Enforcing Fine-grained Authorization Policies for Java Mobile Agents

Giovanni Russello Changyu Dong Naranker Dulay
Department of Computing
Imperial College London
South Kensington London, SW7 2AZ, UK
{g.russello, changyu.dong, n.dulay}@imperial.ac.uk

Abstract

The Mobile Agent (MA) paradigm advocates the migration of agent code to achieve computational goals. MAs require an executable environment on hosts where mobile code can be executed. The execution of foreign code on a host raises security concerns for both the agent and the host. In [1] it has been recognized that most of the approaches for providing security in MA suffer from a limitation of expressing complex security requirements. Thus, approaches have been proposed that introduce the use of a policy language for specifying security policies to control MA's access to host resources. With this paper, we outline a framework where security policies can be uniformly specified for protecting both MAs and host resources.

1 Introduction

Current distributed systems involve a large number of applications which require a variety of security mechanisms to fulfill their needs. In particular, the use of MAs introduces new challenges and security threats that need to be carefully considered [4]. On the one hand it is necessary to protect the host environment where agents are executed from malicious and buggy mobile code. It is necessary to protect the host information and resources from illegal accesses and over-consumption by incoming mobile code. On the other hand, it is necessary to protect the state and behavior of the mobile code from tampering or misuse by malicious hosts. Additionally, it would be desirable that hosts provide QoS-like guarantees on the resources that are made available to mobile code. For instance, if an agent moves to a given host then the host has to make sure that enough memory and processor time is given to the agent for a correct execution.

Most of the research in providing security frameworks for MAs has concentrated only on the first part of the problem. Sandboxing techniques and type-safe languages can be used to rigidly control the interaction between the mobile code and the host. The rigidity of such approaches can be overcome if a language-based approach is used for specifying authorization policies. For instance, in [1] an approach was adopted where the Java security architecture was integrated with a policy language. However, all of these approaches focus on controlling the MA accesses on the host resources.

In this paper, we propose a framework where it is possible to specify policies for both the MAs and the host resources. The framework is currently implemented for Java based MAs. Policies are expressed using an extension of the Ponder language [2] and are enforced using a Ponder interpreter. In our approach, the enforcement of policies is done transparently to the MA code.

The contributions of this paper are twofold. First of all, we provide the description and implementation of a framework where security policies can be uniformly specified for both MAs and host resources. Secondly, the framework is independent from both the actual mechanism used for policy enforcement and the specific policy language.

This paper is organized as follows. Section 2 reviews previous research conducted on policy specification for MAs. In Section 3, we describe our syntax for specifying policies. We implemented our framework and its detailed description is provided 4. We conclude in Section 5 and provide some future directions of our research.

2 Background

Several policy-based approaches have been proposed for the specification of policies to control the interaction of agents. KAoS [14] is a collection of component-based policies and domain management services which provide support for mobile agent, grid computing and web services. KAoS relies on a DAML description-logic-based ontology of the computational environment, application context, and the policies themselves. It makes it possible to represent subjects, actions, and situation at multiple levels of abstrac-

tion and to dynamically calculate relations between policies and environment entities and other policies based on ontology relations. Rei [8] is a policy framework designed for pervasive computing applications, represents policies in a semantic language like RDF-S, DAML+OIL or OWL. Using a semantic language allows different systems to share a model of policies, roles and other attributes. The language is not tied to any specific application and it permits domain specific information to be added without modification. In LGI [10], policies specify which actions the agent has to enforce upon the receipt or sending of messages. Policies use a simple Prolog notation. It assumes that policies are interpreted by trusted controllers at each agent's site. Ponder [2] is a declarative, object-oriented language that supports the specification of several types of management policies for distributed systems. Ponder uses an object-oriented approach which allows users to define different types of policies to meet specific administrative and security management goals.

In [1] Ponder was used for specifying authorization policies for mobile code. The authors describe an extension of the access control mechanism provided by the Java security framework [6]. The extension consists of several modules that have been introduced to map authorization policies specified in Ponder into Java security structures. With the use of the Ponder language, it becomes possible to specify more complex policies. However, the Java security framework is limited to control resource access of the host. Preventing an agent from performing an operation or forcing the agent to reject the result of a request is out of the scope of the Java security framework.

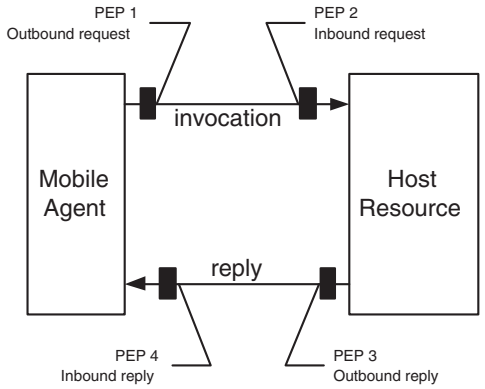

Figure 1. Policy enforcement points.

To fill the gap of the above approaches, we propose a framework where the Policy Enforcement Points (PEP) can be specified for both the agent and host resources. As shown in Figure 1, we specify four points of policy enforcement:

- PEP 1: at this point policies are enforced when the agent sends out a request to a (more generally to any local or remote host or agent) host resources. For instance, the agent is not authorized to invoke a resource of the host unless certain conditions are met. Such conditions could be contextual, such as time of the day or host location. Conditions can be defined on properties of either the agent or the target resource on the host. PEP 1 policy could be used to protect the privacy of the agent's data. For instance, the agent is authorized to invoke the host resource but data passed as parameters should be filtered to remove private or sensitive information. In other words, the enforcement of authorization policies at this point allows us to separately define and control the execution of operations by the agent. Only when certain assumptions hold, can the call be made. We named such policies *Subject Authorization (SA)* policies.

- PEP 2: this point is used for activating traditional authorization policies for access control on the resource. Policies are enforced when the host resource receives a request. We named this type of policies *Target Authorization (TA)* policies.

- PEP 3: this point allows the host to apply policies when the resource sends back the reply. For example, to remove sensitive data from the reply that is sent back to the agent. Just denying the agent the right to perform the operation is not sufficient to cover this case. The resource will provide information to the agent. However the resource administrator defines the conditions under which the information is can be given without compromising confidentiality. We named these policies *Target-Return Authorization (TRA)* policies.

- PEP 4: this point allows us to enforce policies when the agent receives the reply. PEP 4 policies can be used to protect the integrity of the agent from malicious or buggy data sent from the resource. We named such policies *Subject-Return Authorization (SRA)* policies.

Figure 1 shows an agent that is the initiator of a request. However, it could be the case that the agent provides some functionality to the host environment. If this is the case, then the agent becomes the target of an invocation. Therefore, PEP 2 and 3 are also used to enforce authorization policies on the functionalities exposed by the agent.

If specified at the application level, the enforcement points may look different for each application. Such points can be uniformly abstracted as method invocations when seen at the system level (e.g., at the level of the Java virtual machine) where we can intercept any method invocation (and also replies), and it is transparent to the application.

An intercepted method invocation or reply can provide most of the information for policy evaluation. For instance,

most access control policies-base their decisions on (subject, target, action) tuples. This information is included implicitly in the method call or the reply. In addition, the parameters of the method call and the return value of the reply can provide more information if needed. Our approach is independent from the policy language, as long as the language offers a syntax to express the types of policies required by each PEP.

In the following sections, we discuss in more detail the policy language and interpreter used in our framework.

```
SA auth+/- subject.action(p)→target
TA auth+/- subject→target.action(p)
TRA reply+/- subject→target.action(r)
SRA reply+/- subject.action(r)→target
```

Figure 2. Mobile Agent Authorization Policy Syntax.

3 Mobile Agent Authorization Policies

In our approach a positive *authorization policy* defines which *subjects* are granted the permissions to execute *actions* of a given *target*. We also support negative authorization policies. In our examples, subjects typically map to mobile agents and targets to host resources. However, MAs can be targets and host resources can be subjects. A negative authorization policy can be seen as a refinement of more general positive authorization policies. Negative authorization policies are also particular useful when permissions (in the form of a positive authorization) need to be removed to a group of subjects.

When dealing with policy based systems, it is unavoidable that conflicts arise in the set of policies. This issue is more acute in the case of agents migrating through different hosts. As a matter of facts, policy administrators cannot be aware of the policies that agents take along during their migrations. Conflict resolution is fundamental for policy based systems, as discussed in [9]. The study of conflict resolution is one main area of our future research.

The main contribution of this paper that differentiates our approach from previous research is that for a given action authorization policies are uniformly applied to subjects as well as to targets. Figure 2 presents the authorization policies that can be specified in our framework.[1] The keyword **reply**$\pm$ specifies that the authorization policy is to be applied on the reply of the action. In this case, the result of the action is explicitly indicated by the parameter `r` of the action.

[1] Although in the syntax we explicitly identify each type of policies, the position of the action in the policy self-explains whether the policy is to apply to a subject or a target.

In the following, we provide several examples of authorization policies for both MAs and host resources.

3.1 Examples of Policy Specifications

In this section we provide examples of policies that it is possible to specify using our approach. The policies that we consider are for mobile agent for healthcare applications.

Policy 1 shows a "refrain" policy that prevents a mobile patient agent requesting treatment to a medical service provided by the host when the medical service cannot provide a valid certificate signed by the National Health Service (NHS).

Policy 1 *Negative authorization policy for the patient agent to issue a request of a treatment to a medical service.*

```
auth- patientAgent.requestTreat()→MedService
      when !MedService.isNHSCertified()
```

Policy 2 is another negative authorization policy applied on the patient agent. However, this policy denies to the agent access to the treatment returned by the medical service when the returned treatment is signed by a GP that is not recognized by the NHS.

Policy 2 *Negative authorization policy for the patient agent to receive the result of a request issued to a medical service.*

```
reply- patientAgent.requestTreat(prescription)
            →MedService
       when !prescription.GP().isNHSCertified()
```

Positive authorization policies can be used for applying filters to the data that is supplied or returned. The filter is specified by using the **filter** keyword in the action clause. Filtering policies must be positive authorization because no transformation needs to be applied if the action is forbidden. Policy 3 shows a filtering policy for an agent of an employee. The agent of an employee has to provide to the GP of the company where the employee works her medical record. The data is stored on the data base of the company. For privacy reasons, the employee psychiatric data must be removed from her medical record. The policy applies a filter that nullifies the sensitive field from the record. The filter is executed before the action is performed.

Policy 3 *Filtering policy for an agent when providing sensitive data to a database on a host.*

```
auth+ employeeAgent.ins(record)→employeeMedDB
      filter myRecord.psych := NULL
```

It should be noted how the use of our framework realizes a complete separation of concerns [11]. In fact, all the

details about checking the credentials of the target, the target's reply, and the application of filters on sensitive data are specified outside the logic of the application. These details are isolated and captured in the policy specification.

Policy 4 provides an authorization policy for a nurse agent that has to perform accesses on the patients' medical records in a hospital. According to this policy, a nurse agent can access the medical records of a patient when the nurse is on duty on the ward where the patient is assigned.

Policy 4 *Authorization policy for granting access right to a nurse agent on the medical records of patients in a hospital.*

```
auth+
  nurseAgent→medicalRecordDB.accessFor(patient)
  when (nurseAgent.ward = patient.ward)
```

The policy interpreter organizes the entities (agents and resources) that are specified in a policy in hierarchical domains of objects. Domains can be used to specify the subject and target in a policy. When an agent arrives in a host, the local policy interpreter authenticates the agent and adds it in a local domain. The domain where the agent is added depends on the agent's credentials. Using this approach, we can specify the previous authorization policy in terms of domains as shown in Policy 5. In this case, agents representing hospital personnel and patients are organized in domains. Each domain represents the different wards of the hospital. When the nurse starts her shift in a ward, her agent is inserted in the appropriate ward domain (ward10 in our example).

Policy 5 *Authorization policy for access control based on the domain location of the nurse and patient agents.*

```
nurseAgent in /hospital/personnel/ward10/
patientAgent in /hospital/patients/ward10/
auth+ nurseAgent.getRecord()→patientAgent
```

More details on how this policy is enforce are provided in Section 4.2.1.

4 Implementation

This section discusses details of the implementation of our framework. The actual prototype is built mainly in Java, although our framework is conceptually independent of the actual programming language. Java was mainly chosen for a faster integration with our existing policy interpreter.

4.1 MA Migration Details

This section provides insights on some aspects related to the migration of a mobile agent in our framework.

Figure 3 shows the migration of an agent to its destination host. In particular, the figure shows that the unit of

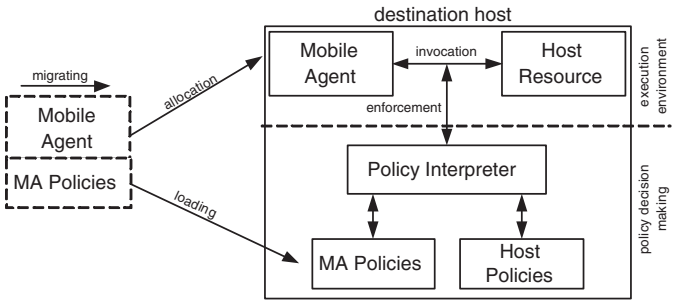

Figure 3. The migration of an agent and its policies.

mobility is composed by the agent logic (that is the executable part) and the agent policies. On arrival on the host, the agent logic is inserted in the executable environment where it can interact with the host resources. The agent policies are loaded by the host's interpreter in its local data structure. When the agent interacts with the resources in the host, the interpreter enforces the policies as required.

The basic assumption in our approach is that the host policy interpreter where an agent moves is *trusted*. A host is trusted when it can provide to the agent credentials to guarantee that the execution will be carried according to the constrains specified by the agent's policies. Which are such credentials and the specific method that an agent has to use for building enough trust on the host integrity is the subject of our future research.

We are also considering a more radical approach which wraps a policy interpreter around the mobile agent.

4.2 Implementing the Policy Enforcement Points

In this section, we describe the architecture of our prototype that we built to demonstrate the feasibility of our approach.

Crucial to our approach is the realization of a PEP mechanism such that (i) it supports a fine-grained level of enforcement point specification and (ii) it is completely transparent to the application logic.

Several techniques could be used for realizing the PEP mechanism of our approach. For instance, using the standard Wrapper Pattern, the agent and resource code is *wrapped* by a piece of code that intercepts all the inbound and outbound calls to and from the component. Each time a call is intercepted, the wrapper passes the necessary information to the Policy Interpreter to activate a policy.

A Java-based solution that supports the wrapper pattern is the Java Management Extension (JMX) [5]. In JMX, the agent and the resources must be managed by a Man-

aged Bean (MBean). A MBean is a special Java bean that exposes via a standardized interface (defined by the JMX specification) attributes and methods of the resource that it manages. MBeans have the capability to emit notifications upon certain events. Such events could call the PEPs in our framework.

Another Java-specific solution is based on the use of the Java Virtual Machine Tool Interface (JVMTI) [7]. JVMTI provides an interface that can be used by user code to control and monitoring Java applications. In JVMTI such user code is called an agent. To avoid confusion, we will refer to it as *ti-agent*. Ti-agents use the functionality exposed by JVMTI to be notified when events occur in the application, and to query and control the application during execution. Among the events that a ti-agent can intercept there are those that capture when the execution enters and exits a method. JVMTI allows ti-agents to retrieve information regarding method call, such as object type of the caller and the callee, the parameter values passed in the method invocation, and the value that the method returns. Given the fine-grained control and monitoring capability, and the fact that it is not required to change any application code, we implemented the PEP mechanism using JVMTI.

An alternative approach would be to use Aspect-Oriented Programming [3]. Such an approach is used by Verhanneman, *et al.* in [13]. They use Java Aspect Components [12] to implement a wrapper to intercept method calls from the caller to callee and to enforce policies as required. This technique requires that the agent code is modified with the injection of aspect-specific code at the host side. Crucial in AOP is the specification of where the aspect code must been injected. In JAC, this is done transparently to the application using descriptor files. The descriptor provides those points to the JAC middleware that then weaves the aspect code with the application code. This is completely transparent to the application.

We decided to use JVMTI mainly for two reasons. The first reason is that JVMTI is a standard Java tool. Using an aspect oriented approach requires us to rely on non-standard Java compilers and tools that are not always so thoroughly developed. The second reason is that the use of JVMTI does not requires any changes in the application code.

It should be noted however, that the design of our framework is independent of the actual mechanism used for implementing a PEP. In principle, all the above approaches could be used to implemented a PEP with enough capabilities that would enable our framework to function as required. This has the main advantage of allowing our framework to enforce policies across systems implemented using different technologies.

Figure 4 gives an overview of our architecture. Given that the JVMTI and policy interpreter modules were already available, the only modules that we implemented are the `ti_a.c` and the `TIAgent.java`.

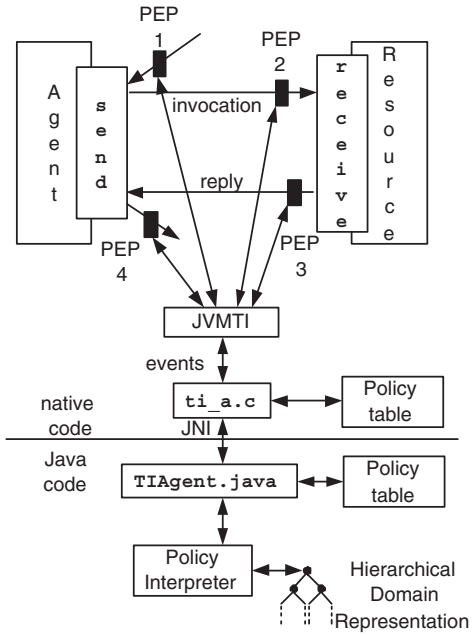

Figure 4. The modules implemented in our framework to provide a complete control over authorization.

The `ti_a.c` is the ti-agent written in C that is injected in the JVM at start-up time as a command line option. Once the `ti_a.c` has been loaded into the JVM, it registers the notification callbacks for JVMTI events. In particular, the agent registers for `JVMTI_EVENT_METHOD_ENTRY` and `JVMTI_EVENT_METHOD_EXIT` to intercept when the execution flow enters and exits a method, respectively. For example, Figure 4 shows an agent that is executing its own method `send` to invoke the method `receive` of the resource. In this example, `ti_a.c` is notified by the JVMTI notification system when the following events occurs:

- entering method `send` (event 1),
- entering method `receive` (event 2),
- exiting method `receive` (event 3),
- and finally exiting method `send` after the call to `receive` (event 4).

Such events are one-to-one mapped to the PEPs that our framework requires.

Method entry and method exit events are notified every time a method is called. This requires that `ti_a.c` that has to filter out all those events relative to methods for which a policy has not been specified. This means that the `ti_a.c`

needs to be interfaced with the policy interpreter because the interpreter organizes the policies in its domain hierarchy. This task is fulfilled by the `TIAgent.java` that provides to the `ti_a.c` information on the policies loaded by the interpreter. The information that the `TIAgent.java` extracts from a loaded policy is concatenated to from a so called *signature*. A signature is the concatenation of the following elements:

- the action that is the name of the method to be invoked,
- the target that is the host object's full class name that contains the method,
- and the subject that is the MA object's full class name that invokes the method.

The signatures are passed to the `ti_a.c` that can use them to intercept the appropriate events. The `ti_a.c` and `TIAgent.java` maintain policy tables where policy signatures are stored. This enables us to minimize the dependencies of our framework from the policy representation used by the specific interpreter.

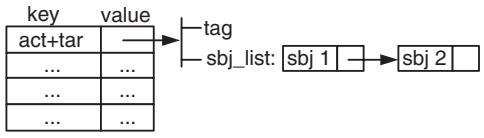

Figure 5. The policy table of the `ti_a.c`.

The policy table maintained by the `ti_a.c` is a hash table represented in Figure 5. The key column contains the concatenation of the action and target. The value column contains structures with the following fields:

- `tag`, that can have three values: 0 for an authorization policy on entering the method, and 2 for an authorization policy on exiting the method;
- `subject_list`, a linked list subjects for which an authorization policy is specified.

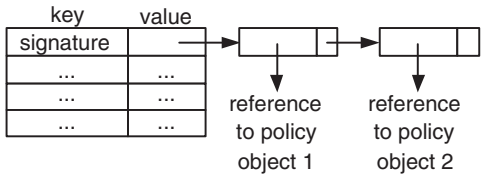

Figure 6. The policy table of the `TIAgent.java`.

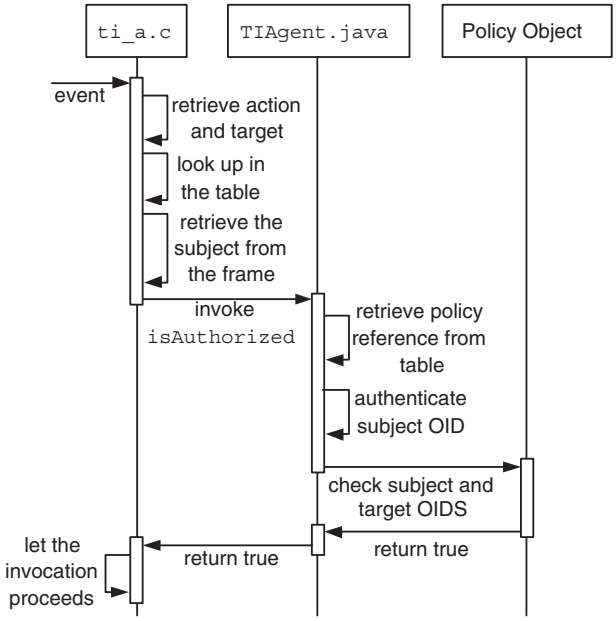

Figure 7. The message sequence chart of the activation of an authorization policy.

The policy table of the `TIAgent.java` is represented in Figure 6. It is a hash table where the key column contains signatures. The value column contains a linked list of references to the actual policy objects as represented by the policy interpreter.

The main reason of having two separate instances of the table is to lower the overhead of JNI calls between the `TIAgent.java` and `ti_a.c`. For each event, the `ti_a.c` retrieves the method name (action) and the full class name (target) of the object that contains the method. Using this information, the `ti_a.c` looks up a matching key in its table. If a match is found, the process to activate the proper policy is started. Otherwise, the `ti_a.c` just ignores the event. By having its local table, the `ti_a.c` can perform locally the search instead of having to use a costly JNI call to the `TIAgent.java`.

Whenever the interpreter updates the set of policies (i.e., for loading new policies, disabling or enabling policies), the `TIAgent.java` is notified that an update was performed. This triggers the update on its own table. As soon as the `TIAgent.java` changes its table, the `ti_a.c` intercepts the changes and update its local table, accordingly.

In the following, we explain with an example the details of the authorization of a method invocation.

4.2.1. An example of Authorization Policy Enforcement. Let us consider one of the policies presented above. In particular, let us consider the authorization policy defined in

Policy 5

The policy specifies that the nurse agent is authorized to get the records of a patient agent when the nurse is on duty on the same ward where the patient is assigned.

Figure 7 shows a message sequence chart of the policy execution for authorizing the nurse access. When the execution flow enters the method `readRecord` of the patient agent, JVMTI raises an event captured by the `ti_a.c`. The `ti_a.c` retrieves the method name (action) that is being invoked and the agent's full class name (target) where the method belongs to. This information is used to look up in its policy table for a matching element. Once the element is retrieved, the `ti_a.c` uses the subject list to identify which agent is invoking the method. For each element in the subject list, the `ti_a.c` scans the frames of the current execution stack (provided by the JVMTI) for a match. If no matching subject is found, then the `ti_a.c` has to notify the `TIAgent.java` that an unauthorized access is being attempted and appropriate actions should be taken (for instance, throwing an exception).

Once the nurse agent is found in the current execution stack, to complete the authorization it is necessary to authenticate the subject.

As for the authentication, in the current implementation of our prototype, we use the authentication mechanism provided by the policy interpreter. As we said, the policy interpreter maintains a domain structure populated with managed objects. Each managed object represents a component that needs to be managed. In this case, a component can be either an agent or a resource. When a new component is discovered, the policy interpreter checks the components's credentials. If the component is authenticated, the policy interpreter instantiates the correspondent managed object in its domain structure. Each managed object is uniquely identified by an ID generated by the interpreter, called *OID*. Thus the subject can be authenticated if it provides a valid OID. This holds also for the target.

The `ti_a.c` retrieves the subject and target OIDs. Afterwards, the `ti_a.c` invokes the `TIAgent.java`'s method `isAuthorized` (using JNI) passing the following information:

- the signature, that is the concatenation of subject and target class name followed by the method name,
- the subject OID,
- the target OID,
- and an array containing the parameters' values of the method invocation (in this case the array is null because no condition clause is specified in the policy).

Using the signature, the `isAuthorized` method retrieves from the policy table the linked list of policy references. Multiple policies could be defined for the same combination of action, target, and subject. All of these policies are contained in the list. The method goes through the list until a policy that authorized the execution of the operation is found.

A policy authorizes an action if the subject and the target OIDs are valid. This also means that the OIDs must be contained in the specified domains. In this case, the nurse agent OID must be contained in the ward10 domain for the hospital personnel, and the patient agent OID must be in ward10 domain for the patients. When the policy that authorizes the action is found, then the `isAuthorized` method immediately returns to the `ti_a.c` that the invocation can proceed. Otherwise the `isAuthorized` method returns false to the `ti_a.c` that does not allow the invocation.

5 Conclusions and Future Work

In this paper, we presented a framework for authorization policy enforcement for mobile agent applications. The main contribution of our approach is that authorization policies can be used to protect both the mobile agents and he host resources. This fills the gap of previous approaches where policies could be specified and enforced on the host resources. This paper also described our implementation of the framework.

As future work, we foresee working in combining a Trust Management System (TMS) with access control. Since the interception mechanism is independent from the actual authorization model, we can easily integrate a TMS in our suite. The TMS will take decisions on whether a given entity should be granted authorization based on the trust level that the owner of the accessed resource puts on the entity. This level can change over time, providing a very flexible framework in comparison with the yes-or-no approach of classical security models.

Another main area of future research is the introduction in our framework of conflict resolution strategies to automatically resolve conflicts that could arise between policies.

Acknowledgments

This research was supported by the UK's EPSRC research grant EP/C537181/1 and forms part of the CareGrid, a collaborative project with the University of Cambridge. The authors would also like to thank the members of the Policy Research Group at Imperial College for their support.

References

[1] Corradi et al. A flexible access control service for Java mobile code. In 16th Annual Computer Security Application Conference (ACSAC'00)s.

[2] N. Damianou, N. Dulay, E. Lupu and M. Sloman. "The Ponder Policy Specification Language." *In Proc. 2nd IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 18–38, 2001.

[3] R. E. Filman and D. P. Friedman. "*Aspect-Oriented Programming is Quantification and Obliviousness*." *Workshop on Advanced Separation of Concerns*, OOPSLA, October 2000.

[4] W. Jansen and T. Karygiannis. "Mobile Agent Security." *NIST Special Publication* 800-19, National Institute of Standard and Technology, 2000.

[5] Java Management Extension Specifications. http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html

[6] Java Security White Paper. http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf

[7] JVM Tool Interface http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html

[8] L. Kagal, T. Finin and A. Joshi. "A policy language for a pervasive computing environment." *In Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 63–74, 2003.

[9] E. Lupu and M. Sloman. "Conflicts in Policy-Based Distributed Systems Management." *IEEE Transaction on Software Engineering*, pp. 852–869, Vol. 25, No. 6, 1999.

[10] N. H. Minsky and P. Pal. "Law-Governed Regularities in Object Systems - Part 2: A Concrete Implementation." *Theory and Practice of Object Systems (TAPOS)*, John Wiley. 2, 1997.

[11] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, 15(12):1053-1058, December 1972.

[12] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. "*JAC: A Flexible Framework for AOP in Java*." In *Reflection'01*, volume 2192 of Lecture Notes in Computer Science, pages 1–24. Springer-Verlag, September 2001.

[13] T. Verhanneman, F. Piessens, B. D. Win and Wouter Joosen "Uniform Application-level Access Control Enforcement of Organizationwide Policies." *In Proc. 21st Annual Computer Security Applications Conference*, pp. 431–440, 2005.

[14] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni and J. Lott. "KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement." *In Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 93–96, 2003.