Figure 5.f). Although the length of the code is reduced, the compressing factor is very low if we take into account the whole agent code. In the case of the example, the length of the MCA before the compressing step is 3472 bits, and the length after the

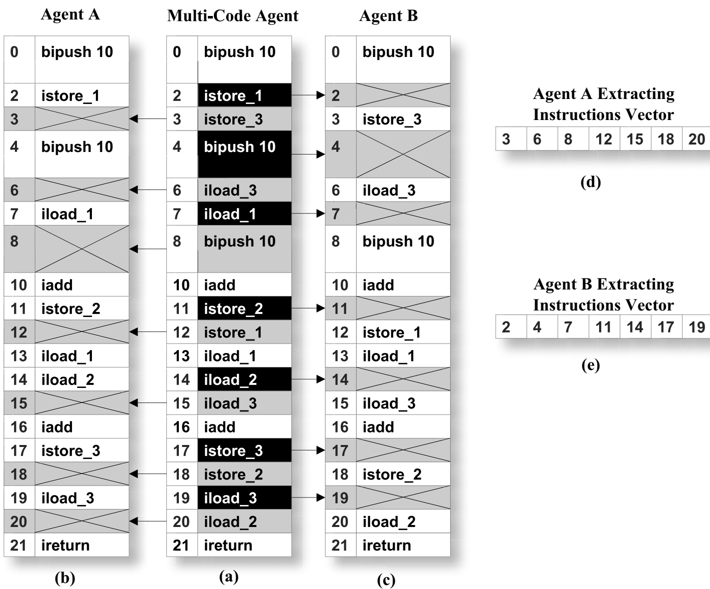

Fig. 6. Generating the Extracting Instructions

compressing step is 3424 bits. This means that we have reduced the MCA's length only by a factor of 1.38%. However, compression is necessary to add some extra complexity to the resulting code. For an attacker, it is more difficult to extract a certain agent if some of the instructions have been erased.

Extracting instructions. When a certain Host receives the MCA, it has to extract its corresponding agent variant. To do so, the Host needs to know which of the Bytecode instructions belongs to its own agent. This information is what we call Extracting Instructions. The information contained in the Extracting Instructions is a list of memory positions. These memory positions indicate which Bytecode instructions do not belong to the agent that is being extracted (blacklist). Figure 6 shows the process of generating the Extracting instructions. The MCA contains a compressed mixture of the different agent variants (see Figure 6.a). The black cells contain the Bytecode instructions of agent A; the grey cells contain the Bytecode instructions of agent B; and white cells contain Bytecode instructions that are shared by both agents. For instance, to obtain agent A, we only need to preserve the cells that contain Bytecode instruction of agent A (Figure 6.b); and the same with agent B (Figure 6.c). Once the cells containing other Bytecode instructions are identified, we store this information in a vector (Figure 6.d, Figure 6.e).

3.2 Distribution Part

Before sending the agent, the origin host must send the whole set of extracting instructions to the TTP (one for each host). The TTP will be in charge of their management. Thus, the origin host is free from having to stay on line while the agent is roaming the net. This process is shown in Figure 7:

- Step 1: The origin host sends the set of extracting instructions (I_1, I_2,..., I_N) to the TTP for their management (in our example $N = 2$).
- Step 2: The origin host sends the MCA to the first host in the itinerary (Host-1). After this step, the origin host stays offline until the MCA arrives from the last host carrying all the results.
- Step 3: Host-1 receives the MCA and request the extracting instructions to the TTP.
- Step 4: The TTP authenticates Host-1, and it sends I_1, which are the extracting instructions that corresponds to Host-1. It also stores the time in which Host-1 asked for its extracting instructions T_1.
- Step 5: Host-1 extracts and executes its variant, and it appends the results of its execution D_1 to the MCA before sending the whole package to the next host Host-2.
- Step 6 and 7: The rest of hosts of the itinerary perform the same actions than Host-1.
- Step 8: When the last host has asked its extracting instructions, the TTP sends back the collected timestamps (T_1, ..., T_N) to the origin host.
- Step 9: The last host sends back the MCA containing all the execution results (D_1, ... ,D_N) to the origin host.

Fig. 7. Distribution of the Multi-Code Agent

3.3 Timestamps Verification

The TTP has been storing a time reference for each valid Instruction Request message sent by the hosts. Thanks to that, it is possible to calculate the period of time in which the agent variant has been available for a host. Figure 8 draws the timeline of part of the distribution part to show that the TTP gets the timestamps just when it sends the Instruction Response message to the hosts. These timestamps are used by the origin host to verify the time coherence of the execution. The origin host can calculate $\triangle T_i = T_{i+1} - T_i$. This period of time ($\triangle T_i$) includes the extraction time of the agent, the execution time of the agent, the transmission time of the Multi-Coded Agent to the next host, the propagation delay, and the time interval between the reception of the MCA, and

Fig. 8. Timestamps storing

the Instruction Response message. With this information, the origin host will be able to estimate accurately the execution time of the agent.

Depending on the computational capabilities of the hosts, and the resources needed for the agent to execute, the origin host can estimate a maximum allowed time of execution ($\triangle T_{Max}$). In case that any $\triangle T_i$ is greater than this value, the hosts will be suspicious of performing malicious activities. Figure 8 shows two time-lines, one in which Host-2 acts honestly, and another one in which acts maliciously. If Host-1 acts honestly, the origin host will detect that $\triangle T_{HonestHost} < \triangle T_{Max}$. On the other hand, if the origin host detects that $\triangle T_{MaliciousHost} > \triangle T_{Max}$, Host-2 is suspicious of performing malicious actions.

3.4 Data Coherency Verification

As mentioned earlier, each agent has a particular Identity Mark, which is codified on the execution results. By checking these Identity Marks, we will be able to verify which variant has been executed by each host. At the end of the process, the Origin Host receives the MCA with the execution results of all the hosts. Then, one by one it takes the results of each Host, and extracts its Identity Mark. If the Identity Mark corresponds to the proper host, there is a positive match and we assume that this particular host has executed its proper variant. Otherwise, we assume that the Host has executed any other agent variant, and we tag it as malicious because it has performed a collusion attack.

3.5 An Example of a Collusion Attack Based on Code Passing

As we explained previously, in a code passing attack, the first colluding host sends the code of its agent to a second colluding host further in the agent's itinerary. This second colluding host will have more time to analyze this code to find possible vulnerabilities. Afterwards, the second host will try to obtain a favorable execution when it receives the agent from the legal route.

We are trying to avoid this kind of attack using a Multi-Code Agent. Let us suppose that the MCA has an itinerary of twenty hosts, and two of them are willing to collude to perform a code passing attack. The origin hosts builds a MCA including twenty variants of the original agent, one for each host in the itinerary. Figure 9 gives an overview of the actions taken by the colluding hosts (Host-1 and Host-13) trying to perform the code passing attack. Host-1 receives the MCA from the origin host, and it asks the TTP for its extracting instructions. Once Host-1 has its extracting instructions, it sends all the information to Host 13 (the MCA, its extracting instructions, its variant, or whatever other information that could be used to attack the agent). After that, Host-1 continues with the execution of its Agent Variant (Host-1 extracts Variant A, Host-2 extracts Variant B, ..., Host-13 extracts Variant M, etc). At this point, while the MCA follows its normal route (see Figure 9.a), Host-13 has received privileged information from the collusion route (see Figure 9.b). Thanks to that, Host 13 has extra time to analyze all this information (the MCA, the extracting instructions of Host-1, Variant A) to guess how Variant M will be and try to tamper it

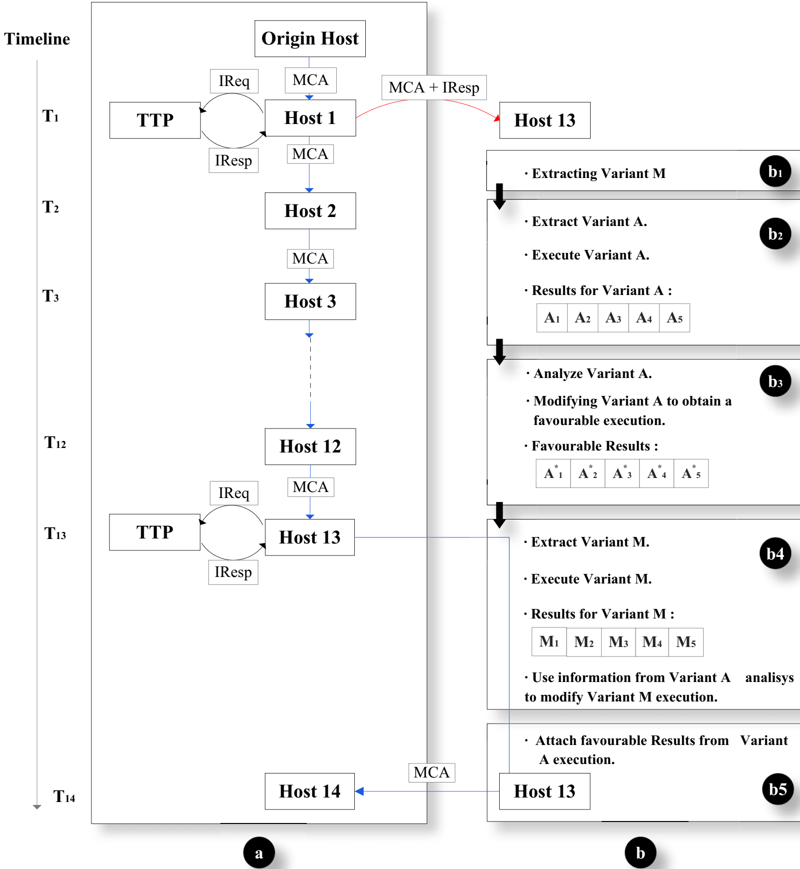

Fig. 9. A collusion attack based on code passing

before the execution. The first attack that Host-13 can attempt is trying to guess which instructions are part of its Variant M (see Figure 9.b1). This is impossible before obtaining the corresponding extracting instructions from the TTP, because all instructions of the MCA are susceptible of being part of Variant M.

Another attack that Host 13 can try to perform is using Variant A to infer which are the intentions of the agent, and trying to understand how the Identity Mark is created to forge variant M when the MCA arrives from the legal route. As Host-1 has sent its extracting instructions to Host-13, it can extract Variant A and execute it. This execution will create a vector with the values $[A_1 A_2 A_3 A_4 A_5]$. But this vector in particular does not correspond to the identity mark of Host-1 (because the values depend on the host in which we have executed the variant, in this case Host-13), nor to the identity mark of Host-13 (because we have executed variant A, not variant M). Host-13 can try to analyze these values to understand how the Identity Mark is constructed (see Figure 9.b2). However, this analysis

will only provide the particular way in which Host-1 should construct the vector, but there are $n!$ possible ways of ordering these values (in this particular example, the number of values of the vector is $n = 5$), and only one correspond to Host-13. In addition, some of these five values correspond to execution results (R_i), and the rest corresponds to intermediate values (V_i), but in fact the host does not know the type of each of these values. As it is shown in Figure 10.a, Host 13 cannot extrapolate the relationship between A_i and V_i or R_i.

At last, Host-13 can also try to modify the code of Variant A to obtain a favorable execution. This tampered execution will generate different values $[A_1^* A_2^* A_3^* A_4^* A_5^*]$ (see Figure 9.b3) that could be used later to compare with the values obtained in a honest execution. When finally Host 13 receives the MCA from Host 12 (from the legal route), it asks the TTP for its extracting instructions and obtain its Variant M. Now, Host 13 has a limited time to execute Variant M, so it cannot expend much time in a deep analysis of the code. For that reason, the only thing it could do is to execute its Variant M honestly to obtain $[M_1 M_2 M_3 M_4 M_5]$, and use the previous experience to modify these values and obtain a favorable execution (see Figure 9.b4). This would be possible to achieve in case that both variant A and M build the values of the Identity Mark vector in the same way, but each agent calculates these values in a different way

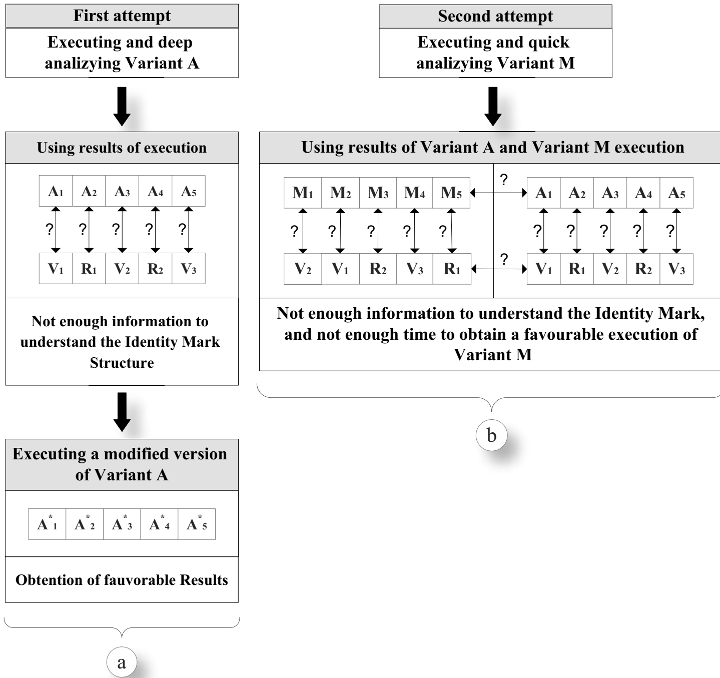

Fig. 10. Understanding the Identity Mark

and in a different order. In conclusion, Host 13 cannot determine the identity mark structure (see Figure 10.b).

4 Conclusions

This paper introduces a new mechanism to avoid collusion attacks based on code passing. The mechanism is based on building a Multi-Code Agent which contains different variants of an original agent code. Each agent variation is executed in a different host, and thanks to that, any information shared by colluding hosts is useless. The introduction of a TTP, which takes charge of distributing the extraction instructions and collecting timestamps, allows us to detect and avoid code passing attacks. Although the Multi-Code Agent contains different agent variants; the length of the Multi-Code Agent is practically equal to the length of a single agent. Therefore, we avoid the wasting of bandwidth, which is a typical drawback when using several agents. With the inclusion of an identity Mark in the results section of the agent variants, we could verify that each host has executed its specified agent variant. Finally, the use of the TTP to obtain trusted timestamps allows us to limit the time used by the hosts to execute the agents.

References

1. Benachenhou, L., Pierre, S.: Protection of a mobile agent with a reference clone. Computer Communications 29(2), 268–278 (2006)
2. Borrell, J., Robles, S., Serra, J., Riera, A.: Securing the Itinerary of Mobile Agents through a Non-Repudiation Protocol. In: IEEE International Carnahan Conference on Security Technology (1999)
3. Westhoff, D., Schneider, M., Unger, C., Kaderali, F.: Methods for Protecting a Mobile Agent's Route. In: Zheng, Y., Mambo, M. (eds.) ISW 1999. LNCS, vol. 1729, p. 57. Springer, Heidelberg (1999)
4. Esparza, O., Muñoz, J.L., Soriano, M., Forné, J.: Punishing Malicious Hosts with the Cryptographic Traces Approach. New Generation Computing 24(4), 351–376 (2006)
5. Esparza, O., Muñoz, J.L., Soriano, M., Forné, J.: Secure brokerage mechanisms for mobile electronic commerce. Computer Communications (Elsevier) 29(12), 2308–2321 (2006)
6. Esparza, O., Soriano, M., Muñoz, J.L., Forné, J.: Implementation and Performance Evaluation of a Protocol for Detecting Suspicious Hosts. In: Horlait, E., Magedanz, T., Glitho, R.H. (eds.) MATA 2003. LNCS, vol. 2881, pp. 286–295. Springer, Heidelberg (2003)
7. Haridi, S., Van Roy, P., Brand, P., Schulte, C.: Programming languages for distributed applications. New Generation Computing 16(3), 223–261 (1998)
8. Hohl, F.: Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 92–113. Springer, Heidelberg (1998)
9. Hohl, F.: A Framework to Protect Malicious Hosts Attacks by Using Reference States. In: International Conference on Distributed Computing Systems, ICDCS (2000)

10. Jansen, W.: Countermeasures for Mobile Agent Security. In: Computer Communications, Special Issue on Advanced Security Techniques for Network Protection (2000)
11. Jansen, W., Karygiannis, T.: Mobile Agent Security. Special publication 800-19, National Institute of Standards and Technology, NIST (1999)
12. Kinny, D.: Reliable agent communication - a pragmatic perspective. New Generation Computing 19(2), 139–156 (2001)
13. Leung, K.-K., Ng, K.: Detection of Malicious Host Attacks by Tracing with Randomly Selected Hosts. In: Yang, L.T., Guo, M., Gao, G.R., Jha, N.K. (eds.) EUC 2004. LNCS, vol. 3207, pp. 839–848. Springer, Heidelberg (2004)
14. Maña, A., Lopez, J., Ortega, J.J., Pimentel, E., Troya, J.M.: A framework for secure execution of software. International Journal of Information Security 3(2), 99–112 (2004)
15. Mir, J., Borrell, J.: Protecting Mobile Agent Itineraries. In: Horlait, E., Magedanz, T., Glitho, R.H. (eds.) MATA 2003. LNCS, vol. 2881, pp. 275–285. Springer, Heidelberg (2003)
16. Oppliger, R.: Security issues related to mobile code and agent-based systems. Computer Communications 22(12), 1165–1170 (1999)
17. Ouardani, A., Pierre, S., Boucheneb, H.: A security protocol for mobile agents based upon the cooperation of sedentary agents. J. Network and Computer Applications 30(3), 1228–1243 (2007)
18. Riordan, J., Schneier, B.: Environmental Key Generation Towards Clueless Agents. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 15–24. Springer, Heidelberg (1998)
19. Roth, V.: Mutual protection of cooperating agents. In: Vitek, J. (ed.) Secure Internet Programming. LNCS, vol. 1603. Springer, Heidelberg (1999)
20. Sander, T., Tschudin, C.F.: Protecting mobile agents against malicious hosts. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 44–60. Springer, Heidelberg (1998)
21. Vigna, G.: Cryptographic traces for mobile agents. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 137–153. Springer, Heidelberg (1998)
22. Wilhelm, U.G., Staamann, S., Buttyán, L.: Introducing trusted third parties to the mobile agent paradigm. In: Vitek, J. (ed.) Secure Internet Programming. LNCS, vol. 1603. Springer, Heidelberg (1999)
23. Yee, B.S.: A sanctuary for mobile agents. In: DARPA workshop on foundations for secure mobile code (1997)
24. Yu, C.M., Ng, K.W.: A flexible tamper-detection protocol for mobile agents on open networks. In: International Conference of Information and Knowledge Engineering (IKE 2002) (2002)