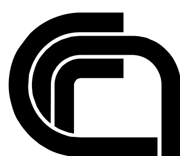
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PUblication MAnagement



Fine Grained Access Control for Computational Services

F. Martinelli, P. Mori, A. Vaccarelli

IIT TR-06/2006

Technical report

Giugno 2006

Istituto di Informatica e Telematica

Fine Grained Access Control for Computational Services*

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

Abstract

Grid environment concerns the sharing of a large set of resources among entities that belong to Virtual Organizations. To this aim, the environment instantiates interactions among entities that belong to distinct administrative domains, that are potentially unknown, and among which no trust relationships exist a priori. For instance, a grid user that provides a computational service, executes unknown applications on its local computational resources on behalf on unknown grid users. In this context, the environment must provide an adequate support to guarantee security in these interactions.

To improve the security of the grid environment, this paper proposes to adopt a continuous usage control model to monitor accesses to grid computational services, i.e. to monitor the behaviour of the applications executed on these services on behalf of grid users. This approach requires the definition of a security policy that describes the admitted application behaviour, and the integration in the grid security infrastructure of a component that monitors the application behaviour and that enforces this security policy.

This paper also presents the architecture of the prototype of computational service monitor we have developed, along with some performance figures and its integration into the Globus framework.

1 Introduction

Grid technology concerns the sharing of a very large set of resources among a dynamic, large and geographically distributed set of entities. The entities that

*Work partially supported by the STREP-project ”S3MS”, the FET-project ”SENSORIA” and the STREP-project ”GRIDTrust”

share their resources in the grid environment are organized in Virtual Organizations (VOs), and could be companies, universities, research institutes and other individuals. Each of these entities exploits a grid toolkit to set up its grid services, such as computational services, storage services, software repository services and so on. Hence, the grid is a very large and dynamic set of services provided by a various and geographically distributed entities. A grid user exploits this environment by composing the available grid services in the most proper way to solve its problem. As an example, the grid environment can be successfully exploited for the execution of computational or data intensive applications, such as very large scale simulations (e.g. earthquake simulation [15]) or image analysis (e.g. astronomy or climate study).

Security is a very important issue in the grid environment. As a matter of fact, a VO includes entities that belong to distinct administrative domains and, consequently, the resources shared by these entities are managed by distinct administration teams that may adopt different security mechanisms and apply distinct security policies. Moreover, no trust relationships exist a priori among the VO participants. Hence, the entity that provides a computational service, executes on the resources it shares unknown applications on behalf of unknown grid users. If adequate mechanisms that allow potential grid participants to use the grid environment with confidence are not provided by the environment itself, the spreading of the grid will be slowed down.

This paper proposes an approach to improve the security of grid computational services that concerns access control enforcement, inspired to the concept of continuous usage control proposed by Sandhu and Park in [41] and [42]. This approach proposes to monitor the usage of the grid computational resource due to application executed on behalf of grid users. This can be executed integrating in the grid security architecture a component that monitors the behaviour of the applications.

The structure of the paper is the following. Section 2 describes the main features the grid environment and of the grid computational services. Section 3 describes some previous works that improve grid security with authorization systems. Section 4 defines the continuous usage control model, and Section 5 describes the security policy specification. Section 6 describes the design and some implementation details of our prototype of grid monitor dedicated to computational services that support the execution of Java application, along with some performance figures. Section 6 also describes the integration of our prototype with the Globus toolkit. Conclusions are drawn in Section 7.

2 Grid Computing

The grid environment is a distributed computing environment where each participant allows any other participant to exploit its local resources by providing a grid service [18], [21]. This environment is based on the concept of Virtual Organization (VO). A VO is a set of individual and/or institutions, e.g. companies, universities, industries and so on, who share their resources. The management of a VO is very complex because the number of participant is typically very large, and VOs are dynamic because during the life of a VO, some participant can leave the VO, while new participant can join the VO. The sharing that the grid environment is concerned with, is not primarily file exchange, but rather direct access to computers, software, data and even other kind of resources [21]. These resources are heterogeneous and geographically distributed, and each of them is managed locally and independently from the others. The heterogeneity and the dynamicity of the resources involved in the grid environment requires considerable efforts for the integration and the coordination that are necessary to create such a computing environment. The solution adopted to guarantee interoperability derives from the Web Service technology, and defines a grid services in terms of a set of standard protocols and a standard syntax to describe the service interface [17].

The Global Grid Forum community developed a standard, the Open Grid Service Infrastructure (OGSI) [50], that details the concept of grid services. The Globus Toolkit 3, [16], is a reference implementation of the OGSI standard, and this paper refers to this implementation as grid environment. However, alternative grid environments are available, such as Gridbus [5], Legion [9], WebOS [51], and Unicore [14]. According to OGSI, each grid service is described in terms of features, behaviour and interface exploiting the Grid Service Description Language (GSDL), an extension of the Web Service Description Language (WSDL) [10].

2.1 Computational Services

Grid services may be of various kind, depending upon the resources they are built on, such as: storage services, data base services, computational services and others. Computational services provide computational resources where the VO participants can execute their applications. To submit its application to a computational service, a VO participant issues a job request, that includes the name of the application and other parameters required for the execution. The Globus Toolkit proposes a Resource Specification Language, RSL, to express job requests. This language allow to express two type of information: the resource requirements, such as the main memory size, the CPU time or the CPU number, and the job configuration, such as the executable name and the input/output files. The following example

shows a simple job request:

```
<rsl:rsl...
<gram:job\>
  <gram:executable> <rsl:path>
    <rsl:stringElement value="my_appl"/>
  </rsl:path> </gram:executable>
  ...
  <gram:maxMemory>
    <rsl:integer value="64"/>
  </gram:maxMemory>
  ...
  <gram:maxCpuTime>
    <rsl:long value="60"/>
  </gram:maxCpuTime>
  ...
</gram:job>
</rsl:rsl>
```

This request concerns the execution of the application `my_appl` and specifies that the application uses not more than 64 MB of main memory and not more than 60 CPU seconds.

Hence, the computational service provider executes on its resource unknown applications on behalf of unknown grid users. From the resource provider point of view, the execution of an untrusted application represents a threat for the integrity of its local resource. From the point of view of the user who submits the job, instead, the execution of its application on an untrusted resource represents a threat for the correctness of the application results. This paper is focused on the protection of the computational resource from malicious applications.

3 Security in Grid Computing

Due to the dynamic, collaborative and distributed nature of the grid, security is a fundamental issue in this environment. As a matter of fact, the resource sharing implemented by the grid environment must be highly controlled, because grid service providers, that grant access to their resources to unknown grid users, want that these accesses are regulated by a security policy that states which actions can be performed on the resources. Security management in the grid environment is complicated by the need to establish secure relationships between a large number of dynamically created participants among which no trust relationships exist a priori,

and across distinct administrative domains, each with its distinct resources and its own local security policy. In the case of computational resources, for instance, grid service providers allow the execution of unknown applications on behalf of unknown grid users on their resources. If an adequate security support is not adopted, the applications could perform dangerous and even malicious actions on these resources.

Even if the security threats to the grid environment fall into the standard categories, some standard solutions cannot be adopted to protect the grid environment because, in the this case, the malicious entity could be anyone, even one of the grid participants. For instance, a firewall cannot be trivially applied, because the fact that the VO includes entities from distinct administrative domain, implies that is not trivial to define what is "inside" the grid environment and what is "outside" [38]. Hence, more complex security solutions, tailored to the grid environment, are required, such as the one proposed in [12] to "create overlaying security perimeters, protecting different virtual collaborations that may exist at a time .., while ensuring the security of each member as defined by its local administrator".

The security requirements of the grid environment are detailed in [20], [28] and [38]. These requirements include authentication, delegation, authorization, privacy, message confidentiality and integrity, trust, policy and access control enforcement. However, these requirements are not fully implemented by the current grid environments.

Most of the approaches that have been proposed to improve the security of the grid environment are all related to Globus, that is one of the most used grid toolkit. These approaches are meant to integrate in the Globus architecture an authorization system that performs a fine grained access control on grid resources. The Globus Security Infrastructure, GSI, assigns a unique grid identity to each VO participant. This identity is represented by a X.509 certificate signed by the VO certification authority, that includes the owner identity string and its public key. This certificate is used by the owner to produce proxy certificates, to authenticate himself on some services. Once the authentication is successfully performed, the grid participant is mapped into a local user with the proper set of right. The strength of this mechanism is given by the secrecy of the private key that, in turn, mainly depends upon the owner local security policy. If the workstation of a VO participant p is violated and the private key of p is stolen, the attacker can impersonate p on the grid, and can execute malicious applications on the grid services provided by other participants.

Moreover, Globus provides a coarse grained access control on the resource, because once the Globus Resource Allocation Manager [16] has authenticated a grid user through an identity certificate issued by the VO Certification Authority, this user is mapped onto a local account, and the security policy that is enforced

is only the one defined by the privileges paired with the local account by the resource operating system. However, native operating system access control models may not be expressive enough to define security policies suitable for the grid environment. Moreover, in general, distinct grid resources have distinct operating systems, and distinct operating systems may not support the same kind of policies. Hence, a proper support to guarantee security in grid computational service must be provided.

The Community Authorization Service, CAS, has been proposed by the Globus team in [19] and [40]. It is an implementation of an authorization service integrated with the Globus toolkit. CAS is a service that stores a database of the VO policies, i.e. the policies that determines what each grid user is allowed to do as VO member. This service issues to grid users proxy certificates that embed CAS policy assertions. The grid user that wants to use a grid resource contacts the CAS service to request a proper credential to execute an action on this resource. The credential returned by the CAS server will be presented by the grid user to the service it wants to exploit. This approach requires CAS-enabled services, i.e. services that are able to understand and enforce the policies included in the credentials released by the CAS server.

An approach that integrates a fine grain authorization system in the grid environment has been proposed by Keahey and Welch. In [31], they describe some of the shortcomings of the Globus current authorization system and they state the need for a fine grain authorization system. In [49], they address this issue by integrating Akenti within the Globus toolkit. "Akenti is an authorization service (PDP) that uses authenticated X.509 certificates to establish identity and distributed digitally signed authorization policy certificates to make access decisions about distributed resources" [48].

Another solution to adopt an advanced authorization system in the grid environment has been presented by Stell, Sinnot and Watt in [47]. They integrate a role based access control infrastructure, PERMIS, with the Globus toolkit to provide fine grained control on user rights. PERMIS is "a a role based access control infrastructure that uses X.509 certificates to store users' role. All access control decisions are driven by an authorization policy, which is itself stored in a X.509 attribute certificate.." [8].

This paper proposes an approach to improve the security of grid computational services that concerns access control enforcement, that is one of the grid security requirements listed in [20], [28] and [38]. This approach is inspired to the concept of continuous usage control. This implies that the applications executed on the computational service are not considered as atomic entities, but all the interactions between these applications and the local resource performed during the execution of the application are monitored according to a security policy. The enforcement of

a security policy to monitor the application behaviour is necessary to detect attacks that try to circumvent the other security mechanisms, such as the authentication one.

4 Continuous and Fine Grain Usage Control

This paper proposes to adopt a *continuous* and *fine grain* usage control model on the grid computational resource to monitor the usage of the resource itself, i.e. the behaviour of grid applications. Here and in the following, we denote as grid applications those applications that are executed on the local grid computational resource on behalf of remote grid users. This model defines the application behaviour that is admitted on the computational resource through a security policy, as described in the next section.

The computational resource usage control is *continuous* because the application behaviour is monitored during the whole execution time. For instance, the main memory occupation, the execution time, the number of processes or threads generated by the application and all the other interactions between the application and the local resource are checked during the application execution. As a consequence of these controls, the right of an application to execute an action on the local resource could be revoked during the execution time. For instance, the right of an application to send data to a remote host could be revoked during its execution if a critical file has been read by the application itself, or the right to create a thread could be revoked if the application has already created the maximum number of threads allowed by the policy. Moreover, also the grid service usage right could be revoked to an application during its execution if its behaviour does not satisfy the enforced policy.

The application monitoring is *history based* because, to decide whether the current interaction is permitted, the whole trace of execution of the application is evaluated. In this way, some dependencies among the operations issued by the application (*obligation*) can be imposed by the policy. For instance, the policy can state that the read and write file operations can be executed by the application only after the related file open operation, or that messages can be sent on a socket only if critical files has not been read. Moreover, a policy obligation could also state that an action can be performed only if a proper set of credentials have been submitted by the remote grid user. Here and in the following, we define a trace of execution of an application as the ordered list of the operations issued by this application from the beginning to the end of its execution. The set of operations that are relevant to define a trace is defined by the security policy, as showed in the next section.

Even *conditions* that evaluate dynamic factors that does not depend upon the

application itself, like the resource status or the current local time, can be included in the policy as well. These conditions are evaluated during the execution of the application. For instance, a policy applied to "not important" users can state that their applications cannot be executed when the system is overloaded, i.e. the overall system load is greater than a fixed threshold. Moreover, a condition could state that communications with remote hosts can be performed during the day time only.

The computational resource usage control is also *fine grained*, because all the interactions of the application with the local resource are controlled. For instance, if the access to a file is granted to the application, the related read and write operation are monitored as well. To this aim, the security policy represents a detailed description of the admitted application behaviour. Hence, the execution of an application is not considered as an atomic operation, but all the interactions with the local resource, such as memory allocations, file accesses and remote communications through sockets, are evaluated according to the security policy.

Hence, in this model the admissibility of an operation does not depend upon a static access matrix, but it depends upon a set of dynamic factors that must be evaluated by the grid monitor system for each access.

The model of service monitoring we propose is inspired to the $UCON_{ABC}$ usage control model, proposed by Sandhu and Park in [41] and [42]. $UCON_{ABC}$ extends the traditional access control model by taking into account three decision factors: authorization, obligation and condition. Moreover, it introduces mutable attributes and recognizes the continuity of access enforcement. In our case, these concepts are mapped onto the grid environment for a specific kind of resource, the grid computational resource. As a matter of fact, the subjects are the grid users and the objects are the grid services. The reputation could be an attribute of the grid user. This attribute is mutable, because the reputation of a grid user could be updated as a consequence of the user interactions with grid services. According to the reputation, a proper security policy can be chosen to be applied to the grid user.

This work extends the approach we described in [4] and [37].

5 Security Policy

Our approach requires the definition of a fine grain security policy detailing the sequences of security relevant actions that the grid applications are allowed to perform on the grid computational service. In other words, the security policy defines the restricted environment provided by the grid computational service. This policy can be local, i.e. paired with the grid service, can be global, i.e. established by the VO, or can be also a combination of these two. For instance, the VO can provide a standard security policy, while the local resource administrator can integrate this

policy by adding other behaviours he believe that are not dangerous for its computational resource. Moreover, distinct policies can be applied on the same service to execute applications on behalf of distinct grid users. As an example, the policy can be chosen according to the trust level paired with the grid user.

The aim of the security policy is the recognition of dangerous or even malicious behaviors of the application that could reduce the availability or could grant unauthorized accesses to the computational service. The security policy is composed of a set of limits over the usage of the local resources of the computational service and a set of rules that defines the permitted behavior of an application. The usage limits involve main memory occupation, CPU time, overall execution time, number of processes or threads, and so on. The rules that describe the application behavior, instead, concern the order in which the security relevant actions can be performed (traces), and other various conditions, represented through predicates, that have to be satisfied during the execution. Any behavior that does not satisfy the rules is denied by default. In other words, any security relevant operation is denied unless explicitly allowed by a policy rule. Some resource usage limits can be derived from the job request, because the features of the required computational service specified in the job request can be exploited to define the maximum limits to the resource exploitation.

Since applications interact with the computational resource through operative system calls, the application behavior can be monitored trough the system calls they issues, and the application environment can be restricted by preventing the invocation of some system calls. Hence, in the following, we assume that the security relevant operations we are interested in correspond to system calls. As a matter of fact, processes exploit system resources, e.g. memory or disk space, by issuing an appropriate request to the operative system through the system call interface.

5.1 Policy Specification

As previously described, the security policy specifies a set of limits over the resource usage and a set of rules that describes the admitted behavior of the applications.

Each rule of the policy results from the composition of system calls, predicates and variable assignments. The composition operators are meant to describe the possible execution orders of the system calls, and are described by the following grammar:

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

where P is a rule, α is a system call 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*, and represents the possibility of performing a system call α and then behave as P;
- $p(\vec{x}).P$ behaves as P in the case the predicate $p(\vec{x})$ is true;
- $\vec{x} := \vec{e}.P$ 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*, and represents the non deterministic choice between P_1 and P_2;
- $P_1 par_{\alpha_1,\ldots,\alpha_n} P_2$ 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 table 1 through semantics rules given as

$$\frac{\text{premises}}{\text{conclusion}}$$

Other derived operators may be considered. For instance, $P_1 par P_2$ is the *parallel operator*, and represents the interleaved execution of P_1 and P_2. It is used when the policies P_1 and P_2 deal with disjoint system calls. Using the constants definition, the iteration operator, $i(P)$, can be easily derived. Informally, $i(P)$ behaves as P x times, for any value of x. It can be modeled in our framework by defining a constant $Z = P par Z$. Thus $i(P)$ is Z. Also the policy sequence operator $P_1; P_2$ may be implemented using the policy languages operators (and control variables) (e.g., see [27]).

As an example, given that $\mathtt{r}$ and $\mathtt{s}$ represent system calls and $\mathtt{p}$ and $\mathtt{q}$ are predicates, the following rule:

$$\mathtt{p}(\vec{\mathtt{x}}).\mathtt{r}(\vec{\mathtt{x}}).\mathtt{q}(\vec{\mathtt{y}}).\mathtt{s}(\vec{\mathtt{y}}).\mathtt{P}_1$$

describes a behavior that includes the system call $\mathtt{r}$, whose parameters $\vec{x}$ enjoy the conditions represented by the predicate $\mathtt{p}$, followed by the system call $\mathtt{s}$, whose parameters $\vec{y}$ enjoy the conditions represented by the predicate $\mathtt{q}$. In turn, $\mathtt{s}$ is followed by a policy P_1. The predicate $\mathtt{p}$ specifies the controls to be performed on the parameters and on the results of $\mathtt{r}$, through conditions on $\vec{x}$, as showed in the

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\frac{p(\vec{n}) \text{ holds where } \vec{x} = \vec{n} \text{ in the global state}}{p(\vec{n}).P \xrightarrow{\epsilon} P}$$

$$\frac{\text{where } \vec{e} \text{ evaluates to } \vec{n} \text{ in the global state}}{\vec{x} := \vec{e}.P \xrightarrow{\epsilon} P(\text{where the values of } \vec{x} \text{ are updated with } \vec{n})}$$

$$\frac{P \xrightarrow{\alpha} P'}{PorQ \xrightarrow{\alpha} P'}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{PorQ \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \text{ and } \alpha \notin Act}{Ppar_{Act}Q \xrightarrow{\alpha} P'par_{Act}Q}$$

$$\frac{Q \xrightarrow{\alpha} Q' \text{ and } \alpha \notin Act}{Ppar_{Act}Q \xrightarrow{\alpha} Ppar_{Act}Q'}$$

$$\frac{Q \xrightarrow{\alpha} Q' \text{ and } P \xrightarrow{\alpha} P' \text{ and } \alpha \in Act}{Ppar_{Act}Q \xrightarrow{\alpha} P'par_{Act}Q'}$$

Table 1: Rules for inferring the admissible behavior, with the symmetric rules for the binary operators

next example. Moreover, the parameters and the results of each system call can be assigned to variables, to be exploited in other rules of the policy. `p` could include also conditions that include variables, current local time, etc. Hence, variables can be exploited to implement interactions between distinct rules of the same policy. They can be used to represent some aspects of the execution status, such as the number of files currently opened by the application.

This rule also defines an ordering among the system calls represented by `r` and `s`. As a matter of fact, `s` can be executed only after `r`. Hence, this rule represents an *obligation*, because it states that the execution of `r` is a requirement for the execution of `s`. The following example shows in more details a simple rule:

[eq(x_1,“/tmp/*”) and less(OF, 9)].open(x_1,x_2,x_3,x_4).OF:=OF+1

This rule allows the execution of the `open` system call if the first parameter x_1, i.e. the file name, is equal to `/tmp/`$\star$. As usual, the symbol $\star$ represents any

string. The other condition in the predicate, `less(OF,9)`, concerns the execution state and says that the `open` system call can be performed only if the number of files currently opened by this application, represented by the variable `OF`, is smaller than 9. Hence, this rule grants to the application the right to open any file in the `/tmp` directory. However, this right is revoked when 9 or more files are currently opened by the application. As a matter of fact, when the right to open the file is granted, the variable OF is incremented.

Many different execution patterns may be described with this language. For instance, if we wish that the system call `d` is performed only after that `a`, `b` and `c` have been performed (in any order) we may define the following policy:

```
(a par b par c); d
```

The difference between the synchronous parallel composition and the asynchronous one may be exemplified as follows. The admissible traces of the policy

```
(a par b)
```

are `a`,`b`,`ab` and `ba`. So the admissible traces of

```
(a par a)
```

are `a` and `aa`. Instead, the admissible traces of

(a $\texttt{par}_a$ a)

is simply `a`, since the two parallel `a`s must synchronize.

An implementation of the Chinese Wall Security Policy (CWSP) [7] using our policy language is shown in table 2.

The critical files are grouped in two sets, S1 and S2, and if the application reads one (or more) files from one of the two sets, then it cannot read any file from the other set. The first two statements represent some limits on the usage of the CPU (secs) and of the main memory (MBs). S1 and S2 are sets of file names that have been previously defined. The variables OS1 and OS2 indicate, respectively, whether a file of the set S1, or S2, has been opened.

The first predicate of the first rule, $\mathtt{p}_1$, states that the `open` system call on a file of the set S1 can be performed only if any file of the set S2 has been opened. As a matter of fact, the condition `eq(OS2, false)` tests the variable OS2, that is true only if a file of S2 has been opened. The condition in(x_1, S1) checks that the first parameter of the system call, i.e. the file name, is included in the set S1. Moreover, the condition eq(x_2, READ) states that the open system call can be executed only in READ mode. If the open is executed, the value true is stored in the variable OS1. fd denotes the file descriptor returned as result of the open system call. The iterative operator allows to perform any number of read system call on the file descriptor represented by fd. At the end, the close system call can be executed on fd. The second rule regulates the opening of file belonging to the S2 set, and it is symmetric to the first one.

```
....
MAX_CPU_TIME=100
MAX_MEMORY=64

OS1 := false.
OS2 := false.

([eq(OS2, false), in(x1, S1), eq(x2, READ)].          (p1)
 open(x1,x2,x3,fd).
 OS1:=true.
 i([eq(x5, fd)].                                       (p2)
   read(x5, x6, x7, x8));
 [eq(x9, fd)].                                         (p3)
 close(x9, x10))
par
([eq(OS1, false), in(x1, S2), eq(x2, READ)].          (p4)
 open(x1,x2,x3,fd).
 OS2:=true.
 i([eq(x5, fd)].                                       (p5)
   read(x5, x6, x7, x8));
 [eq(x9, fd)].                                         (p6)
 close(x9, x10))
```

Table 2: Implementation of the Chinese Wall security policy

This policy also implements the aggressive model of chinese wall security policy (ACWSP) [33]. As a matter of fact, the same file *f* may belong to both S1 and S2. In this case, when the application wants to open *f*, both the rules in the policy allows it, and rule 1 sets the variable OS2 to true, while rule 2 sets OS1 to true. Hence, no other files from S1 and S2 can be opened by the application. This is possible because if two (or more) rules allow the current system call, when their predicates are evaluated, each of these rules sees the same state, i.e. the updates executed by one rule does not affect the evaluation of the predicates of other rules. As a matter of fact, the updates of the variables are executed only after that all the predicates have been evaluated. The updates are executed following the same order of the rules they belong to in the policy.

The opening of a file of S1 or S2 in write mode, instead, is denied by this policy, because it is not explicitly allowed by any rule.

6 A Grid Monitor Prototype

To validate the effectiveness of the framework described in the previous sections, a prototype of grid computational service monitor, Gmon, has been developed. This prototype monitors the behaviour of Java applications by performing a continuous usage control on the shared computational resource where the JVM runs.

We focus on Java language because the platform independence of Java addresses the interoperability problem of the grid environment, due to the high heterogeneity of the resources shared in this environment. The portability of Java applications has been already exploited to develop some distributed heterogeneous environment such as IceT [25], Javelin [39] and Bayanihan [43].

The standard Java security architecture is based on the sandbox model, and includes the bytecode verifier, the class loader and the security manager, [2] [23], that provide an access control mechanism. Gmon integrates the standard access control mechanism by performing a history based monitoring of the application behaviour.

Other attempts have been made to improve the Java security architecture. For instance, [46] extends the JVM with an event logging system, and exploits the log to detect possible attacks using STAT [13], a signature based intrusion detection engine. JSEF [26] is another security framework that integrates the Java security architecture by introducing higher level security policies to enhance the expressiveness of policy rules.

Alternative Java Virtual Machines (JVMs) that follow both the *Java Language Specification* [29] and the *Java Virtual Machine Specification* [34] are currently available. To develop our prototype, we have chosen the Jikes Research Virtual Machine (RVM), that is a research oriented and open source JVM developed by the Jalapeno project of the IBM T.J. Watson Research Center [1] [30], on top of the Linux operating system. Jikes RVM adopts the gnu classpath library [11] to implement the Java core classes.

As previously described, the behaviour of an application can be found out through its interactions with the computational resource, because the only way the application has to interact and to change the status of the computational resource is through operating system calls. Hence, to perform the resource usage control, these calls have to be intercepted and controlled by Gmon before they are issued to the operating system. The Linux standard method to intercept the system calls issued by a process exploits the ptrace() system call. The application, issuing the ptrace() system call accept to be traced. The tracing process, instead, issues the ptrace() system call indicating the application process identifier to catch the next system call issued by the application. However, this method has not been adopted in our prototype because it does not allow to trace only a subset of system calls. As

a matter of fact, the ptrace mechanism stops the traced application and activates the tracing process each time the application issues a system call, and it is not possible to choose which system calls have to be traced.

The proposed architecture exploits the JVM to intercept the system calls issued during the execution of the Java applications. As a matter of fact, the JVM is a Virtual Machine and, consequently, mediates all the interactions between the application and the resource. The JVM executes the Java applications by interpreting the Java operations in the bytecode and implementing them by issuing the proper sequences of operating system calls to the operating system of the resource. Actually, since the JVM is executed in user space, it exploits the libc API to issue system calls. For instance, to execute the `open` system call, the JVM exploits the `open` libc function. Moreover, the JVM does not exploit all the system calls provided by the operating system, but only a subset of them. Exploiting this architecture, the system call wrapper has been embedded in the JVM code. This approach is more efficient and flexible than the ptrace one, and allows us to intercept only the system calls we are interested in.

Gmon runs in a POSIX thread that is created by the JVM during its initialization, before starting the execution of the application. After its initialization, Gmon suspends itself on a semaphore, and it is reactivated by the JVM each time to execute the Java application a system call is required. To reactivate Gmon, the system calls in the JVM code are wrapped by inserting some hooks. An hook is a system call wrapper that, instead of issuing the system call to the operative system, activates Gmon and suspends the JVM.

A first kind of hooks intercept the system calls that the JVM issues to implement the operations of the Java application. These hooks are inserted in the code that implements some methods of the classpath library, because the Java applications interact with the local resource invoking the methods of the Java core classes, such as java.io to access the filesystem and java.net to communicate with remote hosts.

A second kind of hooks involves the resource management operations performed by the JVM, such as threads creation and scheduling, memory management, class loading and so on. Some of these operations are implemented through system calls, that are wrapped by some hook, as in the previous case. As an example, when the classloader loads a new class, it interacts with the filesystems through system calls to open and read the file that includes the class. However, these system calls are not included in the application trace, because they depend upon the chosen JVM implementation. Some JVM operations, instead, are not implemented through system calls. Thread management, as an example, is fully implemented at the JVM level, and does not exploit any operating system function. In these cases, the hook is inserted in the JVM code to activate Gmon even if it does not wrap any

system call. This architecture is showed in figure 1.

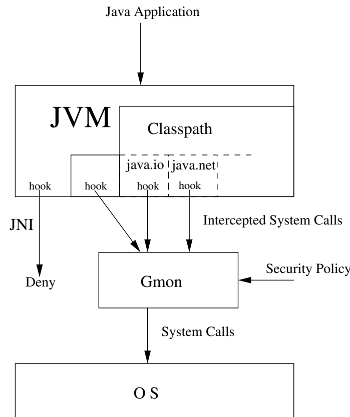


Figure 1: Gmon architecture

In our architecture, we deny the use of the Java Native Interface (JNI), because JNI allows the execution of arbitrary code outside the JVM by invoking it in a Java application. This implies that this code cannot be monitored through the JVM.

Once activated, Gmon checks the intercepted system call *sc* according to the security policy. Since Gmon is a thread started by the JVM, it reads the parameters of *sc* in the stack of the JVM thread, while the system call number has been written in a shared variable by the wrapper. Gmon adopts the default deny model, and the behavior of the grid application has to be consistent with the behavior described by the security policy. An execution history that does not match the permitted behavior is considered malicious. To determine whether, at a given time of the execution of the application, a system call matches the permitted behavior, both the security policy and the system calls that have been previously executed by the application, i.e. the history of the application, have to be considered. In particular, *sc* can be executed only if there is at least a rule *r* in the policy that includes *sc*, if all the system calls that precedes *sc* in *r* have been executed, and if the predicate paired with *sc* are satisfied.

As an example, in the security policy of table 2 the read system call is included in both rules. Hence, at a given time of the application execution, the read system call can be executed only if the open system call has been previously executed initializing the file descriptor *fd* of one of the two rules and if, in the same rule, *fd* is equal to the file descriptor of the read system call.

To efficiently perform the previously described checks, the security policy along with the history of the application are represented by Gmon adopting se-

curity automata [44]. In the initialization phase, Gmon loads the resource usage limits and the rules of the security policy, and creates the corresponding set of security automata. Each automaton represents a rule of the policy and, consequently, an admitted behavior. An example of security automaton, representing the first rule of the policy in table 2 is shown in figure 2.

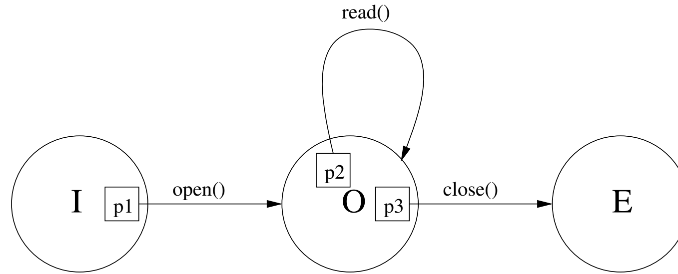


Figure 2: Security automaton

As we can see in the example, each node of the automata represents a possible state of the application. For instance, node I represents the initial state. An edge from a node A to a node B, instead, represents a system call *sc* that can be executed when the application is in the state A, and that leads to state B. The edge is paired with the predicates that represent the controls to be performed on *sc*. Moreover, the edge is paired also with the variable assignments that involve the parameters or the results of the system call. In the example, the edge from the initial state, I, to state O represents the `open()` system call. Hence, the execution of the `open()` system call leads from state I to state O. However, the system call execution is permitted only if predicate `p1` is satisfied.

At the beginning, the current state of an automaton is the initial one, i.e. the one that has not incoming edges. The initial state of each automaton is always considered as a current one during the execution of the application. As a matter of fact, an automaton can be fired more than once, to represent two instances of the same behaviour. Each automaton instance is independent from the others. Since an automaton for each rule is created, at a given time of the execution, the global state of the application is represented by the union of all the states of the automata.

As an example, when a file *f* from set A is opened, the automaton in figure 2 is fired, O become a current state and in this state *fd* stores the file descriptor of file *f*. When another file *g* from set A is opened, the same automaton is fired again, another instance of the same automaton is created, O become the current state of this instance, but in this state *fd* stores the file descriptor of *g*, while in the other instance O stores the file descriptor of *f*. However, to save memory space, the

same automaton is not replicated when it is fired more than once, but a compact representation is adopted.

If the wrapped system call satisfies the security policy, it is executed by Gmon, that reactivates the JVM and suspends itself. Instead, if the system call violates the security policy, the application can be stopped, or an error code can be returned to the application as result of the denied system call.

The predicate control and the system call execution are not executed as an atomic action, because in between the time when the predicate is evaluated and the time when the system call is executed, some actions could be performed. This problem is known as time-to-check-to-time-of-use (TOCTTOU) flaw [6] [22]. As an example, another thread of the application could try to modify the system calls parameters. In this way, the parameters with which the system call is executed are not the ones tested by Gmon. However, when Gmon is active, the JVM is suspended, and consequently, all Java threads, i.e. the application threads, are suspended. Moreover, Gmon copies the system call parameters in its local variables, and the controls and the system call invocation exploit the local copies. Hence, the parameters value cannot be altered by the application.

A similar problem can arise with the `open()` system call and symbolic links to files. As a matter of fact, after that Gmon has tested the name of the file to be opened, and before that the `open()` system call is executed, another application can delete this file, or a directory in its path, and substitute a link with the same name that points to a file or to a directory that cannot be read according to the policy.

This problem is avoided by Gmon because it opens the files before the execution of all the controls, using the file names that have been passed as parameters. When the file is opened, it cannot be deleted or unlinked and re-linked to another file to bypass the file name control. Before the predicate evaluation, the file name that have been passed as parameter of the system call is normalized, i.e. all symbolic links in the path are resolved and the real file name is determined. Hence, the predicate evaluation is executed on the normalized file name. If the `open()` system call satisfies the security policy, Gmon reactivates the JVM and returns the file descriptor to the application. Otherwise, the file is closed by Gmon and the application is stopped or the JVM is reactivated an error is returned to the application.

Another problem could arise because the values of file or socket descriptors are reused by the operating system when the files or the sockets have been closed. For instance, if a file whose file descriptor *fd* had value 9 has been closed, a new file with *fd = 9* could be opened. Hence, the same identifier could be bound to distinct object in distinct time during the execution of the application. The use of a outdated descriptor from Gmon, can be simply avoided by properly intercepting

all the system calls that updates the binding between objects and descriptors and by including them in the policy rules. As an example, when a file is opened, the current state is updated by recording the new value of the resulting file descriptor, and when the file is closed the current state is updated by deleting the value of the closed descriptor.

6.1 Performances

This section evaluates the impact of the application monitoring executed by Gmon on the performances of the Jikes RVM Java Virtual Machine. As a matter of fact, Gmon slows down the execution of the Java applications because, for each system call *sc* invoked by the JVM to interpret the application, the JVM is suspended, to check the policy and determine whether the execution of *sc* is allowed or not.

The performance degradation mainly depends upon two factors: the considered Java application and the enforced security policy. In particular, it depends upon the ratio between invoked system calls and other computation executed by the application itself. For instance, computational-intensive applications, that are the typical applications that can take advantage from the grid environment, issues a number of system calls that is not relevant with respect to the overall execution time of the application. Consequently, the Gmon overhead will be negligible in this case.

The overhead on the execution time of a monitored system call depends also upon the enforced policy. As a matter of fact, if the system call *sc* is included is *n* rules of the policy, to determine whether *sc* is allowed, Gmon have to check all these *n* rules. Hence, in general, complex security policies could take more time to be evaluated by Gmon than simple ones.

To evaluate the performance of Jikes RVM and Gmon, we have chosen the Ashes Suite Collection benchmarks [3]. In particular, we have chosen some benchmarks included in the Ashes Hard Test Suite Collection that are suitable to test our framework, because they perform a large number of system calls. As a matter of fact, other benchmarks we have tested, such as Scimark2 [45] and Linpack [35], perform a very small number of system calls with respect to the overall computation, and they are not a good tests for our purposes.

Figure 3 shows the execution times of the chosen benchmarks. On the x axe of the graph are listed the benchmark names along with the number of system calls they execute, while the y axe indicates the execution time in seconds. Each test compares the execution time of a Java application performed by the original Jikes RVM against the one of the same application executed using the Jikes RVM with Gmon. All benchmarks have been performed on a PC equipped with an Intel Pentium 4 (2.4Ghz) processor with 512Kb of L2 cache, with 512Mb of main memory and Linux (kernel 2.6.5) operating system.

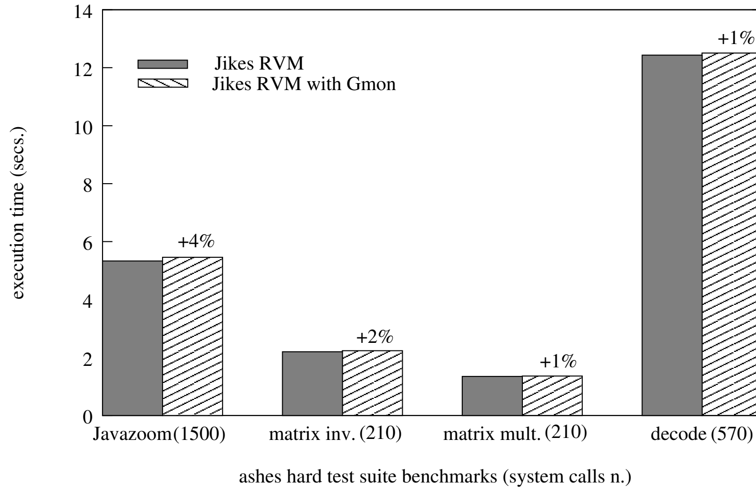


Figure 3: JVM performances

The policy adopted to perform each of these tests is simple, and allows to execute only the system calls required by the benchmark. As previously described, the adoption of more complex policies could result in lower performances.

The first benchmark of the ashes hard test suite collection we have tested is *javazoom*: a mp3 to wav converter that executes about 1500 system calls in about 5.3 secs. Most of these system calls are executed to read the source mp3 file and to write the target wav one. In this case the overhead introduced by Gmon has been measured in about 4% of the JVM time. The second and the third benchmarks execute, respectively, a 200×200 matrix inversion and multiplication. These benchmarks execute about 210 system calls in, respectively, 2.2 and 1.4 secs. The *decode* benchmark, instead, is an algorithm for decoding encrypted messages using Shamir's Secret Sharing scheme. This benchmark executes about 570 system calls in about 12,4 secs. The overhead introduced by Gmon that has been measured in executing these benchmarks is not more than 2% of the JVM time.

6.2 Integration with Globus

The component of the Globus architecture that implements the grid computational service is the Globus Resource Allocation Manager, GRAM, service. "GRAM is a fundamental GT service enabling remote clients to instantiate, manage and monitor, in a secure fashion, computational tasks (jobs) on remote resources" [52]. The GRAM service runs on the shared computational resource. GRAM accepts the

job requests from remote grid users, creates the environment for the job execution, submits the job to the local resource scheduler and monitors the job status. In Globus 3, GRAM consists of two components: the Master Hosting Environment, MHE and the User Hosting Environment, UHE, as described in [24]. The MHE is executed with the privilege of the "globus" user, and receives the request for the creation of a new instance of the computational service coming from a remote grid user. Then, Master Managed Job Factory Service, MMJFS, of the MHE checks the credentials of the grid user to authorize the request, maps the remote grid user in a local user through the gridmap file, and activates the UHE for the local user. The UHE runs with the privileges of the local user and, once activated, it does not terminate, but it waits for other requests from the same local user. Notice that the resource provider can configure the gridmap file in a way such that distinct grid users are mapped on the same local user, sharing the same UHE. Once the UHE is activated, the MHE requests to the Master Job Factory Service, MJFS, that runs in the UHE the creation of a new instance of Managed Job Service, MJS, to manage the job request. The MJS reads the job request, translates and submits it to the local scheduling system, and monitors the job status.

Due to the high modularity of the globus toolkit design, the integration of Gmon has been easy. As previously stated, our prototype implements a computational service for the execution of Java jobs. To properly describe this kind of jobs in the job request, the RSL schema has been be updated to include the "java" jobs as a new job type. The MJS has been modified to invoke the JVM in case of Java job. As a matter of fact, the GRAM perl scripts that submit the application to the local scheduling system have been modified to invoke our security enhanced JVM. Gmon is activated by the JVM, and reads both the policy file and the job request to create the restricted environment for the execution of the application. As a matter of fact, Gmon extracts from the job request the limits to be applied to the resource usage. Since Java application is executed in the local user environment, the MJS, that is executed in the same environment, has been modified to export the job request to Gmon. This modification is straightforward because the MJS Java implementation includes a proper class to manage the gram attributes of the job request. Figure 4 shows the proposed solution, based on the GRAM architecture that is presented in [24].

7 Conclusion and Future Work

This paper presented the adoption of the continuous and fine grained usage control model on grid computational services. As a matter of fact, we improve the security of grid computational services by monitoring the behaviour of the applications

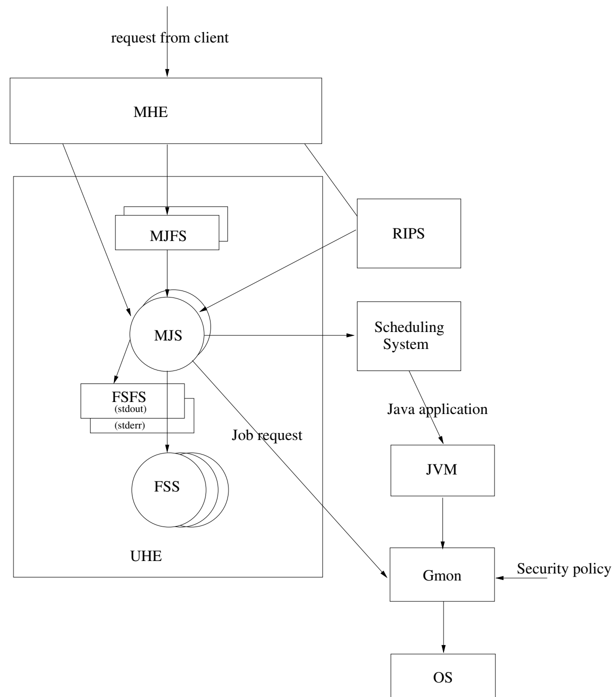


Figure 4: Gmon integration with Globus GRAM

executed on behalf of remote grid users. A prototype implementation demonstrates the effectiveness of this approach.

Our approach is inspired by the concept of continuous usage control proposed by Sandhu and Park in [41] and [42]. We claim that this is a very suitable paradigm to be used in grid environment.

Recently developed mechanisms allow the application monitor to interact with the application, for instance, by performing specific actions to correctly end the program (e.g., see [32]). We plan to test some of these mechanisms in our framework. Moreover, some theoretical results about automatic synthesis (e.g., see [36]) of monitors for enforcing high-level security policies have been recently developed. We plan to test also these results in our framework.

References

[1] B. Alpern, C.R. Attanasio, J.J. Barton, et al. The jalapeño virtual machine. *IBM System Journal*, 39(1):211–221, 2000.

[2] A. Anderson. Java access control mechanisms. Technical report, Sun Microsystems, 2002.

[3] Ashes suite collection benchmarks. http://www.sable.mcgill.ca/ashes/.

[4] 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.

[5] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *International Journal of Software: Practice and Experience*, 32(15):1437–1466, 2002.

[6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[7] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 206–214, 1989.

[8] D.W. 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.

[9] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in Legion. *Future Generation Computer Systems*, 15(5–6):583–594, 1999.

[10] E. Chrinstense, F. Curbera, G. Meredith, and S. Weerawarana. Web service description language. W3C, 2001.

[11] Gnu classpath project. http://www.gnu.org/software/classpath/home.html.

[12] I. Djordjevic and T. Dimitrakos. Towards dynamic security perimeters for virtual collaborative networks. In *Trust Management: 2nd Int. Conference, LNCS*, volume 2995, pages 191–205, 2004.

[13] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. Statl: an attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1-2):71–103, 2002.

[14] D.W. Erwin and D.F. Snelling. UNICORE: A Grid computing environment. In *Lecture Notes in Computer Science*, volume 2150, pages 825–838, 2001.

[15] B. Spencer Jr. et al. Neesgrid: A distributed collaboratory for advanced earthquake engineering experiment and simulation. In *13th World Conf. on Earthquake Engineering*, 2004.

[16] I. Foster and C. Kesselman. The globus project: A status report. In *Proceedings of IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[17] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.

[18] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The physiology of the grid: An open grid service architecture for distributed system integration. Globus Project, 2002. http://www.globus.org/research/papers/ogsa.pdf.

[19] 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.

[20] 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.

[21] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

[22] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings Network and Distributed Systems Security Symposium*, February 2003.

[23] L. Gong. *Inside Java2 Platform Security*. Addison-Wesley, 2nd edition, 1999.

[24] Globus resouce allocation manager. http://www-unix.globus.org/developer/gram-architecture.html.

[25] P.A. Gray and V.S. Sunderam. Icet: Distributed computing and java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 1997.

[26] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure execution framework for java. In *ACM Conference on Computer and Communications Security*, pages 43–52, 2000.

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

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

[29] J.Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, 2000.

[30] Jikes rvm. http://jikesrvm.sourceforge.net/.

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

[32] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[33] T. Y. Lin. Chinese wall security policy — an aggressive model. In *Proceedings of the Fifth Annual Computer Security Applications Conference*, pages 286–293, 1989.

[34] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 1999.

[35] Linpack benchmark – java version. http://www.netlib.org/benchmark/linpackjava/.

[36] F. Martinelli and I. Matteucci. Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties. Tech. report, IIT-CNR, 2005. To appear in Proceedings of the Workshop on Foundations of Computer Security (FCS'05).

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

[38] 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.

[39] M.O. Neary, B. Christiansen, P. Cappello, and K.E. Schauser. Javelin: Parallel computing on the internet. *Future Generation Comp. Systems*, 15:659–674, 1999.

[40] 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.

[41] R. Sandhu and J. Park. Usage control: A vision for next generation access control. In *Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security MMM03, LNCS*, volume 2776, pages 17–31, 2003.

[42] R. Sandhu and J. Park. The $UCON_{ABC}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[43] L.F.G. Sarmenta and S. Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5–6):675–686, 1999.

[44] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[45] Scimark 2.0 benchmark. http://math.nist.gov/scimark2/.

[46] S. Soman, C. Krintz, and G. Vigna. Detecting malicious java code using virtual machine auditing. In *12th USENIX Security Symposium*, pages 153–168, 2003.

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

[48] 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.

[49] M.R. 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.

[50] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open grid services infrastructure (OGSI). Global Grid Forum Recommendation, 2003.

[51] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh Symp. on High Performance Distributed Computing*, 1998.

[52] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *12th IEEE International Symp. on High Performance Distributed Computing*, 2003.