Distributed Policy Management for JDK 1.2

Pekka Nikander
pekka.nikander@ericsson.com
Ericsson Telecom Research[1]

Jonna Partanen
jonna.partanen@hut.fi
Helsinki University of Technology

Abstract

In JDK 1.2, the security architecture supports fine grained access control. In the default implementation, Java runtime modules (classes) are signed, and permissions are configured through a configuration file using the signer's identity and the loading location (URL) of the module. In a large network, the number of applets and the frequency of changes to the security policy will eventually grow very large. In a large organization, changing the configuration file in all Java enabled workstations and devices every time a need arises may be very hard.

In this paper, we describe a better scaling solution. We use authorization certificates to delegate permissions to Java modules. In JDK 1.2, the permissions are attached to the runtime modules through protection domains. In our implementation, each protection domain may be decorated with one or more SPKI certificates. These certificates directly describe the possible permissions of the domain. The actual permissions depend on the currently valid certificate chains leading to these certificates.

In addition to the certificates distributed with the modules, certificates for the chains may be retrieved from a distributed directory service. This approach makes it possible to fully distribute Java security policy management, allowing, among other things, security policy to be changed and new permissions types to be introduced without any modifications to the local configuration. Furthermore, the permissions need not be statically assigned but can be dynamically derived from the SPKI certificates as needed.

Our approach also enables further extensions. In particular, we propose how permissions could be delegated from a domain in one JVM to a domain in another JVM. This could eventually lead to a fully distributed secure Java execution environment.

[1] Pekka Nikander was at Helsinki University of Technology when most of this work was accomplished.

1. Introduction

The Java runtime environment (JRE) seems to be the first widely accepted architecture for mobile code. From the very beginning, Java has addressed the security concerns arising from executing code loaded from the untrusted network on a local computer, mainly assuring that malicious code cannot tamper with the local machine or network.

In the first two releases (1.0 and 1.1) the approach was simple: any untrusted code was placed in a confined environment, the sandbox, where its attempts to communicate with the external world were monitored and restricted. In JDK 1.0, all code loaded from the network was regarded as untrusted, and prohibited from performing any operations considered dangerous. These included, for example, accessing the local file system, opening network connections (to other machines but the one the code was loaded from), and accessing environment variables or Java properties that might reveal information about the local system or user. Java 1.1 enhanced this approach slightly by adding the notion of signed applets. Basically, in the Java 1.1 environment the local user could configure whether signed applets were considered trusted or untrusted. All untrusted code was still executed in the sandbox, equally restricted as before.

From an access control point of view, JDK 1.2 is a huge improvement. As we describe in more detail in Section 2, JDK 1.2 allows fine grained access control in the form of permissions. Whenever a Java class is loaded, it is associated with a number of permissions that represent the access rights the class has. Whenever a controlled resource is accessed, the runtime verifies that all classes in the method call stack have sufficient permissions for accessing that resource.

Unfortunately, the JDK 1.2 default implementation does not address the administrative needs of distributed systems. A configuration file is used to describe how permissions are granted to each class, based on the signature(s) the class file has and the location the class was loaded from. In practical terms, this means that the administrator of a local, distributed Java environment has to anticipate *beforehand* all possibly needed permission combinations, and to create corresponding signature keys and security configurations for them. If need arises to change these, the configuration files must be updated on *all affected machines.*

If we think about the suggested idea of using Java in various kinds of equipment, including embedded devices such as cell phones, PDAs and network routers, the concept of locally managing the Java security configuration in all devices will clearly create an administrative nightmare. Of course, it is possible, at least in theory, to remotely manage the security configurations in the same way as other configuration files are managed. In JDK 1.2, there is the possibility of defining the location of the configuration file as an URL, so the file could be fetched from a Web server. Remote management, however, requires secure management connections, which in a pure Java environment will probably be controlled by the local security configuration files, i.e., the very files the manager wants to modify.

The rest of this paper is organized as follows. In the remainder of this section, we briefly introduce the concept of authorization certificates in general, and SPKI in particular. In the next section, we describe the relevant details of the basic JDK 1.2 security architecture in order to be able to show where our modifications plug in. A more complete description is available in [8]. In Section 3, we discuss some weaknesses of the basic architecture and implementation, mainly from the management point of view, and outline our modification and customizations in conceptual terms. Section 4 describes our architecture in detail. Next, in Section 5, we describe the prototype implementation, and give initial performance measurements. In Section 6, we suggest a way of extending JDK 1.2 security domains across distributed Java Virtual Machine (JVM) environments with the help of SPKI certificates. Finally, in Section 7, we present our conclusions from this research.

1.1. Authorization certificates

Authorization certificates, or signed credentials, are signed statements of authorization, first independently described in the SDSI [16] and PolicyMaker [4] prototype systems and the SPKI initiative [5]. Some of the SDSI and PolicyMaker ideas are being merged to SPKI, which in turn is being standardized by the IETF as an alternative to the rigid X.509 based identity certificate hierarchy.

The basic idea of an authorization certificate is simple. In SPKI terms, a certificate is basically a signed five tuple (**I**,**S**,**D**,**A**,**V**) where

- **I** is the Issuer's (signers) public key, or a secure hash of the public key,
- **S** is the Subject of the certificate, typically a public key, a secure hash of a public key, or a secure hash of some other object such as a Java class,
- **D** is a Delegation bit,
- **A** is the Authorization field, describing the permissions or other information that the certificate's Issuer grants to or attests of the Subject,
- **V** is a Validation field, describing the conditions (such as a time range) under which the certificate can be considered valid.

The meaning of an SPKI certificate can be stated as follows:

Based on the assumption that **I** has the control over the rights or other information described in **A**, **I** grants **S** the rights/property **A** whenever **V** is valid. Furthermore, if **D** is true and **S** is a public key (or a hash of a public key), **S** may further delegate the rights **A** or any subset of them. [6]

Example. Let us consider a simple situation, where Alice wants to allow all applets signed by Bob to be able to access the local temporary directory, `/tmp`, on her local machine. Conceptually, this allowance could be represented by an SPKI certificate (K_{Alice}, K_{Bob}, Yes, (Java-Permission (File-Access "/tmp/*")), Always). Basically, this certificate states that Alice delegates Bob the right to authorize applets to access files in `/tmp`. To complete a certificate loop, two other certificates are needed. First, Bob must create a certificate for the applet in question: (K_{Bob}, hash(applet), No, (Java-Permission (File-Access "/tmp/*")), Always). Second, the local machine must have a local certificate that delegates a right to administer local Java permissions to Alice: ($K_{local\text{-}machine}$, K_{Alice}, Yes, (Java-Permission (All-Permission)), Always).

2. Basic security architecture in JDK 1.2

The JDK 1.2 security architecture contains two parts: an access control architecture and a number of cryptography related classes. Their integration is relatively loose. The components of the access control architecture are enumerated in Table 1 and discussed in more detail in Sections 2.1–2.4. Section 2.5 describes the relevant cryptographic classes.

2.1. Permissions

JDK 1.2 introduces a new type of classes, called Permissions, that are used inside the Java runtime environment to represent access rights to protected resources. Each protected resource in the system has a corresponding Permis-

Table 1: The parts of the JDK 1.2 access control Architecture

Class or classes	The role of the class or classes
`Permission` and its subclasses	Represent different “tickets” or access rights, i.e., permissions.
`ProtectionDomain`	Connects the Permission objects to executing classes.
`SecureClassLoader` and its subclasses	Load classes and create protection domains.
`Policy` and its subclasses	Decide what Permission objects each class gets.
`AccessController`	The reference monitor.

sion object. The Permission object can be seen as a capability or a “ticket” that grants access to the resource. Typically, there are many instances of a given Permission, possessed by and thus granting access to different classes.

Permissions are divided into several subtypes that extend the `Permission` class. Each resource type or category, such as files or network connections, has its own Permission subclass. Inside the category, different instances of the class correspond to different instances of the resource. In addition, the programmers may provide their own Permission subclasses if they create protected resources of their own.

Some permissions are more generic than others. For example, a single permission object may grant access to more than one instance of the controlled resource. Such a more generic permission instance implies a number of more restricted permissions. Thus, for example, the File–Permission(“/tmp/*”, “read,write”) object implies the FilePermission(“/tmp/foo.txt”, “read”) permission. Instances of the class AllPermission imply all other permissions.

2.2. ProtectionDomains

Just as in any capability-based access control system, the Java classes must be prevented from creating permissions for themselves and thus gaining unauthorized access. This is the task of ProtectionDomains.

Each class belongs to one and only one ProtectionDomain. Each ProtectionDomain has a PermissionCollection object that holds the permissions of that domain (see Figure 1). Only these permissions can be used to gain access to resources. The classes cannot change their ProtectionDomain nor the PermissionCollection of the domain. Thus, the classes are free to create any permission objects they like, but they cannot affect the access control decisions and gain unauthorized access.

In the current JDK 1.2 implementation the protection domain of a class is uniquely identified by the Code–Source of the class. A CodeSource consists of the codebase or URL that the class was loaded from, and a set of cryptographic certificates that indicate the signatures the class has. The classes are placed in the protection domains corresponding to their CodeSources. If a class is not signed, or if the signature cannot be verified, the class is placed in a protection domain that has an empty set of certificates.

All classes in the same protection domain get the same permission objects. However, classes with identical permissions may belong to different protection domains, because many protection domains may happen to have been granted a similar set of permissions by the current security policy.

2.3. AccessController

The AccessController is the JDK 1.2 incarnation of the reference monitor concept [1]. That is, when a thread requests access to a protected resource such as a file, the AccessController object is asked whether the access is granted or not. To determine this, the AccessController checks the execution context to see if the caller has the Permission object corresponding to the resource. For example, if a class tries to read the file /home/jhp/myfile, its protection domain must have the FilePermission(“/home/jhp/myfile”, “read”), or some other permission that implies this permission.

Asking for an access that requires a specific permission may be made by a method that was called from another

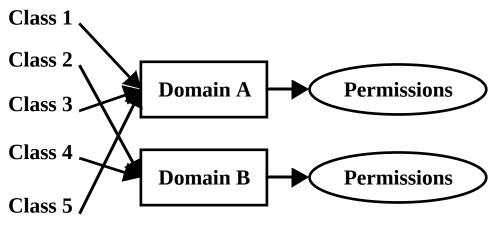

Figure 1: Classes, domains and permissions

class. This class may belong to a different protection domain. Since it is important that a class does not bypass the access control simply by calling another class with more permissions, the AccessController also checks all the previous classes in the call chain. The general algorithm is that if class A calls class B, which in turn calls class C and so on, and finally class M tries to read a file, then the AccessController checks each class from M to A to see if they all have the required permission. If some class in the call chain does not have the permission, AccessController throws an exception. Otherwise it returns quietly, implicating that the request has been accepted.

There is one irregularity to the general access control algorithm. A class may ask the AccessController to mark it as "privileged" while performing a task. This marking creates an artificial bottom to the call stack. When the AccessController reaches a class marked privileged, it checks whether this class has the permission in question and then stops. The preceding callers are not checked.

To further ensure that the access control cannot be bypassed, any thread inherits its parent's access control context. The AccessControlContext object contains all information relevant to making access control decisions.

2.4. Policy

A security policy defines the rules that mandate which actions the actors in the system are allowed or disallowed to do [1]. Java security policy, implemented as a subclass of the class Policy, defines what permissions each protection domain gets. There is a clear separation of duties between the AccessController and a Policy object: the Policy defines the rules and the AccessController enforces them. In other words, the Policy gives you the tickets and AccessController checks them at the gate. This means that we can change the policy according to which we distribute the permissions, without having to touch the AccessController.

A security policy can be static or dynamic. A static security policy is fixed: the permissions of a class cannot change once it is loaded to the JRE. However, the permissions can be different in the next time the class is loaded, during another run of the JRE. Having a static security policy has some performance advantages. On the other hand, if the runtime session is long, the circumstances may change so much that a change in the security policy is needed. Further more, even if the sessions are short, a change in the policy may be so important that it must take effect immediately. Thus, dynamic security policy that can be changed "on the fly" is the preferred solution because it provides better security. However, a dynamic policy requires some means for performing a set of actions in an atomic manner in order to prevent the system from entering an inconsistent state in case the permissions of a class change in mid-action and it is not able to complete the task it has begun.

The security policy in the current JDK 1.2 implementation is semi-static. That is, it does have a `refresh()` method, but it must be called explicitly and it only affects the permissions granted after the method was called. The protection domains that have been granted their permissions prior to the refresh still have the same permissions after it.

The class `Policy` is an abstract class. The actual implementation, which can be changed, defines how the security policy is managed. The default policy implementation of JDK 1.2 uses a set of configuration files to define the security policy.

There is one configuration file for defining a system-wide security policy. Each user may additionally have their own policy file. All the definitions are additive, so permissions can only be granted, not taken away. If the policy files do not exist or their format is incorrect, the classes end up in the sandbox.

The policy configuration file is clearly a kind of an access control list (ACL). As all ACLs, it has the disadvantage that it must be maintained locally, i.e., the access right management cannot be easily distributed while still preserving security. If we want to make this management easier to distribute, changing the configuration files with a capability-based policy definition looks like a promising approach.

2.5. Keys, certificates and certificate management

As mentioned above, the classes are placed in the protection domains according to where they have been loaded from, and what keys they have been signed with. To be able to sign classes and verify the resulting signatures, Java includes a basic set of cryptographic functionality. The concepts of cryptographic keys, digital signatures and certificates are a central part of this functionality. The keys are used as input to the signature functions, and the certificates are used for telling the verifier the key that can be used to verify the signature, and whom the key belongs to.

The `Certificate` interface, which is the Java representation of certificates, has the following methods: equals, getEncoded, getPublicKey, getType, hashValue, toString and verify. Although the interface was designed to be a superclass for identity certificates, with little imagination it is generalizable to authorization certificates as well [14].

JDK 1.2 has general interfaces for public key cryptography, including Key, PublicKey, PrivateKey, KeyPair and KeyPairGenerator. The KeyFactory takes care of converting keys to raw key material, called KeySpec, and vice versa. There are also more specific interfaces for RSA and DSA keys and their handling. The runtime can have several providers of classes that implement the interfaces. Key and certificate management in Java is handled by a KeyStore class that stores keys and the corresponding certificates.

3. Shortcomings and remedies

While the JDK 1.2 access control system provides fine granularity and flexible configuration facilities, its default implementation has a number of weaknesses that diminish its power in practical deployment in a distributed system. First, the permissions associated with each domain must be defined through a (usually local) configuration file prior to loading the classes to the runtime environment. Second, the way classes are divided in security domains is somewhat rigid and arbitrary. The former property is more significant, as it prohibits, among other things, dynamic creation of new permission types. Furthermore, when the number of keys controlling domains grows large, the complexity of the configuration file may become hard to manage. And finally, as mentioned in Section 2.4, the current default implementation is static in the sense that the permissions of a domain do not necessarily reflect changes in the policy file.

Fortunately, these problems are mainly due to the default, one-machine oriented implementation, not the access control architecture itself. This has allowed us to make our customizations with almost no changes to the JDK 1.2 source code.

We will next discuss the above mentioned shortcomings in detail, and show how they can be solved by using authorization certificates.

3.1. Alternatives to local configuration

The basic idea behind JDK 1.2 access control can be summarized as follows:

1. *All executable code, i.e., classes, is divided into security domains. Each class belongs to one, and only one domain.*
2. *Each security domain is assigned permissions.*
3. *The intersection of permissions present in the current method call stack (down to and including the permissions of the current thread with its inherited access control context, or the upmost privileged class) define the operations this method is allowed to perform.*

The problem of local configuration pertains mainly to step 2 (and to some extent also to step 1; this issue is discussed in Section 3.2).

As already described in Sections 2.2 and 2.4, the default implementation of the Policy object in JDK 1.2 runtime environment reads the permissions from a (usually local) security configuration file. This means, among other things, that whenever the user wants to create *a new permission*, to create a *new combination of existing permissions*, to assign permissions to *a newly created domain*, or to *remove permissions from a domain* over which the local organization has no direct control, the user has to edit the security configuration file.

If we think about large scale Java deployment, such as using large numbers of Java terminals within a multinational enterprise, or using Java in embedded devices such as cell phones or PDAs, changing the configuration separately in each device is either impractical or too expensive in practice. Clearly, alternative means are needed.

An obvious, but less-than-optimal solution is to place the configuration file in a directory that is shared, e.g., through NFS, or to use some kind of distributed database or a remote configuration mechanism such as Sun Microsystems Network Information Service (NIS). Optimally, such a mechanism provides adequate protection for the security configuration data through, e.g., preassigned shared keys and shared key cryptography. In such a case it is enough to configure the administrative security keys to the device when it is taken into use. Thereafter the security configuration files of the device can be remotely administered in a secure way, *provided* that the security of the administration system persists.

The default implementation of JDK 1.2 proposes solving this problem by specifying the file location as an URL, and thus fetching the file from a suitable Web server. As HTTP and FTP protocols do not provide any security, TLS or some other method for securing the connection between the host and the server would be necessary to ensure the integrity of the configuration information.

However, even this scheme has a number of problems:

- The security of the Java runtime inherently depends on the security of another, external mechanism. Thus, effectively, the correctness of access permissions assigned to a class depend on two cryptosystems: the signature system used to sign the classes, and the remote administration system used to manage the security files. If either of these is broken, Java security breaks.
- Keeping the configuration files of all Java devices up to date would be hard or impossible. If any of the devices are off-line while a change is made, arrangements would be needed to take care of the devices immediately when they come back on-line. This would be difficult or impossible in Ad-Hoc networks.

In our system, each collection of executable classes (i.e. a jar file) is a self contained domain that carries its own (potential) permissions. That is, each class is placed in a jar file, and the jar file is decorated with one or more SPKI certificates[1]. Each SPKI certificate denotes a number of permissions that the issuer of the certificate wants to assign to the domain. The local security system checks the validity of these certificates, and based on the certificate sequences leading to them, decides which of the permissions are actually assigned to the domain (see Section 4 for details).

3.2. Protection domains

Currently, the main purpose of the protection domains is to divide the classes into groups so that each group can be given distinct permissions. From the access control point of view, this is fine. However, as we will show in Section 6, it would be nice if protection domains could be used for other purposes as well.

In the current JDK 1.2 implementation, classes are divided into protection domains somewhat arbitrarily based on the URL they were loaded from and the X.509 certificates they carry. To us, using URLs seems like a bad choice from a security point of view. An URL consists of a DNS name and an arbitrary string. Until secure DNS is deployed (if ever), DNS names cannot be trusted for security purposes. Therefore, from a security point of view, the URL must be regarded as an arbitrary string that has no security relevance. Nevertheless, from a practical point of view, the usage of URLs may be a reasonable temporary solution until widely deployed PKIs exist.

Signing the code, and using signatures as basis of domain creation, is definitely a better idea. However, the currently used X.509 certificates do not carry any explicit information about why the class was signed, or what kind of permissions the class would indeed need in order to perform its function. The local configurator must get this information through some external channel in order to be able to set up the local policy correctly. That is, the current system leaves two decisions to the local administrator:

- Guessing what permissions a class would need in order to function correctly, and
- Deciding whether the signer is trustworthy enough so that the class can indeed be given the alluded permissions.

Again, as we shall see, using SPKI simplifies this situation. First, the certificate issued by the class writer clearly denotes what permissions the class would desire. Second, SPKI certificates can be used to represent trust and delegate trust decisions, lifting most of the burden of making trust decisions from the local administrator.

[1] At least in theory, we could use X.509v3 certificates or some other form of authorization certificates instead of SPKI certificates, but we have chosen to limit our research to the latter.

3.3. Scalability

Recent history has shown on many occasions that local configuration scales badly to the global Internet. Instead, a system that has been designed to be fully distributed, i.e., both deployed and managed in a distributed way, scales extremely well. A prime example of this is the Domain Name System (DNS): it was taken into use when the static hosts file grew too large to manage, and technically it has not needed any major modifications ever since.

From this point of view, the JDK 1.2 local security configuration file resembles the static hosts file. It will probably serve well in a small network where there are only relatively few trusted applets. However, as the need and usage of somewhat trusted Java code grows, a system that scales better is required.

According to our initial analysis, the suggested SPKI based system of signed capabilities scales extremely well. SPKI allows rights to be delegated, allowing administration to be distributed within organizations and between organizations. [10]

3.4. Pseudostatic vs. dynamic permissions

In the current JDK 1.2 implementation, the permissions assigned to a class are not amended unless the Policy.refresh() method is explicitly called. Furthermore, once assigned, permissions cannot be revoked from a domain in any practical way. When Java is being used in servers, and especially if the architecture is extended so that Java servlets can be delegated more permissions by clients (see Section 6), there arises a need to be able to revise the permissions dynamically.

Independently of the other modifications, we have also made the permission evaluation more dynamic. This is explained in Section 5. As mentioned in Section 2.4, a dynamic policy may create problems if the permissions of a class change while it is performing a set of actions that should be considered as a whole, i.e., that should be performed completely or not at all. For the sake of this study we have assumed that a mechanism for allowing atomic actions can be added to the AccessController in a relatively straightforward manner, following the example set by the doPrivileged-method. We have not, however, implemented this functionality in our prototype.

4. Assigning Java permissions with SPKI certificates

In JDK 1.2, the actual implementation of the access control mechanism is divided between the class loader, the policy manager, and the reference monitor. The purpose of the class loader is to make sure the classes are integral, at least in some sense, and to divide them into security domains. The policy manager, in turn, assigns permissions to the domains, while the reference monitor checks that an attempt to access a resource is indeed authorized.

In our model, the tasks of the class loader are simple. It loads classes from a jar file, and creates a domain from it. If there are any SPKI certificates present in the jar file, they are associated with the new domain. The policy manager and the dynamic permission evaluation are more complex.

4.1. Policy manager

The main task of the policy manager is to attempt to reduce a set of certificates to form a valid chain from its own key, called the Self key, to the hash of the protection domain, and to interpret the authorization given by the chain into Java Permission objects. This chain reduction includes checking the validity of the certificates, checking that all but the last certificate have the delegation bit set, and intersecting the authorization fields to get the final authorization given by the chain. Furthermore, usually more certificates must be fetched from a certificate store in order to get complete chains [13]. If the certificates cannot be reduced or the authorizations reduce to null, no permissions are granted to the class. [10]

The authorization field, or the tag, of an SPKI certificate can be described as an s-expression: [5]

auth:: `(tag (*))` | `(tag` *tag-expr*`)`
tag-expr:: *simple-tag* | *tag-set* | *tag-string*
tag-set:: `(* set` *tag-expr**`)`

The form `(tag (*))` means unlimited authorization, i.e., all permissions. When translated to Java permissions, it becomes java.security.AllPermission.

We have extended the SPKI tag definition to express Java permissions as follows: [15]

simple-tag:: *java-tag*
java-tag::
    `(java-permission` *type target? action?*`)`
type:: `(type` *bytes*`)`
target:: `(target` *bytes*`)`
action:: `(action` *bytes*`)`

That is, the tag specifies that it consists of a Java Permission. The *type* gives the full class name of the permission class in question. This may be a permission type included in JDK or any other class, as long as it is a subclass of the class java.security.Permission. If the constructor of the permission specified by the type takes a target as an argument, that string is given in the *target* field of the tag. Likewise, if the constructor of the permission takes an action as an argument, it is given in the *action* field of the tag. The target and action strings are passed to the constructor as-is, because we cannot expect the policy manager to be able to parse the arguments of all kinds of permissions, as any programmer can define her own types of permissions.

The *tag-set* can be used to pass several permissions in one certificate. This possibility is important, as creating a new certificate for each permission that one wants to delegate would be all too tedious and rapidly explode the number of certificates.

4.2. Dynamic policy

To make the security policy dynamic instead of static or semi-static, our implementation of protection domains no longer has a static set of permissions. When a class tries to access a protected resource the reference monitor asks the protection domain whether it implies the specific permission required, and the protection domain in turn asks the Policy for the permission. The Policy tries to produce a certificate chain reduction that would imply the permission in question. If it fails, the access is not granted.

The SPKI drafts propose that the Prover (i.e. the class) is responsible of presenting a valid certificate chain to the Verifier (i.e. the Policy) at the time of access request or authentication [5]. This approach effectively moves the burden of certificate storage, retrieval and part of the chain reduction from the server to the client software. The server is only left to verify that the chain presented is a valid one. This approach may be suitable to controlling user access, since the user is likely to know which certificates it has been issued and may even be able to store these certificates on a smart card or in some other practical way.

However, *mobile code downloaded from the Web cannot know if it has been issued local certificates or not*, and it certainly cannot possess all these certificates from each site that might want to use it. Thus, this approach is doomed to fail in our architecture and we do not pursue it any further. Instead, we think that the Policy needs to locate the relevant certificates as well as to reduce the certificate chains.

Many different solutions have been proposed to the certificate storage. We have presented one possibility in [13], suggesting storing the certificates in the DNS directory. Furthermore, Aura has analysed several different algorithms for chain reduction [3].

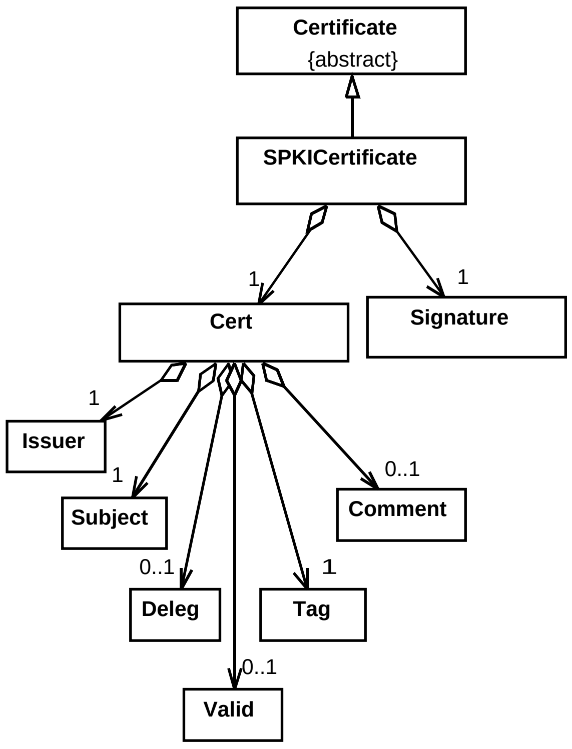

Figure 2: SPKI certificate object structure

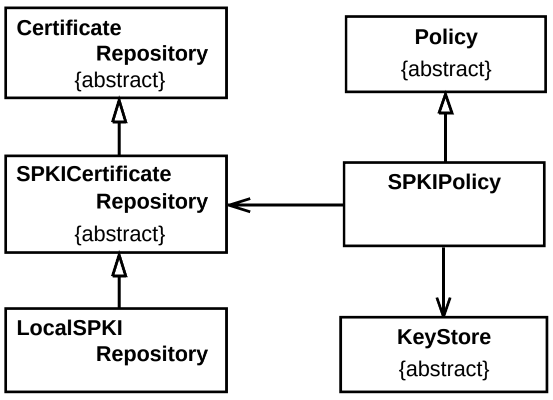

Figure 3: The Policy and certificate repositories

5. Implementation

A number of changes to the Java classes are required in order to allow the administrator to define the Java security policy using SPKI certificates instead of the configuration file. More specifically, we need to change the way the Policy object and the protection domains behave. In addition, we need to create a Java implementation of the SPKI certificates, and a way to store them so that they can be retrieved easily.

The way we implemented the SPKI certificates is depicted in Figure 2. The in-memory representation of the certificate consists of the certificate data and the signature, represented as Java objects. The data in turn includes the issuer, subject and authorization (tag) objects, and may include delegation, validity and comment objects.

Our implementation of the Java Policy object is called `SPKIPolicy`. It gives the protection domains exactly those permissions that are delegated to the domains through valid SPKI certificate chains. A valid chain must start from the Self key. The authorizations given by the certificates are transformed to Java permissions according to the principles given in Section 4.1.

The prototype uses a simple depth first algorithm to find valid certificate chains. Although not optimal for performance, this algorithm is good enough for our prototype; the number of certificates in our database is relatively small. The chain reduction is simple: two certificates form a valid piece of a chain if they are both valid, the first certificate has delegation set to true and the subject of the first certificate is the same as the issuer of the second certificate. The authorization that results from such a chain is the intersection of the two authorization fields. The authorization fields are converted into Permission objects, and `imply()` method is used to intersect the authorization fields. The subset is found by checking if either one of the permissions implies the other. This is sufficient for now, but a more generic method is clearly needed. However, this would require significant modifications to the JDK 1.2 library.

The `SPKIPolicy` uses the Java KeyStore to store its public key, i.e., the Self key for the SPKI certificate chain validation. A separate certificate repository is used to store the certificates. In the prototype, the certificate repository was implemented using a local file (see Figure 3). However, in the future we expect it to use DNS or some other dynamic, distributed directory service.

To implement a dynamic security policy instead of a static one we needed to change the way the protection domains behave. In the default implementation the protection domains get their permissions when they are initialized. We created a subclass of the class PermissionCollection, called DynamicPermissions, that does not have a static set of permission objects at all. An instance of this class is given to the domain instead of a regular PermissionCollection object (see Figure 4). Now, every time the AccessController checks whether the protection domain's PermissionCollection implies a certain permission, the collection asks the Policy object to give it the permission. The check succeeds or fails depending on what the Policy returns.

To make the Java Runtime read SPKI certificates from the jar files and put them to the protection domains we had to create a class of our own that handles the SPKI file veri-

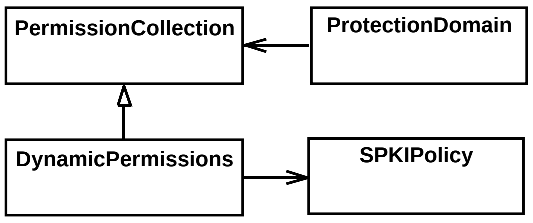

Figure 4: The ProtectionDomain and PermissionCollections

fication. In addition, we had to slightly modify the `java.util.jar.JarVerifier` to make it invoke our SPKI verifier.

The system security properties file `lib/security/java.security` contains several configuration variables for the security architecture, including the policy configuration file locations. A property called `policy.provider` can be used to change the default Policy implementation. This is done by specifying the fully qualified class name of the new implementation in the property:

```
policy.provider=fi.hut.tcm.\
  spki.policy.SPKIPolicy
```

5.1. Performance measurements

As noted in Section 4, a static security policy obviously has some performance advantages when compared to dynamically resolving the permissions. We measured the performance of our prototype and compared it to the performance of the default JDK implementation to see if the difference was unacceptable. Since the main performance changes to the default JDK implementation occur in class loading and permission checking, these two functions are the ones we measured.

Originally, we expected the class loading to get slightly slower or stay the same, as we would not need to figure out what permissions new protection domains would get, but would instead need to resolve the certificates from the jar files. Since class loading is fairly well optimised in the JDK, it was also possible that no change in the performance would be noticed. As to the permission checking, we expected the access right check to be slower, since we not only verify whether the class' permissions imply the permission needed, but also resolve what permissions the class has at the moment.

The actual measurements were made with JDK 1.2 beta 4 in Solaris 2.6 running on Ultra 1 hardware. The results are averages from 10 test runs. We expressed the same security policy in the form of a configuration file and SPKI certificates. The average length of an SPKI certificate chain was 3. The results are given in Table 2.

Table 2: Preliminary performance measurements

	JDK	Our Prototype
Time to load 10 classes (in 10 different domains)	1690 ms	4990 ms
Time to resolve 10000 access rights	38900 ms	39200 ms

Our prototype is not optimised in any means; it does some unnecessary work. At the moment it handles the SPKI certificates of the classes to be loaded *in addition* to the regular signatures and not *instead* of them, although the regular signatures are not used for anything in our system. The results show that our system is about three times slower in class loading and only slightly slower in access checking.

When analysing in more detail where the time is spent during the class loading, about 80% of the JDK loading time seems to be spent on checking the X.509 certificates. In our prototype, the time used in checking SPKI certificates is roughly equal to the time spent in X.509 certificate checking. Thus, this explains only 40% of the increased loading time. Currently we cannot fully explain the other part of the increase; it *seems* to be spent at the Sun provided JAR file handling routines. Unfortunately, the JDK distribution does not include source code for these.

Thus, when the time spent on checking X.509 certificates is substracted from the total time, our prototype is about 2.2 times slower than the default implementation in class loading. Less than half of this time is spent checking the SPKI certificates. Once we understand better the reasons for the degradation, it should be possible to get performance quite close to the default implementation.

6. Creating distributed protection domains

The dynamic and distributed nature of SPKI based Java protection domains opens up new possibilities for their use. In particular, we would like to be able to perform the following functions:

- Dynamically delegate a permission from one domain, executing in one Java virtual machine, to another domain, executing in another Java virtual machine. For example, when a distributed application requests a service from a server, it might want to allow a certain class, an agent, in the server, to execute as if it were the user that started the application in the first hand.
- Create a secure connection between domains executing in distinct Java virtual machines. For example, a banking applet might want to create a secure connection back to the bank, using a proprietary security protocol.

In order to be able to perform these kinds of functions, the domains involved must have local access to some private keys, and a number of trust conditions must be met. The requirement of access to a private key can be easily accomplished by creating a temporary pair of keys for each policy domain. This is acceptable from a security point of view, because the underlying JVM must be trusted anyway, and so it can be trusted to provide temporary keys as well. The temporary key can be signed by the local machine key, denoting it to as belonging the domain involved.

Delegation. Let us now consider the trust requirements of the delegation. The situation here is that Alice has loaded some Java code C to perform a function X that she wants to be performed. However, X cannot be accomplished locally, but it must be performed on a server administered by Bob using Java code S.

From Alice' point of view the trust requirements are the following:

- Alice must trust C and S to be able to perform X on her behalf, independent on their execution location.
- Alice must trust Bob to execute S on her behalf.
- Finally, as a result, Alice must trust S, when run by Bob, to perform X.

Using SPKI certificates, these can be expressed roughly as follows:

Cert$_{C}$: (K$_{Alice}$, hash(C), Yes, X, always)
Cert$_{S}$: (K$_{Alice}$, hash(S), Yes, X, always)
Cert$_{Bob}$: (K$_{Alice}$, K$_{Bob}$, Yes, execute S, always)

Now, the fact that C has a local, temporary key K$_{C}$ and that S has a local, temporary key K$_{S}$ can be expressed as

Name$_{C}$: (K$_{Alice}$, K$_{C}$, Yes, hash(C) at K$_{Alice}$, now)
Name$_{S}$: (K$_{Bob}$, K$_{S}$, Yes, hash(S) at K$_{Bob}$, now)

These certificates can be considered as name certificates, effectively late binding the hashes of C and S, as names in the local namespaces of Alice and Bob, respectively, to the temporary keys K$_{C}$ and K$_{S}$.

Given these, C can check Cert$_{Bob}$ and Name$_{S}$, and thereafter authorize S to perform X

Auth: (K$_{C}$, K$_{S}$, Yes, X, now)

The fact that Alice authorizes S on Bob to perform X can be depicted through the following sequence:

$$\begin{tikzcd}
K_{Alice} \arrow[r, thick, "X \nearrow \mathrm{hash}(C)"] & K_C \arrow[r, "X"] & K_S
\end{tikzcd}$$

Similarly, the checks performed by C before creating Auth can be described as the sequence:

$$\begin{tikzcd}
K_{Alice} \arrow[d] \arrow[dr, "X"] & \\
K_{Bob} \arrow[r, thick, "\mathrm{hash}(S)"] & K_S
\end{tikzcd}$$

From Bob's point of view, on the other hand, the requirements are the following:

- Bob must trust S to perform X on Alice' (or everybody's) behalf.

Again, using SPKI certificates this can be expressed as

Cert$_{Alice}$: (K$_{Bob}$, K$_{Alice}$, Yes, X, always)

Now, given the certificates created, Bob can check that S is permitted to perform X:

$$\begin{tikzcd}
 & & K_S & \\
K_{Bob} \arrow[urr, "\mathrm{hash}(S)"] \arrow[dr] & & & K_C \arrow[ul] \\
 & K_{Alice} \arrow[urr, thick, "\mathrm{hash}(C)"] & &
\end{tikzcd}$$

Secure connection. In the case of a secure connection, Alice wants to allow a class C to open a secure connection to a class S, being run by Bob. Respectively, Bob wants to allow the class C, being run by Alice, to open a secure connection to the class S, running locally.

From Alice' point of view, the trust requirements can be stated as follows:

- Alice must trust C to open secure connections to S.
- Alice must trust Bob to be trustworthy to run S.

Similarly, from Bob's point of view,

- Bob must trust S to accept secure connections from C.
- Bob must trust Alice to be trustworthy to run C.

In a way analogous to the delegation case, temporary keys can be created for the classes C and S, and using suitable SPKI certificates these keys can be seen as proper keys to be used in a key agreement protocol.

7. Conclusions

We have shown how JDK 1.2 access control management can be effectively and securely distributed using SPKI certificates. The new systems allows new permission types to be taken into use dynamically, allows the creator of an application to control the division of Java classes into distinct security domains in a natural way, provides worldwide interorganizational scalability, and allows the permissions of a domain to be dynamically extended.

In Section 3 we analysed the default implementation of the JDK 1.2 access control architecture and suggested some improvements. In Section 4 we described the functional details and modifications needed to implement the improvements. Only one change was needed in the actual library in order to load SPKI certificates in addition to X.509 certificates. The rest of the system consists of the policy manager and a new type of PermissionCollection. The result is a dynamic security policy defined with SPKI certificates. A distributed directory service, such as the one proposed in [13], is needed for storing the certificates.

The actual prototype implementation, described in Section 5, consists of a generic SPKI certificate package that extends the `java.security.cert.Certificate` interface, the custom policy manager, and the minor modifications needed in the library proper. For the purpose of this prototype we only implemented a local certificate repository. Although the prototype is not optimised in any way, its performance was clearly adequate, especially in the permission checking.

Furthermore, we sketched how the new system can be used to delegate permissions dynamically from one Java virtual machine to another, and how SPKI certificates can be used to control the creation of secure connections between classes in separate virtual machines. These can be seen as initial steps towards a secure distributed Java environment. Currently we are building an ISAKMP [11] framework in Java. That will be used to implement the sketched delegation systems. One further possibility would be to design CORBA like security services for interoperating Java virtual machines on the top of the resulting system.

References

[1] E. Amoroso, *Fundamentals of Computer Security Technology*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[2] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[3] T. Aura, “Comparison of Graph-Search Algorithms for Authorization Verification in Delegation”, *Proceedings of the 2nd Nordic Workshop on Secure Computer Systems,* Helsinki, 1997.

[4] M. Blaze, J. Feigmenbaum, and J. Lacy, “Decentralized trust management”, *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy,* Oakland, CA, May 1996.

[5] C. M. Ellison, B. Frantz, B. Lampson, R.Rivest, B. M. Thomas and T. Ylönen, *Simple Public Key Certificate*, Internet-Draft `draft-ietf-spki-cert-structure-05.txt` work in progress, Internet Engineering Task Force, March 1998.

[6] C. M. Ellison, B. Frantz, B. Lampson, R.Rivest, B. M. Thomas and T. Ylönen, *SPKI Certificate Theory*, Internet-Draft `draft-ietf-spki-cert-theory-02.txt` work in progress, Internet Engineering Task Force, March 1998.

[7] C. M. Ellison, B. Frantz, B. Lampson, R.Rivest, B. M. Thomas and T. Ylönen, *SPKI Examples,* Internet-Draft `draft-ietf-spki-cert-examples-01.txt` work in progress, Internet Engineering Task Force, March 1998.

[8] Li Gong, *Java™ Security Architecture (JDK 1.2),* DRAFT DOCUMENT (Revision 0.8), `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html` Sun Microsystems, March 1998.

[9] Li Gong and R. Schemers, “Implementing Protection Domains in the Java Development Kit 1.2”, *Proceedings of the 1998 Network and Distributed System Security Symposium*, San Diego, CA, March 11–13 1998, Internet Society, Reston, VA, March 1998.

[10] I. Lehti and P. Nikander, “Certifying trust”, *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) ’98*, Yokohama, Japan, Springer-Verlag, February 1998.

[11] D. Maughan, M. Schertler, M. Schneider and J. Turner, *Internet Security Association and Key Management Protocol (ISAKMP),* Internet-Draft `draft-ietf-ipsec-isakmp-10.txt`, work in progress, Internet Engineering Task Force, July 1998.

[12] P. Nikander and A. Karila, “A Java Beans Component Architecture for Cryptographic Protocols”, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, Usenix Association, 26-29 January 1998.

[13] P. Nikander and L. Viljanen, “*Storing and Retrieving Internet Certificates*”, *Proceedings of the 3rd Nordic Workshop on Secure Computer Systems*, Trondheim, Norway, November 1998.

[14] J. Partanen and P. Nikander, “Adding SPKI certificates to JDK 1.2", *Proceedings of the 3rd Nordic Workshop on Secure Computer Systems*, Trondheim, Norway, November 1998.

[15] J. Partanen, *Using SPKI certificates for Access Control in Java 1.2,* Master’s Thesis, Helsinki University of Technology, August 1998.

[16] R. L. Rivest and B. Lampson, “SDSI — a simple distributed security infrastructure”, *Proceedings of the 1996 Usenix Security Symposium*, 1996.

[17] *ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, ITU-T, June 1997.