A Secure Execution Framework for Java[*]

Manfred Hauswirth
Distributed Systems Group
Technical University of Vienna, Austria
mh@infosys.tuwien.ac.at

Clemens Kerer
Distributed Systems Group
Technical University of Vienna, Austria
ck@infosys.tuwien.ac.at

Roman Kurmanowytsch
Distributed Systems Group
Technical University of Vienna, Austria
q@infosys.tuwien.ac.at

ABSTRACT

The Java platform facilitates to dynamically load and execute code from remote sources which can threaten the security and integrity of a system and the privacy of its users. To address these problems, Java includes a security architecture which is based on a closed policy model. Although this model is sufficient to specify arbitrary policies, it easily may become cumbersome to use and is not well-suited for administering a consistent security policy for a complete network. The Java Secure Execution Framework (JSEF) overcomes these drawbacks: it introduces higher-level abstractions which enhance the expressiveness of policy rules; it simplifies the maintenance of security configurations; and it provides additional functionality and tools to make administration less error-prone. In JSEF we propose a hybrid policy model which supports additive and subtractive permissions with a denial-take-precedence rule to resolve conflicts. Security profiles can be expressed in terms of hierarchical groups where a subgroup inherits the policy defined by its parent. All members of a group share the same set of permissions and users can be members of an arbitrary number of groups. JSEF's administrative model supports the definition of a network-wide policy which users can tailor to their needs but not break. At runtime JSEF enforces the defined security policy and supports security negotiation in case of insufficient permissions. A set of graphical tools supports the user in defining security policies and configuring JSEF.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection—*Java, management, configuration*; D.4.6 [**Security and Protection**]: Access Controls—*Java, mobile code security and access*; K.6 [**Management of Computing and Information Systems**]: Security and Protection—*Java, management, configuration*; K.6.5 [**Security and Protection**]: Unauthorized Access—*Java, mobile code security and access*

General Terms

Security, Management

Keywords

Java security management, XML-based security configuration, management GUIs

1. INTRODUCTION

Mobile code denotes code that traverses a network and executes at a remote site. The process of traversing can either be active as in the case of mobile agents which move around in a network at their own volition, or it can be passive, i.e., a user downloads the code to a site and executes it there (e.g., applets).

Java can be used as a platform for both types of code mobility and in conjunction with the Internet opens new possibilities for software development, software deployment, and architectural styles. The downside is that it also opens new security threats. For example, downloaded code can include a virus or be a Trojan horse and thus pervert the concept of code mobility over the Internet in a very dangerous way ("portable viruses"). As any mobile code platform Java suffers from four basic categories of potential security threats [2, 7, 16, 17, 18]: (1) *leakage* (unauthorized attempts to obtain information belonging to or intended for someone else), (2) *tampering* (unauthorized changing—including deleting—of information), (3) *resource stealing* (unauthorized use of resources or facilities such as memory or disk space), and (4) *antagonism* (interactions not resulting in a gain for the intruder but annoying for the attacked party).

To deal with these threats Java provides a special runtime environment that tries to protect users from erroneous or malicious mobile code and tries to ensure the integrity, security, and privacy of the user's system. It provides good protection against leakage and tampering but resource stealing and antagonism cannot be fully prevented since it is hard to distinguish automatically between legitimate and malicious actions.

Java's security architecture offers many low-level security mechanisms, e.g., access permissions on resources, and supports the definition of arbitrary security constraints, but provides no higher-level security management concepts such as hierarchical policies or user groups. It does not support

[*]This work was supported in part by the European Commission under contract IST-1999-10288 (OPELIX).

flexible system-wide security policies and offers no concepts for defining security profiles. No notions of users and groups exist and it provides only very limited means for hierarchically organized security configurations. The lack of such higher-level concepts complicates maintenance of a consistent security policy for a complete network and tailoring of security requirements to the needs of a specific user and thus may cause misconfigurations or the introduction of security holes.

This paper presents the Java Secure Execution Framework (JSEF) which solves these shortcomings. JSEF (pronounced Joseph) provides a hierarchical security policy scheme which supports both local, user-specific security policies, and a global security policy defined by the administrator. It supports the definition of user groups with assigned security policies which can be freely structured into a hierarchy. A user can be member of a set of groups with different security profiles which aids administrators in the definition, assignment, and maintenance of security policies for a user or a group of users. JSEF offers several additional features beyond Java's standard capabilities: Policy and group definitions are represented as XML documents; policies, configurations, and mobile code can be retrieved from arbitrary locations; and security conflicts can be negotiated interactively at runtime. JSEF is based on the Java 2 security architecture and is fully compatible with it. It can be used with any Java code and in any environment which is compatible with Java 2, for example, in Java-based mobile agent systems or for extended applet security features in Web browsers. JSEF originally was developed as part of the Minstrel push system project [8] to provide a flexible security environment for executable channel content (so-called pushlets) and agents [9].

This paper is organized as follows: Section 2 provides an overview of Java's security model. This model is discussed in Section 3 and we point out its shortcomings which provided the motivation and requirements for JSEF. Section 4 then presents JSEF's security model and concepts. Some key parts of JSEF's implementation are highlighted in Section 5 and we overview the main tools developed for JSEF which offer easy-to-use access, configuration, and management functionalities for users and administrators. Section 6 presents work related to JSEF and we summarize and give our conclusions in Section 7. JSEF and all associated tools are available at http://www.infosys.tuwien.ac.at/jsef/.

2. JAVA'S SECURITY ARCHITECTURE

According to [20] four practical techniques for securing mobile code exist: the sandbox model, code signing, firewalling, and proof-carrying code. Java uses a hybrid approach which combines *sandboxes* and *code signatures*. The Java core classes act as a security shield and enforce the sandbox model by granting or forbidding access to resources based on a security policy. The rules specified in the security policy define the actions a piece of code is allowed to perform depending on the origin of the code and an optional signature. Not all of Java's powerful security mechanisms are in place per default when launching the Java Virtual Machine (JVM). While some basic checks are performed automatically, the more sophisticated concepts including the sandbox model have to be put into action manually [4] (in the following sections we assume that this has been done).

When a class is loaded the following steps occur: First, the Verifier [22] performs a set of security checks to guarantee properties such as the correct class file format, correct parameter types and binary compatibility before a class is loaded. These checks enhance runtime performance because otherwise they would have to be performed during runtime. Also, they assure the integrity of the Java runtime environment since no malformed class can be loaded. Having passed the Verifier the *class loader* loads the bytecode representation of the class and checks optional signatures. Furthermore, the class's code source is constructed which consists of the location from which the class was obtained and a set of certificates representing the signature.

The class's code source is the key input for the security policy construction for a given class. In Java 2 the security policy is defined in terms of *protection domains* which define what a piece of code with a given code source is allowed to do. Hence, a protection domain contains a code source with a set of associated permissions. Given the code source of a class, the security policy (i.e., a collection of protection domains) is searched to compose the permissions of the class.

Finally, the class is being defined. Defining a class makes it publicly available and adds it to the class loader's cache of classes which is important to ensure class uniqueness. Java considers two classes equal if, and only if, they have the same name and were loaded by the same class loader.

After these initial steps the class can be used in the Java runtime environment. However, every time the class tries to access a system resource its permissions have to be checked by contacting the *security manager*. If the call to the security manager returns silently, the requesting caller has sufficient permissions to access the resource and the execution continues. If not, a security exception is raised and has to be handled by the caller or otherwise the JVM terminates.

The remaining question is, how the security manager decides whether access to a resource is granted. Since Java 2 the security manager is mainly included for compatibility reasons and delegates nearly all of its tasks to the *access controller*. The access controller uses a *stack inspection algorithm* and the security policy to decide how to proceed. The stack inspection algorithm is based on the call stack of the current method. Since every class was assigned an appropriate set of permissions when it was loaded, the stack inspection algorithm can use this information to make its decision. An in-depth discussion of Java's stack inspection algorithm is beyond the scope of this paper. A detailed description of Java's security architecture and the stack inspection algorithm can be found in [14].

3. A CRITICAL VIEW ON JAVA'S SECURITY MODEL

Java's current security model only supports explicit specification of accesses that are permitted. This is sufficient to specify arbitrary security policies but may be impractical, however, if a user needs an advanced security policy. Instead of specifying what is permitted, it is frequently necessary to specify what is *not* permitted. For example, a directory hierarchy may hold configuration files which may only be read and data files which may also be modified. If the number of files and directories is high, it may be cumbersome to explicitly list all files and directories which may be accessed with the according permissions and maintenance of this declarations can become difficult. It may be considerably easier to assign read-write permissions to the whole directory tree and only forbid write access for certain files. Of course this depends on the concrete requirements but having such a feature at hand leaves the decision to the user which way of configuration fits best his/her needs. JSEF supports both ways of specification by its so-called additive and subtractive permissions.

The current security model of Java uses a two-level configuration approach. A global policy file holds the default permissions for any user on a specific site and a user's local policy file can specify additional permissions. Since Java's security model only supports additive policies, only two extremes for meaningful security configuration exist: Either each user must maintain a private security policy file, or a global policy is specified and user-specific configurations are ignored. With the first strategy users can easily introduce security holes—regardless of whether a global policy file exists since the user's local policy can extend the global policy in any way—but can have a personalized configuration. In the second case the administrator has total control over the security policy but cannot tailor it to specific users' needs.

JSEF overcomes these problems by providing a hierarchical security policy scheme which supports both local, user-specific security policies and a global security policy defined by the administrator which takes precedence over user policies. At runtime a user's actual policy is defined by merging the user's local policy with the global policy. The user's policy, however, cannot circumvent restrictions imposed by the administrator in the global policy. This scheme attempts to improve the management of security policies and will be explained in detail in Section 4. For example, the system administrator may define the company-wide security policy in the global policy and JSEF ensures that every user implicitly follows it. However, users can still refine this policy, e.g., by defining a more restrictive policy for their personal data, but cannot overrule it.

Moreover, the Java security model lacks support for user groups while JSEF supports the definition of hierarchical user groups with assigned security policies. A user can be member of several groups that have different security profiles. With user groups being supported, an administrator can easily define a set of profiles in terms of groups and assign these profiles to users depending on their requirements. Additionally, these groups can be freely structured into a hierarchy to simplify maintenance and tailoring of the security policy. Thus policies for the specific user roles can be easily defined, nested, and maintained. For example, developers may be assigned a certain profile; a subgroup for testers may inherit these permissions but be refrained from modifying the source files.

In contrast to Java, JSEF supports the retrieval of policy definitions from arbitrary sources. It currently uses files (which hold the necessary definitions represented in XML), but can easily be tailored to load the policy definitions from other sources such as databases or remote locations. For example, a company may want to keep these definitions in a database on a secure computer which can only be accessed via a special security procedure. Also mobile code that is to be executed can be loaded from arbitrary sources.

In the standard Java security model the requester of an operation receives a security exception whenever an access is denied by the user's security policy. This typically terminates the execution, the user has to exit the program that wanted to perform the access, set the appropriate permissions, restart the program, and retry its execution. This can be tedious and time-consuming especially for applets and mobile agents. For example, a user downloads an applet and the applet wants to access a resource the user does not permit access to yet. Then the user would have to adjust the permissions and reload the applet (possibly again over the network) and restart it. This process has to be repeated for every denied access until all required permissions are available since the applet fails as soon as it encounters such a problem. The required permissions cannot be determined during download or verification time because the security requirements of a Java program depend on its dynamic runtime behavior. Additionally, no formal correctness proof of the Java verifier exists so far. JSEF provides a security negotiation facility: If a forbidden operation is attempted, JSEF intercepts it *before* the actual access and starts a negotiation process which can also be used as a blueprint for other (semi-) automatic negotiation schemes. This supports runtime management of the security policy while still ensuring that the existing policy settings are not violated. To circumvent these runtime security negotiations users may permit all accesses, but this problem also applies to the standard security architecture. It is even more likely to occur there since it requires more effort than with JSEF to adjust permissions correctly.

4. THE JSEF MODEL

One of the main goals of JSEF's security model is to remain compatible with the default Java security model [3, 4, 5, 21]. As a consequence, JSEF's policy concepts extend the loading, execution, and monitoring of a class as described in this section. In-depth descriptions of the JSEF policy concepts are given in [10] and [14].

For the further discussion of JSEF's main features consider the following example scenario: a user called *Charly Brown* is working on a computer in a local area network managed by a system administrator. He has just downloaded a demo version of a promising new Java program from www.infosys.tuwien.ac.at and wants to execute it. For security reasons the system administrator has installed JSEF on all machines in the network. Based on this scenario the following sections present the concepts underlying JSEF and, finally, compare the behavior of Java and JSEF.

4.1 Additive and Subtractive Policy

JSEF introduces the notions of *additive* and *subtractive permissions*. Additive permissions are the class of permissions as used by the Java security model: They grant permission to access a resource. Subtractive permissions are defined in a similar way but specify which resources must not be accessed. As with additive permissions, subtractive permissions are grouped in (subtractive) protection domains to associate a code source with a set of permissions. The collection of additive protection domains defines the user's *additive security policy* and the collection of subtractive protection domains defines his/her *subtractive security policy*.

Figure 1 shows the additive and subtractive policy definition of the user *Charly Brown*. Since Charly Brown trusts code originating from www.infosys.tuwien.ac.at if it is signed by CK, he defines an additive protection domain granting all permissions to such code. Notwithstanding his trust, Charly Brown wants to make sure that his personal data remains untouched. Thus he defines a negative policy item preventing such code from accessing his home directory.

```
<?xml version="1.0"?>
<!DOCTYPE localPolicy SYSTEM "localPolicy.dtd">
<localPolicy userName="Charly Brown" lastChanged="10/03/2000">
  <addItems>
    <policyItem signedBy="CK"
                codeBase="http://www.infosys.tuwien.ac.at/-">
      <permission class="java.security.AllPermission">
      </permission>
    </policyItem>
  </addItems>
  <subItems>
    <policyItem signedBy="CK"
                codeBase="http://www.infosys.tuwien.ac.at/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/home/-"/>
        <actions name="read write execute"/>
      </permission>
    </policyItem>
  </subItems>
</localPolicy>
```

Figure 1: Charly Brown's additive and subtractive policy definition.

This example already indicates that negative permissions overrule additive ones. A complete description of JSEF's policy semantics is given in Section 4.4 after all concepts have been presented.

4.2 Policy Exceptions

Policy Exceptions are another extension to Java's security architecture which is applied in conjunction with the wildcards '`*`' and '`-`' of Java's policy model. Wildcards make it easy to define a permission to access a set of resources but the Java policy model lacks a possibility to also express an *except-for* semantics. For example, if a permission is to be granted to all files in a directory except for few, all (potentially numerous) other files would have to be explicitly listed in the additive permissions. Policy exceptions solve this. For example, policy exceptions allow Charly Brown to grant all permissions for all resources but prohibit access to his home directory (see Figure 2).

```
<?xml version="1.0"?>
<!DOCTYPE localPolicy SYSTEM "localPolicy.dtd">
<localPolicy userName="Charly Brown" lastChanged="10/03/2000">
  <addItems>
    <policyItem signedBy="CK"
                codeBase="http://www.infosys.tuwien.ac.at/-">
      <permission class="java.security.AllPermission">
      </permission>
    </policyItem>
    <policyException signedBy="CK"
                codeBase="http://www.infosys.tuwien.ac.at/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/home/-"/>
        <actions name="read write execute"/>
      </permission>
    </policyItem>
  </addItems>
</localPolicy>
```

Figure 2: Charly Brown's policy definition using a policy exception.

As Figure 1 and Figure 2 indicate, the same semantics can be achieved with either subtractive permissions or policy exceptions. The question whether both concepts are necessary is discussed in the next section.

4.3 Global and Local Policies

As already mentioned in Section 3, Java neither supports the concept of groups nor the enforcement of system-wide security settings. To overcome this drawback the policy concept of JSEF distinguishes between a *global policy* (e.g., defined by the network's security administrator) and a *local policy* (defined by the user) which both can hold additive and subtractive permissions and policy exceptions. A user's local policy settings are under full control of the user and allow a user to define whatever privileges or restrictions he/she wants. All examples presented so far have been taken from Charly Brown's local policy settings. In our example scenario, however, the network is managed by a system administrator who is interested in enforcing a system-wide security policy—the global policy. The global policy is defined as a hierarchical structure of groups. JSEF distinguishes between an additive and a subtractive hierarchy of groups. In an additive hierarchy, permissions are broadened along the inheritance tree, whereas in subtractive hierarchies the restrictions increase along the inheritance hierarchy. Global additive permissions represent a default set of permissions for the members of groups (e.g., the permissions required to make a company's applications work) which can be further adapted by the users. This idea is comparable to the `umask` concept for setting default file access permissions in the UNIX operating system which can also be adapted by the users. Global subtractive permissions, on the other hand, represent global restrictions which are enforced automatically and cannot be circumvented by the users (e.g., network-wide security settings). Permissions are either assigned directly to groups or inherited from a group's parent group. Inheriting in this context means to collect all the permissions and restrictions of all parent groups. For example, Charly Brown can be member of an arbitrary number of such groups and is granted all the permissions defined in them. Figure 3 shows the definition of a subtractive global policy hierarchy and defines negative permissions for the *Admin*, *Developer*, and *User* groups.

Having introduced the notion of a global subtractive policy the question of whether both policy exceptions and subtractive permissions are necessary to achieve an except-for semantics can be answered: When using a global subtractive permission to forbid an action, the user can never overrule this. Using a global policy exception, on the other hand, would give the user the possibility to grant the excluded permission locally. Thus although the same semantics can be achieved, the difference lies in the fact whether the user is allowed to grant the excluded permission locally or not. Local policy exceptions do not differ considerably from local subtractive permissions; the main benefit is to define different qualities of excluded permissions for a semi-automatic negotiation component (see Section 4.5).

4.4 Policy Semantics

To formally describe the semantics of JSEF's policy concepts, a slightly modified version of the *Authorization Specification Language (ASL)* [12] is used. ASL is a logical language to define access control policies and supports the definition of users, groups, authorizations, and the policy according to which access control decisions are to be made. In contrast to ASL's original specification, we have no notion

```
<?xml version="1.0"?>
<!DOCTYPE globalPolicy SYSTEM "globalPolicy.dtd">
<globalPolicy lastChanged="10/03/2000" changedBy="sysadmin">
  <group groupName="Admin">
  </group>
  <group groupName="Developer" parentGroup="Admin">
    <policyItem>
      <permission class="java.io.FilePermission">
        <permissionName name="/system/-"/>
        <actions name="read write execute delete"/>
      </permission>
    </policyItem>
  </group>
  <group groupName="User" parentGroup="Developer">
    <policyItem>
      <permission class="java.net.SocketPermission">
        <permissionName name="*" />
        <actions name="accept connect listen resolve"/>
      </permission>
    </policyItem>
    <policyException>
      <permission class="java.net.SocketPermission">
        <permissionName name="www.sun.com:8080" />
        <actions name="accept connect listen resolve"/>
      </permission>
    </policyException>
  </group>
</globalPolicy>
```

Figure 3: Subtractive global policy for the *Admin*, *Developer*, and *User* groups.

of roles since in JSEF authorizations cannot be activated or deactivated during runtime and we add the code source of a class as new criterion. In the following o is used for an object a given permission applies to, a for an action to be performed on an object, c for the code source of the executing code, and s for a subject requesting a permission. Subjects can either be users (u) or groups (g); if a rule applies to both users and groups s is used. Hence, a permission is represented as a 4-tupel (s,c,o,a) of a subject, a code source, an object and an action. In the following rules all literals L_i (i.e., positive and negative predicates) on the right-hand side must evaluate to *true* to yield the left-hand side of the rule. The $in(s_1, s_2)$ literal used in the following definitions defines that subject s_1 is a member of subject (i.e., group) s_2.

The modified *cando*, *except*, *dercando*, and *do* literals are defined as follows (based on [12]):

A *cando* rule defines the permissions for a subject (either a user or a group) and a code source. The following rule defines that a subject s has a positive (+) or negative (-) permission for a code source c to perform action a on object o if the right-hand side evaluates to true.

$$\texttt{cando}(s, c, o, <+|->a) \quad \leftarrow \quad L_1 \& L_2 \& \ldots \& L_n$$

An *except* rule defines the policy exceptions for a subject (either a user or a group) and a code source. The following rule for a subject s and a code source c defines a policy exception which excludes a positive (+) or negative (-) permission for action a on object o from the policy if the right-hand side evaluates to true.

$$\texttt{except}(s, c, o, <+|->a) \quad \leftarrow \quad L_1 \& L_2 \& \ldots \& L_n$$

A *dercando* (derived cando) rule defines how a subject inherits permissions from another subject for a given code source: A subject s inherits a positive (+) or negative (-) permission to perform action a on object o for code source c if the right-hand side evaluates to true.

$$\texttt{dercando}(s, c, o, <+|->a) \quad \leftarrow \quad L_1 \& L_2 \& \ldots \& L_n$$

A *do* rule defines which permissions of a subject and a code source are applied by resolving potential conflicts. The following rule defines that a subject s is granted a positive (+) or negative (-) permission to perform action a on object o for a code source c if the right-hand side evaluates to true.

$$\texttt{do}(s, c, o, <+|->a) \quad \leftarrow \quad L_1 \& L_2 \& \ldots \& L_n$$

4.4.1 Definition of Permissions and Exceptions

In JSEF permissions and exceptions do not depend on any conditions but are directly assigned to subjects as defined by the following two rules:

$$\begin{aligned} \texttt{cando}(s, c, o, <+|->a) &\leftarrow \; . \\ \texttt{except}(s, c, o, <+|->a) &\leftarrow \; . \end{aligned}$$

For example, in Figure 2 an additive permission and an exception are defined in the local policy. These definitions can be directly mapped onto `cando` and `except` rules by inserting the values given in the figure into the above rules.

4.4.2 Derivation of Permissions

Derivation of permissions in JSEF is defined as follows:

$$\begin{aligned} \texttt{dercando}(s, c, o, a) &\leftarrow \texttt{cando}(s, c, o, a) \& \\ &\quad\; \neg\texttt{except}(s, c, o, a) \\ \texttt{dercando}(u, c, o, a) &\leftarrow \texttt{dercando}(g, c, o, a) \& in(u, g) \\ \texttt{dercando}(g_2, c, o, a) &\leftarrow \texttt{dercando}(g_1, c, o, a) \& in(g_2, g_1) \end{aligned}$$

The first rule specifies the derivation of a permission from the set of specified permissions and exceptions. A permission (s,c,o,a) can only be derived if the permission is defined and no policy exception nullifies the permission. In Figure 2 for example, the permission to access the home directory is nullified by a corresponding policy exception. The second rule states that a user u has to be member of group g to derive a permission from the group. Finally, the third rule says that groups in the group hierarchy inherit the permissions of their parent groups. Figure 3, for example, defines that the *User* group is a subgroup of the *Developer* group and, thus, inherits all its permissions.

4.4.3 Conflict Resolution

The following rules describe how JSEF resolves conflicts in the case of conflicting additive and subtractive permissions or if no permission can be derived.

$$\begin{aligned} \texttt{do}(s_1, c, o, -a) &\leftarrow \texttt{dercando}(s_2, c, o, -a) \& in(s_1, s_2) \\ \texttt{do}(s_1, c, o, +a) &\leftarrow \texttt{dercando}(s_2, c, o, +a) \& \\ &\quad\; in(s_1, s_2) \& \neg(\texttt{dercando}(s_3, c, o, -a) \& \\ &\quad\; in(s_1, s_3)) \\ \texttt{do}(s_1, c, o, -a) &\leftarrow \neg\texttt{dercando}(s_2, c, o, +a) \& in(s_1, s_2) \end{aligned}$$

The first rule states that denials take precedence: If a subtractive permission can be derived, it is enforced. This means that if a subtractive permission is defined either in the global or the local policy, the action is forbidden. If the system administrator forbids an action in the global policy, no user can override this setting by a local policy entry (enforcement of system-wide restrictions). The second rule says that an additive permission is to be granted if and only if the additive permission and no subtractive permission can be derived. This means that a user can apply a locally defined additive permission only if no corresponding subtractive permission is defined. On the other hand, a system administrator can globally grant a permission by defining it in the global policy. Still the user can overrule this global additive permission with a local subtractive one. Finally, it can occur that neither an additive nor a subtractive permission can be derived. In this case the third rule defines the *default decision* which does not grant the permission.

Having formally defined the semantics of JSEF's policy definitions, the benefit of having both subtractive permissions (see Section 4.1) and policy exceptions (see Section 4.2) becomes clear: Excluding privileges using policy exceptions in a global policy definition allows users to individually grant the excluded settings; a globally forbidden action, however, cannot be overruled by the user.

4.5 Interactive Policy Negotiation

Once a user's policy has been constructed by merging the local and global policy settings, a class can be executed. Since JSEF includes concepts that extend the standard Java security policy, a specialized JSEF security manager is used to monitor a class during runtime. As mentioned in Section 2, a security violation in Java's security model normally results in an abnormal termination of the Java virtual machine. If this occurs, a user would have to manually adapt his/her security settings and restart the application. This has to be repeated until all permissions required by an application have been granted. In JSEF an (interactive) policy negotiation component can take care of this. A security violation in the context of JSEF is due to one of three reasons: access to a resource was (1) forbidden by a global policy setting, (2) forbidden by a local policy setting, or (3) not forbidden but neither granted by a global nor by a local setting (default decision). In the first case a user cannot overrule the security decision since system-wide, subtractive settings cannot be influenced by the user. If, however, access to a resource is forbidden locally or merely the appropriate additive permission is not present in the policy settings, policy negotiation is started: The user can dynamically update the policy settings, for example grant missing permissions or adapt the policy exceptions to make the execution of a locally forbidden action possible. A change in the policy means that according changes must be applied to all classes on the call stack which currently do not allow the requested action. Furthermore, the user can decide whether the policy adaption shall apply only once, during the runtime of the application, or shall be permanently added to the user's policy settings. In terms of the language semantics presented in Section 4.4 two situations have to be distinguished: If the user adds a missing permission, the appropriate cando rule for an additive action is added to the user's policy. If a locally forbidden action is allowed, the cando rule with the corresponding subtractive action has to be removed or—in case wildcards are used in the subtractive permission—a new except rule has to be added.

Although the current implementation of JSEF questions the user how to proceed, any other decision making process could be used instead, e.g., automatically deny all requests to emulate Java's default behavior in the case of missing permissions.

The advantages of this concept are that an application need not be aborted due to missing permissions and the automatic policy update frees the user from manually adapting the policy settings.

4.6 Java vs. JSEF in a simple Scenario

Considering the scenario presented at the beginning of this section, the Java security model allows an application to be started and access all resources. To prevent this, user Charly Brown can launch the application with a security manger installed. It will abort the application when the first access to a resource is attempted which was not explicitly permitted. Charly Brown then would have to manually add the requested permission to his policy file and restart the application. This tedious task has to be repeated until all required permissions have been added to Charly Brown's policy definition.

Using JSEF instead would enhance system security since any application immediately would be subject to security restrictions. Additionally, JSEF would support Charly Brown in adapting his policy settings. Thus, the demo application must only be started once and whenever a security violation occurs Charly Brown can decide how to proceed by clicking a button. Furthermore, a system-wide security policy could be enforced in which system administrators could protect the most vital resources against undesired accesses.

5. JSEF IMPLEMENTATION AND TOOLS

JSEF is fully implemented and available under the terms of the GNU General Public License from http://www.infosys.tuwien.ac.at/jsef/ where also exhaustive documentation can be found. The implementation consists of the complete JSEF runtime environment and a set of graphical tools as described in Section 5.2 to support operation and maintenance of JSEF (including a context sensitive help facility).

5.1 JSEF – Java Integration

In Java's default security architecture every attempt to access a system resource results in a call to a check method of the security manager which relays the decision to the access controller (see Section 2). Since JSEF extends the Java security model, the process of handling access requests had to be extended as shown in Figure 4 (a detailed description is given in [14]).

Whenever a check method of JSEF's security manager is invoked by any of the Java core classes, it asks Java's access controller to check the appropriate permission. If the access controller finds a class on the stack which is not granted the appropriate permission, an access control exception is raised to the JSEF security manager which in turn starts the interactive policy negotiation (see Section 4.5) if the denial is not caused by a global subtractive permission which cannot be overruled.

The key issue in JSEF's security model implementation is how the access controller applies JSEF's enhanced policy semantics in the stack inspection algorithm. As depicted in Figure 4, the access controller queries the protection domains of all classes on the call stack whether they grant the requested access. This means that the permission collection stored in the protection domain of the class is checked whether it implies the requested permission. In the case of JSEF this is a specialized `JSEFPermissionCollection` object that knows how to deal with JSEF's policy concepts. This permission collection object is associated with the pro-

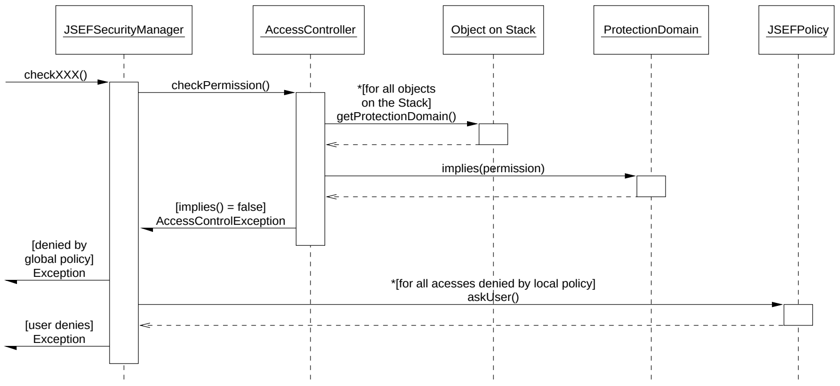

JSEFSecurityManager
AccessController
Object on Stack
ProtectionDomain
JSEFPolicy
checkXXX()
checkPermission()
*[for all objects on the Stack] getProtectionDomain()
implies(permission)
[implies() = false] AccessControlException
[denied by global policy] Exception
*[for all acesses denied by local policy] askUser()
[user denies] Exception

Figure 4: Processing of an access request in JSEF (UML sequence diagram)

tection domain of the class when the class is being defined by a special JSEF class loader. Figure 5 shows the extended `implies` method of the `JSEFPermissionCollection` class. Its task is to decide whether a requested permission is included in the set of permissions stored in its protection domain based on the policy semantics introduced in Section 4.4.

```
public boolean implies(Permission p) {
  if ((p is globally forbidden) &&
      (p is NOT contained in a global subtractive exception)) {
    return false; // globally forbidden
  }
  if ((p is locally forbidden) &&
      (p is NOT contained in a local subtractive exception)) {
    return false; // locally forbidden
  }
  if ((p is globally allowed) &&
      (p is NOT contained in global additive exception)) {
    return true; // globally allowed
  }
  if ((p is locally allowed) &&
      (p is NOT contained in local additive exception)) {
    return true; // locally allowed
  }
  // neither forbidden nor allowed means not allowed!
  return false;
}
```

Figure 5: The extended `implies` method of the `JSEFPermissionCollection` class

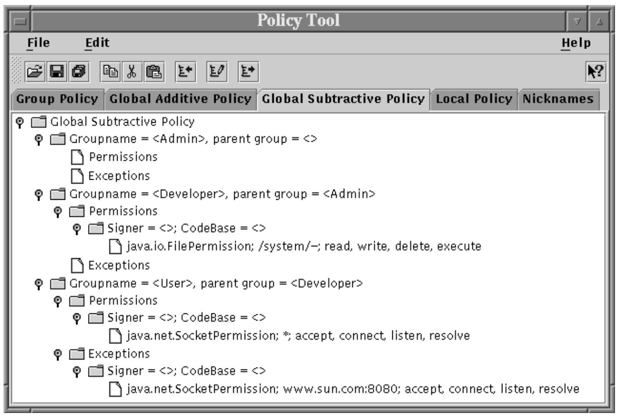
Every `JSEFPermissionCollection` objects stores a set of global additive, global subtractive, local additive, and local subtractive permissions. Thus, the `implies` method has to check each of these four sets to determine whether a given permission is implied according to the policy rules in Section 4.4.

One major constraint concerns the interactive policy negotiation. Java's stack inspection algorithm has a special feature called *privileged mode*. This mode allows classes on the stack to execute according to their permissions without being restricted by less privileged classes (see [14] for a detailed description). Since JSEF cannot figure out which classes on the stack run in privileged mode (the necessary information is private to the `AccessController` class and cannot be accessed without modifications of the JVM), missing privileges are added to all classes on the stack instead of only those called by a privileged class. Thus, permissions are potentially granted to more classes than absolutely necessary. Since in the process of policy negotiation the current policy is only weakened but never made more restrictive, this problem applies only to additive permissions and policy exceptions as discussed in the description of the semantics of the policy negotiation process above.

5.2 JSEF Tools

JSEF offers three graphical tools to manage security policies and its configuration and operation: the *Policy Tool*, the *Secure Application Launcher*, and the *Configuration Tool.* All tools include a context-sensitive help facility based on JavaHelp.

The *Policy Tool* shown in Figure 6 can manage all policy related settings such as the definition of group policies, local and global policy settings, and nicknames (nicknames allow the user to assign simple string names to Distinguished Names which are used to identify certificates but are hard to remember).

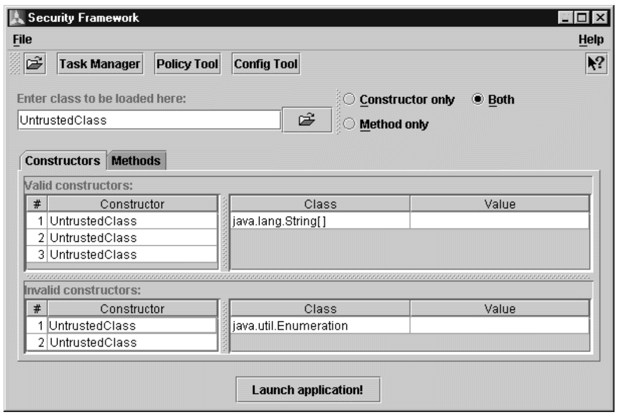


```
Global Subtractive Policy
  Groupname = <Admin>, parent group = <>
      Permissions
      Exceptions
  Groupname = <Developer>, parent group = <Admin>
    Permissions
      Signer = <>; CodeBase = <>
        java.io.FilePermission; /system/-; read, write, delete, execute
    Exceptions
  Groupname = <User>, parent group = <Developer>
    Permissions
      Signer = <>; CodeBase = <>
        java.net.SocketPermission; *; accept, connect, listen, resolve
    Exceptions
      Signer = <>; CodeBase = <>
        java.net.SocketPermission; www.sun.com:8080; accept, connect, listen, resolve
```

Figure 6: JSEF's Policy Tool

It provides comfortable ways to edit the policy settings, context menus, and support for copy and paste of policy subtrees and the contained settings. Figure 6 shows an example view of a global subtractive policy configuration including the hierarchy of user groups, the permissions, and the defined policy exceptions.

The *Secure Application Launcher* (SAL) shown in Figure 7 is the main front-end of JSEF and allows the user to execute classes inside the JSEF environment.



UntrustedClass

[● Both]

Valid constructors:

#	Constructor
1	UntrustedClass
2	UntrustedClass
3	UntrustedClass

Class	Value
java.lang.String[]	

Invalid constructors:

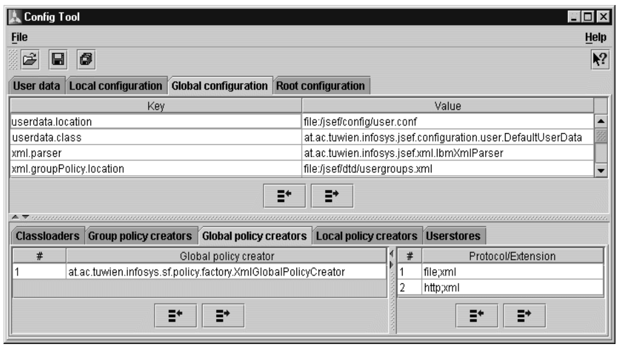
#	Constructor
1	UntrustedClass
2	UntrustedClass

Class	Value
java.util.Enumeration	

Figure 7: JSEF's Secure Application Launcher

SAL does not require the class which is to be started to fulfill special requirements such as having a `main` method. Instead it utilizes Java's Reflection API, examines the given class and allows the user to choose any of its static methods, public constructors, or a combination of these as a "start" method. Like the Policy Tool, SAL offers an easy-to-use GUI. Since both the constructors and methods might require parameters, SAL allows the user to specify values for those parameters. SAL's integrated task manager provides a feedback of all the tasks that currently use JSEF.

The *Configuration Tool* shown in Figure 8 provides a GUI which allows the user to configure JSEF itself.



Key	Value
userdata.location	file:/jsef/config/user.conf
userdata.class	at.ac.tuwien.infosys.jsef.configuration.user.DefaultUserData
xml.parser	at.ac.tuwien.infosys.jsef.xml.IbmXmlParser
xml.groupPolicy.location	file:/jsef/dtd/usergroups.xml

#	Global policy creator
1	at.ac.tuwien.infosys.sf.policy.factory.XmlGlobalPolicyCreator

#	Protocol/Extension
1	file;xml
2	http;xml

Figure 8: JSEF's Configuration Tool

The most important setting is the root configuration which defines where the global policy and the user definitions can be found. This configuration must be secured specially to ensure that it can only be changed by an administrator. The user data view provides the user management front-end of JSEF. Users can be added or removed and their privileges and the location of their local policy definitions can be defined. The privileges of a user include whether the user is allowed to change the various configuration settings and whether the user may alter the local or global policy settings. Additional configuration views are available to define low-level system configurations such as the XML parser used.

6. RELATED WORK

Java's security model is well-documented [4, 5, 6] and many approaches exist to extend or replace this basic model in terms of new security features and capabilities. For example, [15] describes an approach which uses protected domains, so-called playgrounds, to protect machines and resources from mobile code. A playground is a dedicated machine on which the mobile code is executed, with its input and output re-directed to the user's machine. This creates the illusion that the mobile code is executed on the user's computer while it is actually run on the playground machine which is physically separated from the user's machine and thus the mobile code has no access to the user's resources.

The J-Kernel [11] goes even further by replacing the standard Java security architecture with a capability-based system that supports multiple cooperating protection domains inside a single Java virtual machine. While Java's protection domains are closer to the notion of a user, J-Kernel defines them more like processes which considerably changes the security semantics. Via protection domains J-Kernel separates objects into local ones and capability objects which are shared among domains. It provides capability-based communication channels and supports revocation of capabilities.

The security model for aglets [13] uses concepts closely related to JSEF but targets the specific needs of mobile agents. Aglets are mobile agents which execute in a certain context on any aglet-aware host they visit. The aglet security model defines the concept of principals to separate the security requirements of the owner, the manufacturer, and the context master of an aglet. The principals define layers of security in which security settings can be refined but not overruled. If an aglet matches several policy definitions, a "consensus voting rule" combines the policy settings. Since aglets are mobile agents the privileges which define access to local resources are augmented with privileges defining inter-aglet behavior and allowances. Allowances are privileges encapsulating system resources such as memory usage and CPU time, whose implementation and enforcement require (incompatible) modifications to the Java virtual machine. Similar to JSEF, users may be grouped in named groups and share a set of permissions. Policy definitions may be combined with simple boolean operators which supports composite privileges and negation of privileges. Furthermore, black-lists exist to disallow suspicious aglets and contexts. JSEF's subtractive permissions and policy exceptions can be expressed in the aglet model by the use of boolean operators. The different principals imposing policy restrictions on an aglet relate to local and global policies in JSEF. While the aglet security model extends the Java security model to provide inter-aglet permissions, JSEF only builds on the permissions defined by Java. In contrast to the aglet model, JSEF facilitates to structure user groups hierarchically which supports simpler and less error-prone administration of security profiles.

An interesting conceptual approach to extend Java's security features and simplify definition of security profiles is presented in [19]. This approach suggests a constraint language which allows the user to specify security constraints which are a combination of subject-based, object-based, and history-based policy statements. History-based constraints are a powerful concept and support the definition of policy rules over time. For example, it could be specified that an applet can only make 5 write accesses to a file. Additionally rules can specify conditional constraints, such as if a piece of mobile code wishes to access a protected file it no longer can make a network connection. Constraints can be combined with simple logical operators and can define both additive and subtractive permissions (similar to JSEF). Policy exceptions are not an explicit concept but can be defined implicitly. No grouping mechanisms and hierarchies exist which makes it difficult to assign layered security profiles to users. The definition of constraints is cumbersome since an S-expression type language is used and no graphical tool support is available. This approach has been applied to and tested only with JDK 1.1. However, we plan to further investigate the addition of a constraint language as suggested in [19] to JSEF to further extend it.

In contrast to these approaches, JSEF does not introduce incompatible Java security features. Instead it uses the existing Java security architecture but enhances its usability, introduces higher-level abstractions and hierarchical policies, and offers new ways of configuration as described in the previous sections. It simplifies the definition and maintenance of security policies at the system and at the user levels. Such simplification facilitates to prevent the introduction of security holes and thus improves a system's overall security characteristics. None of the above approaches supports (interactive) runtime security negotiation and offers simple ways to define system-wide (or even network-wide) security policies as JSEF. Also tool support for security maintenance is very limited or does not exist at all.

The idea of overruling authorizations is also presented in [1] where a multipolicy access control system for databases is discussed. Users and groups can be assigned positive and negative authorizations which can either be strong or weak. Authorizations can be overridden by more specific authorizations (according to their position in the group membership graph). In JSEF a user inherits all (even conflicting) permissions and conflicts are resolved afterwards. In [1] strong authorizations always overrule weak ones and the set of strong authorizations must be consistent, i.e., no conflict among strong authorizations may exist. Conflict resolution rules only apply to conflicts among weak authorizations. The main difference to JSEF's policy model is that in JSEF global permissions are not necessarily stronger than local ones, e.g., a local subtractive permission is allowed to overrule a global additive one while a global subtractive permission can never be overruled by a local additive one. The strength of a permission in JSEF is thus dependent on both the scope (global or local) and the type (additive or subtractive) of the permission.

7. CONCLUSIONS

The Java Secure Execution Framework (JSEF) presented in this paper is built on top of Java's standard security architecture and extends it with powerful features in a compatible way. It uses a hybrid policy model which supports additive and subtractive permissions with a denial-take-precedence rule to resolve conflicts. JSEF's policy semantics is formally defined in an ASL-based notation. It provides a security framework which simplifies the maintenance of security profiles and provides graphical tool support for security administration. Better maintenance support may improve overall system security since it helps to prevent sloppy configurations or the introduction of security holes by erroneous configurations.

While in standard Java only permitted accesses can be defined, which can blow up configurations and makes them cumbersome to maintain, JSEF additionally supports the specification of forbidden accesses (subtractive policy) and policy exceptions. JSEF's hierarchical groups provide a concept to aggregate users into groups, freely structure these groups into a hierarchy, and assign security profiles to them. The concept of global and local policies in JSEF enables the definition of network-wide security policies that define a security corset for users while still allowing them to freely adjust their configurations inside these mandatory security standards. Thus users can tailor their local policy towards their needs but cannot break the system-wide policy.

In the standard Java security model, a forbidden access typically terminates the execution, whereas JSEF offers the possibility to negotiate security at runtime. JSEF intercepts forbidden accesses and the user (or a special security control component) can negotiate with the relevant Java code about the requested permissions. This can avoid tedious trial-and-error cycles to find out about the actual permissions required by a piece of mobile code.

JSEF, all associated tools, and full documentation are available under the terms of the GNU General Public License from http://www.infosys.tuwien.ac.at/jsef/.

8. REFERENCES

[1] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, Califorinia*, pages 94–107, May 1996. http://isse.gmu.edu/~csis/publications/oklnd96-samarati.ps.

[2] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems – concepts and design*, chapter 16, pages 477–516. International Computer Science Series. Addison-Wesley, Reading, Mass. and London, 2nd edition edition, 1994.

[3] S. Fritzinger and M. Mueller. Java security, 1996. Sun Microsystems, Incorporated. White Paper. http://java.sun.com/security/whitepaper.txt.

[4] L. Gong. Secure Java Classloading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.

[5] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: an overview of the new security features in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997*. USENIX Association, 1997.

[6] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security, San Diego, CA, USA*, 1998.

[7] S. Gritzalis and D. Spinellis. Addressing threats and security issues in world wide web technology. In *Proceedings of CMS'97, 3rd IFIP TC6/TC11 International Joint Working Conference on Communications and Multimedia Security, Athens, Greece*, pages 33–46, September 1997.

[8] M. Hauswirth. *Internet-Scale Push Systems for Information Distribution—Architecture, Components, and Communication.* PhD thesis, Distributed Systems Group, Technical University of Vienna, October 1999.

[9] M. Hauswirth and M. Jazayeri. A Component and

Communication Model for Push Systems. In *Proceedings of the ESEC/FSE 99 – Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), Toulouse, France, September 6–10, 1999*, pages 20–38, September 1999. http://www.infosys.tuwien.ac.at/Staff/pooh/papers/PushIssues/.

[10] M. Hauswirth, C. Kerer, and R. Kurmanowytsch. *Minstrel Client Security Framework*, 1999. http://www.infosys.tuwien.ac.at/Minstrel/Receiver/CSF/.

[11] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, June 1998.* USENIX Association, 1998.

[12] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 4–7, 1997, Oakland, CA*, 1997.

[13] G. Karjoth, D. B. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, 1(4), July 1997. http://computer.org/internet/ic1997/w4068abs.htm.

[14] C. Kerer. A flexible and extensible security framework for Java code. Master's thesis, Distributed Systems Group, Technical University of Vienna, Austria, October 1999.

[15] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure Execution of Java Applets using a Remote Playground. In *Proceedings of the IEEE Symposium on Security and Privacy, Los Alamitos, California*, May 1998.

[16] G. McGraw and E. Felten. Java security and type safety. *Byte*, 22(1):63–64, January 1997.

[17] G. McGraw and E. W. Felten. *Java security: hostile applets, holes, and antidotes.* John Wiley, New York, 1997.

[18] G. McGraw and E. W. Felten. *Securing Java: getting down to business with mobile code.* John Wiley, New York, 1999.

[19] N. V. Mehta and K. R. Sollins. Expanding and Extending the Security Features of Java. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas, January 26-29, 1998.* USENIX Association, 1998.

[20] A. Rubin and D. E. Geer. Mobile Code Security. *IEEE Internet Computing*, 2(6):30–34, November/December 1998.

[21] Sun Microsystems, Incorporated. *Secure computing with Java: now and the future*, September 1998. White Paper. http://java.sun.com/marketing/collateral/security.html.

[22] F. Yellin. Low level security in Java. In *Proceedings of the Fourth International World Wide Web Conference, Boston, Massachusetts, USA, December 11–14, 1995*, volume 1 of *World Wide Web Journal.* O'Reilly & Associates, Incorporated, November 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.