Fine-grained and History-based Access Control with Trust Management for Autonomic Grid Services *

Hristo Koshutanski
CREATE-NET
Via Solteri 38, Trento 38100, Italy
hristo.koshutanski@create-net.org

Fabio Martinelli, Paolo Mori, Anna Vaccarelli
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Via Moruzzi 1, Pisa 56124, Italy
{fabio.martinelli,paolo.mori,anna.vaccarelli}@iit.cnr.it

Abstract

Grid technology provides an Internet-wide environment where a very large set of entities share their resources. The main feature of a Grid environment is that resource providers belong to distinct administrative domains each with its own security policies and enforcement mechanisms. Even more, service providers and entities, exploiting the Grid infrastructure, typically have incomplete information about each other mainly because each administrative domain manages its policies and resources with high degree of autonomy.

Thus, controlling access to Grid resources has become a major security issue and a Grid infrastructure has to provide a proper set of mechanisms and tools that allow for a fine-grained and history-based access control management.

This paper proposes a comprehensive access control and enforcement framework for Grid computational resources. The framework is based on a behavioral model that defines fine-grained and history-based monitoring and on a trust management model that provides access decisions and proper access rights management.

The framework provides dynamic and context-aware access control enforcement by generating temporal credentials at run time while user's applications are exploiting Grid's resources.

1 Introduction

Having a look on the IT sector over the last decade middleware was a trendy word used to describe integration of distributed resources. Nowadays new paradigms for lightweight integration of enterprise resources have emerged. Among them we find Web Services technologies as a unified way of describing (WSDL), discovering (UDDI) and invoking (SOAP) resources in a heterogeneous and platform independent manner. Moving up in the paradigm from intra-enterprise to inter-enterprise integration of business resources we find virtual organizations (VOs) to result.

Grid or Grid infrastructure services [4] have emerged as a technology for providing an environment where geographically dispersed participants share their resources. Those participants cooperate with each other according to particular business needs and form VOs. The nature of the participants could be various, e.g. universities, research centers, companies or private members. Thus, Grid environment comprises resources shared by entities belonging to different administrative domains where each entity adopts different security policies and mechanisms with a high degree of autonomy.

Partners involved in a VO typically have incomplete information about their counterparts and, as so, they have no a priori established trust relationships. Autonomic Grids pose new security challenges of both access control and behavioral control on the computational resources.

There is a need for a security framework that properly controls and enforces the behavior of users (applications running on behalf of users) when using Grid resources. Furthermore, since we are in an autonomic scenario, the framework should provide proper access rights management and trust relationships establishment between resource providers and users.

1.1 Paper Contribution

In this paper we propose a comprehensive framework for fine-grained and history-based access control for autonomic Grid resources. The framework monitors the behavior of

*This work is partially funded under the IST program of the EU Commission by the STREP-project "ONE" (INFSO-IST-034744), the NoE-project "OPAALS" (INFSO-IST-034824), the STREP-project "S3MS", the FET-project "SENSORIA" and the STREP-project "GRIDTrust"

applications running on Grid computational resources according to local, site dependent, security policies. On each operation performed by an application, the framework determines whether the operation is allowed according to a behavioral policy and if access decision needs to be taken then the framework enforces proper access rights management according to user's credentials and local access policies.

The proposed framework integrates two main concepts:

- fine-grained and history-based monitoring and enforcement of Grid resources with
- access control management for proper trust relationships establishment.

The structure of the paper is the following. Section 2 describes security problems in Grid environments and approaches that have been adopted to solve them. Section 3 presents the model for fine-grained and history-based behavioral control. Section 4 introduces the model on interactive access control and access rights establishment. Then, Section 5 presents and discusses the architecture of the overall access control enforcement. To make the contribution of the paper clearer, Section 6 shows an example of the security polices and Grid usage scenario. Finally, Section 7 concludes the paper and draws possible future work.

2 Security in Grid Computing

Security is a fundamental aspect in a Grid environment mainly because such an environment instantiates interactions between a large number of participants among which no trust relationships exist. The reason for that is that those participants belong to distinct administrative domains each with its own security policies and mechanisms.

The security requirements of the Grid environment include authentication, delegation, authorization, privacy, message confidentiality and integrity, trust, policy and access control enforcement [6, 8, 15]. All these requirements have to be addressed by the Grid security architecture in order to define a secure environment. However, the current Grid toolkits does not fully implements all the requirements especially those covering authorization, access control and trust management.

Globus [4] is the widely used Grid toolkit nowadays. The Globus Security Infrastructure (GSI) provides a coarse-grained access control model [9]. The Globus Resource Allocation Manager authenticates a grid user by exploiting an identity certificate issued by a trusted VO's Certification Authority. Then, the only controls that are performed on the user's actions are those defined by the access control model on operating system level of the grid resource. Hence, from the authorization point of view, this access control model is not expressive enough to define a fine-grained resource control and to support trust management.

Several attempts have been made to improve the GSI. One of them is the Community Authorization Service (CAS) [5, 16] that was integrated within the Globus toolkit. CAS determines which actions each grid user is allowed to do. This approach requires CAS-enabled grid services that are able to understand and enforce the security policies issued by a CAS server in a form of certificate.

There are a number of systems for distributed access control proposed in the literature and we refer to [10] for a comprehensive survey. The most cited and discussed ones are PERMIS [3], Akenti [19] and XACML [21] specification.

The general idea behind PERMIS and Akenti is that the information needed for an access decision, such as identity, authorization, and attributes is stored and conveyed in certificates, which are widely dispersed over the Internet (e.g., LDAP directories, Web servers etc.). The authorization engine has to gather and verify the certificates relevant to the user's request and then to evaluate them against the access policy in order to take a decision. Based on those systems Keahey and Welch [9, 18], and Stell, Sinnot and Watt [17] proposed fine-grained authorization solutions for the Grid environment.

However, most of access control systems provide:

- static access control decisions (not history- and context-aware),
- static, a priori fixed, trust relationships establishment (no interactive and dynamic access management of credentials).

The goal of this paper is to synthesizes the above aspects into one comprehensive access enforcement framework.

3 The Fine-grained and History-based Behavioral Model

The model described in this section improves the security of grid computational resources. Grid computational resources are typically executed by applications on behalf of (unknown) grid users. These applications perform actions on resources that are history- and context-sensitive and, as so, each action modifies the state of the resources. Therefore it is necessary to preserve the integrity of Grid resources by fine-grained monitoring of actions performed on them.

The approach is based on a fine-grained and history-based model introduced in [2]. The monitoring is fine-grained because instead of considering the execution of an application as a single atomic action we split down the monitoring on the basic operations performed by an application

during its execution. In particular, we refer to the system calls that a grid application invokes on the operating system level. Hence, an application execution corresponds to a sequence of actions that form the behavior of the application.

During an application execution each action changes the state of the application that reflects its execution further. Thus the notion of history must be taken into account.

In our model actions that an application is allowed to perform depend on the behavior of the application from previous actions. In fact, during an application execution we determine for each action *a* that the application tries to perform, whether *a* is included in the current set of allowed actions that, in turn, is determined according to the actions that have been already performed from the beginning of the execution. The main advantage of the history-based monitoring is that it defines dependencies among the actions that an application is allowed to perform. In this way we can state that a given action *a* is not allowed during the whole execution of the application, but only when some other actions that *a* depends on have been already executed.

We introduce a *behavioral policy* in order to define the set of actions and dependencies among them that applications must conform to. The behavioral policy defines traces of executions (sequences of actions) that are allowed to be performed by applications when using local Grid resources.

As an example, if the policy includes the sequence $a.b.c$, where a, b and c are actions and the dot represents the sequential operator, the application can execute the action b only if a has been already executed, and it cannot execute c if only a has been executed. To define complex policies, a set of operators is required to compose the actions, i.e. the system calls, in the most proper way to represent a given behavior. The following grammar shows the operators to define a behavioural policy:

$$P ::= \alpha.P \parallel p(\vec{x}).P \parallel \vec{x} := \vec{e}.P \parallel PorP \parallel Ppar_{\alpha_1,..,\alpha_n} P \parallel Z$$

where P is a rule, α is an action in a set Act, $p(\vec{x})$ is a predicate, $\vec{x}$ are variables and Z is a constant process definition $Z = P$. The informal semantics is the following:

- $\alpha.P$ is the *sequential operator* that represents the possibility of performing an action α and then behave as P;
- $p(\vec{x}).P$ behaves as P in the case the predicate $p(\vec{x})$ is true. $p(\vec{x})$ can also consist of a sequence of predicates that are included in square brackets;
- $\vec{x} := \vec{e}.P$ assigns to variables $\vec{x}$ the values of the expressions $\vec{e}$ and then behaves as P;
- $P_1 or P_2$ is the *alternative operator* that represents the non deterministic choice between P_1 and P_2;
- $P_1 par_{\alpha_1,...,\alpha_n} P_2$ is the *synchronous parallel operator*. It expresses that both P_1 and P_2 policies must be simultaneously satisfied;
- Z is the constant process. We assume that there is a specification for the process $Z = P$ and Z behaves as P.

The rigorous semantics is defined in [14] through semantics rules. Derived operators may be considered, such as $P_1 par P_2$, that is the *parallel operator* and that represents the interleaved execution of P_1 and P_2, and $i(P)$, which is the iteration operator. Informally, $i(P)$ behaves as $P\ x$ times for any value of x. Also the policy sequence operator $P_1; P_2$ may be implemented using the policy language operators (and control variables) (e.g., see [7]).

The previous grammar shows that our policy can also include properties that have to be verified before the application is allowed to invoke a given system call on the resource. These properties are represented by predicates that precede the systems calls they refer to. As an example, $a.[p(\vec{x}), q(\vec{x})].b$ means that the properties $p(\vec{x})$ and $q(\vec{x})$ must be verified before performing the action b, but it is not required that are verified before the execution of the action a. These properties could involve the evaluation of conditions of various nature. For instance, they can force the value of some system call parameter. In the following rule:

[eq(x_1, READ)].open(x_0,x_1,x_2,x_3)

the predicate that precedes the `open` system call states that the second parameter of this action, i.e. the open mode, must be equal to READ. This allows the application to open files in read mode.

Predicates could also include properties that concern the evaluation of factors that does not involve only the system calls and their parameters. The exploitation of external factors provide a flexible way to evaluate distinct kind of conditions from the ones provided by the behavioural policy, i.e. to integrate and exploit other policies with the behavioral one. A very simple example could be a property that evaluates whether the current time is within a given time interval.

The definition of an integrated framework for the specification and analysis for security and trust in complex and dynamic scenarios was introduced in [13]. In this way, the behavioural policy could include properties that involve the evaluation of the set of credentials submitted by the user. In particular, a new predicate, $\mathtt{tm}^{\mathcal{C}_\mathcal{A},\mathcal{C}_{tmp}}(\beta)$ is embedded in our policy to state that the evaluation of β is delegated to a trust management engine. $\mathcal{C}_\mathcal{A}$ represents the set of credentials that have been submitted by a grid user and which are active during their life time (until they expire). In contrast, $\mathcal{C}_{tmp}$ represents the set of temporal credentials that have

been dynamically generated according to the behavioural policy and recent behavior of the user. $\mathcal{C}_{tmp}$ reflects the additional rights that an application has. For the sake of space limitation we omit the details on the rules used to generate temporal credentials and refer the reader to Section 6 for a comprehensive example of their usage.

Let us consider the following example.

[eq(x_1,READ),tm$^{\mathcal{C}_{\mathcal{A}},\mathcal{C}_{tmp}}$(open(normal))].open($x_0$,$x_1$,$x_2$,$x_3$)

It allows an application to open files in read mode only if the property open(normal) is verified according to an access policy. As soon as an external property has to be evaluated, the behavioural policy delegates the decision to an external access control module. For doing so, it first allocates the sets $\mathcal{C}_{\mathcal{A}}$ and $\mathcal{C}_{tmp}$ regarding the current application (running on behalf of a user) and then offloads the real access decision process to the external module specifying as input the sets of active and temporal credentials. Once an access decision is taken the monitoring is resumed and further enforced by the evaluation of the behavioural policy.

The enforcement of behavioural policy has to be strictly coupled with a proper access control model described in the next section.

4 Credential-based Access and Trust Management

Controlling access to Grid resources has become one of the major security issues for the last several years. The term *credential* has emerged as a widely used way of expressing digital access rights in a distributed environment. Management of credentials for access control, also refereed in the literature as *trust management* [20], emerged as a key issue for distributed Grid environment.

This section describes a novel access control prototype, called *i*Access, that leverages access and trust management in a Grid computational infrastructure. *i*Access is based on the interactive access control model introduced by Koshutanski and Massacci [11],[12].

In the *i*Access framework each Grid domain has a *security policy for access control* $\mathcal{P}_{\mathcal{A}}$ and a *security policy for disclosure control* $\mathcal{P}_{\mathcal{D}}$. $\mathcal{P}_{\mathcal{A}}$ protects Grid's resources by stipulating what credentials a requester must satisfy to be authorized for a particular resource while, in contrast, $\mathcal{P}_{\mathcal{D}}$ defines which credentials among those occurring in $\mathcal{P}_{\mathcal{A}}$ are disclosable so, if needed, can be demanded from the client.

Policies are written as normal logic programs [1]. A logic program is a set of rules of the form:

$$A \leftarrow B_1, \ldots, B_n, \ not\, C_1, \ldots, \ not\, C_m \qquad (1)$$

A is called the *head* of the rule, each B_i is called a *positive literal* and each $not\, C_j$ is a *negative literal*, whereas the conjunction of B_i and $not\, C_j$ is called the *body* of the rule. If the body is empty the rule is called a *fact*.

The intuition is to interpret the rules of a program P as constraints on a solution set S (a set of ground atoms) for the program itself. So, if S is a set of atoms, rule (1) is a constraint on S stating that if all B_i are in S and none of C_j are in it, then A must be in S.

We also need to keep a memory of past credentials submitted by a user. This is the role of $\mathcal{C}_{\mathcal{A}}$, the set of *active credentials* that have been presented by a client in past requests to other services within the Grid's domain. We note that this set is stored, activated and cleared externally to *i*Access module and accordingly to the Grid's business logic. Grid infrastructure is also responsible to properly update $\mathcal{C}_{\mathcal{A}}$ with newly submitted credentials by the client.

Extending client's profile, we introduce the notion of *declined credentials* $\mathcal{C}_{\mathcal{N}}$. $\mathcal{C}_{\mathcal{N}}$ keeps track of what credentials a client has declined to provide within an access control session for a particular service. $\mathcal{C}_{\mathcal{N}}$ is an internal set to *i*Access and, as so, it is activated when initially an access decision is requested.

To request for an access decision, Gmon (the Grid enforcement module) presents as input: the resource r, the set of client's active credentials $\mathcal{C}_{\mathcal{A}}$ and a set of temporal (dynamic) credentials generated by Gmon, namely $\mathcal{C}_{tmp}$. Here the use of temporal credentials reflects the fine-grained access and enforcement control that is to be performed on Grid resources.

Figure 1 shows that *i*Access decision protocol. The intuition behind the protocol is the following. When an access decision is requested, *i*Access loads the client's set of declined credentials $\mathcal{C}_{\mathcal{N}}$, the set of missing credentials requested in the last interaction $\mathcal{C}_{\mathcal{M}}$, the policy for access $\mathcal{P}_{\mathcal{A}}$ and policy for disclosure control $\mathcal{P}_{\mathcal{D}}$. If there is no session associated with the requested resource then *i*Access sets up $\mathcal{C}_{\mathcal{N}}$ and $\mathcal{C}_{\mathcal{M}}$ to an empty set.

Step 1 updates the declined credentials with those requested in the last session interaction ($\mathcal{C}_{\mathcal{M}}$) minus what the client has accumulated in his set of active credentials $\mathcal{C}_{\mathcal{A}}$. Next, *i*Access checks whether the requested resource is granted by $\mathcal{C}_{\mathcal{A}}$ and $\mathcal{C}_{tmp}$ according to $\mathcal{P}_{\mathcal{A}}$. If the check succeeds then the client has enough access rights and *i*Access returns grant r.

If the client does not have enough access rights then *i*Access protocol computes the set of disclosable credentials $\mathcal{C}_{\mathcal{D}}$ (step 3a). This step is the staring point of the interactive access control model [11]. $\mathcal{C}_{\mathcal{D}}$ contains all credentials disclosed by the disclosure policy $\mathcal{P}_{\mathcal{D}}$ together with $\mathcal{C}_{\mathcal{A}}$ and $\mathcal{C}_{tmp}$. The set difference in this step is to assure that *i*Access does not ask for credentials that have already been presented or declined by the client.

Once the disclosable credentials are computed, *i*Access performs a special reasoning over $\mathcal{P}_{\mathcal{A}}$ according to $\mathcal{C}_{\mathcal{D}}$ in

Internal Vars: $\mathcal{C}_\mathcal{N}, \mathcal{C}_\mathcal{M}$; Initially $\mathcal{C}_\mathcal{N} = \mathcal{C}_\mathcal{M} = \emptyset$
Input: $r, \mathcal{C}_\mathcal{A}, \mathcal{C}_{tmp}$;
Output: grant/deny/ask($\mathcal{C}_\mathcal{M}$);

*i*Access($r, \mathcal{C}_\mathcal{A}, \mathcal{C}_{tmp}$){

1. update $\mathcal{C}_\mathcal{N} = \mathcal{C}_\mathcal{N} \cup (\mathcal{C}_\mathcal{M} \setminus \mathcal{C}_\mathcal{A})$, where $\mathcal{C}_\mathcal{M}$ is from the last interaction,
2. check whether $\mathcal{P}_\mathcal{A}$ together with $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_{tmp}$ grant r,
3. if the check succeeds then return ***grant*** else
   (a) compute the set of *disclosable credentials* $\mathcal{C}_\mathcal{D}$ from $\mathcal{P}_\mathcal{D}$ according to $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_{tmp}$. From the resulting set remove all already presented and already declined credentials, namely $\mathcal{C}_\mathcal{D} = \mathcal{C}_\mathcal{D} \setminus (\mathcal{C}_\mathcal{N} \cup \mathcal{C}_\mathcal{A})$.
   (b) compute a set of *missing credentials* $\mathcal{C}_\mathcal{M}$ among the disclosable ones ($\mathcal{C}_\mathcal{M} \subseteq \mathcal{C}_\mathcal{D}$) such that $\mathcal{P}_\mathcal{A}$ together with $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_{tmp}$ and $\mathcal{C}_\mathcal{M}$ grant r,
   (c) if no such set exists then return ***deny*** else
   (d) return ***ask($\mathcal{C}_\mathcal{M}$)***.

}

Figure 1. *i*Access Decision Protocol

order to find a set of missing credentials that grants the resource (step 3b). We refer the reader to [11] for details.

If such a set is found then *i*Access returns it back specifying that there is a potential solution for the client to get the resource. If no missing set is found then *i*Access denies access to r. The reason for denial would be that either the client's set of active credentials explicitly denies access to the resource, according to $\mathcal{P}_\mathcal{A}$, or the client does not have enough access rights to disclose more information from $\mathcal{P}_\mathcal{D}$ so that step 3b can find a solution for the resource.

We remark that $\mathcal{C}_{tmp}$ plays an important role for a fine-grained and context-aware access control process, as we shall see in Section 6.

5 The Architecture

We have not seen yet the architecture that enforces the overall access control process. This section describes the architecture that enforces proper access decisions when Java applications are executed by the Globus toolkit [4].

Informally, we have divided the architecture in three levels: resource level, execution level and operating system level. Figure 2 shows our architecture and its components.

The operating system level represents local Grid resources and their low level execution. Rather, the resource level comprises the Globus toolkit and the JVM. It is responsible for handling user's requests and executing user's applications in a mobile and transparent way wrt the user's location.

The execution level resides between the resource level and the OS of local resources. This level is where all the security components that build up our architecture take place. There are four security components:

User's Profile module keeps the profiles of active users, i.e. users' active credentials that have been presented in past interactions with other services within the local Grid domain. This module is responsible for validating, storing, updating and clearing[1] user's credentials.

Policy Repository module stores the local site's security policies. In particular, it stores the behavioral policy, the access policy and the disclosure policy.

Gmon module is the policy enforcement point and the behavioural policy evaluation point of our architecture.

***i*Access** component is the trust management decision point of the architecture.

We have modified the Globus toolkit such that whenever access to a Grid resource is performed Gmon intercepts it and evaluates whether access to that resource is allowed to be performed or not. Whenever a client requests a service, presenting some credentials, the Globus toolkit allocates the Java application and starts its execution. Gmon intercepts the Globus service request and extracts information relevant to the behavioral policy. During the execution of the Java application, for each system call invoked by the JVM, the architecture performs the following steps:

1. Gmon intercepts the system call and the Java application is suspended.

[1] We note that user's active credentials are time dependent and should be periodically cleared up from the already expired credentials.

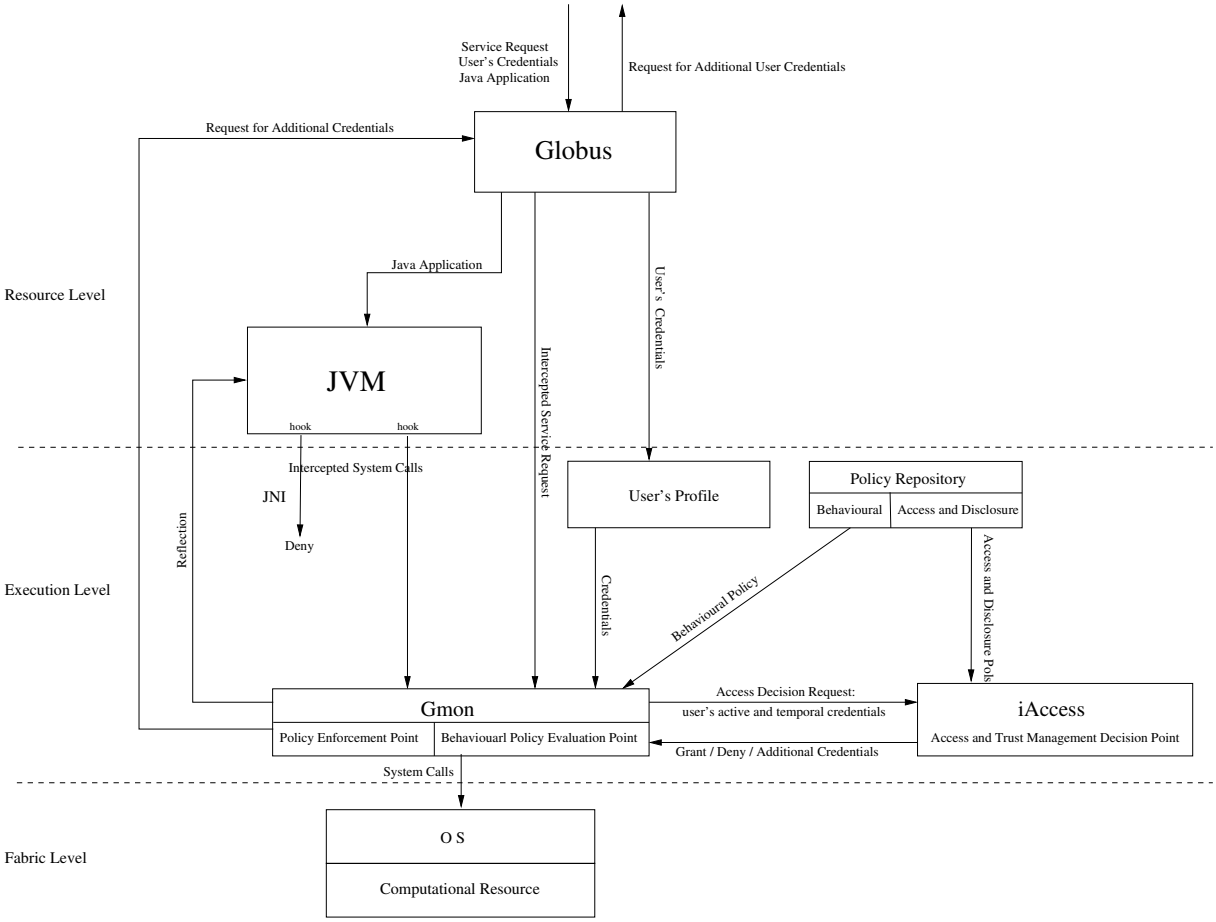

Figure 2. The Architecture

2. Then, Gmon checks whether the call is allowed according to the history of executions and the behavioural policy, i.e. if in the behavioural policy there is at least a rule that includes this call and all the previous calls in this rule have been already successfully performed by the application. If the call is not allowed according to the history then the Java application is terminated.

3. Next, Gmon checks whether an access decision needs to be taken for the intercepted system call and invoked *i*Access with the following input:

   - request r: <system call, mode of execution>,
   - temporal credentials generated by Gmon, $\mathcal{C}_{tmp}$, and
   - user's active credentials, $\mathcal{C}_{\mathcal{A}}$.

4. *i*Access loads the local access and disclosure control policies in order to take an access decision.

5. when Gmon receives back the result of *i*Access it enforces the access decision by the following ways:

   - if Grant then invoke the system call on Operating System (OS) level and resumes the execution of the Java application,
   - if Deny then terminate the execution of the Java application,
   - if Missing Credentials then return back to the client the need to provide additional credentials and suspend the application until the client replies with new credentials.

6 Integrated Security Policy Example

To show the practical relevance of the framework this section presents an example of the security policies introduced so far and a Grid usage scenario.

Figure 3 shows the three polices that define the right sequence of actions to be performed and enforced on the sets of critical, $S_{critical}$, and non-critical resources, S.

Access Policy

$\texttt{dominate}(user, user) \leftarrow .$
$\texttt{dominate}(admin, user) \leftarrow .$
$\texttt{dominate}(admin, admin) \leftarrow .$
$\texttt{grant}(open, normal) \leftarrow \texttt{cred}(Holder, Attr), \texttt{dominate}(Attr, user).$
$\texttt{grant}(send, normal) \leftarrow \texttt{cred}(Holder, Attr), \texttt{dominate}(Attr, user).$
$\texttt{grant}(open, critical) \leftarrow \texttt{cred}(Holder, Attr), \texttt{dominate}(Attr, admin).$
$\texttt{grant}(send, critical) \leftarrow \texttt{cred}(Holder, Attr), \texttt{dominate}(Attr, admin), \texttt{tmp\_cred}(Holder, well\_behaved).$

Disclosure Policy

$\texttt{cred}(Holder, user) \leftarrow .$
$\texttt{cred}(Holder, admin) \leftarrow \texttt{cred}(Holder, user), \texttt{tmp\_cred}(Holder, well\_behaved).$

Behavioral Policy

S := {/usr/share/file0.txt, /usr/share/file1.txt}
$S_{critical}$:= {/sys/kernel/config0.txt, /sys/kernel/config1.txt}
AH := {host1.iit.cnr.it, host2.iit.cnr.it, host3.iit.cnr.it}
OS := false

[eq(x_1,AF_INET),eq(x_2,SOCK_STREAM),eq(x_3,IPPROTO_TCP)].socket(x_1,x_2,x_3,sd).
[eq(x_4, sd),eq(x_6,2000)].bind(x_4, -, x_6, -, -).
[eq(x_9, sd), in(x_{10}, AH)].connect(x_9, x_{10}, -, -).
i([eq(OS, false),**tm**$^{\mathcal{C}_A,\mathcal{C}_{tmp}}$**(send(normal))**,eq($x_{13}$, sd)].send(x_{13}, -, -, -, -) or
[eq(OS, true),**tm**$^{\mathcal{C}_A,\mathcal{C}_{tmp}}$**(send(critical))**,eq($x_{18}$, sd))].send(x_{18}, -, -, -, -) or
[eq(x_{23}, sd)].recv(x_{23}, -, -, -, -)).
[eq(x_{28}, sd)].close(x_{28},-).

[in(x_{30},$S_{critical}$),**tm**$^{\mathcal{C}_A,\mathcal{C}_{tmp}}$**(open(critical))**,eq($x_{31}$, READ)].open($x_{30}$,$x_{31}$,-,$fd$)
OS:=true.
i([eq(x_{33}, fd)].read(x_{33}, -, -, -)).
[eq(x_{37}, fd)].close(x_{37}, -)

[in(x_{39}, S),**tm**$^{\mathcal{C}_A,\mathcal{C}_{tmp}}$**(open(normal))**,eq($x_{40}$, READ)].open($x_{39}$,$x_{40}$,-,$fd$)
i([eq(x_{42}, fd)].read(x_{42}, -, -, -)).
[eq(x_{46}, fd)].close(x_{46}, -)

Figure 3. Example of Integrated Security Policy

The *behavioural policy* consists of three sets of rules that define the sequences of actions that can be performed by an application. For the sake of simplicity, the parameters and the results of the system calls are explicitly represented by a symbol x_i only if their values are exploited by a predicate otherwise a placeholder is used.

The first set of rules allows communications with remote hosts predefined within the set AH. The system call in the first rule is `socket(`x_1`,`x_2`,`x_3`,`sd`)`. x_1,x_2, x_3 represent the system call parameters and sd represents the result. The `socket()` system call is preceded by the predicate `eq(`x_1`,AF_INET)`, `eq(`x_2`,SOCK_STREAM)`, `eq(`x_3`,IPPROTO_TCP)`, that define the allowed values for the `socket` parameters. In our case they state that the socket has to be a TCP one. Here, the evaluation of user's credentials is not required to open a socket. After the socket call, the application can issue a `bind` and a `connect` system calls to establish a connection with a remote server. The predicates that precede these two system calls do not require any credentials evaluation as they state that these calls must exploit the socket previously opened (represented by the variable *sd*). The value of the socket local port must be 2000 and the connection can be established only with hosts within the set AH.

In the following lines of the behavioural policy, the application can iteratively send or receive data on established connections. To send data, the evaluation of the user credential is required. $\texttt{tm}^{\mathcal{C}_A,\mathcal{C}_{tmp}}\texttt{(send(normal))}$ evaluates whether the user can send non critical data to remote hosts. $\texttt{tm}^{\mathcal{C}_A,\mathcal{C}_{tmp}}\texttt{(send(critical))}$, instead, evaluates whether the grid user is trusted to send critical information to remote hosts. In particular, the operator $\texttt{tm}^{\mathcal{C}_A,\mathcal{C}_{tmp}}\texttt{(send(normal))}$ indicates that the evaluation of the property send(normal) is delegated to a trust manage-

ment engine, where $\mathcal{C}_{\mathcal{A}}$ is the set of credentials submitted by the user, and $\mathcal{C}_{tmp}$ is the set of temporary credentials.

In the behavioural policy there are two entries concerning the `send` system call. The predicate of the first entry, $\text{tm}^{\mathcal{C}_{\mathcal{A}},\mathcal{C}_{tmp}}$(send(normal)), requires that the value of the variables OS is false. The second entry, instead, requires the evaluation of $\text{tm}^{\mathcal{C}_{\mathcal{A}},\mathcal{C}_{tmp}}$(send(critical)) if the value of the variables OS is true.

If the variable OS is false it means that no critical file has been opened by the application otherwise if it is true then the application has opened a critical file during its execution.

Alternatively of the `send`, the application can perform `recv` system call to receive data from its communication partner. In this case, there are no predicate that involve credential evaluation.

The second and third sets of rules define the behavior to read files from the sets S and $S_{critical}$. Here, when a critical file is opened the value of the variable OS is set to true.

As already described, the credentials evaluated by the *i*Access engine include not only those submitted by the user but also some credentials that have been dynamically generated by Gmon according to some predefined rules. The temporal credentials are stored in the User's Profile component of the architecture.

As an example, the behavioural policy could grant special $well-behaved$ rights to applications that are running from two hours without doing anything in conflict with the behavioral policy.

The *access policy* shown in Figure 3, uses the predicate $grant(Operation, Operation\_Mode)$ to express that access is granted on operation $Operation$ executed in a mode $Operation\_Mode$ if the conditions in the body of the rule are satisfied. We represent variables with staring capital letter (e.g. $Holder$, $Attr$ etc) and constants starting with small case letters (e.g., $normal$, $send$ etc). A variable indicates any value in this field. We explicitly note that for a finer-grained access control one can extend the predicate $grant(Operation, Resource, Operation\_Mode)$ so that it can be specified different conditions on different resources when operated in different modes. This is possible because one of the advantages of *i*Access model is that it does not pose any restrictions on policy structure (refer to [11] for details).

The first three rules (facts) of the access policy state that the role $user$ dominates itself, the role $admin$ dominates the $user$ one as well as itself. The fourth and fifth rules say that operations *open* and *send* performed on all resources in a $normal$ mode are granted to any client having presented a credential for a role that dominates the $user$ one. Here the Holder variable represents any client.

Rule sixth grants operation *open* in a $critical$ mode only to roles dominating the $admin$ one. Rather, the last rule of the access policy says that operation $send$ in a $critical$ mode is granted to those clients that have the $admin$ role and, at the same time, has not behaved suspiciously by the time of access.

The *disclosure policy* says that the need for a $user$ credential is disclosed to any client, i.e. it is not sensitive information. Rather, the need for $admin$ credential is disclosed only to those clients that have presented a credential identifying them as legitimate users and, at the same time, the monitoring system attests them as $well-behaved$.

The following example shows how the three policies cooperate in order to enforce proper access rights. Here we assume that the grid application has already established a client socket connection with the Grid application domain.

Example 1 (Grid Resource Usage Scenario) *Let us have a Java application running on behalf of a grid user in a Grid domain. Since we are in an autonomic scenario so the client does not have any a priori information of what the requirements are to execute the application. So, initially the client does not have any active credentials in the system, i.e. his profile of active credentials $\mathcal{C}_{\mathcal{A}}$ is the empty set.*

Now, let us suppose that the application tries to open a file "/usr/share/file0.txt". At this point Gmon detects that "file0.txt" is a non-critical one and then, enforcing the behavioural policy specified by predicate $\text{tm}^{\mathcal{C}_{\mathcal{A}},\mathcal{C}_{tmp}}$(open(normal)), it invokes iAccess for an access decision.

The input to iAccess is: the request r=<$open, normal$>; client's set of active credentials $\mathcal{C}_{\mathcal{A}} = \emptyset$; and the set of temporal credentials, generated by Gmon, $\mathcal{C}_{tmp} = \emptyset$.

Once iAccess is invoked, it checks whether the client has enough access rights for the request r. Loosely speaking, we check whether $grant(open, normal)$ can be deduced from a rule in the access policy $\mathcal{P}_{\mathcal{A}}$. Since the client has no active credentials the check fails (step 2 in Fig. 1) and iAccess computes the set of disclosable credentials $\mathcal{C}_{\mathcal{D}} = \{cred(Holder, user)\}$. Next, iAccess computes the set of missing credentials, among the disclosable ones, that grant r. In our case $\mathcal{C}_{\mathcal{M}} = \{cred(Holder, user)\}$. Then, Gmon returns back to the grid user the need for $cred(Holder, user)$ and awaits until the user replies.

On the next interaction the client presents a credential that attests him as a legitimate user of the Grid domain, i.e. $cred(paolo\_mori, user)$. Then, Gmon requests iAccess for an access decision presenting client's active credentials, in our case $\mathcal{C}_{\mathcal{A}} = \{cred(paolo\_mori, user)\}$. Next, iAccess returns grant according to $\mathcal{P}_{\mathcal{A}}$ which is then enforced by Gmon. □

The following example shows the role of temp credentials for fine-grained access control.

Example 2 (Use of Temporal Credentials) *Let us assume that Example 1 has already taken place and now the application, running on behalf of the user, tries to open a file "/sys/kernel/config0.txt". Gmon detects that the file is a critical one and, according to the behavioural policy, invokes iAccess for an access decision presenting:* r=<$open, critical$>; $\mathcal{C}_{\mathcal{A}} = \{cred(paolo\_mori, user)\}$; *and* $\mathcal{C}_{tmp} = \emptyset$.

Now, iAccess returns deny. It is because according to $\mathcal{P}_{\mathcal{A}}$ *the client does not have enough access rights to get* r *and according to the disclosure policy he does not have enough credentials to disclose more credentials so that iAccess can find a missing solution. At that point* $\mathcal{C}_{\mathcal{D}} = \{cred(Holder, user)\}$ *and step 3b in Fig. 1 fails to find a solution for* r. *Next, Gmon enforces the denial answer by terminating the application.*

Now, let us have the above scenario but in case the client's application has behaved well within the last 2 hours according to the behavioral policy. Then when the application performs open operation on the file "/sys/kernel/config0.txt" Gmon invokes iAccess with input: r=<$open, critical$>; $\mathcal{C}_{\mathcal{A}} = \{cred(paolo\_mori, user)\}$; *and* $\mathcal{C}_{tmp} = \{tmp\_cred(paolo\_mori, well\_behaved)\}$. *At this point, iAccess instead of denial returns the need for additional credentials* $\mathcal{C}_{\mathcal{M}} = \{cred(paolo\_mori, admin)\}$. *It is because now the disclosure policy discloses the need for* $cred(Holder, admin)$ *and iAccess can find a solution for* r. *Then Gmon returns back to the client the need for admin credential and suspends the application. On the next interaction, if the client presents a certificate for admin then iAccess grants access and Gmon enforces the system call* $open$.

Now, let the user's application, having already admin status, start behaving suspiciously and within the time interval of 2 hours open more critical files than those considered for well-behaved applications. Then, if the application tries to send information to a remote host, Gmon will detect that it is a $send$ *operation in a* $critical$ *mode (*$critical$ *because at least one critical file was open) and requests iAccess for an access decision. At this point* $\mathcal{C}_{tmp} = \emptyset$ *and according to the access policy grant on* $send$ *in* $critical$ *mode is allowed only to applications with administrator privileges which have not behaved suspiciously, i.e. which are well-behaved certified. In this case iAccess returns denial and Gmon terminates the application.* □

The example above shows that if a user behaves well according to some rules then he is more trusted to obtain sensitive information for missing credentials when accessing Grid resources. On the other side, even a user has high access rights, he may not have all operations allowed if he behaves suspiciously.

In this section we presented the dynamic nature of our access enforcement framework and how history and context play an important role for a fine-grained access control.

7 Conclusions and Future Work

In this paper we presented a comprehensive framework for access control monitoring and enforcement on Grid computational resources. The framework provides fine-grained and history-based behavioral model which captures dynamic and context-aware access properties by generating temporal credentials each time when an access decision needs to be taken.

During an application execution if user's credentials are not enough to execute a resource the framework creates a session within which it interacts with the user in order to establish enough trust (access rights) needed to execute the resource.

Future work is in the direction of extending the framework to capture security policies on the level of Virtual Organizations. Typically each VO has security requirements that are to be enforced on the partners involved in the VO. In this way the framework will provide a way that global security policies can be combined with local site ones in order to obtain the final policy to be enforced.

References

[1] K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[2] F. Baiardi, F. Martinelli, P. Mori, and A. Vaccarelli. Improving grid service security with fine grain policies. In *Proceedings of On the Move to Meaningful Internet System 2004: OTM Workshops, LNCS*, volume 3292, pages 123–134, 2004.

[3] D. Chadwick and A. Otenko. The permis x.509 role based privilege management infrastructure. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 135–140, New York, NY, USA, 2002. ACM Press.

[4] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, LNCS 3779, 2005.

[5] I. Foster, C. Kesselman, L. Pearlman, S. Tuecke, and V. Welch. A community authorization service for group collaboration. In *Proceedings of the3rd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 50–59, 2002.

[6] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[7] C. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[8] M. Humphrey, M. Thompson, and K. Jackson. Security for grids. *Proceedings of the IEEE*, 93(3):644–652, 2005.

[9] K. Keahey and V. Welch. Fine-grain authorization for resource management in the grid environment. In *GRID '02: Proc. of the Third International Workshop on Grid Computing - LNCS*, volume 2536, pages 199–206, 2002.

[10] H. Koshutanski. A survey on distributed access control systems for web business processes. *International Journal of Network Security (IJNS)*, To appear.

[11] H. Koshutanski and F. Massacci. Interactive access control for Web Services. In *Proceedings of the 19th IFIP Information Security Conference (SEC 2004)*, pages 151–166, Toulouse, France, August 2004. Kluwer Press.

[12] H. Koshutanski and F. Massacci. Interactive credential negotiation for stateful business processes. In *Proceedings of the 3rd International Conference on Trust Management (iTrust)*, volume 3477 of *LNCS*, pages 257–273, Rocquencourt, France, May 2005. Springer-Verlag Press.

[13] F. Martinelli. Towards an integrated formal analysis for security and trust. In *FMOODS*, pages 115–130, 2005.

[14] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *Proc. of International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services 2005, IEEE Computer Society*, page 82, 2005.

[15] N. Nagaratnam, P. Janson, J. Dayka, A. Nadalin, F. Siebenlist, V. Welch, I. Foster, and S. Tuecke. Security architecture for open grid services. Global Grid Forum Recommendation, 2003.

[16] L. Pearlman, C. Kesselman, V. Welch, I. Foster, and S. Tuecke. The community authorization service: Status and future. *Proceedings of Computing in High Energy and Nuclear Physics (CHEP03): ECONF*, C0303241:TUBT003, 2003.

[17] A. J. Stell, R. O. Sinnott, and J. P. Watt. Comparison of advanced authorisation infrastructures for grid computing. In *Proc. of High Performance Computing System and Applications 2005, HPCS*, pages 195–201, 2005.

[18] M. Thompson, A. Essiari, K. Keahey, V. Welch, S. Lang, and B. Liu. Fine-grained authorization for job and resource management using akenti and the globus toolkit. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP03)*, 2003.

[19] M. Thompson, A. Essiari, and S. Mudumbai. Certificate-based authorization policy in a pki environment. *ACM Transactions on Information and System Security, (TISSEC)*, 6(4):566–588, 2003.

[20] S. Weeks. Understanding trust management systems. In *IEEE Symposium on Security and Privacy (SS&P)*. IEEE Press, 2001.

[21] XACML. eXtensible Access Control Markup Language (XACML), 2004. www.oasis-open.org/committees/xacml.