ANR programme ARPEGE 2008

Systèmes Embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011 (extension jusqu'à fin août 2012)

Projet SELKIS ANR-08-SEGI-018 Implémentation des plugins de translation de politiques de sécurité

Livrable 5.3

Editeur : Institut Mines-Télécom / Télécom Bretagne / SERES

T0 + 36

Résumé

Le présent rapport constitue le livrable 5.3 de la tâche WP5 du projet SELKIS. L'objectif est de définir et implanter un plugin de transformation et d'implémentation d'une politique de sécurité. La politique de sécurité est exprimée conformément à un modèle de contrôle d'accès permettant une spécification ne dépendant pas de la mise en œuvre et une gestion des conflits potentiels avant le déploiement de cette politique.
Dans le cadre du projet SELKIS, deux plugins ont été mis en oeuvre. Le premier concerne la transformation d'une politique abstraite en des règles XACML utilisées ensuite pour configurer les composants de contrôle d'accès au niveau d'un web service. Ce Plugin a été notamment utilisé dans le cadre d'une des études de cas du projet. Il s'agit de la gestion des accès aux dossiers des patients dans le CHU de Brest en l'intégrant dans le portail de MEDECOM (l'un des partenaires industriels du projet). Ainsi, une politique OrBAC est transformée au moyen du plugin, puis le service fait appel au serveur qui interprète la requête et applique la politique XACML (*cf* Livrable 4.2).
Le second plugin développé et implémenté dans le cadre du projet SELKIS est présenté dans le présent livrable. Il s'agit de la transformation et le déploiement donc d'une politique dynamique et contextuelle de type OrBAC pour le gestionnaire de sécurité d'une machine virtuelle Java. Tout comme le plugin XACML, ce transformateur est un plugin de MotOrBAC, le module de spécification et d'analyse des politiques OrBAC.

Abstract

The Java execution environment includes several security mechanisms. They are found in the language itself, in the class loader, in the class verifier and in the sandbox in which bytecode is executed. The sandbox isolates the executed bytecode from the host on which the Java virtual machine is executed. The security policy enforced by the sandbox can be configured depending on who runs a program and the origin of the program and offers fine-grained mechanisms to control resource access. However the security policy language offers no higher-level paradigms, such as the abstraction of users into roles, to enable the management of java security policies into large infrastructures. Moreover those policies are static and cannot change depending on the state of the environment into which they are deployed. In this report we present an approach to use of the OrBAC model to configure the sandbox security policy, allowing the use of an implementation independent policy language which offers facilities to manage large sets of JVMs, enables the expression of dynamic security policies and offers an advanced administration model.

1 Introduction

This report presents an approach to use the OrBAC model to configure the security policy of a Java Virtual Machine (JVM). This permits the use of an implementation-independent policy language which offers facilities to manage large sets of JVMs, enables the expression of dynamic security policies and offers an advanced administration model. An implementation of this approach is presented as well as a performance evaluation.

The Java security model relies on several mechanisms. The Java language itself provides strong type checking, a garbage collector and access control to class members and methods. The Java Virtual Machine (JVM) also implements a class loader and a class verifier which checks various properties of the loaded bytecode. The component which isolates the JVM from the operating system in a sandbox is called the security manager. The security manager handles the external boundary of the JVM It controls how code executed by the JVM interacts with resources outside of the JVM. This security manager is configured by a security policy specified in a file loaded when an instance of the JVM is created. This security policy is static and expressed into a language specific to the JVM which offers many low level security mechanisms. For example access control can be done on, among other resources, the file system, network connections, thread management and the AWT framework. If we consider the fact that the security policy language offers no mechanisms to manage the expression and deployment of policies on large infrastructures, configuring this security policy for one user on a given machine is feasible but doing it for hundreds of users with various profiles on hundreds of machines becomes an impossible task.

Given the increasing size of infrastructures into which more and more complex information systems are integrated, system administrators must often configure the security of a wide range of components using ad-hoc configuration languages. This requires them to learn many languages and prevent them from having a global view on the whole system security policy. We believe that the administrators should use ideally one language to express the security policy of a whole system, which should then be enforced through a set of translators that generate configuration files for all the components.

The OrBAC [KBB+03] model attempts to address this problematic by offering several abstract concepts such as abstract entities and contextual security rules which can be used to express a dynamic security policy independently from its implementation. Moreover The OrBAC model offers a framework to analyze and solve rules conflicts, which is impossible to do using multiple security policy languages. The OrBAC model also feature an administration model, the AdOrBAC |CBCC07| model, which can be used to decentralize security policy administration.

We present in this report an approach to express and enforce dynamic security policies for the JVM security manager. Section 2 presents the existing work related to the expression and management of JVM security policies. Section 3 presents the JVM security policy language and how it is used. Section 4 presents the OrBAC model. Section 5 shows how the OrBAC model can be used to express JVM security policies. Section 6 illustrates how policies expressed in OrBAC can be enforced in a JVM. Section 7 concludes this paper and presents future works.

2 Related work

To our knowledge there are few proposal that address the problems related to the expression of JVM security policies. The standard Java Runtime Environment (JRE) includes *Policy Tool*, a very simple application which generates JVM security policies. This application aims at making the policy specification easier by providing a user interface instead of editing directly policies in a text editor. In [Sam04] the authors define a new policy model which includes both positive and negative authorizations. The authors use those two kinds of authorizations to define exceptions, which are not supported in the standard JRE policy language. They also define constraints as temporal constraints exclusively, enabling them to associate temporal conditions to the policy. The concept of permission delegation is also introduced in the model. Since the authors focus on the use of Java in distributed systems by using the Jini[Riv10] framework, the notion of delegation is here restricted to the case of two JVMs communicating through the network and exchanging permissions. However the only constraint on multi-step delegation is the delegation duration time, not the depth of a delegation chain. The

authors also propose to use the notion of groups, a group being a set of principals. This group concept is close to the notion of role in RBAC[FSG+01] or OrBAC. The policies are expressed in XML. Although this work addresses many problems in the expression and in the management of JVM security policies, the fact that it focuses on distributed systems and that the security model is an extension of the existing one makes the contribution less interresting in the context of large infrastructure administration. Actually a system administrator still has to use the new ad-hoc language to configure the JVM security policy and has no global view over the security policy of the whole system.

In [ZPPS06] the authors use an authorization specification language (ASL [JSSB97]) to express security policies for mobile devices. The implementation is done using a modified JRE security manager which parses XACML policies translated from the ASL representation. Unfortunately the authors only present the XAMCL format and not the abstract policy expressed in ASL.

In [DC07] the authors address the lack of flexibility of the Java 2 Micro Edition (J2ME) security model. They extend the J2ME security model and use the SPL language [RZFG99] to express the security policy enforced on mobile devices. The J2ME is only used on mobile devices and does not use the security manager of the standard JRE so this work is not applicable in our context. The use of SPL offers policy administrators a wider view of the security deployed on a mobile device.

In [CCBSM04], the authors present an approach to express firewall security policies using the OrBAC model to translate them into native firewall configuration languages. The model is independent from the targeted firewall implementation.The authors choose to represent each firewall by an organization, each firewall defining its own security policy through the specification of abstract rules in the corresponding organization. Although this paper focuses on the abstraction of firewall policy and does not address JVM policy modeling, we adopt a similar approach by defining the security policy enforced in sets of JVM hosts into organizations (cf section 5.2).

3 JVM security policies

The current security model implemented by the JVM security manager relies on different security mechanisms. In this paper we focus on the closed security policy which defines the sandbox boundaries. This policy specifies the permissions granted to bytecode depending on its source and the principal as which it is executed, a principal being the identity assigned to an entity, which can be the result of an authentication. This contrasts with previous JDK version before version 1.4 where access control was based only on which code is executed.

3.1 Policy syntax

By default a single system-wide policy is defined in a file and user specific policies can optionaly be defined. The system-wide policy file defines an optional keystore entry which is used to check the public keys associated with signed bytecode. The rest of the policy file defines the permissions granted to code through the specification of "grant" entries. A grant entry specifies the sources and principals which are granted a list of permissions. The policy file syntax is defined as follows:

```
grant signedBy "signer_names", codeBase "URL",
  principal principal_class_name "principal_name",
  ... {
    permission permission_class_name "target_name", "action",
      signedBy "signer_names";
    ...
  };
```

Permissions have an *action* parameter which is not mandatory for all permissions. Note that the JRE security policy is static by construction, no dynamic condition can be associated with permissions. We believe that current security requirements met by system administrators show the need for dynamic policies. System administrators should not have to learn such configuration language but should instead use higher level paradigms to express the security of an information system. Note that in this paper we do not address the security problems related to the lack of protection of a Java Runtime Environment (JRE) default installation. For example if the underlying operating system's access control mechanisms do not correctly restrict the access to a JRE setup, some of its components could be changed and/or the policy file could be easily modified [WCLX01].

Actually the *target_name* parameter may also implicitly contain an action. For example the *SecurityPermission* type defines the *createAccess ControlContext* target which contains the *create* action and the *AccessControlContext* object. Note that the JRE security policy is static by construction, no dynamic condition can be associated to any permission. We believe that current security requirements met by system administrators show the need for dynamic policies. System administrators should not have to learn such configuration language but should instead use higher level paradigms to express the security of an information system. The following grants are taken from a JDK7 standard installation policy file:

```
grant codeBase "file:/usr/lib/jvm/java-7-openjdk-common/

jre/lib/ext/*" {
  permission java.security.AllPermission;
};
grant {
  ...
  permission java.lang.RuntimePermission "stopThread";
```

```
    permission java.net.SocketPermission "localhost:1024-",
     "listen";
    ...
};
```

The first grant entry specifies that all classes installed in the *ext* subdirectory of the JDK7 installation are given all permissions. This subdirectory has been created so that system administrators can install standard libraries which are trusted without having to write specific permissions for them. Notice that the source associated with this grant is a filesystem path. The second grant entry specifies that any code can stop a thread and listen for connections on ports above 1024. The standard policy file also includes rights for any code to read standard system properties like the JVM version, the file separator, the OS version, etc... Note that in this paper we do not address the security problems related to the lack of protection of a Java Runtime Environment (JRE) default installation. For example if the underlying operating system's access control mechanisms does not correctly restrict the access to a JRE setup, some of its components could be changed and/or the policy file could be easily modified [WCLX01]. Another example of vulnerability in a default JRE setup is the fact that the policy file can be overriden when starting the JVM through the *java.security.policy* option. This can be disabled by changing a property in the *java.security* file.

3.2 Permission types

The Java 2 security manager default implementation defines a set of permission types which define the granularity of the JVM sandbox boundary. We do not present the complete Java security policy due to space limitations and since most of the permissions types are related to very specific use cases. The use case presented in section 5 focuses on network access and filesystem access hence we only use the security manager *FilePermission* and *SocketPermission* types . We do not review in details all permission types here but the table presented on figure 1 lists all the permission types and give some information about their semantic.

The *AllPermission* type has been defined to provide a simple way to completely open the sandbox security policy as it implies all the permission types. For instance the *ext* directory presented in the previous section example is granted the *AllPermission* permission.

4 The OrBAC model

The OrBAC model addresses many problems faced by system administrators in big infrastructures when specifying security policies. For example, a

Permission type	Description
AllPermission	grants all permissions
SecurityPermission	the SecurityPermission object is used to guard access to the Policy, Security, Provider, Signer, and Identity objects. It can be used for example to allow some bytecode to replace the security policy
AWTPermission	guard access to the Abstract Windowing Toolkit, for example access to the clipboard or the application display surface
FilePermission	control access to the filesystem. It supports the *read*, *write*, *execute* and *delete* operations
SerializablePermission	if granted some bytecode can replace the serialization implementation
ReflectPermission	controls the behaviour of the reflection mechanisms, i.e it can disable the standard language access check for public, private and protected class members
RuntimePermission	controls many aspects of the behaviour of an application. For example code can be granted the right to create a new class loader, load native libraries or define new classes at runtime
NetPermission	grants the ability to retrieve authentication information
SocketPermission	controls access to the network via sockets. It supports the *accept*, *connect*, *listen* and *resolve* operations
SQLPermission	if granted a bytecode can replace the logging object used when accessing databases
PropertyPermission	controls access to the system properties
LoggingPermission	controls access to the standard loggin system
SSLPermission	if granted some bytecode can modify the behaviour of SSL connections or get access to SSL sessions data
AuthPermission	AuthPermission object is used to guard access to the Subject, SubjectDomainCombiner, LoginContext and Configuration objects
PrivateCredentialPermission	used to protect access to private credentials belonging to a particular subject
DelegationPermission	restricts the usage of the Kerberos delegation model, ie, forwardable and proxiable tickets
ServicePermission	used to protect Kerberos services and the credentials necessary to access those services
AudioPermission	guards access to the audio system resources for playback and recording
UnresolvedPermission	used to hold permissions for which the bytecode has not been yet loaded when the security policy is loaded

Figure 1: List of all the permission types defined in the standard JRE

company infrastructure may be spread accross different countries with different legislations, the employee turnover can be very high, its contractors may need to access its information system and more generally, the security policy must be modified on a regular basis. In such context, the need for multiple administrators and a dynamic security policy become central.

OrBAC aims at modelling a security policy centered on the organization which defines it or manages it. An OrBAC policy specification is done at the organizational level, also called the abstract level, and is implementation-independent. The enforced policy, called the concrete policy, is inferred from the abstract policy. This approach makes all the policies expressed in the OrBAC model reproducible and scalable. Actually once the concrete policy is inferred, no modification or tuning has to be done on the inferred policy since it would possibly introduce inconsistencies. Everything is done at the abstract policy specification level. The inferred concrete policy expresses security rules using subjects, actions and objects. The abstract policy, specified at the organizational level, is specified using *roles*, *activities* and *views* which respectively abstract the concrete subjects, actions and objects. The OrBAC model uses a first order logic formalism with negation. However since first order logic is generally undecidable, we have restricted our model in order to be compatible with a stratified Datalog program [Ull89]. A stratified Datalog program can be evaluated in polynomial time.

Each organization specifies its own security rules. Some *role* may have the permission, prohibition or obligation to do some *activity* on some *view* given an associated *context* is true. The *context* concept [CCB08] has been introduced in OrBAC in order to express dynamic rules. Contexts are defined through logical rules which express the condition that must be true in order for the context to be active. In the OrBAC model such rules have the predicate *hold* in their conclusion. As suggested in [CCB08], contexts can be combined in order to express conjunctive contexts (denoted &), disjunctive contexts (denoted $\oplus$) and context negation (denoted $\overline{ctx}$, ctx being a context name). Once the security policy has been specified at the organizational level, it is possible to instantiate it by assigning concrete entities to abstract entities.

5 Expressing JVM policies in OrBAC

In this section we show how the JVM policy model can be represented using OrBAC. The main idea motivating this initiative is that ideally system administrators should be able to use the same security policy model to specify the security policy of a whole system. In the case of the OrBAC model, administrators should be able to use the same abstract entities to define the security rules which are enforced by heterogeneous security components. For example let us consider a generic *doctor* role used in a hospital policy

to express the security rules common to all physicians. This role is refined by defining sub-roles such as *surgeon*, *radiologist* or *anethesist* to specify security rules that only apply to specialists. A person empowered in one of the sub-roles of the *doctor* role may access various data about his/her patients using various peripherals and applications which all must enforce the security policy. Among those applications some are executed in the Java runtime environment and others are nativelly executed.

In this report, we consider as a use-case a Java client application which accesses a database containing the patients medical files. Informally, the security policy associated with this application is the following: physicians can use the application , the application can open network connections to the database and can make some modifications to the local filesystem in order for it to correctly run.

5.1 Supported permissions types

As said in section 3.2, in this paper we only focus on the *FilePermission* and *SocketPermission* types. We review here in details what they represent and how they are expressed in the JVM security policy model. Although we do not model other permission types in this report, the policy translation and deployment mechanism would still be the same.

5.1.1 *FilePermission* permission type

The *FilePermission* type represents an access to a file or directory. An instance of this permission consists of a pathname and the set of actions which can be done on the pathname. A pathname is either a file or a directory and the syntax allows the use of wildcards. The $*$ indicates all the files in a directory and $-$ indicates all the files in a directory plus recursively all files and directories contained in the directory. The possible actions on a file or directory are *read*, *write* (which implies the permission to create), *execute* and *delete*.

Section 3.1 gives an example of such a permission to specify the access to the *ext* sub-directory of the JDK7 installation. Here is another example which uses a Java system property to make the policy more generic:

```
permission java.util.PropertyPermission
    "user.home", "read";

permission java.io.FilePermission
    "${user.home}${/}.somefile", "read,write";
```

The *user.home* system property is used to compute a platform-independant path in which the application can read and write but not execute files.

5.1.2 *SocketPermission* permission type

The *SocketPermission* type represents an access to the network via sockets. An instance of this permission consists of a host specification and a set of operations which specifies how connections can be estabiliched with the host. A hostname is specified as follows:

```
host = (hostname | IPaddress)[:portrange]
portrange = portnumber | -portnumber | portnumber-
[portnumber]
```

Four connection methods can be specified: *accept*, *connect*, *listen* and *resolve*. *resolve* is implied by the first three methods, i.e if the JVM can connect to other machines, accept connections or listen to connection then it can resolve host names. Note that this representation of network activities do not take into account network protocols and their attributes and states, limiting considerably the expression of network security policy compared to personnal firewalls.

5.2 OrBAC representation of JVM policies

This section focuses on the definition of OrBAC abstract entities (organizations, roles, activities, views, contexts, rules) necessary to model the filesystem and network permissions of the Java security policy model.

5.2.1 Organizations

In the context of this report, a possible modeling choice would be to represent each machine running the JVM by an organization but this would possibly lead to a huge number of organizations, making the specification of JVM policies more complicated and error prone than the manual configuration of the machines. We argue here that a set of machines running the same Java applications can be abstracted into one organization in which the common security policy is defined. For example if we consider tablet PCs used by physicians when visiting their patients in a hospital, we can assume that they will all be running the same set of Java applications, or at least they can be grouped in sets of machines running the same applications. We use organization attributes to infer the set of JVM hosts on which a JVM security policy must be deployed. More precisely, let us consider a set S_{vm1} of machines hosting the JVM on which a set of Java applications will be ran. The predicate jvm_target is used to associate each element of S_{vm1} with an organization O_{vm1} modeling this set.

To avoid defining several times the same subset of the security policy in different organizations, an organization hierarchy should be defined. This

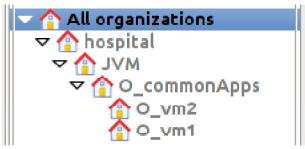

Figure 2: A simple example of an organization hierarchy defined in an hospital

way common security rules can be defined in the super organization of organizations representing different machine sets sharing some common Java application(s). We chose to define a default JVM organization as the root of the hierarchy representing the sets of machines running java applications. Figure 2 shows an example of such hierarchy defined using the MotOrBAC[ACCBC08] tool where O_{vm1} and O_{vm2} represent two subsets of machines sharing some common applications for which the policy is defined in $O_{commonApps}$. MotOrBAC is a security policy editor which implements the OrBAC model.

5.2.2 Roles

In our approach we do not define specific roles linked to the specification of JVM security policies. The roles are completely defined by the use case for which a security policy is specified. However since a java application may send and receive network packets, some activities, defined further below, model those operations. We do not propose a new model from scratch for the modeling of network operations but rather use the approach defined in [CCBSM04]. Actually we do not follow exactly the same semantic regarding the modeling of network traffic direction. In [CCBSM04] a role models a machine sending network packets to a machine modeled as a view. This requires to create views corresponding to some roles to be able to specify traffic going from and to a machine. In our case the roles model users using a java application running on a machine which sends and receive traffic. We do not create corresponding views for each role to be able to express the security policy for incomming traffic but choose to encode the network traffic direction in the activities modeling the traffic emission and reception.

5.2.3 Activities

As said previously, we only model in this paper the *FilePermission* and *SocketPermission* types. The actions defined by the *FilePermission* type are already very generic operations that do not need much abstraction to define the corresponding activities. In fact in our experience, we noticed that the *read*, *write* and *delete* activities which abstract actions consist-

ing in accessing various data storage entities are often present in OrBAC policies. The action of executing something is more specific to the use of software and is also easy to abstract into the *execute* activity. We choose to model the $FilePermission$ actions by making the hypothesis that in a super-organization of the JVM organization defined previously, generic *read*, *write*, *delete* and *execute* activities are defined. A super-activity of *read*, *write* and *delete* can be modeled as the *handle* activity (cf figure 3) , thus simplifying the specification of filesystem security rules for a JVM.

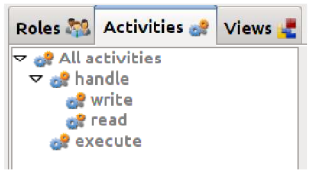

Figure 3: The activity hierarchy modeling the actions defined by the $FilePermission$.

Regarding the modeling of the $SocketPermission$ type, we follow the approach presented in [CCBSM04] to specify network security policies with OrBAC, i.e activities are seen as abstraction of network services. The activity hierarchy defined in [CCBSM04] is represented on figure 4 and consists of activities *all_protocols*, *tcp*, *udp* and *icmp*. However we have seen in section 5.1 that the $SocketPermission$ type offers coarse granularity and do not take into account network protocols. We define a sub-activity of the *all_protocols* activity called *bidirectionnal* defined in the JVM organization and two sub-activities of the *bidirectionnal* activity, *send* and *receive*, also defined in the JVM organization. This way we modify locally, i.e in the JVM organization, the semantic of the activity modeling proposed in [CCBSM04] but retain the ability to use the structuration of views related to the firewall security policy. This allows us to use the aforementioned approach along with our approach to integrate the security policy specification of JVM hosts inside a more global security policy. The $SocketPermission$ type does not allow to use the network protocol type to express the network security policy but port numbers can be used to identify network services. Actions considered as the *bidirectionnal*, *send* and *receive* activities have a *port* attribute which expresses a port or a port range. For example the following assertions represent two actions modeling the *ssh* and *mysql* services:

$action(ssh)$. $action(mysqlV5)$. $port(ssh, 22)$. $port(mysqlV5, 3306)$.

Port ranges are expressed the same way they are in the $SocketPermission$ type syntax.

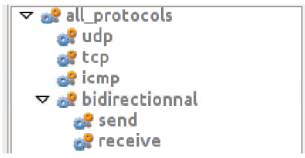

Figure 4: The activity hierarchy modeling the network protocols used to model the *SocketPermission*

5.2.4 Views

To model the *FilePermission* type, the set of views to be defined and their hierarchies depend mainly on the design of the Java applications the subjects will use. We choose to represent files and directories by objects having a *path* attribute expressing the target path. Such an attribute is modeled using the following predicate:

$targetPath(obj, path)$

This models the fact that object *obj* has a target path represented by *path*. We use the same syntax as the *FilePermission* type to express the path. For example to represent a directory *application/log* located in the current user home directory by an object called $fooDir$, the following assertion would be true:

$targetPath(fooDir, \${user.home}\${/}application\${/}log)$

This object would then be used in a view representing a set of directory or files belonging to some applications having the same right on them. Here the ${/} substring is a platform-independent representation of the file separator.

Views for the *SocketPermission* type are defined in a similar way as in [CCBSM04]. Such views represent sets of network machines, identified by their address or name. For example the *toDatabase* view can be defined to represent a set of machines on which databases are installed. Objects representing network machines have an *address* attribute which represents the machine ip address or network name. View definitions can be used to manage large sets of machines. Indeed, instead of manually inserting large numbers of objects into views, view definitions can be used to automatically insert objects depending on the network address. For instance the following view definition, defined in the *hospital* organization of the example used earlier, says that an object representing a network machine H is used in the *toDatabase* view representing databases if H is part of some subnet and that it is not used in the *toBackup* view which models network backup hosts:

$use(hospital, H, toDatabase)$:-
$networkAddress(H, A)$,
$subNet(A, 10.0.0.0, 24)$,
$\neg use(hospital, H, toBackup)$.

Using the same model as in [CCBSM04] allows system administrators

to use the same views that have been defined when expressing the network security policy in OrBAC, thus giving administrators a more global view of their security policy.

5.2.5 Contexts

In section 3.1, we saw that the JVM security manager can grant right to code depending on the location from which the bytecode is fetched (from the local filesystem or the network) and the identity of the subject who has signed the bytecode. This feature can be modeled by OrBAC contexts: we define two contexts types corresponding to the two conditions.

The first context type, named *codeBase*, models the code source condition expressed by the *codeBase* keyword in the JVM security policy syntax. The following derivation rule shows an example of such context:

$hold(JVM, S, \_, \_, codeBaseFoo)$:-
$byteCode(S, B)$,
$codeBase(B, database.intranet.mycompany.com)$.

This context is true in organization JVM if the bytecode executed by the subject S has been downloaded from a server *database.intranet.mycompany.com* in the intranet of some company.

The second context type, named *signedCode*, models the code source condition expressed by the *signedBy* keyword in the JVM security policy syntax. The following derivation rule shows an example of such context:

$hold(JVM, S, \_, \_, signedCodeFoo)$:-
$byteCode(S, B)$,
$signedBy(B, peter)$.

This context is true in organization JVM if the bytecode executed by the subject S has been signed by Peter. The JVM security policy syntax support the specification of code signed by multiple subjects, which can be easily taken advantage of in our modeling. For example the following context models a condition where at least one of the developpers of some bytecode must have signed it in order to be true:

$hold(JVM, S, \_, \_, signedCodeFoo)$:-
$byteCode(S, B)$,
$developpedBy(B, D)$,
$signedBy(B, D)$.

Here the $developpedBy(b, d)$ predicate is true if bytecode b has been developped by subject d.

Those contexts can be defined and used in the specifications of security rules inside the JVM organization.

5.2.6 Security rules

In our approach we use the standard implementation of the JVM security manager, which implements a closed policy. Hence in this paper the transla-

tion process only translates permissions specified in the JVM organization and its sub-organizations. The JVM security policy is specified inside the organization hierarchy defined in the JVM organization using the roles defined in the super organizations of JVM. The activities, views and contexts defined previously are used to define all the JVM abstract permissions.

However the system administrators are not limited to the use of the previously defined context types when specifying the abstract permissions. As said in the introduction of this paper, JVM security policies are static. In our approach any other OrBAC context type can be used to make the policy dynamic. Actually contexts can be composed using conjunction and disjunction operators to associate complex contextual conditions with permissions. When a context state changes for some concrete entity triple $\{subject, action, object\}$ in the JVM organization or one of its sub-organizations, the new infered concrete policy is pushed on the hosts specified in the corresponding organization attributes. Such context can be, but is not limited to, a temporal context, a spatial context expressing a condition on the position of a subject in space, a condition on some concrete entity attribute or a condition on the system state. Since the JVM security manager standard implementation does not refresh the security policy if the policy file is modified after a JVM instance is started, we have modified the standard implementation to trigger this refreshment.

6 Enforcing JVM OrBAC policies

In this section we present the OrBAC JVM policy to JVM policy translation algorithm and illustrate it with an example and an implementation. The translator which also updates the security policy files on target hosts running a JVM is implemented as a MotOrBAC plugin. MotOrBAC is used to specify the abstract security policy and associate concrete entities with abstract entities. It is also used to specify the list of hosts on which the security policy must be deployed.

6.1 Translation algorithm

The algorithm does not translate the abstract security policy but rather the concrete security policy which is infered by the OrBAC Application Programming Interface (API) inference engine. The OrBAC API is used by MotOrBAC to process OrBAC policies. The concrete permissions infered by the OrBAC API have many attributes like the contexts to which they are associated and the organization in which they have been infered. Each infered concrete permission is parsed to generate a grant entry. The JVM security policy syntax does not support the specification of different policies for different users in one file. Hence the translation process generates one policy file per subject.

Let us consider a subject for which a set P of concrete permissions related to a JVM security policy has been generated. For each permission p in P, the translation process generates a grant entry for each contextual condition on the origin of the code. Then for each of those entries, the list of signers are added if the contextual condition contains such condition. The type of permission to add to the grant entry and its attributes are extracted from the parameters of p.

When the list of permissions for a subject changes because some contexts have been activated or deactivated, the corresponding security policy file is generated and pushed on the hosts the user may use.

6.2 Example

We consider an example based on the one presented in section 5.2 of an OrBAC policy specified in an hospital. We assume that physicians use tablet PCs when they visit their patients to access their files. The client application is a Java applet, which must be signed by the main developper *bob*, running inside a web browser which connects to a database where patient files are stored. The applet can connect to the database but not the opposite. The applet used by physicians uses a directory structure created in the user home directory. This directory is called *appletDir* and contains three other directory storing specific files: the *resource* directory, which can only be read, the *log* directory, which can only be written and the *temp* directory which can be read and written. We assume that the OrBAC policy is already structured according to the roles defined in the hospital and that a network security policy has been defined according to the approach in [CCBSM04]. Hence we assume a *doctor* role has been defined in organization *hospital*. We define a sub-organization of the JVM organization, called *appletOrg*, in which the security policy applied to peripherals running the applet is deployed. The list of hosts on which the policy is deployed is specified in the *appletOrg* organization.

The technical details for the considered use case are the following:

- a DNS server is used in the private network
- the database server and the client Java applet are run on linux machines
- the database is a Mysql 5 database listenning on port 3306 and hosted on machine *database.intranet.hospital.com*
- the webpage from which the applet is retrieved is:

  http : *//applet.intranet.hospital.com*

A *mysql* action models the action of using the Mysql database. Its *port* attribute is set to 3306. It is considered as the *send* activity in the

JVM organization because the applet connects to the database but does not accept connections. Two actions *readFilesystem* and *writeFilesystem* are considered respectively as the *read* and *write* activities in the JVM organization.

We assume that the database server has already been modeled by a view *database* in the network related part of the OrBAC policy. The object *db*1 is used in this view in the *hospital* organization. Its *address* attribute is set to the host name specified above. Three views are defined to model the applet directories: *resource*, *log* and *temp*. Three objects *resource_applet*1, *log_applet*1 and *temp_applet*1 are used respectivelly in the *resource*, *log* and *temp* views in the JVM organization. Their *targetPath* attribute is set respectivelly to $\${user.home}\${/}appletDir\ \${/}resource$, $\${user.home}\${/}appletDir\${/}log$ and $\${user.home}\${/}\ appletDir\${/}temp$.

We define a *codeBase* context to model the condition on the applet source bytecode:

$hold(appletOrg, S, \_, \_, cbCtx)$:-
  $byteCode(S, B)$,
  $codeBase(B, applet.intranet.hospital.com)$.

The following *signedBy* context models the condition on the applet signed bytecode:

$hold(appletOrg, S, \_, \_, scCtx)$:-
  $byteCode(S, B)$,
  $signedBy(B, bob)$.

We also define a *visitTime* temporal context in the *hospital* organization which is only active when doctors are visiting their patients. Using the previously defined abstract entities we can write the abstract permissions corresponding to the example:

$permission(appletOrg, doctor, send, database, scCtx\&cbCtx\&visitTime)$
$permission(appletOrg, doctor, read, resource, scCtx\&cbCtx)$
$permission(appletOrg, doctor, write, log, scCtx\&cbCtx)$
$permission(appletOrg, doctor, handle, temp, scCtx\&cbCtx)$

Assuming that a subject *daniel* is empowered in the doctor role in the *hospital* organization and that the *visitTime* is active for *daniel* in the *hospital* organization, the following concrete permissions are infered:

$permission(daniel, mysql, db1)$
$permission(daniel, readFilesystem, resource\_applet1)$
$permission(daniel, writeFilesystem, log\_applet1)$
$permission(daniel, readFilesystem, temp\_applet1)$
$permission(daniel, writeFilesystem, temp\_applet1)$

6.3 implementation

We have developped a MotOrBAC plugin implementing the translation process and the configuration of the JVM hosts. Four virtual machines have

been created to represent the database server, a tablet PC running the applet, a webserver from which the applet is downloaded and the administrator host running MotOrBAC and the plugin. Generated policy configuration files are uploaded by the plugin into the users home directory through ssh connections using public key authentication. The list of hosts to which the files are transfered is infered from the *appletOrg* organization attributes as specified in section 5.2. Configuration files are transfered to the hosts whenever a change in contexts state have been triggered. We have modified the standard JVM security manager implementation to reload the security policy while a JVM instance is executed when a change is detected in the policy file. This way JVM security policies are dynamically updated as the concrete policy evolves in time.

From the concrete permissions infered in the previous section and the concrete entities attributes, a JVM security policy configuration files is generated:

```
grant signedBy "bob", codeBase "http://applet.intranet.hospital.com" {
  permission java.io.FilePermission "\${user.home}\${/}appletDir\${/}resource", "read";
  permission java.io.FilePermission "\${user.home}\${/}appletDir\${/}log", "write";
  permission java.io.FilePermission "\${user.home}\${/}appletDir\${/}temp", "read,write";
  permission java.net.SocketPermission "database.intranet.hospital.com:3306", "connect";
};
```

Note that generated grant entries having the same *signedBy* and *codeBase* conditions are grouped to generate smaller files. We use rsync to generate less network traffic when uploading the configuration files. Note that although we have used only linux machines in our proof of concept, the generated policy files could directly be used on other operating systems as we used generic variables to identify the current user home directory and the file system separator.

Although our proof of concept uses a small policy deployed on few hostss, we tested the translation algorithm on larger policies. More precisely we used the previously defined policy and added subjects empowered in the *doctor* role to increase progressively the number of concrete rules to translate. Figure 5 shows the time needed to translate the OrBAC policy in milliseconds on the vertical axis and the number of concrete rules to translate on the horizontal axis.

7 Conclusion

In this report we presented an approach to abstract the JVM security policy model into the OrBAC model and an implementation as a MotOrBAC plugin. This allows system administrators to use a powerful dynamic security model to express the security requirements applied to JVM instances instead of applying the ad-hoc policy language defined in the standard JRE. We think that the main advantage of this approach is that system administrators can use the same model and the same abstract entities to define

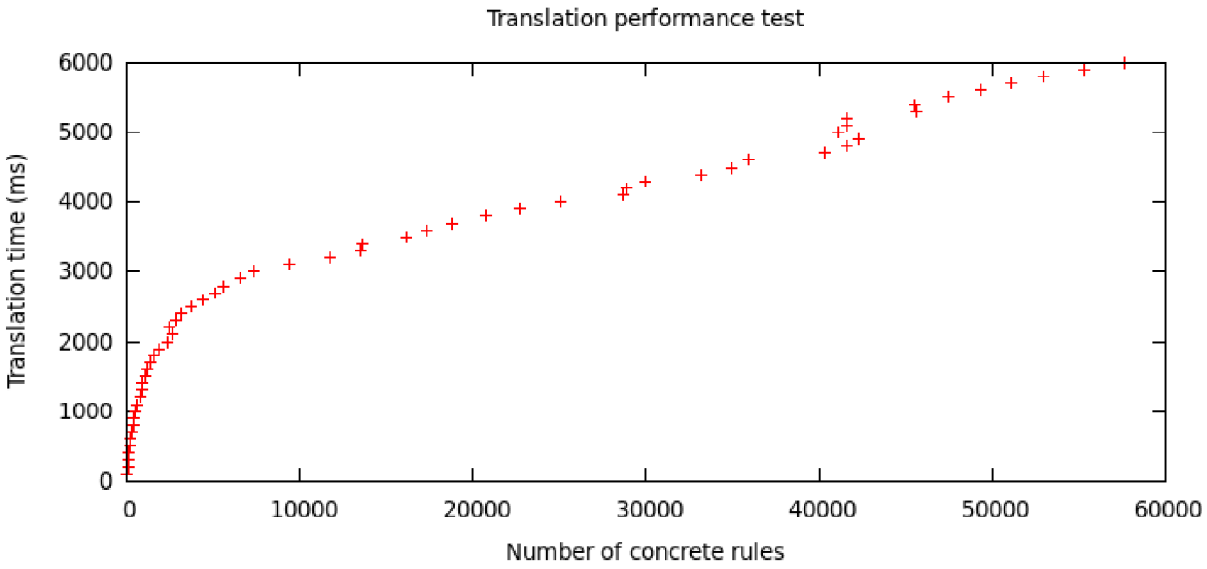

Figure 5: Experimental results with larger policies

security policies applied to heterogeneous security components, thus giving them a global view of their information system without having to specify separate policies in different languages for each component. Moreover the dynamic nature of OrBAC policies and the dynamic deployment of configuration files implemented in our approach provides means to change the security properties of running java applications, which is not possible for a standard JRE.

Another main advantage of this approach is that system administrators can use the AdOrBAC [CBCC07] model to administrate the specification of JVM policies in OrBAC. MotOrBAC implements the AdOrBAC model, including the delegation model, which means that administration tasks can be managed in our proof of concept. For example a system administrator can delegate to another subject the right to define only permission related to the network policy of JVMs.

Using the standard security manager implementation limits the granularity of the policies we can express, especially regarding the network policies. We plan to modify the security manager implementation to refine its boundaries and directly integrate the OrBAC API inside it. This way OrBAC policies specified with MotOrBAC could be directly interpreted without the need for a translator.

References

[ACCBC08] F. Autrel, F. Cuppens, N. Cuppens-Boulahia, and C. Coma. Motorbac 2: a security policy tool. In *Third Joint Conference on Security in Networks Architectures and Security of Information Systems (SARSSI)*, 2008.

[CBCC07] N. Cuppens-Boulahia, F. Cuppens, and C. Coma. Multi-granular licences to decentralize security administration. In *First International Workshop on Reliability, Availability, and Security (WRAS). Paris, France*, 2007.

[CCB08] F. Cuppens and N. Cuppens-Boulahia. Modeling contextual security policies. In *International Journal of Information Security (IJIS). Vol. 7, no. 4. August*, 2008.

[CCBSM04] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Second Workshop on Formal Aspects in Security and Trust (FAST)*, 2004.

[DC07] Iulia Ion Boris Dragovic and Bruno Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *In Proceedings of the Annual Computer Security Applications Conference ACSAC*, 2007.

[FSG+01] David F. Ferrailo, Ravi Sandhu, Serban Gavrila, D.Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for rbac. In *ACM Transactions on Information and System Security*, 2001.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Eliza Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 474–485, New York, NY, USA, 1997. ACM.

[KBB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarteand A. Miège, C. Saurel, and G. Trouessin. Organization based access control. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, 2003.

[Riv10] Apache River. Jini: a network architecture for the construction of distributed systems, 2010.

[RZFG99] Carlos Ribeiro, André Zúquete, Paulo Ferreira, and Paulo Guedes. Spl: An access control language for security policies with complex constraints. In *In Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 1999.

[Sam04] Frédéric Samson. *Alternative Java Security Policy Model*. Phd. thesis, Université Laval, 2004.

[Ull89] Jeffrey D. Ullman. Principles of database and knowledge-base systems. In *Computer Science Press*, 1989.

[WCLX01] D. Wheeler, A. Conyers, J. Luo, and A. Xiong. Java security extensions for a java server in a hostile environment. In *Proceedings of the 17th Annual Computer Security Applications Conference*, ACSAC '01, pages 64–, Washington, DC, USA, 2001. IEEE Computer Society.

[ZPPS06] Xinwen Zhang, Francesco Parisi-Presicce, and Ravi Sandhu. Towards remote policy enforcement for runtime protection of mobile code using trusted computing, 2006.