Review of Computer Engineering Research
2018 Vol 5, No. 1, pp. 1-11
ISSN(e): 2410-9142
ISSN(p): 2412-4281
DOI: 10.18488/journal.76.2018.51.1.11
© 2018 Conscientia Beam. All Rights Reserved.

A SECURITY SCHEME FOR PROTECTING AGENT SOCIETIES

Antonio Muñoz[1]

[1]*Computer Science Department University of Malaga, Málaga, Spain*
Email: amunoz@lcc.uma.es

ABSTRACT

Article History
Received: 16 March 2018
Revised: 16 April 2018
Accepted: 20 April 2018
Published: 23 April 2018

Keywords
Agent security
Protected computing
Trusted computing
Mobile agents
Multi-agent systems
BCEL
Mutual protection.

Mobile agent paradigm, in particular multi-agent systems represent a promising technology for emerging Ambient Intelligent scenarios and Cyber Physical Systems sharing a huge number of heterogeneous devices interacting. Unfortunately, the lack of appropriate security mechanisms, both their enforcement and usability, is hindering the application of this paradigm in real world applications. This paper presents a software based solution for the protection of multi-agent systems based on mobility property of agents. This approach focuses on the cooperative agent-based model and the core of this concept is the protected computing approach. Finally, one aspect of this approach is its user friendly style for agent based system developers who are not security experts.

Contribution/Originality: This study is one of very few studies which have investigated a mutual scheme mechanism for agent protection. This approach provides the theoretical foundations and a complete description of its implementation, which is supported with a set of tools to facilitate agent based system coding protection by a friendly graphical user interface.

1. INTRODUCTION

Today security is one of the most relevant topics. Recently, the number of computing attacks has been triggered and as a consequence the number of protection systems needed to face this issue. Along this paper it is proposed an agent based protection mechanism. Particularly, this paper focuses on static mutual security schemes Man˜a, et al. [1]. Hewitt and Baker [2] created an agent model he defined as an autonomous object that interacts and executes concurrently with an internal state and communication capability. This fact evolved to the coined of a new concept known as the Multi-Agent System (MAS), which allow two or more entities to join forces to perform a common task, which is very difficult to complete individually.

Today an heterogeneous variation of software agents exists according to their features, abilities or properties. We focus on mobile agents as part of multi-agent systems. Mobile agents are implementations of remote programs, that is, those programs developed in a computer and distributed in other computers to continue their execution [3]. The migration capability provokes different security risks and makes controlling the following aspects essential: Protection of hosts against agents; Protection of agents against the host; and Network protection. Several protection approaches exist for each of these points, but this paper will address the related with the protection of agents against the host. We have decided those concerns do not trust the agencies that host mobile agents when these migrate, producing security risks in the whole multi-agent system.

One strategy that faces these security issues, as well as add a higher level of security to the whole system, is the protected computing concept [4]. This strategy lays its foundations on separating the code in mutually dependent parts, some of these parts to be executed in a trusted processor, and some parts can be executed in any processor. Actual implementation of this strategy involves to build a model in which agents mutually collaborate with one or more remote agents. Agents are executed in different host (agencies),that should be trusted. This strategy forces that every agent in the system is codependent of, at least, two agents, generating a virtual linking network of agents. However, a successful attack has been identified. This requires the cooperation of every agency in the system, but this case is not applicable in real cases. We differentiate between two strategies in mutual protection, according to the context and conditions to be applied.

- Static mutual protection: This solution is more simple and is fully implemented and described in this paper. The application of this strategy is recommended to restricted systems in which the number of elements in the system is known a priori.
- Dynamic mutual protection: This approach is an evolution of the static, which provides a higher level of flexibility. This approach is applicable for any real multi-agent system.

Mutual protection has its foundations on the protected computing approach. Along this paper we present a set of assisting tools that have been fully designed and implemented to carry out this strategy by an automatic process. Final target of this set of tools is to provide auto-mechanism to protect multi- agent system. The idea behind this work is to start from any multi-agent system and setting the mutual protection scheme using tools' interface to produce a protected mobile agent system coherent with specified settings.

This paper is structured as follows: section 2 reviews the state of the art and presents the most relevant technologies and platforms to build this solution. Section 3 presents the main approach of this paper; the automatic generation of a MAS making use of the mutual static strategy. In section 4, we describe the features and architecture of the tools developed, and finally we conclude.

2. RELATED WORK

Different approaches exist for secure software migration, but it is interesting techniques that provide two-way protection. Among these techniques is Trusted Computing, based on building a system in which the security of each component is checked by a trusted component. This trusted component is frequently a hardware component [5]. Based on this idea came the development of the Protected Computing approach [6].

It was mentioned that protected computing approach is based on the division of code in two or more parts. Some parts are executed in a trusted processor, but the others will be executed in a regular processor. These divisions are done in such a way, so that application execution is only enabled with a collaborating trusted processor. I identified as one of the most relevant advantages of this technique how the code is separated. This task might be carried out in mutually dependent parts, but it is essential we follow some considerations for that:

- The public part of code never will be able to be used to get information from the private part.
- Any communication trace can be done between both parts, in order to capture some kind of information from the private part.

Once we have a protected mobile agent-based system, many possible applications can be found to use our solution. We posed the possibility to apply this idea to a completely dynamic environment as mobile agents. These systems tend to be compounded of several agents that migrates to non- trusted host with the final target to complete collaborative tasks. A deep study has demonstrated the high difficulty found to guarantee agent and its execution integrity since agencies (hosts) are mostly unacknowledged. Along this section we describe how to address the protection of the agents versus potentially malicious agencies. Figure II shows agent's inter- action with other remote agents and in turn these are running in different hosts. We propose a solution for known malicious hosts problem by means of a mutual protection between every agent involved in the system.

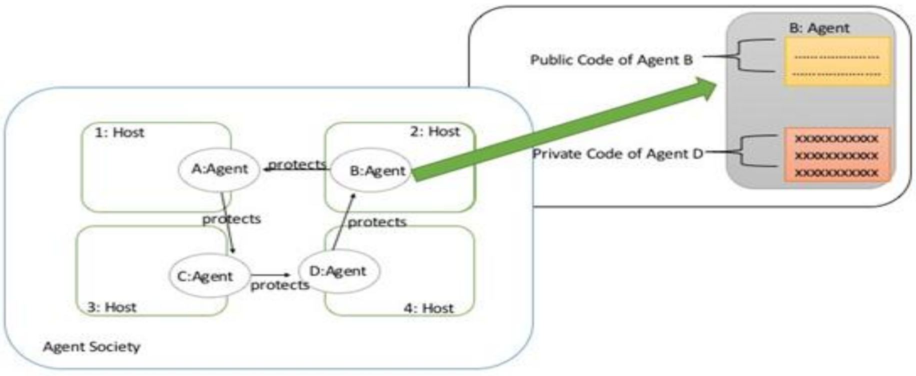


Figure-1. Agent society mutually protected

It was mentioned that two different schemes are within mutual protection strategy, that is, one static and one dynamic. Next sections include a comparison between both alternatives describing the advantages and disadvantages of each of them. In Static Mutual Protection, the first scheme, a predefined collaboration among agents exists. This approach is applied forcing each agent to split its code with a private part of at least one agent in the collaboration. Dynamic mutual protection, where a predefined collaboration does not exist, approach makes possible that every agent in the system acts as a trusted processor for other agents.

A. Static Mutual Protection

Both strategies have been presented, but this paper focuses on static scheme. Therefore, we have to consider how to enable assisting mechanisms to describe predefined collaborations. We recommend that agent's protected parts to be included in agent's code or as part of protector's agents before execution takes place.

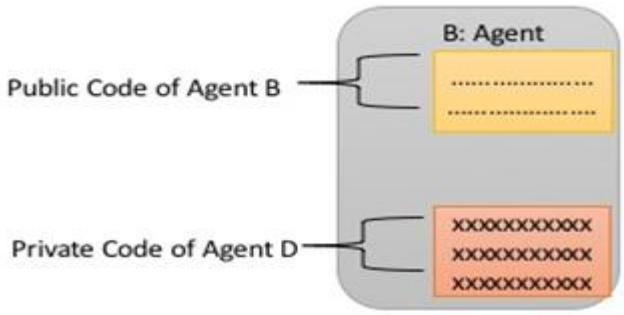


Figure-2. Agent Structure

We claim the most important advantage of this scheme is an improvement in performance since protected parts of agents are reallocated before the system execution starts. This previous redistribution of code avoids overloads in the transmissions of protected parts preventing bottlenecks. However, we are conscious about the flexibility boundaries of this solution since a previous setting, known as code distribution according to some parameters, of the system is mandatory.

In figure 3 we show how message exchange should be performed between two agents to implement our protection approach.

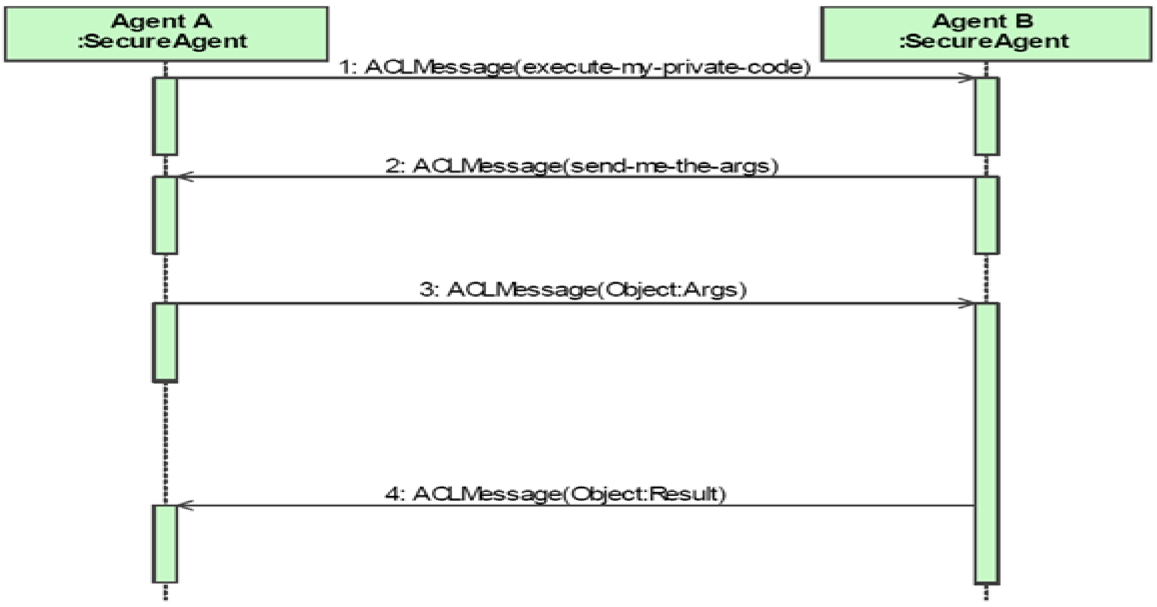

Agent A
:SecureAgent
Agent B
:SecureAgent
1: ACLMessage(execute-my-private-code)
2: ACLMessage(send-me-the-args)
3: ACLMessage(Object:Args)
4: ACLMessage(Object:Result)

Figure-3. Message interchange for protected execution

We defend that security engineering tasks should be considered in the whole process of software productions. Nevertheless, we are aware of the difficulty to face many security challenges for coders. Among those hard tasks we have the identification protected code parts. We made a study consisting of eleven java coders with the target to identify those parts to be protected in random codes. Identification of these parts for security experts familiarized with java coding is around 30 minutes with 90 percentages of successful. Our group of eleven coders spent more than two hours with a 20% of right decisions. After this brief study we encouraged to develop tools for assisting in identification and separation of code.

3. STATIC AGENT PROTECTION APPROACH AND RELATED ASSISTING TOOLS

This section describes our solution as an integrated JADE platform effort that provides mechanisms to develop and execute mobile agents implementing mutual static strategy approach. Our solution's main goal is ease all tasks involved in agent protection to agent system coders. We pretend that a developer should be able to protect his agent-based system using the mutual static libraries producing an equivalent version of his solution. Nevertheless, this job implies to record classes, parts of data and instructions for the actual protection. It is relevant to consider that reason of hard application of this methodology is derived from its repetitive job nature but it is not due to its really complicated task to perform. This implies that its application is tedious and certainly not efficient. Manual application of this strategy is not realistic since many unpractical cases can be found in which the final protected MAS performance is not efficient and we have to modify initial setting. Assisting tool is focused on speeding up and automating the whole process. Following a list with the structure related requirements of the system and select the needed tools is included. An overview of the system scheme is shown in figure 4.

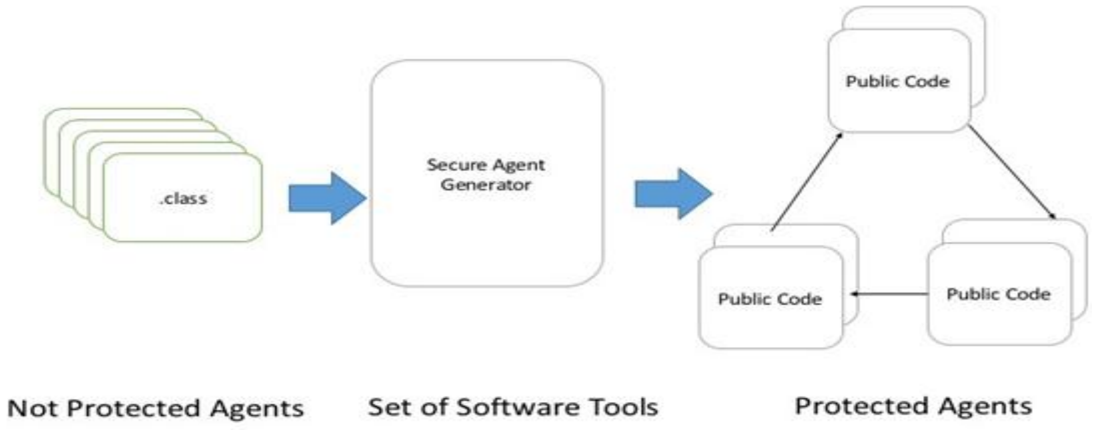


Figure-4. Secure Agent Generator Tool

Main use case identified takes original MAS system as input for our assisting tool. The procedure is as follow; system is composed of a set of agents defined by Java classes (. class files) and these are classified and analyzed. According to settings code is separated and distributed. As final result we have an equivalent MAS in terms of functionality, but it has been protected according to static mutual protection strategy. Most of MAS are based on java then we assume that we have a *MAS* java coded as input. We delivered a set of assisting tools that provides the following functionalities; read, analyze, modify and create, which are steps that compounds the application of protection of *MAS*. This tool entails working with sources java files, particularly we handle *BCEL* [7] ; *Javassist* [8] and *ASM* [9]. *BCEL* is full developed by Apache software foundations, this fact makes possible a better adoption in java coding community.

A. Byte Code Engineering Library (BCEL)

The task to work with ".class" files in not trivial, it has a quite complex and hard to manage internal structure. A huge number of references and the low level code they have made it a hard and tedious task for analysis and production of a protected MAS.

As we have mentioned, we rely on the use of BCEL library to work with source files since BCEL provides a wide and well documented API with clearly three differentiated parts:

- A pack of classes that contains "static" limitations of class* files. This includes classes might be used to read and write class files to and from a specified file. It is especially useful to analyze Java classes when the source code is not available for any reason. Finally, the main class of this pack is *JavaClass*.
- A second pack dedicated for generating or dynamically modifying class files. It can be used to add or analyze code from class files, etc.
- The third pack is a set of code samples and utilities, a viewer for class files and a converter tool from class to HTLM and Jasmin assembler language.

The component in charge to handle static classes is com- posed of some classes for supporting the modeling of internal class file structure. Let us review how is the way of working, *JavaClass* is the main class built from a .class file. This offers a wide spectrum of functions that enables browsing through different components, fields, methods, local variables, internal classes, etc.

A set of patterns is provided to control and analyze every element, i.e. the visitor pattern or the observer pattern. The visitor pattern has been widely productive in our case, and particularly for the instruction analysis. More patterns have been applied as the visitor pattern and allows an easy organization of the classes since the byte-code instructions hierarchy fits perfectly. Let us introduce an example to show its appliance; a LOAD/STORE instruction is required to get the field to read or the method to call in an INVOKE instruction.

However, we found some limitations using the static component because it does not allow the modification nor creation of new .class files. We notice that this step is essential to achieve our target. For this purpose, we use the dynamic component. This component has *ClassGen* as the main class of the dynamic component. This provides functionalities for the creation of an empty class. Additionally, we have the possibility to add dynamically components to pack into a ".class" file. Once the tool has been introduced, let us analyze how to digest input files, that is, the set of input agents and the output is the set of classes that represent protected agents using the static strategy.

4. PROTECTION ASSISTING TOOL FOR MULTI AGENT BASED SYSTEMS

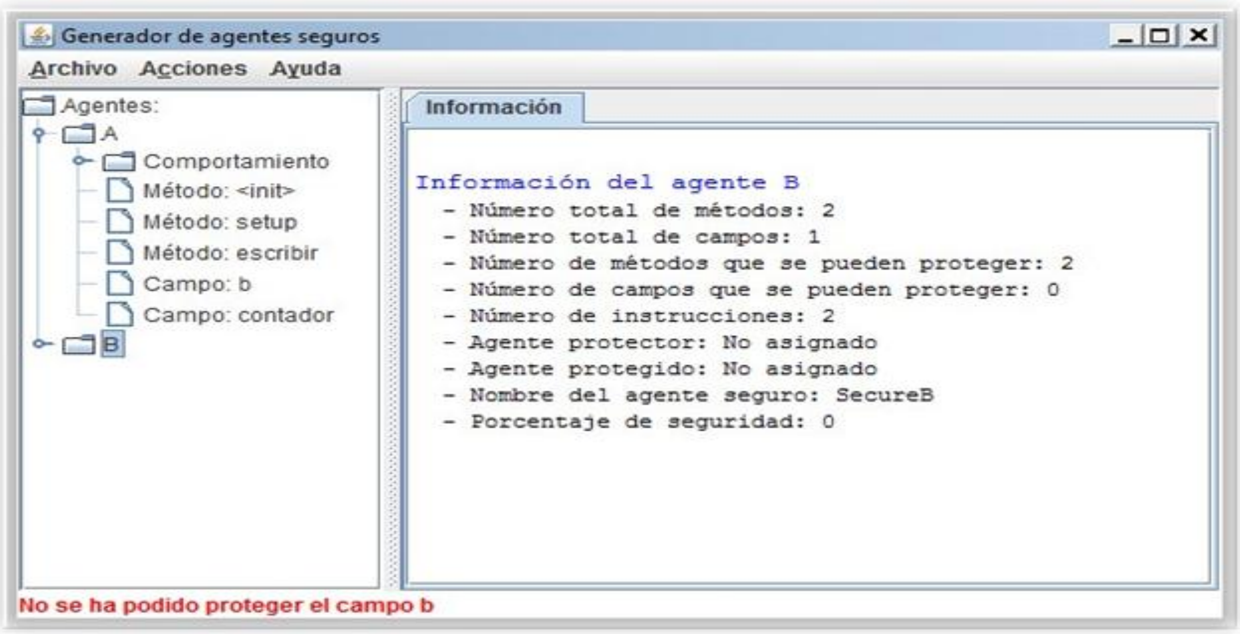
This section provides an full description with the main features and functionalities provided by our tool. We notice that we are focused on the development of a tool for the automatic generation of secure MAS. For this reason, we pay special attention to every step involved in the actual protection of agents during development of these systems. As input we have a set of non-secure agents, these agents fulfill a set of restrictions referred as preconditions, below we describe every of them that must be followed:

- Precompiled and stay in ".class" format of every file.
- Representation by means a class inherited from *jade.core.Agent* of every file.
- Not allowed internal anonymous classes in these files.

In these terms, output files must follow the following conditions:

- A predefined protector agent is assigned, which cannot be changed at runtime for every recently created agent in protected MAS.
- As key principal is essential that resulting MAS is equivalent in terms of behavior to initial MAS we had as input.

In terms of design, these tools follow model view controller pattern to achieve a higher abstraction level and decoupling between every component to easily evolve the solution if required. However, the view is a simple graphic user interface to facilitate the use of the tool, it is shown in figure 5.



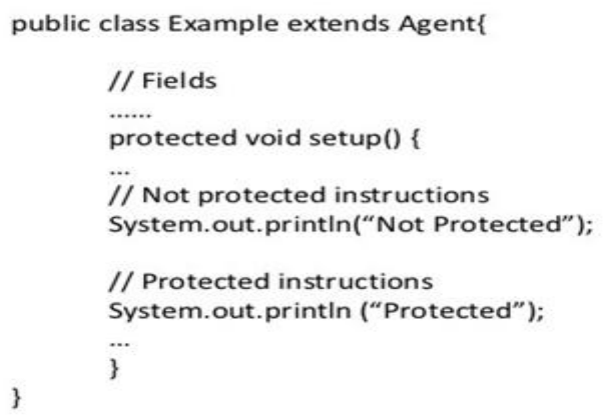
```
Información del agente B
  - Número total de métodos: 2
  - Número total de campos: 1
  - Número de métodos que se pueden proteger: 2
  - Número de campos que se pueden proteger: 0
  - Número de instrucciones: 2
  - Agente protector: No asignado
  - Agente protegido: No asignado
  - Nombre del agente seguro: SecureB
  - Porcentaje de seguridad: 0
```

No se ha podido proteger el campo b

Figure-5. Graphics User Interface

Next, let us describe de use case of an agent based system developer, who is an expert in agent field, but only basic security principals know. For this case, we have modeled our tool with three differentiated phases in its activity. The first step is the loading of original agents as input, second step is related to security settings, this includes all desired security requirements to include in protected system, and we emphasize the desired concept since there are some cases in which security requirements are hardly restricted and not applicable. To face both phases a set of classes in charge of providing a correct execution were implemented. Let us illustrate the whole process using a real example to understand every role for each phase in the process, some minimum guidelines are needed to understand how the code actually works. Inheritance between the code of a regular agent in JADE and the actual agent class, additionally its execution code is encapsulated in *setup()* method. We mentioned the importance of each agent has a protector agent associated in the protected MAS as result. This fact imposes the requirement of a minimal secure MAS of two agents, at least a protector and a protected agent is required. A detailed description of every phase individually is provided in the given example and can be found below.

The class that illustrates the figure *ExampleCode* shows non protected agent source code, which inherits from Agent class and its execution code is inside the *setup()* method.



```
public class Example extends Agent{

        // Fields
        ......
        protected void setup() {
        ...
        // Not protected instructions
        System.out.println("Not Protected");

        // Protected instructions
        System.out.println ("Protected");
        ...
        }
}
```

Figure-6. Example code

Every phase of the process is described and presented by means of a particular example.

A. Phase I: Loading

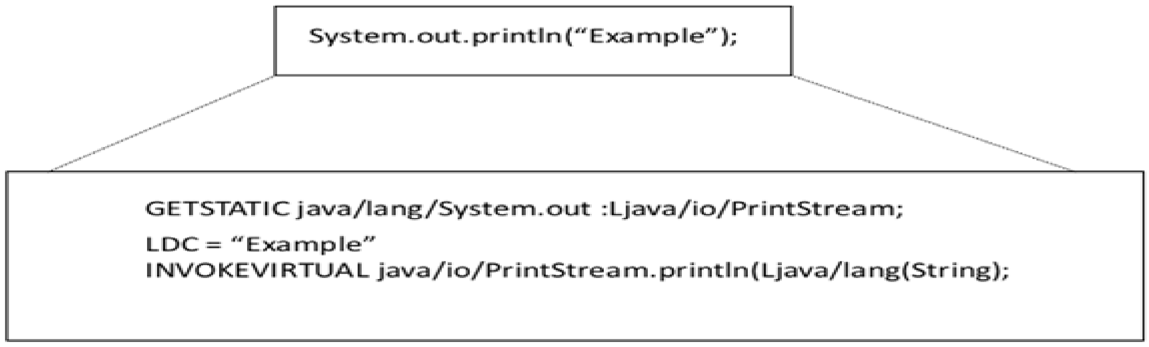
Loading is the first step in the production of secure agents in the process. Class files are then loaded and their content are analyzed. To this end, we have selected the set of non-secure agents and a deep study if every element (methods, fields, instructions, internal classes, etc.) that compounds that are scanned.

For this analysis, static component of the BCEL libraries are used. API from the library provides all required methods to load a .class file and the automatic generation of the structure as an output class file [10]. Many iterations of this process are repeated until, for each of the elements from the file class (methods, fields, internal classes, instructions, etc.), a recently modeling object is created and this is used as handler.

One limitation found was that every element generated with BCEL component is read only one. Therefore, it is important to save information for each elements. Annotation mechanism provided the functionality to save this information. Next, we created classes inheriting from BCEL classes, these new classes included all the useful information for the next phases because we manually inserted.

We have shown the particular case of instructions in the analysis process. However, we are aware of this is a more complex case than the rest of the elements. Classes files include a section dedicated for class methods, we can found bytecode instructions among other elements inside. Nevertheless, we consider these instructions are useless when they are executed separately. These has a strong dependency on the previous one and the next one. The relationship is stated between a java instruction and the set of instructions in bytecode. This fact lead us to decide grouping them in sets corresponding to a java instruction ended with ";".

Once all agents are loaded in the system we start the setting phase. We notice the importance of this phase since includes the actual description of security we pretend for our final protected MAS. During this phase we describe the protection links involved and how are these established. The information of this phase is very relevant since elements to be protected are chosen in this precise instant.



```
System.out.println("Example");
```

```
GETSTATIC java/lang/System.out :Ljava/io/PrintStream;
LDC = "Example"
INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang(String);
```

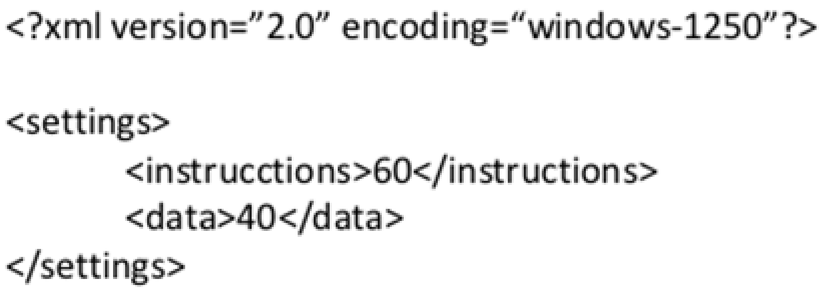
Figure-7. Matching

B. Phase II: Setting

Setting phase of this process is the simplest of the three and the easiest to applied. This stage enables establishing security specification parameters for the creation of the new secure agents. From all existing parts of a JADE agent, there are two elements that is mandatory to protect instructions and data.

Information required to determine the security degree is indicated in the percentage of instructions and data to be protected. *SecureAgentCon- figuration* class is responsible to model all this information. We have implemented two different alternatives have been described for establishing those security parameters. The first method is needed to specify for each of the loaded agents (first phase) which are the security parameters in the system. This fact includes the selection of every agent and the insertion of the percentage of instructions and selection of fields to protect to actually compute the protection. The second method is more complex due to the number of agents loaded and the number of testing proofs to perform. For this reason, we have implemented an optional alternative that enables the application of a security template to every agent using an external XML file. Structure of this template is variable according to many different parameters extracted from the agent system topology, as a complete template catalogues are out of the scope of this paper, we have included three basic templates that fits to a huge number of cases and we propose a customized template for particular cases. In figure 8, we present an example of the format of a setting file.

This example shows the percentage of instructions and data as to be protected. We notice the importance of considering that there are two different cases in which the percentage value is different for instructions and data and how we applied for each of them can alter the final result in the analysis. Next step is in charge of select those protection links, these are meaning to indicate the protection to establish among agents in output MAS. Likewise, it is not required to implement this action manually since there an automate mechanism has been developed as part of the set of assisting tools. In example, we only have two agents, for this simplified case one protects the other and vice versa, final result is really simple but it can be gradually

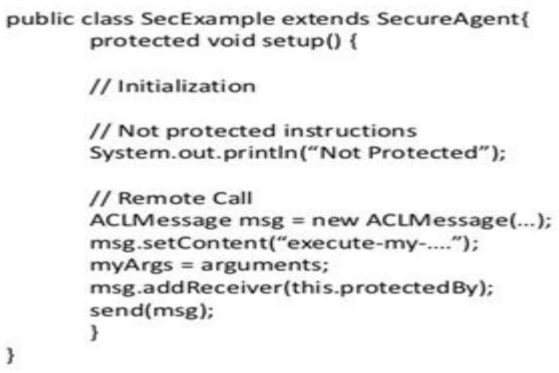


```
<?xml version="2.0" encoding="windows-1250"?>

<settings>
        <instrucctions>60</instructions>
        <data>40</data>
</settings>
```

Figure-8. XML file format

C. Phase III: Protected System Generation

Protected system generation phase takes as input the in- formation collected from both previous stages to build the protected system. Several considerations are taken to guarantee system integrity. Two new classes are generated at least, one of them is related to new secure agent with its public code (data and instructions) and the other with the private code, as we pointed out. Additionally, to these classes, as many internal classes contain the original agent, then is required the creation of some new classes. These are precompiled classes that are generated taking advantage of BCEL dynamic component. This part of the BCEL API facilitates tasks related to the creation class files skeleton and, depending on the security parameters settings in the previous phase, the original code is distributed among different parts. Figure 9 depicts the content of an example with the public part of a new protected agent based in the example in figure 10. We notice that the class code has been modified as we below described:



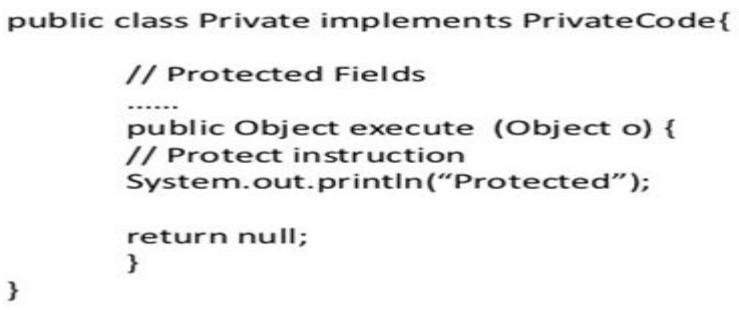
```
public class SecExample extends SecureAgent{
          protected void setup() {

          // Initialization

          // Not protected instructions
          System.out.println("Not Protected");

          // Remote Call
          ACLMessage msg = new ACLMessage(...);
          msg.setContent("execute-my-....");
          myArgs = arguments;
          msg.addReceiver(this.protectedBy);
          send(msg);
          }
}
```

Figure-9. Secure Agent

- Recently created class is in inheritance with *Secure Agent* class.
- A recently initialization section was included to setup methods for define those agents that plays protector and protected roles respectively in every relation established.
- At this point non protected code is included.
- Then requests to remote code execution are included. For each code section protected it is necessary to include those instructions needed to set arguments, invoke calls, halt and get results when needed.

A recently class is generated for protected code. This class is in charge to implement Private Code interface. We notice the importance of this new class to include:

- Those protected fields (when they are required by user).
- The "execute ()" method that contains the protected code in separated sections. The information required to know which sections execute is included 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 Private implements PrivateCode{

          // Protected Fields
          ......
          public Object execute  (Object o) {
          // Protect instruction
          System.out.println("Protected");

          return null;
          }
}
```

Figure-10. Private code

5. ONGOING WORK

The Static Mutual Protection strategy can be successfully applied to many different scenarios. However, there are scenarios where it is not possible to foresee the potential interactions between the agents; where the agents are generated by different parts, or involve very dynamic multi-hop agents. In these cases, the Static Mutual Protection strategy will be difficult or impossible to apply. In the Dynamic Protection Strategy each agent is able to execute arbitrary code sections on behalf of other agents in the society. As shown on top of figure 11, each agent includes a public part, an encrypted private part and a specific virtual machine similar to the one described in Maña, et al. [6]. 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 11. We illustrated how in the first exchange ag1 plays the role of protected agent, while ag2 plays protecting agent role (secure coprocessor). While the exchange is actually taking place, 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. We claim the attention of the scalability of this scheme since only a few agents (one in most cases) are involved in the protection of any other agent and can be applied to cases with disregard of the number of agents involved.

Henceforth we have put efforts on tools for the administration of the static mutual protection scheme, but we have considered as next step in the road of this research to face how to build tools for the implementation of dynamic mutual protection scheme.

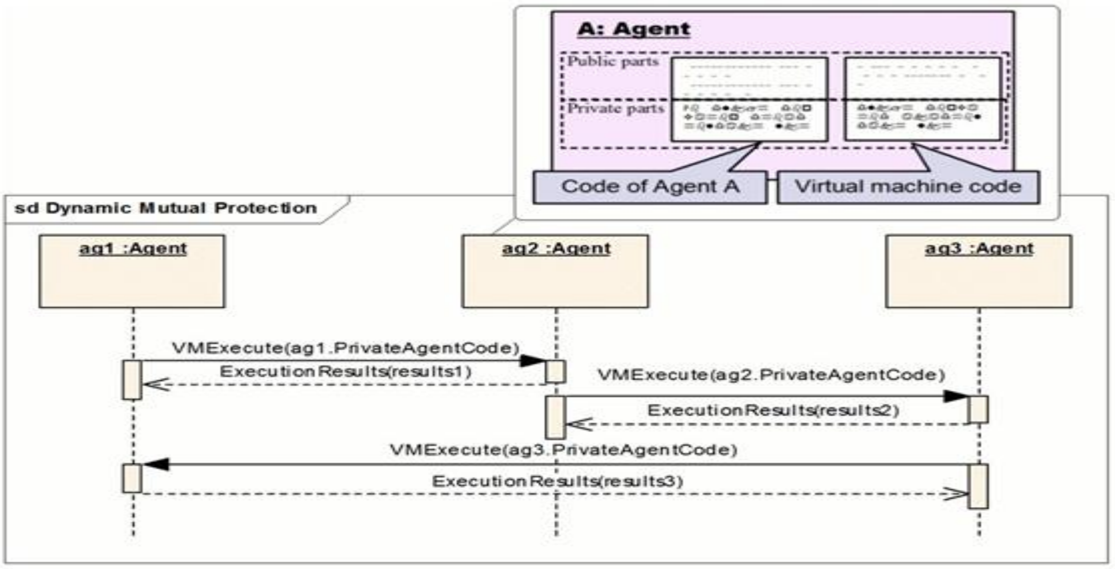

A: Agent
Public parts
Private parts
Code of Agent A
Virtual machine code
sd Dynamic Mutual Protection
ag1 :Agent
ag2 :Agent
ag3 :Agent
VMExecute(ag1.PrivateAgentCode)
ExecutionResults(results1)
VMExecute(ag2.PrivateAgentCode)
ExecutionResults(results2)
VMExecute(ag3.PrivateAgentCode)
ExecutionResults(results3)

Figure-11. Dynamic Mutual Protection agent Structure - The inclusion of a Java Virtual Machine

We are aware of the complexity of this proposal in terms of real implementation. This requires the inclusion of a small java virtual machine, but this java virtual machine is required to be trusted. Restrictions are not easily achievable but we ongoing work includes this development.

6. CONCLUSION

This paper provides a complete methodology based on the protected computing approach for multi-agent systems protection. In particular, this approach proposed a solution based on the mobile agent systems since they propose a more interesting challenge. This methodology covers from conceptual level of abstractions, establishing the foundations of the methodology to implementation details. This contribution provides some graphic tools to easily implement the mutual protection among agents by means of a friendly interface. We expect that this set of tools open up a new field of research that is concerned with performing a number of analytical and statistical studies about the kind of protection to implement, depending on the system requirements.

Funding: This study received no specific financial support.
Competing Interests: The author declares that there are no conflicts of interests regarding the publication of this paper.

REFERENCES

[1] A. Man˜a, A. Mun˜oz, and D. Serrano, "Towards secure agent computing for ubiquitous computing and ambient intelligence," *Computer Science*, vol. 4611, pp. 1201–1212, 2007. *View at Publisher*

[2] C. Hewitt and H. Baker, "Actors and continuous functionals, MIT-LCS-TR-194." Retrieved from http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-194.pdf, 1978.

[3] H. S. Nwana, "Software agents: An overview," *Knowledge Engineer- ing Review*, vol. 11, pp. 205–244, 1996. *View at Google Scholar* | *View at Publisher*

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

[5] P. Siani and B. Boris, *Trusted computing platforms: TCPA technology in context.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[6] 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, pp. 99–112, 2004. *View at Google Scholar* | *View at Publisher*

[7] The Apache Software Foundation, "BCEL (Byte Code Engineering Library)." Retrieved from http://jakarta.apache.org/bcel, 2006.

[8] S. Chiba, "Javassist (java programming assistant). Sun Microsystems, Inc." Retrieved from http://www.cgs.is.titech.ac.jp/projects/index.html, 2009.

[9] OW2 Consortium, ASM. Retrieved from http://asm.ow2.org/, n.d.

[10] T. Lindholm and F. Yellin, *The JavaTM virtual machine specification*: Sun Microsystem, Addison-Wesley, Copyright 1997 Sun Microsystems, Inc, 1999.