A Distributed Access Control Model for Java

Refik Molva, Yves Roudier

Institut Eurécom, BP 193, 06904 Sophia-Antipolis - France
{molva,roudier}@eurecom.fr

Abstract. Despite its fully distributed and multi-party execution model, Java only supports centralized and single party access control. We suggest a new access control model for mobile code that copes with the shortcomings of the current access control model of Java. This new model is based on two key enhancements: the association of access control information with each mobile code segment in the form of attributes and the introduction of intermediate elements in the access control schema. The combination of the current ACL-based approach with the capability scheme achieved through mobile code attributes allows the new access control model to address dynamic multi-party scenarios while keeping the burden of security policy configuration at a minimum. We finally sketch the design of an access control system based on the proposed model using Simple Public Key Infrastructure (SPKI) certificates.

Keywords: Java, access control model, distribution, SPKI, capabilities

1 Introduction

The Java runtime environment (JRE) offers a rich set of security mechanisms for mobile code. The security features of the JRE evolved from the confinement-based sandbox approach of release 1.0 and 1.1 to a full-fledged access control model [GMPS97,Sun98,KG98] as implemented in release 1.2, also called Java 2. Java 2 offers fine-grained access control whereby the operations of mobile code segments on the local resources are controlled via access control lists (ACL) represented by a set of permissions. Each mobile code segment is granted a set of access rights defined as permissions. Access control is enforced during runtime by verifying the permissions of the mobile code for each operation attempting an access to a protected resource.

This access control solution suffers from a strong limitation: despite the truly distributed nature of its execution model, access control in Java 2 is based on a centralized security model. The JRE and the Java language offer a perfect environment for distributed computing in the sense that not only components from various sources can be dynamically integrated with an application at runtime, but also mobile programs can seamlessly run in remote environments like the Java virtual machine. The basic assumption underlying the execution model of Java is that each program may consist of various components and that each component can be generated by independent parties that can be geographically distributed. Such an execution model can be qualified as truly distributed, whereas the access control model of Java 2 seems to be based on a centralized model. In the security model of Java 2, each distributed component needs to be designed in compliance with the security policy of the target environment and the security policy of the target environment needs to fulfill the requirements of each potential component that might be imported by the target environment. The only reasonable way to assure a meaningful collaboration between distributed components and a local environment governed by the security policy seems to place both the remote components and the local security policy under the jurisdiction of a single party. This is a serious limitation in the face of a fully distributed scenario taking advantage of the distributed nature of Java that affords a multi-party execution model including components from multiple sources and runtime environments with varying security policies.

In this paper, we suggest a new access control model for mobile code that copes with the shortcomings of the current access control model of Java. Section 3 describes the new model that is based on two key enhancements: the association of access control information with each mobile code segment in the form of attributes and the introduction of intermediate elements in the access control schema. The combination of the current ACL-based approach with the capability scheme achieved through mobile code attributes allows the new access control model to address dynamic multi-party scenarios while keeping the burden of security policy configuration at a minimum. Section 4 presents the design of an access control system based on the proposed model using SPKI certificates.

2 Access Control in Java 2

If we summarize the goal of access control as the enforcement of a model defining the authorized operations between active components called *subjects*, and resources called *objects*, in the JDK 1.2 access control model, subjects correspond to *protection domains*, and objects to local resources like files or system services. The protection domain of a class is identified by the CodeSource of that class and the CodeSource consists of the CodeBase or URL from which the class was loaded, and the signature of the CodeBase on the class file. In the recent JAAS extension to JDK, the protection domain concept also encompasses the identity of the user who runs the code within the JVM, thus allowing multi-user operations within a single JVM. Access rights are in turn represented by Java 2 constructs called *permissions* that include the details of an operation authorized on a protected resource. Access rights are granted to subjects by assigning permissions to protection domains in a policy file during configuration. The default implementation of the JVM supports two policy files defined respectively by the system administrator and the user.

The actual enforcement of access control at runtime is based on the reference monitor concept. The reference monitor can be implemented either by the *SecurityManager* class or the *AccessController* class. During its execution each object is labeled as belonging to a protection domain and will be granted an access based on this annotation.

Different permissions can be granted to a <protection domain, resource> tuple. The default policy is to deny all access to protected resources unless otherwise stated. The effective access permission as derived by the reference monitor corresponds to the intersection of all permissions of objects that are part of the execution thread, that is, the intersection of permissions from several protection domains. The only exception to this rule is the "privileged code" mark that enables a trusted code to keep its permissions from being shared by its callers'.

3 Shortcomings

Unlike the Java execution model that is fully distributed, the access control mechanisms of Java 2 are based on a centralized security model. As depicted in the previous section, in the default implementation of JDK 1.2, the access rights of each mobile component are defined in a configuration file located in the runtime environment. This file includes the mapping between each mobile component and the permissions granted to the component during its execution on the local runtime environment. The reliance of access control decisions on the local configuration file results in two major limitations from the point of view of a distributed environment. First, each possible remote component or mobile code that can be authorized to access local resources must be identified beforehand and registered in the policy configuration of the runtime environment. Second, each component designer or mobile code programmer must take into account the access control restrictions of all potential target runtime environments at the design stage.

Apart from numerous practical difficulties in terms of programming, these limitations hinder the deployment of a dynamic distributed environment by requiring a static definition of all distributed components and their security attributes at once.

Further analysis points to the fact that these limitations are due to the centralized nature of the underlying access control model for Java 2. This model, which is based on the access control list (ACL) concept, requires that all the access control information be located near the resources that are subject to access control. Since the access control information includes both the identities and the attributes of all the potential subjects that might issue access requests, the resulting access control system necessarily needs to keep a centralized and static information base including all potential components and their attributes. Figure 1 depicts the centralized and single party access control model of Java 2.

Even though an ACL-based model can quite efficiently suit a centralized organization whereby a single party manages the security policy of all the components,

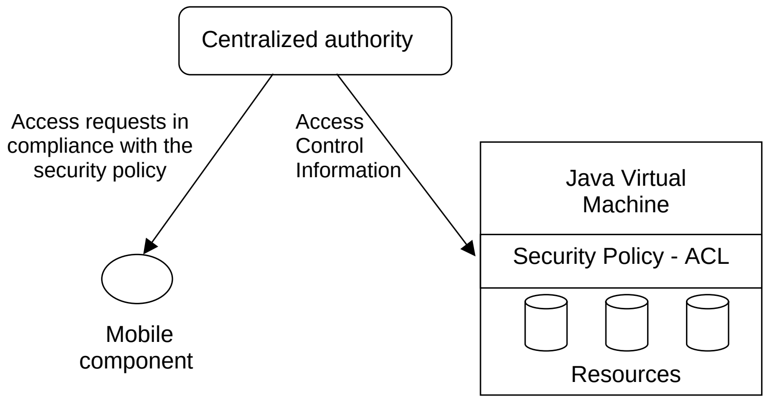

Fig. 1. Centralized and single party access model of Java 2.

it does not meet the requirements of dynamic multi-party scenarios akin to Internet applications. Multi-party environments require that the access control decisions be shared between the local party that manages the resources, or *objects* in the access control terminology, and the remote parties that generate mobile code components, or *subjects*. Dynamic scenarios on the other hand call for a solution that suits dynamic populations both in the subject and object categories. A dynamic solution would then allow mobile code components to join and leave the set of potential subjects as well as a variable number of runtime environments to offer a variable set of resources for access by mobile components.

Another shortcoming of the single party model of Java 2 is that it does not properly exploit the public key infrastructure (PKI) on which it relies for authentication. Java 2 requires authentication of each mobile component based on digital signature verification using identity certificates defined as part of an X.509 PKI. The main objective of a PKI based on X.509 is to allow any two parties to be able to authenticate one another without any prior knowledge or without any bilateral trust relationship between those parties other than a trust chain that can be established through the global certification tree. Because of its centralized and single party access control model, Java 2 does not take advantage of the underlying PKI's global communication capabilities. Instead of a global PKI, a simple authentication model based on shared keys or a flat public key infrastructure using a single certification authority would as well be sufficient to meet the authentication requirements of Java 2.

As a result of the limitations in its access control model, Java 2 does not seem suitable for scenarios that are inherently multi-party or applications that involve dynamic populations of mobile code producers and consumers. For example, an extranet that consists of two or more interconnected corporate networks or intranets raises an access control problem that is inherently multi-party. Let's consider a mobile code access scenario whereby a mobile code component from intranet A attempts to access the resources of intranet B. The access control decision should take into account not only the security policy of intranet B with respect to its local resources but also the attributes of the mobile code component as defined by intranet A. Using Java 2's single party model in this scenario would require intranet B to incorporate into its access control information the attributes of authorized mobile code components from intranet A.

4 Towards a Distributed Access Control Model

In order to alleviate the limitations of Java's centralized and single party access control model, this paper suggests a new access control model that allows for distributed and multi-party operations.

This model enhances the existing access control model with two new concepts:

- access control attributes located with the mobile code segments, in addition to the access control information located within the runtime environment;
- intermediate access control elements that allow for the independent configuration of access control attributes accompanying the mobile programs and access control information located in the runtime environments.

4.1 Access Control Attributes

As depicted in section 2, a mobile code producer and its consumer are different parties: this requires both parties to cooperate in order to define a security policy that suits both parties. Each mobile code component has its own resource usage pattern that is a priori unknown by potential code consumers. In the current access control model of Java 2, the mobile code programs do not carry any access control information other than some identity certificate and a digital signature required for the purpose of authentication.

The first enhancement suggested in this paper consists in associating part of the access control information with the mobile code using a new type of component annotation called **attribute**. Attributes define a mobile code's authorizations in terms of access requirements, behavior, and software compatibility. Attributes are the basic means through which multi-party access control can be achieved, i.e. parties other than the runtime environment can participate in the access control process.

Similarly, the security requirements of dynamic distributed environments can be met thanks to the distributed definition of access control information using attributes. The other part of the access control information is not defined for specific components, but rather as general rules for generating a specific permission. A typical access control process using attributes should thus include an additional operation called attribute resolution. The main purpose of attribute resolution is to combine the information contained in the attribute with the access control rules stored in the local security policy in order to derive local access permissions and rules for the compatibility of the mobile component with the runtime environment or with other components. Once attribute resolution is complete, access enforcement during runtime is performed based on the existing Java 2 model.

4.2 Intermediate Elements

Even if each mobile code was tagged with attributes, the configuration of access control information in a dynamic environment might still be very complex. If access control information included in the attributes were defined in terms of (*subject*, *object*, *right*) tuples, each party defining an attribute for a mobile code would need to be aware of individual resources (*objects*) available at potential target runtime environments and, conversely, access control information included in each runtime environment would still have to enumerate all possible mobile code components (subjects). In this case, the advantage of introducing attributes over the existing Java 2 access control model would merely be limited to multi-party extension. Because of the inherent complexity, this solution would still not scale to large populations of code producers and consumers.

In order to cope with this complexity, we suggest a second enhancement that consists in factoring the access control information represented by the basic (subject, object, right) relation used in Java 2 into two simpler relations: (subject, *intermediate_element*) and (*intermediate_element*, object, right) (Figure 2). Parties involved in the access control process have to agree on **intermediate elements** that are abstractions of existing subjects and objects. Suitable instances for intermediate elements could be source authorizations (roles, groups) or predefined levels of execution contexts (library requirements, dynamic resource requirements, acceptable behavior, etc.).

The resulting access control information offers several advantages in terms of reduced complexity and independence of multi-party operations:

- an access control model for n subjects, m objects using p intermediate_elements can be described using $(n+m) \times p$ entries whereas the simple model would require $n \times m$ entries; there is a clear advantage when n and m are very large with respect to p;
- the two relations can be defined independently; in particular, subject and object populations can be managed in a totally independent manner;
- the first relation, (subject, *intermediate_element*), lends itself perfectly to the definition of attributes whereas the second one, (*intermediate_element*, object, right), is suitable for the description of access control rules stored with the runtime environment.

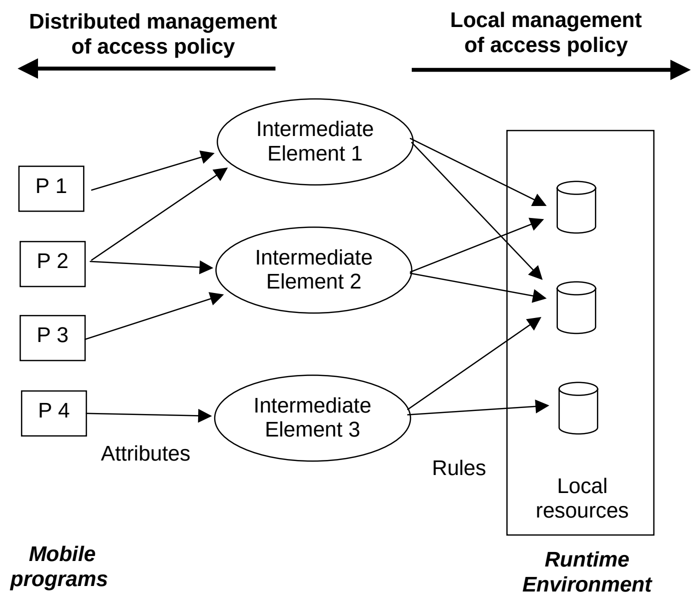

Fig. 2. Intermediate elements.

Another argument speaks for the use of intermediate elements. In the suggested distributed model, a mobile code is to carry its security attributes. If a simple model were used, these attributes would grow with the number of potential execution sites, since each would have different resources and different policies applied to them. With the use of intermediate elements, we can achieve to define the resources needed by a mobile code for several runtime environments simultaneously.

4.3 Advantages over the Centralized Model

The main improvement of the suggested model is the independence between code producers and code consumers with respect to the definition of access control policy. Unlike the current Java 2 model, the new model does not require the mobile code consumer to keep track of each mobile code component that can potentially be integrated or executed on the local runtime environment. Conversely, the code producer does not need to know any detail about the runtime environment at the time of code writing. Nonetheless, he can specify as part of the attributes the features relevant for the successful execution of his code. As a result of this independence, access control for mobile code can be achieved in very dynamic and complex environments with a large number of code producers and code consumers. In particular, the security policy of the runtime environments does not need to be updated when new parties produce mobile code components destined to these environments.

Thanks to the use of intermediate elements, access control does not rely on mobile components' identities. Like in *capability* schemes, the verification of attributes granted to a mobile code component does not require the knowledge of any identity. Consequently, this model can potentially achieve anonymity with mobile code components.

4.4 Deployment Cases

Possible deployment cases of the proposed model are depicted for some generic scenarios.

Single Party Case. This case is similar to the usual Java 2 scenario whereby a single party, the code consumer, defines all the access control information for the runtime environment. Even in this simple case, our model offers an advantage over the access control of Java 2 in that a dynamic mobile code population can be supported through the grouping of their common features with intermediate elements without increasing the complexity of the security policy configuration on the code consumer side.

Two-Party Case. The two-party case fully takes advantage of the new access control model.

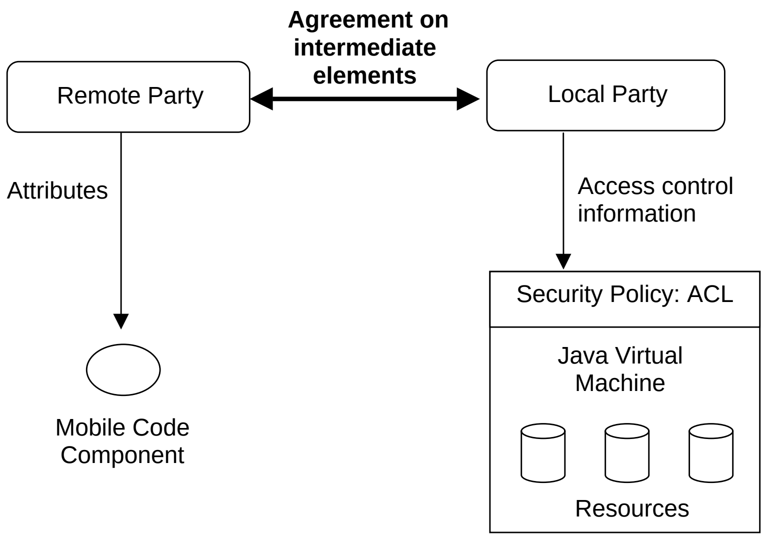

Fig. 3. Two-party case.

As depicted in Figure 3, access control is ruled by two different parties: the *local party* associated with the run-time environment sets the access control information concerning the resources located at the run-time environment whereas the *remote party* associated with the mobile code defines the attributes of the mobile code. The intermediate elements used in the definition of the access control information and the attributes can either be defined through a negotiation between the local party and the remote party or they can simply consist of predefined values. For instance, the local party might define two permission sets, one for games, and one for professional applications, each corresponding to several smaller-grained permissions about local resources. It might also define attributes to enable the mobile code to check if a given freeware library is installed or not before running, but forbid the same inquiry about a commercial library, and so on. If a number of such intermediate elements were established as a minimal standard, existing mobile code programs might be retrofitted with a flexible yet simple access control policy.

Three-Party Case. In the three-party case (Figure 4), the local party associated with the runtime environment sets the access control information governing the access to the resources managed by the runtime environment as in the two-party case.

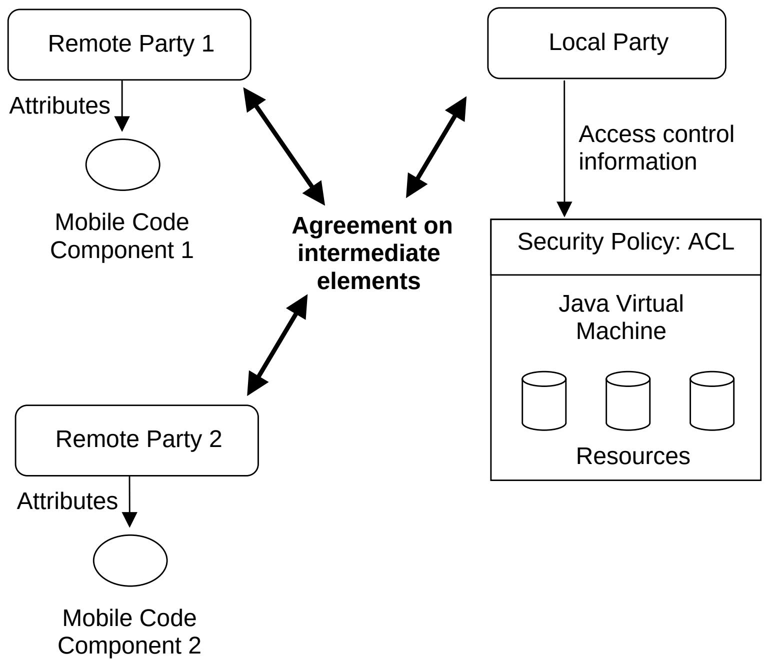

Fig. 4. Three-party case.

In this scenario a different remote party is associated with each of the two mobile code segments. Most interactions between the three parties occur in fact between one of the remote parties and the local party. These interactions thus resolve to two-party case interactions. However, some interactions involve the three parties at the same time. This occurs for instance in the following conditions: a component from remote party 1 that introduces new permissions is used in a program from remote party 2; remote party 2 ignores the identity of remote party 1, the access to component 1 being enforced by the local party. In that instance, the agreement on intermediate elements mentioned in the previous scenario involves the three parties. After an agreement has been reached on a given intermediate element determined either implicitly or explicitly, each remote party can annotate its mobile code components separately, in a distributed manner. Reaching an agreement might be as simple as defining a role for each participant.

5 Design of a Distributed Access Control System for Java

The previous section presents a distributed access control model as an alternative to the centralized Java 2 access control model. This section presents the design of a solution based on the new model using the Simple Public Key Infrastructure (SPKI) framework as a basis for implementing intermediate elements.

5.1 SPKI

The Simple Public Key Infrastructure (SPKI) [EFL$^+$98a,EFL$^+$98b,EFL$^+$99] was started as an authorization-based infrastructure destined to answer access control problems in wide-scale networks. SPKI is now supported by IETF.

The focus of SPKI is the definition of authorization certificates. An SPKI authorization certificate is the encoding of an access control capability: it states that a given subject is granted a set of permissions by an authority, called the issuer, and under some conditions (for instance a given duration), called the validity. This statement is corroborated by the accompanying signature of the issuer and the certificate also includes the public key of the issuer.

As opposed to X.509 [IT88], SPKI does not require the identification of a party as a prerequisite to the access control decision concerning an operation requested by that party. SPKI allows instead to verify the rights of a party regardless of the party's identity. In addition, like X509, SPKI also allows for the representation of identities in public key certificates.

Furthermore, systems designed using SPKI often rely on delegation. Delegation means issuing an authorization certificate for a certain set of rights to another issuing authority. This authority can then itself issue certificates granting a subset of these rights, and so on. Delegation provides support for the distributed definition of authorizations. SPKI defines a precise semantics describing how to reduce a chain of delegated authorization certificates.

In addition to the basic authorization and identity certificates, SPKI has borrowed a mechanism for group certificates from the SDSI framework [RL96,Aba98]: an SDSI group certificate refers to a group of certificates. Group certificates allow treating a set of entities as a single entity. It is thus possible to grant or revoke rights to/from a set of users in a single authorization certificate. A key, that is, a subject, is considered to possess its own namespace corresponding to a group.

In summary, three types of certificates coexist in SPKI:

- public key certificates that can be modeled as $< K_{issuer}$, $name_{subject}$, $K_{subject}$, *validity* $>$ 4-tuples: the issuer states that subject $name_{subject}$ is identified by key $K_{subject}$
- group certificates, corresponding to $< K_{issuer}$, $name_{group}$, $name_1 ... name_n$, *validity* $>$ 4-tuples: the issuer says that the name chain "$name_1$ ... $name_n$" is identified by $name_{group}$ in his namespace; "$K_{issuer2}$ $name_{group2}$" is an example of a name chain defining the name group2 in the certificate namespace of issuer2.
- authorization certificates, that is, $< K_{issuer}$, $name_{subject}$, *delegation*, *authorization, validity* $>$ 5-tuples: the issuer states that $name_{subject}$ has been granted some authorization; if *delegation* is true, it states that the subject has also been granted the right to issue certificates stating the same privileges than those he was granted, or a subset of these privileges, to another subject.

An issuer can conclude the correctness of a set of certificates only when it can establish a chain of certificates starting from a self-signed certificate. The process through which a certificate chain is verified is named "reduction" in SPKI.

For many access control matters, SPKI now supersedes X.509. The delegation mechanism of SPKI is far superior to the simple cross-certification, and can implement it in a straightforward way. Moreover, with group certificates, SPKI can now specify role-based access control policies.

5.2 Components of the Design

Let's now focus on how SPKI certificates are used to support the annotation of mobile code attributes in our design. SPKI group certificates offer a perfect ground for the definition of intermediate elements, as depicted in Figure 5(a). Group certificates link the namespace of the local party with the namespaces of remote parties. Group certificates may also be used as a declaration of the intermediate element to the remote party. For instance, the certificate that we abbreviate as $< K_{LP}$, $group1$, K_{RP1} $group1 >$ means that the local party (identified by its key K_{LP}) will declare $group1$ to the remote party $RP1$ (identified by its key K_{RP1}). *Group 1* will be referred under the same name in the namespaces of both parties and in subsequent certificates.

A mobile code component is identified by a public key certificate issued by its producer. It can also be identified by an SPKI group certificate issued by its producer and that can be ultimately identified by a public key certificate. Attributes will be written as SPKI group certificates attached to components. The latter group certificates will be issued by a remote party, but they must be chained with a certificate issued by the code consumer in order to be interpreted. An attribute reflects which intermediate element - be it a role, a user community, or an execution context - a mobile code is mapped to. A given mobile code can carry several such certificates. In Figure 5(b), remote party 2 issues the certificate $< K_{RP2}, group3, K_{MC2} >$ associating mobile code $MC2$ with *group 3*.

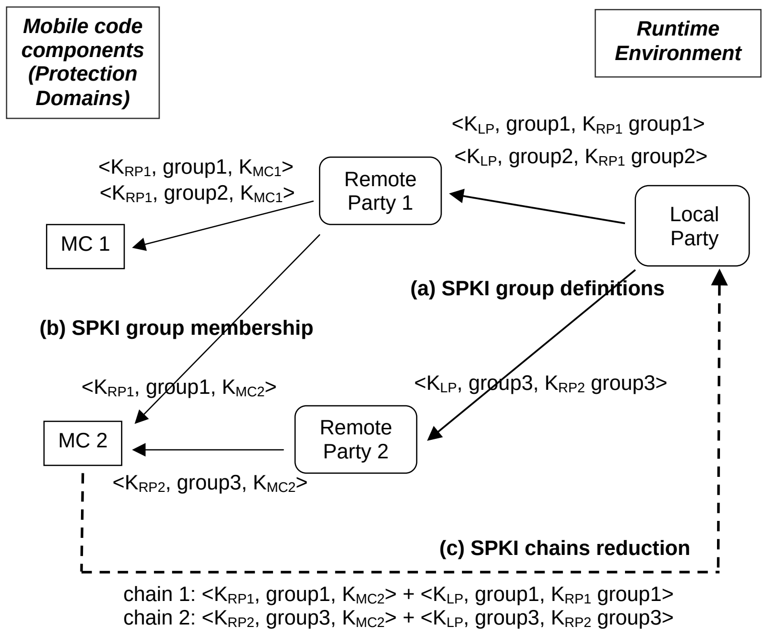
Mobile code components (Protection Domains)
Runtime Environment
<K_LP, group1, K_RP1 group1>
<K_LP, group2, K_RP1 group2>
<K_RP1, group1, K_MC1>
<K_RP1, group2, K_MC1>
Remote Party 1
Local Party
MC 1
(a) SPKI group definitions
(b) SPKI group membership
<K_RP1, group1, K_MC2>
<K_LP, group3, K_RP2 group3>
MC 2
Remote Party 2
<K_RP2, group3, K_MC2>
(c) SPKI chains reduction
chain 1: <K_RP1, group1, K_MC2> + <K_LP, group1, K_RP1 group1>
chain 2: <K_RP2, group3, K_MC2> + <K_LP, group3, K_RP2 group3>

Fig. 5. Issuing and resolving access control attributes.

Attribute-policy resolution rules specify how intermediate elements should be translated into resource access rights, i.e. effective Java 2 permissions concerning the JVM local resources. These rules can be stored within SPKI authorization certificates. With intermediate elements interpreted as subjects, the rules provide a summary of the potential ACL entries for the runtime environment. The rules also seem to be a useful tool for making the inspection and revocation of access rights an easier task.

Let's now outline a typical attribute-policy resolution as depicted in Figure 5(c). Each certificate chain carried by the mobile code is reduced with the SPKI engine into a summary certificate that is issued by the local party. The summary certificate defines the mapping of the attribute to an intermediate element. Certificates issued locally may not be included with the mobile code in order to assure the compatibility of mobile components with a wide range of run-time platforms, and in particular to avoid the need for a specific version of components for each consumer. In that case, the certificate missing from an SPKI chain must be stored locally, possibly in an intranet server. This certificate can be retrieved at the time of reduction. The second part of resolution occurs when the mapping between the attribute and the intermediate element has been established. The local party has then to map the intermediate element obtained to an effective set of Java 2 permissions. This is achieved using the translation rules mentioned above. The resulting permissions are then included in the existing ACL definitions.

A problem might persist in this scenario due to the verification of validity conditions included in the certificates: checking the summary certificate only once at class loading is not sufficient to verify the validity of all the individual certificates referenced by the summary certificate. The validity of the summary certificate should be limited to the intersection of validity conditions of each individual certificate. Two alternative solutions can be envisioned to solve this problem:

– The validity of the summary certificate is re-evaluated at every access. In Java 2, a class is marked as pertaining to a protection domain, which means that the implementation associates a set of access control permissions to a protection domain. Based on this technique, the runtime environment could be extended so that it also stores and checks the validity conditions of the certificates along with the permissions of the protection domains.
– The policy configuration is refreshed periodically. This can be achieved by adding a background thread periodically calling the Policy.refresh() method. However, the refreshed policy may not change the policy set for classes already loaded, depending on the caching strategy of the class loader. Additional mechanisms might be required in existing Java 2 implementations.

The latter solution is preferable, because of its relatively minor impact on the performance of access control operations and the flexibility it offers with respect to the definition of the refreshment frequency.

5.3 Example

In this section we turn to an example where our design is used to implement a role-based access control system. Let's assume that a remote party, Bob, needs to define security restrictions on the applets he downloads from the Internet.

Applets should be enabled to perform read operations on disk, but strictly forbidden write operations except for the gaming applets written by Alice or Brian. Bob might define two different applet roles, namely "browsing" and "gaming".

Suppose Alice and Brian have defined a role named "game" corresponding to gaming applets. Alice and Brian would then issue SPKI group certificates under this name and attach them to their various gaming applets. Bob will also issue an SPKI group certificate for each of the roles that he defined, as well as corresponding rights: the "browsing" certificate group will be used as the role attached by default to any applet entering Bob's intranet with read access. The "gaming" group certificate will be issued for Alice's and Brian's applets which share the same access needs. This certificate will link Bob's certificate namespace to Alice's (respectively Brian's) certificate namespace so that Alice's "game" might be seen as Bob's "gaming" role. Roles of authorized applets thus form a chain starting at Bob, who acts as his own authority. Based on the SPKI principles, Alice and Brian are only identified by their public keys, as stated in the group certificate issued by Bob.

As part of the local access control policy, Bob must have configured the rights associated with the "gaming" role. When an applet from Alice is loaded into a machine in Bob's intranet with Alice's "game" certificate, a chain starting from Bob using the "gaming" certificate stored in Bob's machine can be established and matching writing rights are associated with the applet. Alice's and Brian's applets share the same role: the same translation will therefore result in the same set of Java rights for Brian's applets.

Within the global definition of the "gaming" role, sub-roles can be defined by Alice and Brian who might decide to use routines from other parties and further grant them the same rights their applet was granted by Bob. SPKI thus allows Bob to delegate Alice or Brian the management of rights originally granted by Bob. This would be achieved for instance by Alice's issuing new certificates indicating that Mike's "high-score routine" is part of Alice's "game".

In this example, intermediate elements are just roles, but provide a simple common denominator: Alice's applets would run on Fred's browser as well, provided that he configured his runtime environment so that Alice's "game" is recognized as a role and is granted enough access rights. Other types of intermediate elements might be introduced, for instance, Alice might also state in a certificate that a particularly complex game needs a powerful CPU. Reducing such a certificate would mean that Bob has installed a plugin so that the SPKI engine performing the attribute-policy resolution can check the microprocessor of his machine.

5.4 Application: An Extranet Using Mobile Code

A federation of intranets or extranet offers another interesting example highlighting the suitability of our design for distributed and multi-party scenarios. An extranet access control system has to cope with an inherently multi-party scenario in that objects belong to several domains and each object can be accessed by several subjects from different domains. Suppose that applications programs in that extranet consist of Java mobile codes. In such an extranet, application deployment will require the granting of rights to mobile code components. These rights will concern the resources of the runtime environments where each mobile code can possibly be executed. The same mobile code must however be granted rights for local resources of all potential consumers, that is, all intranets that are part of the extranet. Enumerating all the combinations would be too complex and cumbersome. Even worse, adding a new intranet to the extranet would impact the attribute definitions in all the existing mobile codes segments.

Using the intermediate elements of our model, this deployment scenario becomes much simpler. For instance, instead of defining a hard disk or a printer as a resource, an intranet might advertise two sets of resources, one dedicated to a professional use, and another to a personal use; or define a set of resources available to all users of a given intranet. These sets will be declared as intermediate elements of access control.

In that particular application, each intranet advertises intermediate elements as SPKI group certificates to other intranets that are part of the extranet. These elements can then be viewed as standard service interfaces between the different intranets. As in X.509-based solutions, a cross-certification process is needed. It amounts to the exchange of the keys of certificate issuers. Our model otherwise directly supports the distributed nature of an extranet. Certificate reduction (the resolution of access control attributes into effective permissions) can be automated. Another major benefit of intermediate elements in this example is that new intranets can join the extranet without any impact on the access control definitions of existing components.

6 Related Work

A proposal by Nikander and Partanen [PN98,NP99] also addresses the issue of how to enhance access control in Java. In order to cope with the current Java's requirement on each end-user to individually set and update the security policy file on his machine, the authors suggest to store permissions in a distributed fashion together with applets. Their design is based on a modified version of SPKI whereby SPKI authorization certificates serve as a tool to store Java 2 permissions. This solution only applies to applets stored in an intranet or destined to an intranet, because permissions defined as part of that solution only refer to local resources and to a local access control policy. It cannot provide a solution for defining the access control in a distributed way, as in the examples provided in the previous section. It should be noted in particular, that the suggested model still relies upon Java's centralized and single party security model.

In comparison, our work focuses on the definition of a truly distributed and multi-party access control model. The definition and resolution of access control attributes attached to Java mobile code are completely independent from the definition of permissions associated with target resources. Like Nikander and Partanen, we also presented a possible implementation using SPKI. However, unlike their solution, our design is not based on authorization certificates of

SPKI, but on the group mechanism of SPKI. In addition, in our design, SPKI certificates are used to store only attributes, not access control information, these attributes being only interpreted at attribute-policy resolution. Since the bulk of the access control information remains in the policy file as in Java 2, very few certificates need to be resolved when loading a mobile code component. This should be contrasted with the access control checking performed in Nikander and Partanen's work: it amounts to an SPKI chain reduction for every permission, performed each time the code requests access to a protected resource of the runtime environment.

[WBDF97] also proposed to integrate predefined sets of typical privileges to web browsers in order to help non-technical users. Our goal is similar, but we believe that even technical users, and especially network administrators, need new tools to cope with the wide-scale and pervasive deployment of mobile code. This is why our proposal puts the emphasis on the distributed and multi-party nature of the definition of the mobile code access control policy, which is a much broader concept than the grouping of access rights.

[AF99] argues that the information used for authentication might be specialized on demand for particular applications without the requirement for a special infrastructure for each new application. Based on this idea, specific intermediate elements might probably be encoded in a cleaner manner than with SPKI. However, the focus of our proposal is quite different from that of [AF99] since the latter does not address access control, but only how to introduce specialized attributes in authentication infrastructures.

Proof-Carrying Code (PCC) [Nec97,NL98] aims at verifying the safety of a mobile program with an original approach based on type checking. In this approach, the code consumer specifies a set of safety properties that should be met by the mobile code. The code producer demonstrates that its mobile code conforms to the properties explicitly indicated and bundles this proof together with the mobile program. When the runtime environment receives the mobile code and the proof, it decides whether the mobile program can be safely executed based on the verification of the proof. Checking that the proof is well formed gives the assurance that the program sent indeed corresponds to the program on which it has been proven.

Even though it relies on an approach fundamentally different from our solution, PCC shares some similarities with our proposal: safety properties are a kind of intermediate elements, on which an agreement must be reached between the code producer and the code consumer before establishing any proof. After verifying that the behavior is "safe", no safety checks are needed anymore. Although very promising, PCC has some drawbacks, the first being the basic difficulty of generating proofs. PCC also does not seem to address the distributed and multi-party access control issues discussed in this paper.

7 Conclusion

We proposed a new access control model for Java components addressing the problem of multi-party policy definition that has not been solved by the current model of Java. This new model is based on two key enhancements. Each mobile code component bears associated access control information or attributes. Intermediate elements are introduced in the access control schema to factorize access control policy definition. This model combines the current ACL-based approach with a capability scheme achieved through mobile code annotation with attributes to enable the description of dynamic multi-party systems while keeping the burden of security policy configuration at a minimum. We presented a possible design based on this model using SPKI certificates and the existing Java 2 run-time environment.

References

[Aba98] Martin Abadi. On SDSI's Linked Local Name Spaces. *Journal of Computer Security*, 6:3–21, 1998.

[AF99] Andrew Appel and Edward Felten. Proof-Carrying Authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security, Singapore*, November 1999.

[EFL+98a] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple Public Key Certificate, Internet Draft <`draft-ietf-spki-cert-structure-05.txt`>, March 1998.

[EFL+98b] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI Examples, Internet Draft <`draft-ietf-spki-cert-examples-01.txt`>, March 1998.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI Certificate Theory, RFC 2693, September 1999.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java^{TM} Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California*, December 1997.

[IT88] ITU-T. Recommendation X.509: The Directory - Authentication Framework, 1988.

[KG98] Lora Kassab and Steven Greenwald. Towards Formalizing the Java Security Architecture in JDK 1.2. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'98), Leuven-la-Neuve, Belgium*, LNCS. Springer, September 1998.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France*, January 1997.

[NL98] George C. Necula and Peter Lee. Safe, Untrusted Agents using Proof-Carrying Code. Number 1419 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[NP99] Pekka Nikander and Jonna Partanen. Distributed Policy Management for JDK 1.2. In *Proceedings of Network and Distributed System Security Symposium, San Diego, California*, February 1999.

[PN98] Jonna Partanen and Pekka Nikander. Adding SPKI Certificates to JDK 1.2. In *Proceedings of the Nordsec'98, the Third Nordic Workshop on Secure IT Systems, Trondheim, Norway*, November 1998.

[RL96] Ron Rivest and Butler Lampson. SDSI - A Simple Distributed Security Infrastructure. In *Proceedings of the 1996 Usenix Symposium*, 1996.

[Sun98] Sun Microsystems Inc. Sun. JDK 1.2 Security Documentation, `http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html`, April 1998.

[WBDF97] Dan Wallach, Dirk Balfanz, Drew Dean, and Edward Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating Systems Principles, Saint-Malo, France*, October 1997.