RZ 3227 (# 93273) 04/17/00
Computer Science/Mathematics 10 pages

Research Report

An Operational Semantics of Java 2 Access Control

Günter Karjoth

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

An Operational Semantics of Java 2 Access Control

Günter Karjoth

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Java 2 Security enhanced with the Java Authentication and Authorization Service (JAAS) provide sophisticated access control features via a user-configurable authorization policy. Fine-grained access control, code-based as well as user-based authorization, and implicit access rights allow the implementation of real-world policies, but of the cost of increased complexity. In this paper we provide a formal specification of the Java 2 and JAAS access control model that helps remove ambiguities of the informal definitions. It defines Java 2 access control in terms of an abstract machine, whose behavior is determined by a small set of transition rules. We illustrate the power of Java 2 access control by showing how commonly encountered authorization requirements can be implemented in Java 2.

1 Introduction

Since JDK 1.2, the Java Software Development Kit provides sophisticated access control features via a user-configurable authorization policy and implemented by protection domains [4]. When a Java class is loaded, it is associated with a number of permissions based on the signer's identity and the loading location. Whenever a controlled resource is accessed, the runtime verifies that all classes in the method call stack have sufficient permissions for accessing that resource. The recent Java Authentication and Authorization Service (JAAS) augments JDK 1.2 with support to authenticate the principal who runs the code and to enforce new access controls on who was authenticated [8].

The Java computing platform and the Java Beans component architecture make it feasible to build large-scale systems. Commercial application developers will need sophisticated access control functionality that can deal with the security requirements of such systems, for example in the enterprise world. Vendors that implement Java 2 Security and JAAS will need an unambiguous specification to implement access control correctly. In particular, when vendors develop more efficient implementations to improve performance, a formal specification is necessary to show its equivalence.

This paper provides a formal model of Java 2 access controls with JAAS that abstracts from possible implementations. It defines the basic data structures and gives an operational semantics in terms of abstract transitions. The model enables software engineers to define precisely the required security of their system, to develop efficient decision procedures, and to show their correctness.

Stack inspection, an essential part of the Java 2 security architecture, was described and analyzed first by Wallach and Felten [10]. They use a belief logic to define the security mechanisms as implemented in Netscape, which additionally allows permissions to be enabled/disabled one by one. They also show that a new and more efficient form of stack inspection, called "security-passing style", is equivalent to the original stack inspection system.

In [7], Kassab and Greenwald develop a formal model of a beta version[1] of JDK 1.2 security architecture. It is a state-based model that uses access control matrices to model protection states. For each thread, there is a domain matrix representing its protection state. Whenever a thread calls a new method, the corresponding domain is added to the domain matrix. The domain (row) is calculated from the policy matrix taking the nesting of domains into account. To cope with privileged code, they regard a domain matrix to behave like a stack. Their access control decision function recognizes corresponding marks and terminates evaluation of the intersection of domains on the stack. However, permission implication and the JAAS framework are not considered in their model.

This paper is organized as follows. In Section 2, we review the Java 2 access control model. Section 3 shows a formal model of Java 2 access control combined with the JAAS framework. In Section 4, we analyze the expressiveness of Java 2 authorization. We present various policy implementations that take advantage of principal hierarchies as well as permission hierarchies. Finally, Section 5 concludes the paper.

2 Java 2 Authorization

In Java 2, a set of policy files defines the authorization state that determines whether a given request has to be considered authorized. Authorization is semi-static, as the authorization state is already specified before the Java system starts.[2] Another policy can be loaded later via the `refresh` method of the Policy object, but it is implementation-dependent how this is done. In the way the policy files and thus the authorization state can be changed, Java follows the administration paradigm more closely than the owner paradigm.

A Java *subject* represents a grouping of related information for a single entity, such as a person. Such information includes the subject's identities as well as its security-related attributes, for example passwords and cryptographic keys. Each identity is represented as a *principal* within the subject. Examples of Java principals include names such as e-mail addresses or employee numbers, groups such as departments, or public keys that provide a very scalable name representation. Principals bind names to a Subject.

The specification of access controls is based on static properties of authorization units. Permissions, which are access rights on resources, are granted to locations of code, set of signers of code, and set of principals. Note that permissions are granted to classes, which are static Java code, and not to instances, which are instances of classes [4, p. 67]. Only permissions that

[1] Privileged code is based on the `beginPrivileged()` and `endPrivileged()` methods, which where replaced by the `doPrivileged` method in the final release of Java 1.2.

[2] For optimization, a Java system might delay the instantiation of the policy file until the first permission check. However, this may lead to a different behavior when the content of the policy file should change between the time the policy class is instantiated and the time the first security check is invoked [4, p. 67].

represent approvals can be given. There are implicit access rights, as an ordering on permissions and/or principals can be defined. Signers as well as principals can be composed into compound entities by set intersection, i.e. permissions are only granted if all elements are present.
Threads are the active entities of a Java system. They generate requests which are validated by the reference monitor in accordance with a given access control policy. Authorized requests are mediated to the corresponding receiver object, whereas unauthorized requests have to be rejected. Although a request is always issued by a thread, a thread does not of itself constitute an authorization unit. Threads execute in a context that is determined by the set of protection domains given by the chain of callers. Optionally, when using JAAS, a Subject that can contain several principals is associated with the thread.
The access control model of Java 2 has many similarities with those of CORBA, another well-established distributed object computing technology [6]. Security-aware objects can exercise their own security policy by calling a user-configurable access controller. CORBA rights as well as Java permissions are assumed to be (globally) defined and their semantics are precisely described, but implemented within the objects. Both systems lack the support of (standardized) policy management tools. Whereas Java 2 at least provides a default policy implementation, the normative part of CORBA does not mandate the way policies are managed.
But there are also a number of differences. Although both systems use domains as a means to structure, they use different ordering principles. In the CORBA security model, objects that have common security requirements are grouped in security policy domains. In Java, protection domains are collections of principals with the same security requirements. CORBA provides transparent access control to security-unaware objects; all method invocations are mediated by invocation interceptors to enforce access control. There is no concept of delegation in Java; only the immediate source of the object is considered.

3 Formal Model

A Java security policy is essentially an access control matrix that describes code according to its characteristics (where code came from, who signed it, and who runs it) and the permission it is granted. The content of a matrix is deduced from the policy file(s). The authorization state of a Java system is derived from the above policy together with the permission and principal hierarchies.
We assume that the permission hierarchy as well as the principal hierarchy is partially ordered. Note that this assumption must not be trivially true as the semantics of the `implies` method of the classes that extend the `Permission` class or implement the `PrincipalComparator` interface is implementation-dependent. For example, a permission class could define temporal or other non-static constraints. In our specification, we abstract from the implementation of `implies` methods of the presumed underlying Java system but expect that evaluation is purely applicative.
In the following, we make use of a number of mathematical terms and notations. In general, S shall denote a set and s ranges over elements of S. The term $\mathcal{P}(S)$ denotes the power set of S : $\mathcal{P}(S) \equiv \{X \mid X \subseteq S\}$. Variable $\hat{s}$ ranges over subsets of S and $\vec{s}$ denotes sequences of elements of S. A sequence over a set S is a function from $\mathbb{N}$ to S whose domain is an interval $1 \ldots n$ for some natural number n. The operator :: is used for adding one element to a sequence:

$$x :: \langle x_1, x_2, x_3, \ldots, x_n \rangle \equiv \langle x, x_1, x_2, x_3, \ldots, x_n \rangle.$$

We regard sequences to be like sets but imposed with an order on its elements. In particular, set operations are defined over sequences.

3.1 Protection Domains

A Java access control policy associates every code with a set of permissions. It thus determines sets of classes, called permission domains, whose instances are granted the same set of permissions. Code is distinguished whether it comes from a particular origin, signed with a specific set of public keys, and executed by a specific set of principals. The relation between the origin and the set of public keys is called code source. Principals are names associated with subjects, the users of a computing service.
A protection domain is defined by a location (the code base), a set of public keys (signer names), and a set of principals:

$$D = L \times \mathcal{P}(K) \times \mathcal{P}(S).$$

A location is expressed by a URL; if it is omitted or `null`, it stands for "any location" and shall be denoted by the empty string ε in the following. A signer is an alias for a public key and a certificate that was used to sign the code. If there are no

signers then it stands for "any signer". Principals are class names that are associated with a subject. If there are no principals then it stands for "any principal". Depending on the last characters of the URL, a location denotes either class files and/or JAR files in a single directory or in all subdirectories:

- `"/"` matches all class files (not JAR files) in the specified directory;
- `"/*"` matches all files (both class and JAR files) in the specified directory;
- `"/-"` matches all files (both class and JAR files) in the specified directory and recursively all files in subdirectories contained in that directory.

Let $\preceq$ be a URL prefix relation that obeys the above matching rules. For example, the URL `http://www.puzzles.com/-` is a prefix of URL `http://www.puzzles.com/BurrPuzzles/*`. In fact, JDK 1.2 implements an even more rigorous comparison algorithm [4, p. 43f] that takes the components of a location into account (protocol, host, ports, and anchor). To be a prefix of another URL, that URL must have the same protocol and anchor, and, if specified, it must have the same ports. For example, URL `http://www.puzzles.com:9999/-` is not a prefix of URL `http://www.puzzles.com/BurrPuzzles/*`. In general, URL matching is purely syntactic and, for example, does not deal with proxies or redirects.
Principals are names associated with subjects. The set S of principals is sorted by the classes that implement principals. Principal classes that implement (the `implies` method of) the `PrincipalComparator` interface induce a partial order on S. For example, a group principal may imply a particular subject if that subject belongs to the group. We use $s \Rightarrow s'$ to denote that principal s implies principal s'. This order can be generalized into a partial order on sets of principals:

$$\hat{s} \Rightarrow \hat{s}' \text{ iff } \forall s' \in \hat{s}' : \exists s \in \hat{s} : s \Rightarrow s'.$$

A set of principals $\hat{s}$ implies all principals of another set $\hat{s}'$ if for any principal of set $\hat{s}'$ there is a principal of set $\hat{s}$ that implies that principal. Note that this relation obviously holds if $\hat{s}$ contains all principals of set $\hat{s}'$ ($\hat{s}' \subseteq \hat{s}$).
Finally, we combine the partial orders on components of domains into a partial order $\sqsubseteq$ on domains. We say that domain $\langle l_1, \widehat{k_1}, \widehat{s_1} \rangle$ contains domain $\langle l_2, \widehat{k_2}, \widehat{s_2} \rangle$ if l_1 is a prefix of l_2, $\widehat{k_2}$ contains all keys of $\widehat{k_1}$, and $\widehat{s_2}$ implies all principals of $\widehat{s_1}$:

$$\langle l_1, \widehat{k_1}, \widehat{s_1} \rangle \sqsubseteq \langle l_2, \widehat{k_2}, \widehat{s_2} \rangle \text{ iff } l_1 \preceq l_2 \wedge \widehat{k_1} \subseteq \widehat{k_2} \wedge \widehat{s_2} \Rightarrow \widehat{s_1}.$$

For example, $\langle l, \hat{k}, \{s_1\} \rangle \sqsubseteq \langle l, \hat{k}, \{s_1, s_2\} \rangle$ because the principal set $\{s_1\}$ of the left domain is a subset of the principal set $\{s_1, s_2\}$ of the right domain. Furthermore, $\langle l, \hat{k}, \{s_1\} \rangle \sqsubseteq \langle l, \hat{k}, \{s_2\} \rangle$ if $s_2 \Rightarrow s_1$ holds.

3.2 Permissions

Let R be the sorted set of *resources*. In particular, elements of R model the resource types defined in the Java APIs, such as files or network connections. Let $T \subseteq \mathcal{P}(R)$ denote the set of *targets* in a Java system. A target is a set of resources of the same type. With each resource type, there is associated a (possibly empty) set of *actions*. *Permissions* usually comprise a target and an action. We fix the set of permissions, P, to contain the special permission `AllPermission`; let p range over permissions.
Permissions are ordered, and this order is used to infer implicit access rights. However, this order relation is left entirely up to each subclass of the `Permission` class. As Java 2 defines a `Permission` class hierarchy, there is in fact a family of relations, one for each supported resource type. See, for example, the description for file permissions [4, p. 52ff] and for socket permissions [4, p. 56ff]. Both permission classes even define a suborder on targets (path names and host names, respectively) and on the corresponding actions. In general, one might say that a permission p implies another permission p' ($p \Rightarrow p'$) if both the target of p contains the target of p' and the action of p implies the action of p'. For example, the `java.io.FilePermission` class is implemented such that for the same resource a write permission implies a read permission. Above definition also implicitly states that it should not be possible that a permission of one class implies a permission of another class.
The semantics of permissions is thus modulo to the respective resource types. Besides the properties of the partial order on the permission hierarchy, there is the following axiom:

$$\forall p \in P : \texttt{AllPermission} \Rightarrow p$$

By definition, `AllPermission` permission implies all permissions [4, p. 65].

The family of implies relations on permissions must be further extended to permission collections and to sets of permission collections (`Permissions`). Only the latter relation is explicitly defined:

$$\hat{p} \Rightarrow p \text{ iff } \exists p' \in \hat{p} : \; p' \Rightarrow p.$$

A set of permissions $\hat{p}$ implies a permission p if there is a permission p' in $\hat{p}$ that implies permission p.

3.3 Policy

The Java runtime maintains a mapping from code (classes and objects) to their protection domains and then to their permissions. Authorizations are granted to code sources (origins and signers) and principals.
The system security policy file, possibly (additively) combined with a user security policy file and JAAS policy file, sets the security policy of a Java system. A policy file consists of a number of grant entries that are made up of a code source and/or principals and the associated permissions. The set of grant entries in a policy maps each declared domain to its permissions. All code that is considered part of the system core belongs to the system domain. The system domain is granted all permissions.
A Java policy $\mathbb{P} = D \rightarrow \mathcal{P}(P)$ maps domains to sets of permissions. As a policy file may contain grant entries with nested domains, we require that $\mathbb{P}$ satisfy the transitive closure on relation $\sqsubseteq$ on domains, i.e. each nested domain d contains the union of all permissions in $\mathbb{P}(d')$ for every domain d' that contains d:

$$(p \in \mathbb{P}(d) \wedge d \sqsubseteq d') \supset p \in \mathbb{P}(d').$$

Note that all the definitions are additive, so permissions can only be granted, not revoked.
Thus, each protection domain entry in $\mathbb{P}$ holds all of its permissions defined by the current policy. Furthermore, each protection domain encloses a set of classes whose instances are granted the same set of permissions. The above definition ensures that classes from different sources belong to different domains even if they have the same permissions.
There are two predefined protection domains. All system code is assumed to run in a domain that possesses all permissions (`AllPermission` $\in \mathbb{P}(system)$). If there is no protection domain that matches a newly loaded class, then this class will be mapped to a protection domain that defines a default policy, the original Java sandbox.

3.4 Threads

Threads are the active elements in Java that may execute code from different sources and may be associated with different principals (protection domains), and are thus regarded as the correct context for access control [3]. A thread is a chain of multiple method invocations, and its effective permissions are defined to be the intersection of the permissions of all methods involved in the call sequence. This approach implements the least-privilege principle in that a domain cannot gain additional permissions as a result of calling more "powerful" domains, whereas a more powerful domain must lose its power when calling a less powerful domain [3].
The effective permissions of a thread depend on the protection domains it crossed. Thus, if the call chain of a thread contains code associated with protection domains d_1, d_2, and d_3, the effective permissions of the thread (the most recent method) are given by $\mathbb{P}(d_1) \cap \mathbb{P}(d_2) \cap \mathbb{P}(d_3)$. Note that the order in which domains are crossed is not relevant for the access control decision.
The current (execution) context is entirely represented by its current sequence of method invocations, where each method is defined in a class that belongs to a protection domain [4, p. 92]. Or equivalently, a context is set up by the sequence of protection domains. Thus the context of a thread may change whenever it calls a new method or returns from the current method. Additionally, there are special methods that explicitly affect the current context. For example, method `javax.security.auth.Subject.doAs` assigns (additional) principals to current context, whereas method `java.security.AccessController.doPrivileged` creates a new context that consists only of the protection domain of the method to be executed next. Whenever execution returns from those special methods, the old context will be restored.

3.5 Security Context

Note that the order of domains is not relevant for the access control decision as long as there is no static method `doPrivileged` in the call chain. By calling `doPrivileged`, a piece of code tells the Java runtime system to ignore the

permissions of its callers and that it itself is taking responsibility for exercising its permissions. However, the protection domains of the code that is subsequently called by the "privileged" code are still considered in the access control decision.
The subject affiliated with the current access control context may have a number of associated principals that were successfully authenticated. The static method `Subject.doAs(Subject s, PrivilegedAction a)` associates a subject with the current access control context and then executes the action.
The security context of a thread consists of a set (sequence) of protection domains (call sequence), the inherited access control context, and a set of associated principals:

$$C = \mathcal{P}(D) \times \mathcal{P}(D) \times \mathcal{P}(S).$$

Let $\vec{d}$ range over sequences $\mathcal{P}(D)$ of protection domains and let $\hat{d}$ range over sets $\mathcal{P}(D)$ of protection domains. Let $\mathbb{C}$ range over sequences of security contexts, where only the security context added last will be used for permission checking.
Let $D(f,\hat{s})$ determine the protection domain of method f under the set $\hat{s}$ of current active principals. Function D is total, i.e. if there is no grant entry in the policy file, function D returns protection domain *sandbox*.
When a new thread is created, the initial call sequence is given by the protection domain $D(f)$ of the first method f to be executed. Access control context and associated principals are inherited from the parent thread.
Let $\vec{d} = \langle d_n, \ldots, d_2, d_1 \rangle$ be the 'current' state of the call sequence in a security context of a thread, where d_n is the protection domain of the most recent called method. When a stack frame makes a method call f, this creates a new stack frame and updates the current security context:

$$\langle \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{call(f)} \langle D(f,\hat{s}) :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \tag{1}$$

The updated security context of the thread is computed by adding the protection domain $D(f,\hat{s})$ of the called method f using the currently assigned principals $\hat{s}$ to the call sequence.

Enabling Privileges. When a stack frame calls the special method `doPrivileged(new PrivilegedAction() ...)`, this creates a new stack frame f for the `run` method of the inner class object.

$$\langle d :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{doPrivileged(f)} \langle \langle D(f,\hat{s}) \rangle, \{d\}, \hat{s} \rangle :: \langle d :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \tag{2}$$

Method f is performed with all of the permissions possessed by the caller, as determined by its protection domain d. Note that the fact that the protection domain d of the calling method becomes the new context of the thread prevents code from acquiring more rights than they own themselves.
When a stack frame calls the special method `doAs`, this creates a new access control context based on the current access control context c that is assigned the Subject-based permissions

$$\langle \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{doAs(\hat{s}', f)} \langle \langle D(f,\hat{s}') \rangle, \vec{d} \cup \hat{d}, \hat{s}' \rangle :: \langle \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C}. \tag{3}$$

If the current context contains at least one protection domain (code source) that does not get more permissions via the assigned principals, the set of effective permissions does not increase. Subsequent method calls might only decrease this set (intersection of the permissions of all protection domains; least privilege principle).
When a stack frame calls the special method `doAsPrivileged`, this also creates a new access control context but only based on the protection domain of the called method f under the assigned principals $\hat{s}'$.

$$\langle d :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{doAsPrivileged(\hat{s}', f)} \langle \langle D(f,\hat{s}') \rangle, \{d\}, \hat{s}' \rangle :: \langle d :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \tag{4}$$

Returning from a method. When a stack frame f returns control to its calling frame, $D(f,\hat{s}) = d$, we distinguish between two cases:

$$\langle d :: d' :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{return} \langle d' :: \vec{d}, \hat{d}, \hat{s} \rangle :: \mathbb{C}. \tag{5}$$

If the call sequence consists of more than one protection domain, then we reconstruct the former context by removing the protection domain of the returning stack frame.

$$\langle \langle d \rangle, \hat{d}, \hat{s} \rangle :: \mathbb{C} \xrightarrow{return} \mathbb{C}. \tag{6}$$

If the call sequence is a singleton list, then we simply pop the current context from the sequence of security contexts of the thread.

Definition 1. A Java 2 authorization system $\mathbb{A}$ is an abstract machine $\langle Q, q_0, \Delta, \rightarrow \rangle$, where

- Q is the set of *states*;
- $q_0 \in Q$ is the *initial* state;
- $\Delta = \{call, return, doPrivileged, doAs, doAsPrivileged\}$ is the set of *labels*;
- $\rightarrow \subseteq Q \times \Delta \times Q$ is the *transition relation*.

The operational semantics for the Java authorization system is given by the transition relation $\rightarrow$ defined by the above inference rules 1–6 over the state space $Q = \mathbb{C}$ and initial state $q_0 = \langle\langle system \rangle, \{system\}, \emptyset \rangle$. []

In the following, the expression $q \xrightarrow{\delta} q'$ is a shorthand for $\langle q, \delta, q' \rangle \in \rightarrow$. We also define the derived transition relation $\stackrel{\sigma}{\Rightarrow}$ ($\sigma \in \Delta^*$) of sequences of actions: $q \stackrel{\langle\rangle}{\Rightarrow} q$ and $q \xrightarrow{\delta :: \sigma} q''$ iff $q \xrightarrow{\delta} q'$ and $q' \stackrel{\sigma}{\Rightarrow} q''$.

3.6 Access Control Decision

In Java, a requester is represented by a security context that is composed of protection domains, and, possibly, activated principals. Therefore, requesters do not correspond with authorization subjects given in the policy. An access control rule determines whether a requester must be allowed or denied access. This rule is implemented by the `checkPermission` method of class `AccessController` whose algorithm can be given as:

$$checkPermission(\langle \vec{d}, \hat{d}, \hat{s} \rangle, p) = true \equiv (\forall d \in \vec{d} : \mathbb{P}(d) \Rightarrow p) \wedge (\forall d \in \hat{d} : \mathbb{P}(d) \Rightarrow p)$$

Access is granted only if the required permission p can be derived from the execution sequence $\vec{d}$ as well as from the security context $\hat{d}$. Note that the set $\hat{s}$ of principals in the given security context has already been taken into account whenever a new frame was pushed on the stack. This access control checking can be performed in linear time w.r.t. the number of domains in the execution sequence and in the security context.

Any Java authorization system is complete because all `Permission` and `Principal` classes used have to implement the `implies` methods, provided they do not implement an infinite loop. Furthermore, if there is no matching protection domain in the policy specification, the default protection domain that implements the Java sandbox is taken.

A few concluding remarks on the complexity of the described algorithm are in order. JDK 1.2 uses a lazy evaluation strategy to implement the access control decision algorithm. Whenever permission checking is requested, the algorithm searches the frames on the caller's stack in sequence, from newest to oldest. The search terminates, forbidding access (and throwing an exception), upon finding a stack frame that is forbidden by the policy engine from accessing the target. Otherwise, the search terminates, granting access, when all stack frames are allowed to access the target, either reaching the end of the stack or a frame whose code is marked privileged.

Stack inspection has high run-time costs. At worst, the cost is proportional to the current stack depth. The JDK 1.2 algorithm iterates over the stack frames to determine the associated protection domains (depending on URL prefix, signers, and active principals). For each protection domain, the algorithm again iterates over the contained permissions to find one that implies the required permission. Thus, the performance of the access control decision algorithm will depend on the number of protection domains, the number of permissions per protection domain, and the stack depth.

Looking at the data structures of our model, we can easily identify simplifications. For example, the first component of the security context of a thread is a sequence of protection domains. The result of the access control decision, however, is independent of the order and frequency of the protection domains. In the model, the call sequence is only used to keep track of the evolution of the Java stack frame. An advanced data structure such as a multi-set might therefore lead to a more efficient implementation. The model also allows us to identify where symbolic representations can be used as an efficient implementation technique.

Based on our definition of the Java authorization system $\mathbb{A}$ we can now define what it means to implement $\mathbb{A}$ correctly.

Definition 2. An authorization system $\mathbb{A}' = \langle Q', q_0', \Delta, \rightarrow \rangle$ correctly implements Java authorization $\mathbb{A}$ if $checkPermission(q_0, p) = checkPermission'(q_0', p)$ and for any action sequence σ if $q_0 \stackrel{\sigma}{\Rightarrow} q$ and $q_0' \stackrel{\sigma}{\Rightarrow} q'$ then $checkPermission'(q, p) = checkPermission(q', p)$. □

Note that our definition of correctness is independent of the structure of the states and of the algorithm of the corresponding *checkPermission*′ function of the authorization system $\mathbb{A}'$.

4 Expressiveness of Java 2

Authorization is an independent semantic concept that should be separated from its implementation in system-specific mechanisms. In this section, we analyze whether the Java 2 policy language is expressive enough to specify commonly encountered authorization requirements.

Our formal model shows that the Java access control decision depends on a number of relations. Some of them have a fixed meaning: URL prefix ($\preceq$) and key inclusion ($\subseteq$) as well as the partial orders on resources and actions as implemented by the `Permission` classes that belong to the core Java platform API, for example `java.io.FilePermission`. The others are given by the system-specific implementation of `implies` methods of derived `Permission` and `Permissions` classes as well as of the `Principal` classes implementing the `PrincipalComparator` interface.

To analyze the expressiveness of the Java 2 access control model, this section presents Java policies that express three distinct security policies: identity-based access control, multi-level security, and role-based access control (RBAC). Identity-based access control uses the identity of principals and resources to define an explicit relationship representing access rights. Multi-level security classifies principals and resources and uses a set of rules to infer the authorization state from these classifications.

The specification of access rights in Java is identity-based as well as classification-based. Principals with similar security properties are grouped into protection domains, and permissions are granted to protection domains, thus establishing an indirect relationship between principals and rights. Java access control is not discretionary as it enforces a system-wide access control policy, where the authorization state cannot be changed at the discretion of users.

4.1 Identity-based Access Control

In Java 2, permissions are granted to protection domains, and principals belong to protection domains. This indirection, where permissions are not granted to principals directly, is by design [3]. On the one hand, it facilitates management of authorization; on the other hand, it does not allow permissions to be enabled/disabled one by one as in the Netscape security model [10].

Each grant entry in a policy file specifies a code base, code signers, and principals triplet. For example, the grant entry

```
grant CodeBase "http://guapo.com",
      SignedBy "tony",
      Principal NTPrincipal "kent" {
  Permission java.io.FilePermission
                "/user/kent", "read,write";
};
```

defines that code from *guapo.com*, signed by *tony*, and running as principal *kent* has the permission to read and write files in the directory */user/kent*. In our model, assuming that values (like *kent*) implicitly carry their type information (`NTPrincipal`), the policy would have the following mapping:

$$\mathbb{P}(\langle \texttt{http://guapo.com}, \{tony\}, \{kent\} \rangle) = \{\langle /\text{user/kent}, read \rangle, \langle /\text{user/kent}, write \rangle\}.$$

In [8], Lai *et al* present a usage scenario where a service authenticates a remote subject, and then performs work on behalf of that subject. Using JAAS, the server runs in an access control context bound by the subject's permissions. Having available the work to be performed as a `java.security.PrivilegedAction`, it uses the `Subject.doAs` method to associate the Subject with the current access control context. Thus the server is able to convey some "stack information", i.e. access control context, between platforms.

4.2 Multi-level Security

To implement a label-based access control policy, we introduce permissions of resource type *Label*. Let us assume three security levels; let r_1 stand for permission $\langle \textit{level1}, \textit{read} \rangle$, and permissions r_2, r_3, w_1, w_2, and w_3 are built in the same way. Permission type *Label* then consists of the individual permissions $\{r_1, r_2, r_3, w_1, w_2, w_3\}$. For example, permission r_3 expresses a higher security level than r_2.
Security labels are attached to subjects and targets. A label on subjects is called a security clearance. A label on targets is called a security classification. We model security clearances by granting each subject label rights according to her clearance. For example, if subject *alice* has clearance 2 she gets permissions r_1, r_2, w_2, and w_3, denoting that she can read all targets of classification 2 or lower, and that she can write on all targets with the same classification or higher.

```
grant Principal LabelPrincipal "alice" {
   Permission Label "level1", "read";
   Permission Label "level2", "read,write";
   Permission Label "level3", "write";
};
```

The above grant entry defines that code running as principal *alice* has clearance level 1 and thus the permission to read and write targets with security level 1, and to write to targets with higher classification. In our model, the policy would have the following mapping:

$$\mathbb{P}(\langle \varepsilon, \emptyset, \textit{alice} \rangle) = \left\{ \begin{array}{l} \langle \text{level1}, \textit{read} \rangle \\ \langle \text{level2}, \textit{read} \rangle \\ \langle \text{level2}, \textit{write} \rangle \\ \langle \text{level3}, \textit{write} \rangle \end{array} \right\}.$$

As access control granularity in Java is on the method level, we have to determine the classification level of each method. For all multi-level security schemes it is assumed that each method has either read or write characteristics. Checks for these levels are inserted in the methods. Assume the above subject *alice* wants to invoke method m_1, which has classification level 1 and thus requires permission r_1. Access is allowed as the permission r_1 is granted to subject *alice*. Assume further that method m_2 requires permission w_1 and method m_3 requires permission w_3. Then the same subject *alice* is allowed to invoke method m_3 (write-up) but is not allowed to invoke method m_2 (no write-down, *-property).
Administration can be simplified if we introduce a permission hierarchy as follows: $r_3 \Rightarrow r_2$ and $r_2 \Rightarrow r_1$, and, by transitivity, $r_3 \Rightarrow r_1$. Similarly, it shall hold that $w_1 \Rightarrow w_2$, $w_2 \Rightarrow w_3$, and $w_1 \Rightarrow w_3$. This means that an administrator would give a subject with clearance level 2 the permissions r_2 and w_2 instead of the permissions r_1, r_2, w_2, and w_3. As classification and clearance relations are static, there is no loss of flexibility by implementing the permission hierarchy in the corresponding permission class.

4.3 Role-based Access Control

RBAC models associate permissions with roles and assigns users with appropriate roles. A major difference between groups and roles is that roles can be "activated" and "deactivated" by users at their discretion, whereas group membership always applies [5]. JAAS treats both groups and roles simply as named principals [8]. Thus there no distinction is made between the two concepts. A role as well as a group membership can be enabled with the `Subject.doAs` method that dynamically associates an authenticated subject with the current access control context.
The grant entries in the Java policy file implement the RBAC permission to role assignment relation. The user to role assignment relation, however, is managed by user administrators outside of Java. The `subject.doAs` method dynamically associates principals with the current AccessControlContext and thus defines a session during which a subset of roles is simultaneously activated, of which the user is a member. The permissions available to the user are the union of permissions of all roles activated in that session, i.e. the role principals given to the `Subject.doAs` method.
The next example shows how to formulate an access control policy where a role is combined with a group membership, e.g. "Only somebody who is a manager and a member of project-X is allowed to change project X's time schedule".

```
grant Principal Role "manager",
      Principal Group "project-X" {
  Permission SchedulePermission "change";
};
```

The above grant entry lists two principals and thus requires that a subject provided to `Subject.doAs` must have both principals associated with it to be granted the specified permissions.

$$\mathbb{P}(\langle \varepsilon, \emptyset, \{\mathit{manager}, \mathit{project\text{-}X}\}\rangle) = \{\langle \text{schedule}, \mathit{change}\rangle\}$$

Role and group hierarchies can be built with principals that implement the `PrincipalComparator` interface. For example, *view* permission below is granted to any subject that class RoleComparator, implementing the `PrincipalComparator` interface, *implies*:

```
grant Principal RoleComparator "member" {
  Permission SchedulePermission "view";
};
```

If we assume that a manager role is senior to a member role, $\mathit{manager} \Rightarrow \mathit{member}$, then managers inherit all the permissions granted to members. To put up this hierarchy, RoleComparator has to implement the PrincipalComparator interface, and its `implies` method must return true if the provided subject has an associated "manager" role principal.
Compared with the approach taken by Giuri [2], where a role is implemented as a permission, principal roles solve the problem of dynamically enabling roles but move the definition of the role hierarchy into the class that implements the `PrincipalComparator` interface, and thus is not user-definable.
In addition to role hierarchies, constraints are another fundamental aspect of RBAC. For example, principal classes that form a group/role hierarchy should implement an acyclic membership relation, i.e., if a principal p_i is a member (directly or indirectly) of another principal p_j, with $p_i \neq p_j$, then p_j cannot be a member of p_i. Above property must be properly implemented in the `Principal` classes. RBAC models with constraints on the assignment of users to roles and of roles to permissions support security principles such as separation of duty. Static separation of duty requires that certain roles cannot be granted together to the same subject and can be implemented by the principal authenticator (the login module of JAAS). Dynamic mutual exclusion of roles can only be checked by the `doAs` method. In both cases, the entities that authenticate role principals as well as activate role principals must be aware of (the constraints on) the role hierarchy.
Above discussion shows that the elementary features of RBAC can be implemented in Java 2. However, it requires extensions to vital components of the Java 2 security architecture to implement dynamic mutual exclusion of roles.

5 Conclusion

At the core of Java authorization, protection domains characterize sets of principals to which the same set of permissions are granted. These policies are user-definable but (implicitly) refer to a number of relations, such as resource, permission, or principal hierarchies. Implemented in miscellaneous classes, be they within Java or application libraries, these relations are not user-definable. Finally, user administrators manage the assignment of principals to users. However, such management functionality is beyond the scope of Java.
In this paper we have presented a formal model of the above Java 2 access controls with JAAS. The model provides an abstract machine in terms of a small number of transitions. It was designed as far as possible in an implementation-independent way, to work out the underlying concepts. However, this is quite hard to achieve as the Java access control decision function is defined as a low-level checking algorithm.
The provided formal description of the Java access control model serves the basis for the understanding of the fundamental concepts. By mechanizing our model within a verification environment we could increase assurance that our definition of Java 2 access control is indeed the model implemented by Sun's distribution. It also opens the possibility to compare Java access control model with other access control models and to prove properties about the model. Serving as a reference specification, our model can be used to verify the correctness of implementations based on different data structures and algorithms, Erlingsson and Schneider's in-lined reference monitor for example [1].
We showed that Java 2 Authorization and JAAS can implement different access control policies, including multilevel operations. However, policies that dynamically adopt to changing conditions cannot be directly implemented in Java 2. As the authorization state is derived from the policy file together with the permission and principal hierarchies, all static in nature, there are no means to express that a subject can gain or lose some permissions, except for temporarily adding principals to the current security context. A possible approach to implement history-based policies, like the Chinese Wall policy, would be to record changes in the user registry and thus to implement parts of the access control policy within the authentication service.

The development of Java authorization will certainly continue over time, for example by introducing negative permissions. Our model could serve as a sound base to study the implications on the logic as well as on complexity.
The default policy language supported by the standard JDK 1.2.2 platform does not enable users to define their own principal or permission relations. As principal as well as permission classes are usually quite simple, a policy processing tool may automatically generate these classes from higher-level descriptions. For example, the Authorization Specification Language (ASL) [5] is a logical language in which above relationships together with integrity constraints such as incompatible group memberships can be expressed. The presented formalization of the Java 2 access control model provides a basis to identify the subset of ASL that could be implemented in Java 2.

Acknowledgments

This work was partially supported by the Defence Evaluation & Research Agency (DERA) under programme Beacon. It represents the views of the author. The author thanks Larry Koved and Charles Lai for their feedback on JAAS. Anthony Nadalin and the anonymous referees gave valuable comments and suggestions on this work.

References

[1] Ú. Erlingsson and F.B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, May 2000.

[2] L. Giuri. Role-based access control in Java. In *Third ACM Workshop on Role-Based Access Control (RBAC'98)*, ACM Press, pages 91–100, 1998.

[3] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Security*, IEEE Computer Society Press, pages 125–134, 1998.

[4] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. The Java Series. Addison-Wesley, 1999.

[5] S. Jajodia, P. Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, pages 31–42, 1997.

[6] G. Karjoth. Authorization in CORBA security. In Quisquater *et al*. [9], pages 143–158.

[7] L.L. Kassab and S.J. Greenwald. Towards formalizing the Java architecture for JDK 1.2. In Quisquater *et al*. [9], pages 191–207.

[8] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, IEEE Computer Society Press, pages 285–290, 1999.

[9] J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors. *Fifth European Symposium on Research in Computer Security (ESORICS)*, Springer, Lecture Notes in Computer Science 1485, 1998.

[10] D.S. Wallach and E.W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, pages 52–63, 1998.