Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2011, Article ID 281517, 9 pages
doi:10.1155/2011/281517

Research Article

Multiagent Systems Protection

Antonio Muñoz, Pablo Anton, and Antonio Maña

Escuela Técnica Superior de Ingeniería Informática, Universidad de Málaga, Spain

Correspondence should be addressed to Antonio Muñoz, amunoz@lcc.uma.es

Received 1 December 2010; Revised 14 March 2011; Accepted 9 June 2011

Academic Editor: Kamel Barkaoui

Agent-systems can bring important benefits especially in applications scenarios where highly distributed, autonomous, intelligence, self-organizing, and robust systems are required. Furthermore, the high levels of autonomy and self-organizations of agent systems provide excellent support for developments of systems in which dependability is essential. Both Ubiquitous Computing and Ambient Intelligence scenarios belong in this category. Unfortunately, the lack of appropriate security mechanisms, both their enforcement and usability, is hindering the application of this paradigm in real-world applications. Security issues play an important role in the development of multiagent systems and are considered to be one of the main issues to solve before agent technology is ready to be widely used outside the research community. In this paper, we present a software based solution for the protection of multiagent systems concentrating on the cooperative agents model and the protected computing approach.

1. Introduction

In the area of information systems, security is one of the most interesting topics. Recently, with the huge growth in the number of distributed systems, the number of computing attacks has increased and therefore so has the number of protection systems. The first work done on software agents was in the mid 1970s by Hewitt and Baker [1]. Hewitt created an agent model (named Actor), which he defined as an autonomous object that interacts and executes concurrently with an internal state and communication capability. Since that initial conception, and due to the work developed in Distributed Artificial Intelligence (DAI), a new concept has arisen known as the Multi-Agent System. The main appeal of these systems is that they allow two or more entities to join forces to perform a common task, which is very difficult to complete individually. Nowadays a huge variation of software agents exists according to their features, abilities, or properties. Mobile agents are implementations of remote programs, that is, those programs developed in a computer and distributed in other computers to continue their execution [2]. The migration capability provokes different security risks and makes controlling the following aspects essential: the protection of hosts against agents and the protection of agents against the host and authors to define the network protection.

The firsts MAS applications appeared in the middle of the 80s. These first systems covered a wide variety of environments (manufacturing systems, process control, air traffic control, information management). But most of them were built upon nonsecure infrastructures [3, 4]. Agent technology developers assumed that the underlying infrastructure was secure at that time, but evidently it is not now. Some other examples of agent-based applications that lacked a secure infrastructure are found in nuclear plants [5] and aircraft control [6] applications.

Regarding the infrastructures for agent-based systems development, the situation is quite similar. Some of the platforms for agents are Aglets http://aglets.sourceforge.net/, Cougaar http://www.cougaarsoftware.com/agents/agents-1.htm, the flagship product of the Agent Oriented Software Group JACK http://www.agentsoftware.co.uk/products/jack/index.html, the popular JADE (Java Agent DEvelopment Framework http://jade.tilab.com/, JAVACT http://www.javact.org/JavAct.html, and Jason for AgentSpeak(L) http://jason.sourceforge.net/Jason/Jason.html. All these tools and methodologies share a common negative

point namely poor security against an attack on the platform in which the agency is running.

Some of the general software protection mechanisms can be applied to the *agent protection*. However, the specific characteristics of agents make mandatory the use of tailored solutions. First, agents are most frequently executed in potentially malicious pieces of software. Therefore, we cannot simplify the problem as is done in other scenarios by assuming that some elements of the system can be trusted. So the security of an agent system can be defined in terms of many different properties such as confidentiality, nonrepudiation, and so forth. but it always depends on ensuring the correct execution of the agent on agent servers (a.k.a agencies) within the context of the global environments provided by the servers.

The main approach presented in this paper is based on the "protected computing" technique, which is based on the partitioning of the software elements into two or more parts. The basic idea is to divide the application code into two or more mutually dependent parts. Some of these parts (which we will call private parts) are executed in a secure processor, while others (public parts) are executed in any processor even if it is not trusted. The main appeal of the solution presented in this paper is that users define the rules to make this division of code by means of an easy to use front end. Thus users can select those variables or those parts of code that are critical and must be protected. Additionally a batch protection tool is included that allows the protection of a portion of code or data.

We apply the protected computing model in order to protect agent societies in a multiagent setting, where several agents are sent to different (untrusted) agencies in order to perform some collaborative task. Because agents run in potentially malicious hosts, the goal in this scenario is to protect agents from the attacks of malicious hosts. The basic idea is to make agents collaborate, not only in the specific tasks they are designed to perform, but also in the protection of other agents. In this way, each agent acts as secure coprocessor for other agents.

Therefore, using the protected computing model, the code of each agent is divided into public and private parts. For the sake of simplicity, and without loss of generality, we will consider the simplest case where the code of each agent is divided in two parts: a public one and a protected one. From this description, it is easy to derive the possibilities of the division of the code into more parts. In particular, the inclusion of multiple private parts, which could even be designed to execute in different coprocessors, is especially relevant for the scenarios that we are considering. Usually, the private part, of each agent, has to be executed by another agent in another host. This scheme is suitable for protecting a set of several mutually dependent agents. Consequently, in this case, a conspiracy of all hosts is necessary in order to attack the system.

This paper focuses on multiagent systems and the security within them. More specifically, our work deals with static mutual security schemes [7] and is organized as follows: in Section 2 we review related publications and we introduce the MAS (multiagent system), mobile agents, JADE platform, and security schemes. Section 3 presents the application of the protected computing approach in the agent protection. Section 4 presents the main approach of this paper; the automatic generation of a MAS making use of the mutual static strategy. In Section 5, we describe the features and architecture of the tools developed, and finally we present our conclusion and future work.

2. Related Work

The purpose of this section is to provide a view on the main agent-based systems and agent-oriented tools, focusing on their security mechanisms. This paper covers a wide range of works from the first approaches to the more recent ones.

Several mechanisms for secure execution of agents have been proposed in the literature with the objective of providing security in the execution of agents. Most of these mechanisms are designed to provide some type of protection or some specific security property in a generic way. In this section, we will focus on solutions that are specifically tailored or especially well-suited for agent scenarios. Some protection mechanisms are oriented to the protection of the host system against malicious agents. Among these, SandBoxing is a popular technique that is based on the creation of a secure execution environment for nontrusted software. In the agent world, a sandbox is a container that limits, or reduces, the level of access its agents have and provides mechanisms to control the interaction between them.

Another technique, called proof-carrying code, is a general mechanism for verifying that the agent code can be executed in the host system in a secure way [8]. For this purpose, every code fragment includes a detailed proof that can be used to determine whether the security policy of the host is satisfied by the agent. Therefore, hosts just need to verify that the proof is correct (i.e., it corresponds to the code) and that it is compatible with the local security policy. This technique shares some similarities with the constraint programming technique; they are based on explicitly declaring what operations the software can or cannot perform. One of the most important issues of these techniques is the difficulty of identifying which operations (or sequences of them) can be permitted without compromising the local security policy.

Other mechanisms are oriented towards protecting agents against malicious servers. Sanctuaries [9] are execution environments where a mobile agent can be securely executed. Most of these proposals are built with the assumption that the platform where the sanctuary is implemented is secure. Unfortunately, this assumption is not applicable in our scenario. Several techniques can be applied to an agent in order to verify self-integrity in order to avoid the code or the data of the agent being inadvertently manipulated. Antitamper techniques, such as encryption, checksumming, antidebugging, antiemulation among others [10, 11] share the same goal, but they are also orientated towards the prevention of the analysis of the function that the agent implements. Additionally, some protection schemes are based on self-modifying code, and code obfuscation [12].

In agent systems, these techniques exploit the reduced execution time of the agent in each platform.

Software watermarking techniques [13] are also interesting. In this case, the purpose of protection is not to avoid the analysis or modification but to enable the detection of such modification. The relationship between all these techniques is strong. In fact, it has been demonstrated that neither perfect obfuscation nor perfect watermarking exists [9]. All of these techniques share the fact that they only provide short-term protection.

Many proposals are based on checks. In these systems, the software includes software and hardware-based "checks" to test whether certain conditions are met. However, because the validation function is included in the software, it can be discovered using reverse engineering and other techniques. This is particularly relevant in the case of agents. Theoretic approaches to the problem have demonstrated that self-protection of the software is unfeasible [14]. In some scenarios, the protection required is limited to some parts of the software (code or data). In this way, the function performed by the software, or the data processed, must be hidden from the host where the software is running. Some of these techniques require an external offline processing step in order to obtain the desired results. Among these schemes, function hiding techniques allow the evaluation of encrypted functions [15]. This technique focuses on protecting the data processed and the function performed, thus it is an appropriate technique for protecting agents. However, it can only be applied to the protection of polynomial functions.

The case of online collaboration schemes is also interesting. In these schemes, part of the functionality of the software is executed in one or more external computer. The security of this approach depends on the impossibility for a part to identify the function performed by the others. This approach is very appropriate for distributed computing architectures such as agent-based systems or grid computing, but has the important disadvantage of the impossibility of its application to off-line environments.

Additionally, there are techniques that create a two-way protection. Some of these are hardware-based, such as the Trusted Computing Platform. With the recent appearance of ubiquitous computing, the need for a secure platform has become more evident. Therefore, this approach adds a trusted component to the computing platform, usually built-in hardware used to create a foundation of trust for software processes [16].

The protected computing concept [17] at the core of the strategy presented in this paper. This approach is based on the idea of dividing the code in two or more mutually dependent parts that will be executed in a trusted processor, while remaining parts can be executed in any other processor, whether trusted or not. In the application of this strategy for the security of multiagent systems, we have achieved a model in which each agent collaborates with one or more remote agents that are executed in different agencies, trusted or not. The approach presented in this paper is based on the collaboration feature of multiagent systems. However contrary to the online collaboration schemes in which the selection of those parts of the functionality of the software to execute in external computers, in our approach the difficult task of selecting those parts is carried out by the developer of the multiagent system that although is not a security expert can decide which are the most critical parts of his software.

Thus a unique successful attack requires the cooperation of every agency in the system, which, in practice, does not make sense. In mutual protection, we can differentiate between two schemes.

(i) Static mutual protection: This solution is the simplest and is fully implemented and described in this paper, which is fundamentally for restricted systems in which the number of agents in the system is previously fixed.

(ii) Dynamic mutual protection: this approach consists of an evolved solution from the static mutual approach, which is more flexible. This approach is applicable for any real multiagent system.

3. Protected Computing Approach Applied to Agent Protection

The Protected Computing approach is based on the division of code in two or more parts. Some of these parts will be executed in a trusted processor, but the others will be executed in a regular processor. These divisions are performed in such a way that the execution of the application is not possible without the collaboration of the trusted processor. One of the most important aspects of this technique is the way in which the code is divided. This might be carried out in mutually dependent parts, but it is essential that

(i) the public part of code will not be able to be used to get information from the private one;

(ii) a communication trace is not possible between each part to get information from the private part.

The Protected Computing scheme can be applied in order to protect a society of collaborating agents by making every agent collaborate with one or more remote agents running in different hosts. These agents act as secure coprocessors for the first one. We call *Mutual Protection* to the application of Protected Computing to MAS.

In Figure 1, we show how every agent interacts with one or more remote agent, and these are executed in different agencies. Then the agents protect each other one by one. A possible attack of this scheme would need the collaboration of every agency to hack the system, but this case is beyond the scope of this paper due to its irrelevance in real applications.

Mutual Protection strategy presents two different schemes according to the requirements of the system. In the first scheme, called *Static Mutual Protection*, the collaboration among agents is predefined. This means that each agent has the private code of at least one agent in the collaboration. Secondly, *Dynamic Mutual Protection* offers the possibility that every agent in the system pretends to be a trusted processor for the remaining agents. In which case, the interaction among agents is not predefined.

To split the code into parts, the developer has to use the Automatic Tool for Code Partitioning (CPT). Since the

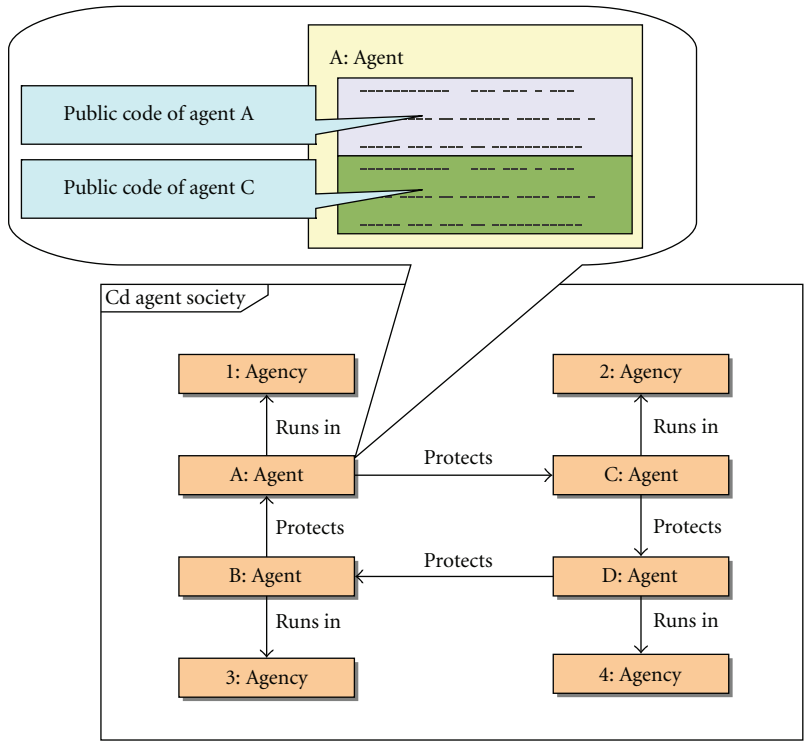

Figure 1: An agent society with mutual protection.

code partition is a difficult task, and specialised expertise is required to carry it out, this tool makes code partitioning easier (creating the public and private parts) according to a set of rules that we call *Protection Profile*. The result of the operation of this tool is a set of public parts and a set of private parts. These parts will be used in a different way depending on the Mutual Protection scheme applied (static or dynamic). Figure 2 shows two different protection schemes as two different ways. A detailed description of these approaches can be found in the subsequent subsections.

3.1. Static Mutual Protection. The Static Mutual Protection strategy can be successfully applied to many different scenarios. However, there will be scenarios where it will not possible to predict the possible interactions between the agents, where the agents will be generated by different parts, when that will involve very dynamic multihop agents. In these cases, the Static Mutual Protection strategy will be difficult or impossible to apply. Therefore, we propose a new strategy named Dynamic Protection where each agent will be able to execute arbitrary code sections on behalf of other agents in the society. The work presented in this paper is based on this static scheme. In this case, private parts of the agent must be included in the agent or in the protector's agents before the execution starts.

The main appeal of this scheme is the increased performance of the system since a split of code between private and public parts is done. Code parts are distributed before system start replacing those parts of private code by their associated call.

Therefore, the efficiency of the system is hardly influenced. Nevertheless, the system is very restricted and the previous setting of the system is mandatory, and agents are protected before their execution.

An example of the possible applications of this scheme is that of a competitive bidding. In this scenario, a client requests bids from several contractors to provide goods services. It is important that the bidding takes place simultaneously, so that none of the contractors can access the offer from the others, because this would give them an advantage over the others. The client can use several single-hop agents to collect the offers from the contractors. Each agent will be protected, using the Static Protection strategy, by other agents. This is possible since the client generates the set of agents, which is static and known a priori. We can also safely assume that a coalition of all contractors will not happen. In fact, no technological solution can prevent all contractors to reach an external agreement. Because each agent is protected by other agents running in the hosts of the competitors, and because the protected computing model ensures that it is neither possible to discover nor to alter the function that the agents perform and it is also impossible to impersonate the agents, we know that all agents will be able to safely collect the bids, guaranteeing the fairness of the process.

A different example is given by a monitoring system based on agents of a power plant control software. In this system, each agent controls a parameter and the fact that the value of this is in a specific range beyond which is risky. Evidently the number of agents in this system is deterministic but the global system requires a high level of security. Static mutual protection strategy fits in the requirements of this scenario.

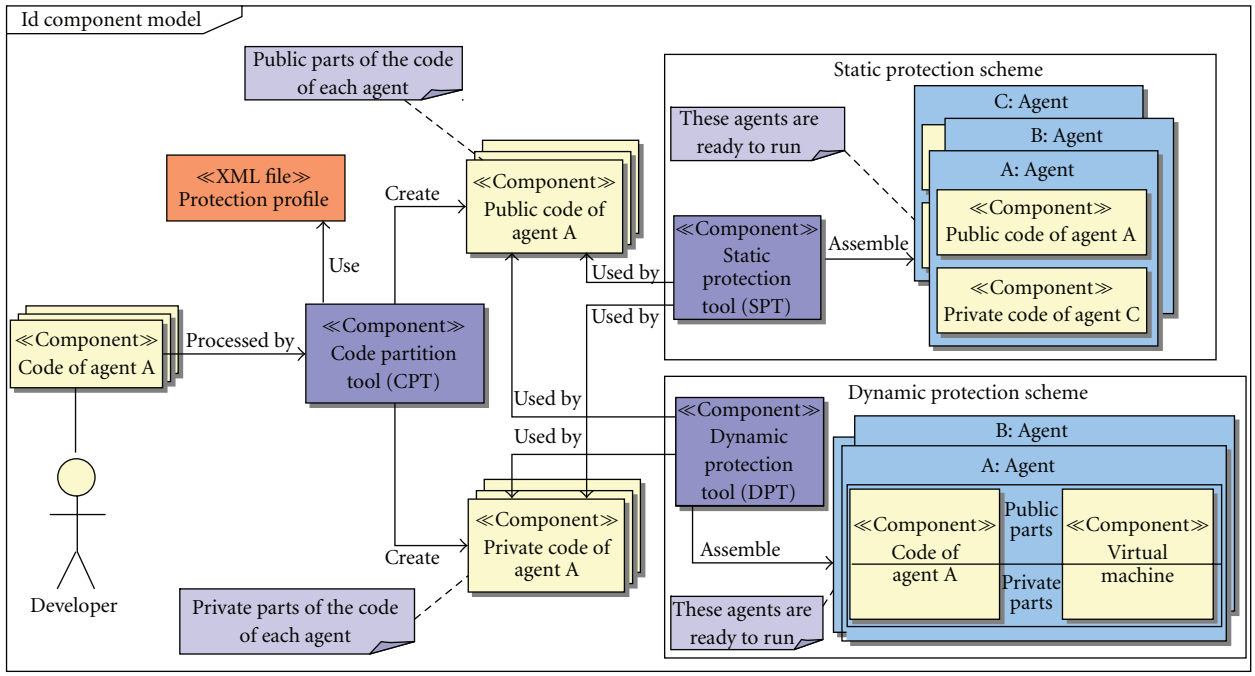

Figure 2: Mutual protection of agents process.

3.2. Dynamic Mutual Protection. The Static Mutual Protection strategy can be successfully applied to different scenarios. However, there are many real-world scenarios where it is not possible to foresee the potential interactions between the agents due to the agents being generated by different parts, or it involves very dynamic multihop agents. In these cases, the Static Mutual Protection strategy is not suitable. The Dynamic Protection Strategy is able to execute arbitrary code sections on behalf of other agents in the society. As shown at the top of Figure 3, each agent includes a public part, an encrypted private part and a specific virtual machine similar to the one described in [18]. This virtual machine allows agents to execute on-the-fly code sections (corresponding to the private parts) received from other agents.

The Dynamic Protection Strategy process is illustrated in Figure 3. In the first exchange, ag1 acts as the protected agent, while ag2 acts as the protecting agent (secure coprocessor) for the first one. In the exchange, ag1 sends a private code section to the virtual machine of ag2. This virtual machine processes the private section and returns some results (results1). Subsequent exchanges illustrate ag3 acting as protecting agent for ag2 (in this case the protected agent), and finally ag1 protecting ag3. The scalability of this scheme is very good since only a few agents (one in most cases) are involved in the protection of any other agent.

4. Paper Contribution

The general aim of the work presented here is to allow the system developers to create secure agent-based systems, namely, a developer should be able to protect his MAS using the mutual static libraries producing an equivalent version of the system. Despite the fact that code distribution tasks imply the selection and protection of different parts iteratively it is not difficult, but it is tedious and certainly not efficient if it is performed manually. We consider the possibility of changing the security setting, studying results and deciding the most appropriate settings for our specific MAS to be useful points. For this reason, Secure Agent Generator tool is focused on automating the whole process increasing the efficiency.

Results of this contribution are fully integrated in the JADE platform providing solutions that allow the development and execution of securely multiagent systems based on the mutual static strategy. Figure 4 shows an overview of the protection process.

Secure Agent Generator tool has as feedback the insecure MAS, that is, a set of nested agents developed to achieve certain goals. This solution is composed of a set of agents defined by Java classes (.class files). Evidently, the output of this tool is an equivalent MAS in functionality. However, the new version of the system is secure according to the strategy used and the parameters set in the protection profile. Secure Agent Generator tool is made up of several sequential tasks: read, analyse, modify and create .class files. A ".class" file has a quite complex and hard to manage internal structure. A huge number of references and the low level code made it a hard and tedious task to analyse and create. Different approaches to handle these files exist, among them we highlight BCEL [19], Javassist [20], or ASM [21]. We bowed to use BCEL due to several reasons; it is the most popular in the community, well documented, fully developed in Java and deployed in Apache software foundations, which provides an easier integration.

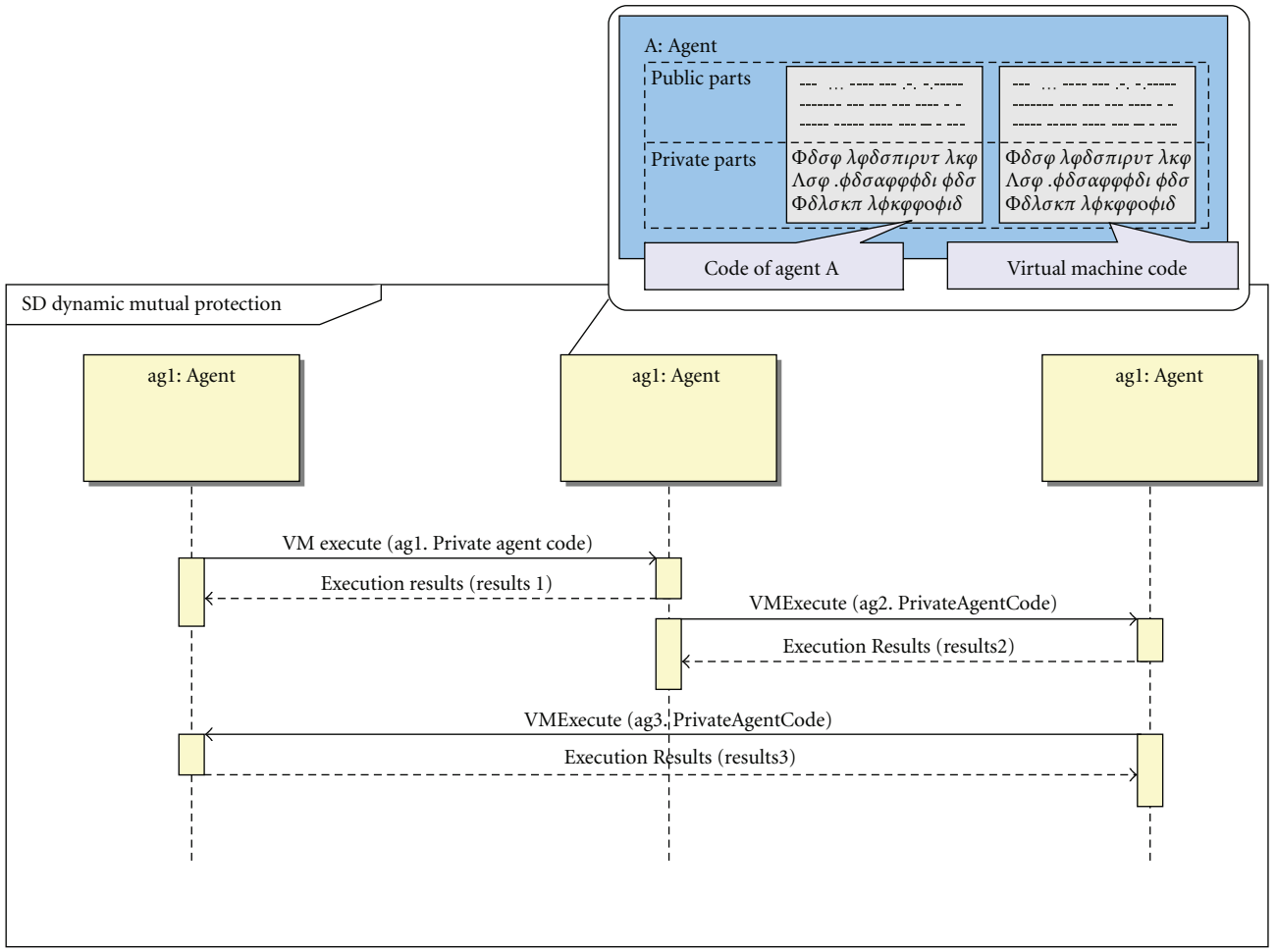

Figure 3: Structure of an agent with dynamic mutual protection.

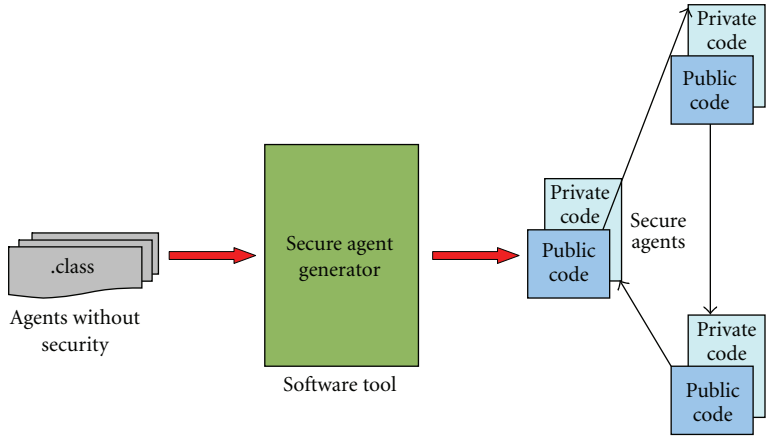

Figure 4: Secure agent generator tool.

This paper presents a methodology to protect agent-based systems using the protected computing approach. This methodology is split to two different strategies. The first approach, namely, the Static Mutual Protection, is based on the restriction that the number of agents in the system is fixed and the protection is performed at compiling time. This approach provides good results in practice because the efficiency is not really affected. The second approach, the Dynamic Mutual Protection is based on protection at runtime. This strategy is more flexible but the efficiency can

be affected considerably. Static Mutual Protection strategy is fully implemented and a set of assisting tools are created to facilitate the protection task. Meanwhile the Dynamic strategy is fully designed and actually being implementated.

5. Architecture

In this section we provide an in-depth description of the most important features and characteristics of the functionalities provided by our tool. It is important to highlight the fact that we are focused on the development of a tool for the automatic generation of secure MAS, implementing the mutual static strategy that as input has a set of agents that compound the nonsecure MAS.

It is important to note that the feedback files, the set of nonsecure agents, must fulfil a set of restrictions (preconditions) as described; every file must be precompiled and stay in ".class" format; every file must represent a class inherited from jade core Agent. And internal anonymous classes are not allowed in these files.

Similarly, there are some output conditions to take into consideration; each of these new agents will have a protector agent preassigned, which cannot be changed at runtime. However, our aim is that the output MAS is equivalent to the input MAS, meaning the behaviour of the new MAS with the security incorporated will be exactly the same as that of the input MAS. Another aspect to mention is the fact that the main architecture of the system follows the model view controller pattern. However, the view is a simple graphic user interface to facilitate the use of the assistant tool.

This process has three clearly differentiated phases: the loading of original agents, security settings that meet requirements and the final creation of secure agents. Each of these phases was implemented with a set of classes in charge of providing a correct execution.

Despite the description, we think that the most appropriate way to illustrate the whole process is by means of a practical example, thus showing the role of every phase in the process. The code of a nonsecure JADE agent inherits from Agent class and its execution code is inside the setup() method. As we have previously stated, it is necessary that each agent has a protector agent associated, thus the minimum number of agents for a secure MAS is two. Let us also suppose that this example contains an empty agent (no instructions nor data) with the role of protector. The class that illustrates Figure 5 shows the code of a nonprotected agent, which is inherited from the Agent class.

Following, every phase of the process is described by means of a concrete example.

5.1. Phase I: Loading. This is the first phase in the generation of the secure agents process. The goal of this step is to load a .class files and analyse their content. To this end, we have selected the set of nonsecure agents, and then we have identified and analysed all its elements, that is, methods, fields, instructions, internal classes, and so forth.

In this class files analysis stage, we have used the static component of the BCEL libraries. This part of the API

```
public class Example extends Agent  {

    // Fields definition
    ...
   protected void setup(){

      // Code not to be protected
      System.out.println("Not to be..");

      // Code to be protected
      System.out.println("To be...");

   }
}
```

Figure 5: Example code.

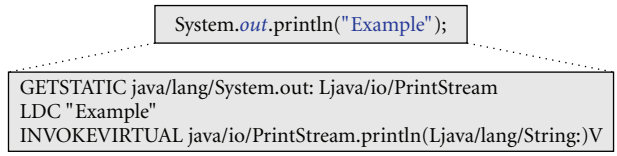

Figure 6: Matching.

provides the methods to load a class file and the automatic generation of the structure of a class file [22]. This process is repeated until, for each of the elements from the file class (methods, fields, internal classes, instructions, etc.), a modelling object is created and is used to handle it.

Each element generated with BCEL component is a read only one. However, it is important to save information from each of these elements by means of annotations. In order to do this, we created classes that inherit directly from the BCEL classes. In these new classes, we introduce all the useful information for the next phases. In this analysis process, we clearly saw the special case of the instructions, this is a more complex case than the rest of the elements. Inside these class files, there is a section dedicated to class methods. Among other elements inside these methods, we found bytecode instructions. These instructions are useless if they are executed separately because normally they depend on the previous one to them and the next one. A relationship is stated between a java instruction and the set of instructions in bytecode. For this reason, we decided to group them in sets corresponding to a java instruction finalised in ";".

Once all agents are loaded in the system, we progress to the setting phase. This phase is very important because we indicate the degree of security and describe the protection links in it. The information of this phase is very relevant due to the fact it selects the elements to be protected.

5.2. Phase II: Setting. The setting phase is the simplest of the three and the easiest to implement. This stage controls the specification of security parameters for the creation of the new secure agents. Among the compounding parts of a JADE agent, there are two elements to be protected instructions and data (Class fields.). The information needed to

determine the security degree is indicated in the percentage of instructions and data to protect. All this information is modelled with a class called SecureAgentConfiguration. With regards to how to indicate the security parameters, we have implemented two separate ways. The first method is needed in order to specify for each of the loaded agents (first phase) the security parameters in the system. This fact implies the selection of every agent and the insertion of the percentage of instructions and selection of fields to protect. This method is a bit more tedious due to the number of agents loaded and the number of testing proofs to perform. For this reason, we have implemented an option that allows us to apply a security template to all agents by means of an XML file. We have developed three basic templates and the possibility to build a customised template.

This example shows the percentage of instructions and data, 50 is the value in this case, to protect. We must take into consideration that there are two different cases in which the percentage value is different for instructions and data. Sometimes the static data or the brunch instructions are clear examples of that. Before progressing to the third phase, we have to select the protection links, meaning to indicate what the protection is like among the agents. It is not necessary to carry out the action manually since there is an option to automate it in the graphic tool. In example, we only have two agents, so one protects the other and vice versa.

5.3. Phase III: Creating Secure Agents. Finally, we have the secure agent creation phase. Thanks to the previous stages, in this stage we collected all the information needed for this creation.

For each of the original already gathered, at least two new classes may be created, one related to the new secure agent with its public code (data and instructions) and the other with the private code. In addition to these classes, we will have as many agents containing the original agent as the number of new classes created.

To create these new precompiled classes, we make use of the dynamic component of BCEL. This part of the BCEL API will facilitate the creation of the skeleton of class files and, depending on the security parameters set in the previous phase, the original code is inserted in one part or the other. In Figure 7, we can see the content of the public part of the new secure agent based in the example in Figure 8. Nevertheless, the class code has been modified as indicated in Figure 7

The new class is an inheritance of the SecureAgent class providing a complete integration in the JADE framework. An initialization section is added in the setup method to mark the protector and the protected agent. Any additional code to perform the protection is needed to be inserted. There will be as many internal agents containing the original agent as the number of new classes are created.

For the protected code, a new class is created that implements the PrivateCode interface, but in this new class it is essential to insert: protected fields (in our example they do not exist); The "execute()" method that contains the protected code divided by sections. The information to know which section to execute is in the method arguments. In the example, the code to protect has only one section then this is directly in the execute() method.

```
public class SecEj extends SecureAgent{

    protected void setup()  {

      // Init
    ...

      // Not protected instruction
      System.out.println("Not to be..");

      // Call to remote code
     ACLMessage msg = new ACLMessage(.);
     msg.setContent("execute-my-..");
     myArgs = argument;
     msg.addReceiver(this.protectedBy);
      send(msg);
    }
}
```

Figure 7: Secure agent.

```
public class Prv implements PrivateCode{
    public Object execute(Object o){

       // Protected code
      System.out.println("To be...");
      return null;
    }
}
```

Figure 8: Private code.

5.4. Phase IV: Validating Secure Agent System. An important aspect of this approach is the validation of the secure system, that is, that we can ensure that the generated system is secure. According to the model proposed by the static mutual protection strategy a correct execution of the system implies that every agent complete its goals. If any error occurs during the execution of the multi-agent system the whole system halts. Obviously this error can be produced by an attack or simply a misunderstanding error in the communication (i.e., an inconsistency of data due to any fail in the communicating channel).

6. Conclusion and Ongoing Work

A full methodology to protect multiagents in multi-agent systems based on the protected computing approach is the main contribution of this paper. In previous sections, we have shown some assisting tools to implement the mutual protection strategy using a friendly interface. A new field is concerned with performing some analytical and statistical studies on the kind of protection to implement, according to the set of requirements of each system.

Methodology proposed in this paper is based on the partitioning of the code of every agent of a system and a distribution of these parts. For the sake of efficiency, the part of code inserted in other agents (private part of code)

are not too long, according to each specific example. This distribution of code could decrease in efficiency especially in those systems composed of a high number of agents but a balance between security and usability is a traditional issue to solve in any real world system.

We have highlighted some of the problems (interoperability, limit to free competition, lack of owner control, lack of assurance, etc.) of the Trusted Computing approach, which make it unacceptable as it is now. Our view is that none of these problems is without solution. We strongly believe that the cooperation of the scientific community can help in solving these problems in the right way.

Obviously the static mutual protection scheme is a deterministic system, that is, a fixed number of agents at entry. However, this approach provides excellent results for those systems in which it fits as shown in the examples in the paper. In order to improve the dynamism of this methodology, we are working on the development of tools to implement the dynamic strategy.

Once we have implemented automatic tools for assisting the deployment of the static mutual protection scheme, next step is the development of assisting tools for the implementation of the dynamic mutual protection scheme. This approach is more complex due to the needed implementation of a restricted specific virtual machine to execute remotely parts of code of agents. Currently, this work is ongoing and it is in its final development stages.

Finally, this paper has demonstrated the existence of alternatives and add ons to current approaches that the scientific community should explore in order to guarantee the best possible solution for the agent protection problem.

References

[1] C. Hewitt and H. Baker, "Actors and continuous functionals," 1977.

[2] H. S. Nwana, "Software agents: an overview," *The Knowledge Engineering Review*, vol. 11, no. 3, pp. 205–244, 1996.

[3] N. R. Jennings and K. Sycara, "A roadmap of agent research and development," 1998.

[4] B. Chaib-draa, "Industrial applications of distributed AI," *Communications of the ACM*, vol. 38, no. 11, p. 4, 1995.

[5] H. Wang and C. Wang, "Intelligent agents in the nuclear industry," *Computer*, vol. 30, no. 11, pp. 28–34, 1997.

[6] U. M. Schwuttke and A. G. Quan, "Enhancing performance of cooperating agents in real-time diagnostic systems," in *Proceedings of the 13th international Joint Conference on Artifical intelligence (IJCAI '93)*, pp. 332–337, Chambery, France, 1993.

[7] A. Maña, A. Muñoz, and D. Serrano, "Towards secure agent computing for ubiquitous computing and ambient intelligence," in *Ubiquitous Intelligence and Computing*, vol. 4611 of *Lecture Notes in Computer Science*, pp. 1201–1212, Springer, New York, NY, USA, 2007.

[8] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pp. 106–119, ACM, New York, NY, USA, 1997.

[9] B. S. Yee, "Secure Internet programming," in *A Sanctuary for Mobile Agents*, pp. 261–273, Springer, London, UK, 1999.

[10] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater, "Robust object watermarking: application to code," in *Proceedings of the 3rd International Workshop on Information Hiding (IH '99)*, pp. 368–378, Springer, 2000.

[11] G. Hachez, *A comparative study of software protection tools suited for E-commerce with contributions to software watermarking and smart cards*, Ph.D. thesis, Universite Catholique de Louvain, March 2003.

[12] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[13] S. Katzenbeisser and F. A. Petitcolas, Eds., *Information Hiding Techniques for Steganography and Digital Watermarking*, Artech House, Norwood, Mass, USA, 1st edition, 2000.

[14] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC '87)*, pp. 182–194, ACM, 1987.

[15] T. Sander and C. F. Tschudin, "On software protection via function hiding," in *Information Hiding*, D. Aucsmith, Ed., vol. 1525 of *Lecture Notes in Computer Science*, pp. 111–123, Springer, 1998.

[16] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[17] A. Maña and A. Muñoz, "Mutual protection for multiagent systems," in *Proceedings of the 3rd International Workshop on Safety and Security in Multiagent Systems*, p. 37, Citeseer, Hakodate, Japan, 2007.

[18] 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*, vol. 3, no. 2, pp. 99–112, 2004.

[19] Apache Software Foundation, *BCEL (Byte Code Engineering Library)*, 2006.

[20] S. Chiba, *Javassist (Java Programming Assistant)*, Sun Microsystems, 2009.

[21] OW2 Consortium. ASM.

[22] T. Lindholm and F. Yellin, *The JavaTM Virtual Machine Specification*, Sun Microsystem, 1999.