In Proceedings of the Internet Society Symp. on Network and Distributed System Security, San Diego, CA, March 1988

Implementing Protection Domains in the Java™ Development Kit 1.2

Li Gong and Roland Schemers

JavaSoft, Sun Microsystems, Inc.
{gong,schemers}@eng.sun.com

Abstract

The forthcoming Java™ Development Kit (JDK1.2) provides fine-grained access control via an easily configurable security policy. In this paper, we describe the design and implementation in JDK1.2 of the concept of protection domain*, which is a cornerstone of the new security architecture. We present design rationales, implementation details, and performance data, which demonstrate the utility and efficiency of the new security architecture.*

1 Introduction

The original Java security model [5, 7], known as the sandbox model, provides a very restricted environment in which to run untrusted code (called applet) obtained from the open network. Overall security is enforced through a number of mechanisms, including language type-safety, bytecode verification, runtime type checking, name space separation via class loading, and access control via a security manager.

The essence of the sandbox model is that local application is trusted to have full access to system resources while applet is not trusted and can access only the limited resources provided inside the sandbox. JDK1.1 introduced the concept of signed applet. A digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet.

The new security architecture in JDK1.2 provides fine-grained access control, easily configurable security policy, easily extensible access control structure, and extension of security checks to all Java programs, including applets as well as applications. In other words, there is no longer a built-in, hardwired notion of which code is trusted. In particular, not all signed code are equally trusted. Moreover, the original binary trust model is extended so that all code (remote or local, signed or not) runs under a security policy that potentially grants different permissions to different programs, and these sets of permissions may or may not overlap, as shown in Figure 1.[1]

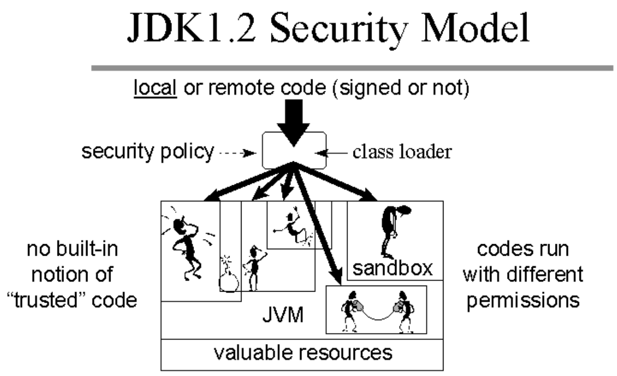

Figure 1: JDK1.2 Security Model

The concept of a protection domain [9] is a vital component of the new architecture in that it is the basis for making access control decisions. Moreover, its design and implementation must be efficient to meet the stringent standard required by certain applications.

In the rest of this paper, we first give a quick overview of the new security architecture in JDK1.2. Then we present details of algorithms, implementation, and performance data. We also explain how protection domain is utilized, and compare it with existing work.

2 Overview of Security Architecture

Looking from a high level, the new architecture is symbolized by security policy, access permission, run-

[1] For a broader overview of the architecture and its motivation, please refer to [3, 4].

ning code that is granted (some) permissions, and access control checking.

First, there is a system security policy, set by the user or by a system administrator, that is represented by a policy object, which is instantiated from the class `java.security.Policy`.

In abstract terms, the security policy is a mapping from a set of properties that characterize running code to a set of access permissions that is granted to the concerned code.

A piece of code is fully characterized by its origin (its location as specified by a URL) and the set of public keys that correspond to the set of private keys that have been used to sign the code using digital signature algorithms. Wild cards are used to denote "any location" or "unsigned".

Second, we introduced a new hierarchy of typed and parameterized access permissions. The root class is an abstract class `java.security.Permission`. Other permissions are subclassed either from the root class or one of its subclasses. Generally, a permission class belongs to the package in which it is mostly used. For example, the permission representing file system access, `FilePermission`, is found in the `java.io` package.

Typically, each permission consists of a target and an action, although either can be omitted. For file access, a target can be a directory or a file. The actions include `read`, `write`, `execute`, and `delete`. A file target can be expressed as `path`, `path/file`, `path/*` (which denotes all files and directories in the directory `path`), or `path/-` (which denotes all files and directories under the subtree of the file system starting at `path`).

We also created, inside the `java.security` package, abstract classes `PermissionCollection` and `java.security.Permissions`, where the former is a homogeneous collection of `Permission` objects and the latter is a heterogeneous collection of collections of `Permission` objects.

Finally, before a controlled resource is to be accessed, an access control decision is made, based on the permissions the executing code has. Note that anyone can create any number of permission objects, but holding such objects do not imply having gained the corresponding access rights. Only those permission objects that the Java runtime system manages signifies granted permissions.

To fully understand the new security architecture, it is critical to know what algorithms are used for granting permissions, how the permissions are stored internally to the Java runtime system, what algorithms are used for access control, and how such algorithms are implemented efficiently. We answer these questions in the rest of this paper.

3 Protection Domain

According to the classical definition of a protection domain [9], a domain is scoped by the set of objects that are currently directly accessible by a principal, where a principal is an entity in the computer system to which authorizations (and as a result, accountability) are granted. The Java sandbox in JDK1.0 is in a sense a protection domain with a fixed boundary.

In JDK1.2, permissions are granted to protection domains, and classes and objects belong to protection domains. This indirection, where permissions are not granted to classes and objects directly, is by design because, in the future, protection domains can be further characterized by user authentication and delegation so that the same code could obtain different permissions when running "on behalf of" different users or principals.

The new class `java.security.ProtectionDomain` is package-private, and is transparent to most Java developers.

3.1 Creating Domain and Its Permissions

In JDK1.2, protection domains are created "on demand", based on code source – the code base where the code originated from and the set of public keys that are used to sign the code.

For example, suppose the following security policy, given in its ASCII representation, is in force.

```
grant CodeBase
            ``http://java.sun.com/people/gong/'',
      SignedBy ``*'' {
      permission java.io.FilePermission
                 ``read,write'', ``/tmp/*'';
      permission java.net.SocketPermission
                 ``connect'', ``*.sun.com'';
};

grant CodeBase ``/home/gong/bin'',
      SignedBy ``self'' {
      permission java.io.FilePermission
          ``read,write,delete'', ``/home/gong/-'',
};
```

The first entry in this simple policy says that any applet that is loaded from the web page `http://java.sun.com/people/gong/`, whether signed or not, can read and write any file in the `/tmp` directory, and can connect to any host within the DNS domain sun.com.

The second entry says that any applet loaded from the local file system under the directory `/home/gong/bin`, signed by a key that is referenced through an alias `self`, can read and write any file in the home directory rooted at `/home/gong`.

When an applet class file in Java bytecode format, `http://java.sun.com/people/gong/demo.class`, is first loaded, the classloader consults the policy object (whose content is an internal representation of the policy earlier given in ASCII form), sees that a matching entry is found but there is no protection domain that corresponds to this set of permissions, and proceeds to create a new domain object. This new object has reference to the permissions granted to it.

When another applet that fits the same profile is loaded later, that applet will be granted the same set of permissions, and no new protection domain needs to be created. Note that if a third applet that does not fit the profile but ends up obtaining the same set of permissions, a new domain will be created.[2]

A policy entry is matched if *both* the code base *and* the signer(s) of the applet (or application) code match the entry. To be more precise, the actual code base of an applet matches a policy entry if the latter URL is a prefix of the former. The signer of an applet matches a policy entry if the public key used to verify the signature matches the key that corresponds to the alias given in the policy entry. (The mapping from an alias to an actual key or certificate is maintained, for example, via a local key store.)

If an applet matches multiple entries in the policy, the applet obtains all permissions granted in those entries. Specifically, for code signed with multiple signatures, permissions granted are also additive. For example, if code signed with key `a` gets permission `x` and code signed by key `b` gets permission `y`, then code signed by both `a` and `b` gets both permissions `x` and `y`.

Note that often a class may refer to an another class and thus cause the second class to be loaded that belongs to another domain. Typically the second class is loaded by the same classloader that loaded the first class, except when either class is a system class, in which case the system class is loaded with a null classloader [5]. The special `null` system classloader is a historical feature – system classes (i.e., those classes on CLASSPATH) are loaded via a mechanism implemented in C.

[2]The current implementation may create more domains than absolutely necessary. This does not compromise security, and further optimization should reduce unnecessary domain creation.

3.2 Mapping Objects to Domains

Each object or class belongs to one and only one domain. Otherwise, the security decision is hard to make. If the same class (represented by a piece of Java bytecode) is reused as part of two applets that belong to two different domains, and if the same classloader is loading both applets, then the class is defined only once and is shared by the two domains.

However, if two different classloaders are loading the applets, then effectively each domain will contain a distinct copy of the class, unless the class is a system class, in which case only one copy is defined because a system class is always defined by the `null` classloader. For example, suppose the same web page contains `a.class`, `b.class`, and `c.class`. Further assume that `a.class` and `b.class` are signed with different keys but both refer to `c.class`. In this case, two protection domains will be created to run `a.class` and `b.class` separately, and two copies of `c.class` will be defined.

The Java runtime maintains the mapping from code (classes and objects) to their protection domains and then to their permissions, as shown in Figure 2.

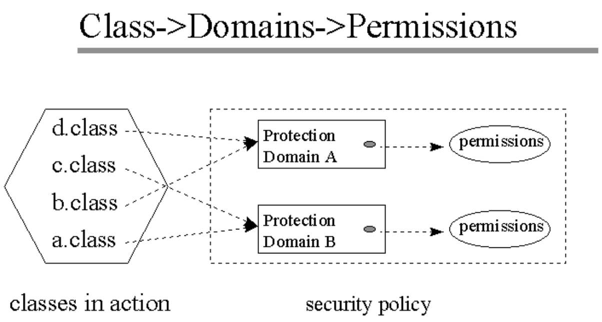

Figure 2: Mapping A Class to Its Protection Domain

This mapping from a class to its protection domain is first registered when the class is defined. This is done by calling a private method `setProtectionDomain(Class, ProtectionDomain)` on class `ProtectionDomain`. Obviously this method is security sensitive; in fact it is not publicly callable from outside the `java.security` package.

From within Java code, the protection domain of a given class is obtained by invoking the static method `getProtectionDomain(CodeSource)`, again a private method within the package. For each class, its protection domain is set only once, and will not change until the class object is garbage collected.

Once a domain is created, the permission set it is granted is made read only, so that further tampering is impossible. This is achieved by calling the `setReadOnly()` method defined in class `java.security.Permissions`.

3.3 System Domain

One protection domain is special: the system domain consists of system code that is loaded with a `null` classloader (i.e., everything on `CLASSPATH`) and that is given special privileges. It is common (but probably not wise) to use the `null` classloader to identify system code:

```
if (x.getClass().getClassLoader() == null) {
   (object x is an instance of system code)
}
```

It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, are only accessible via system code.

Prior to JDK1.2, all local applications had to be installed on `CLASSPATH` in order to be located and run. This arrangement mandates that any local application is treated as system code and by default enjoys special privileges.

To apply the same security policy to applications found on the local file system, in JDK1.2 we provide a new class path, which is specified as a Java property named `java.app.class.path`. For example, one can set the path as the following:

```
java.app.class.path=/usr/classes:/usr/bin
```

Non-system classes that are stored on the local file system should all be on this path, not on the `CLASSPATH`. We also provide a class called `java.security.Main`, which can be used in the following fashion in place of the traditional command `java application` to invoke a local application:

```
java java.security.Main your_application
```

This usage ensures that any local application on the `java.app.class.path` is loaded with a `SecureClassLoader` and therefore is subjected to the security policy (e.g., the second entry in the example policy we give earlier) that is being enforced. Separate protection domains are created for such local applications.

4 Access Control on Resources

The decision of granting access to controlled resources can only be made within the right context, which must provide answers to questions such as "who is requesting what, on whose behalf". In JDK1.2, the decision is based on the protection domains involved in any computation.

For example, when an access request is made, if code involved in the execution all belong to the same protection domain, then the decision is trivial. If the protection domain has been granted the said permission, the request should succeed. Otherwise, it must fail. However, in most cases things are not so straightforward and may involve multiple domains and various types of permissions.

In the rest of this section, we first describe how comparisons are made between permissions. We then give the algorithm for making access control decisions. We also report implementation details and performance data.

4.1 Comparing Permissions

The protection domain does not have to be granted the identical permission as requested, but only one or more permissions that imply the said permission. For example, if the protection domain is granted a permission to read all files in directory `/tmp`, then a request to read file `/tmp/README.txt` should succeed. Moreover, two separate permissions, one for "read" and one for "write", should imply a combined permission for "read,write".

Such implication reasoning is encapsulated in the implementation of the `implies(Permission)` method that is defined in the abstract base class `java.security.Permission` and must have a concrete implementation for each type of permission classes.

For example, the `java.io.FilePermission` class implements the `implies()` method such that it correctly interprets wild card for file names, whereas the `java.net.SocketPermission` class's `implies()` method understands IP addresses and DNS names.

The `implies(Permission)` method must be implemented carefully, as it can compromise security. For example, the method should always check that the permission it is checking against (the permission passed in as a parameter) must be of the same type as itself,[3] as in the following code fragment from `java.io.FilePermission` class:

```
public boolean implies(Permission p) {
    if (!(p instanceof FilePermission))
        return false;

    FilePermission fp = (FilePermission) p;
    ... (continue checking other things)
}
```

Suppose we have the following permissions:

```
FilePermission p1 =
    new FilePermission(``/tmp/*'', ``read'');
FilePermission p2 =
    new FilePermission(``/tmp/x.txt'', ``read'');
FilePermission p3 =
    new FilePermission(``/usr/bin'', ``read'');
SocketPermission p4 =
    new SocketPermission(``*'', ``accept'');
```

Then `p1.implies(p2)` is true, `p1.implies(p3)` is not true, and `p4.implies(p2)` is not true.

Note that although it may be convenient to have a catch-all permission that "implies" other permissions of different types, such super permissions, if ever created, must be handled very cautiously, as they are risky to give out.

When an access control permission is requested, the `AccessController.checkPermission()` method will go through all the permissions that the protection domain has, identify permission objects of the appropriate types, and then compare them with the said permission to see if the latter is implied by any of the former.

Moreover, when comparing two permissions `p1` (which is created as part of the access request) and `p2` (which is stored with the protection domain object), we always invoke checking in the form of `if (p2.implies(p1) == true)` and not `if (p1.implies(p2) == true)`, as the implementation of `p2` (which is controlled by system code) is not less dependable than that of `p1`, which is controlled by the calling application.

4.2 Access Control Algorithms

The right context for access control is a thread of computation, which may involve one or more Java threads, for example, when child threads are created. A thread of computation may occur within a single protection domain (i.e., all classes and objects involved in the thread belong to the identical protection domain) or may involve multiple domains.

For example, an application that reads a file via a `FileInputStream` is in effect interacting with the system domain that mediates access to the underlying file system. In this case, the application domain must not gain additional permissions by merely calling the system domain. Otherwise, security can be compromised.

As another example, suppose that a system domain invokes a method from an application domain, such as in a call-back situation, again the application domain must not gain additional permissions merely because it is being called by the system domain.

The above observation applies to a thread that traverses multiple protection domains. Therefore, for security reasons, a domain cannot gain additional permissions as a result of calling a more "powerful" domain, while a more powerful domain must lose its power when calling a less powerful domain.

Prior to JDK1.2, any code that performs an access control decision relies on explicitly knowing its caller's status (i.e., being system code or applet code). This is fragile in that it is often insufficiently secure to know only the caller's status but also the caller's caller's status and so on. Moreover, having only a broad distinction between system code and applet code limits us to supporting a binary, all-or-nothing access control policy. Finally, placing this discovery process explicitly on the typical programmer becomes a serious burden, and can be error-prone.

To relieve this burden, we automate the most common scenario for checking access in a new class `java.security.AccessController`. Instead of trying to discover the history of callers and their status within a thread, any code can query the access controller as to whether a permission would succeed if performed right now. This is done by calling the `checkPermission()` method of the `AccessController` class with a `Permission` object that represents the permission in question. Below is a small code example for checking file access.

```
FilePermission p =
    new FilePermission("path/file", "read");
AccessController.checkPermission(p);
```

The access controller will, as a default, return silently only if all callers in the thread current history belong to domains that have been

[3] This particular rule usually applies only when comparing individual permission objects. When comparing sets of permission objects of mixed types, similar precaution must be taken.

granted the said permission. Otherwise, it throws a `java.security.AccessControlException` that is a subclass of `java.lang.SecurityException`.

Note that the history of a thread includes not only all classes on the current call stack but also all classes in its parent thread when the current thread is created. This inheritance is transitive, and is natural in that a child thread is simply a continuation of the parent thread. If we do not enforce this inheritance, a child thread can unexpectedly gain additional access that is not permitted in its parent. This phenomenon would require a programmer be extra careful when creating threads, which is an undesirable burden.

The default behavior to grant a permission only when all protection domains involved have that permission, although the most secure, is limiting in some cases where a piece of code wants to temporarily exercise its own permissions that are not available directly to its callers. For example, an applet may not have direct access to certain system properties, but the system code servicing the applet may need to obtain these properties in order to complete its tasks. The default mode will prevent this from happening, as the system code called by the applet code would lose its system privileges.

For such exceptional cases, we provide an overriding primitive, in the `AccessController` class, in the form of static methods `beginPrivileged()` and `endPrivileged()`. By calling `beginPrivileged`, a piece of code is telling the Java runtime system to ignore the permissions of its callers and that it itself is taking responsibility in exercising its permissions. Note that the protection domains of the code that is subsequently called by the "privileged" code are still considered when deciding whether to grant permissions.

Looking from another angle, the permission of an execution thread is the intersection of the permissions of all protection domains traversed by the execution thread, except that, when a piece of code calls the `beginPrivileged()` primitive, the intersection is performed only on the protection domain of the privileged code and the domains of its subsequent callees. This is illustrated in Figure 3. The actual implementation of the access controller can take various shapes, but it must follow the above basic principle.

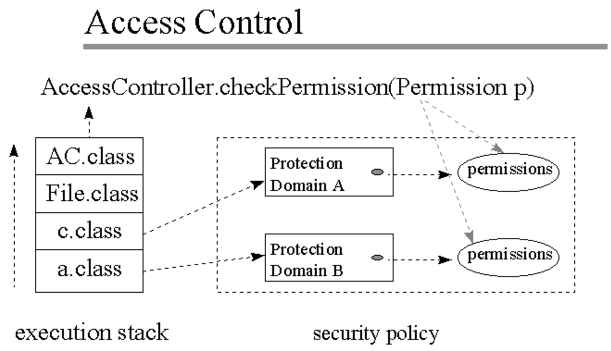

Figure 3: Making Access Control Decisions

4.3 Implementation of AccessController

The most important method in the `AccessController` class is `checkPermission()`, and there are at least two strategies for implementing it. In an "eager evaluation" implementation, whenever a thread enters a new protection domain or exits from one, the set of effective permissions is computed dynamically. The benefit is that checking whether a permission is allowed is simplified and can be faster in many cases. The disadvantage is that, because permission checking occurs much less frequently than cross-domain calls, a large percentage of permission updates will be useless effort. This observation is critical as every method call in Java can potentially be a cross-domain call, and computing the intersection of permissions can be expensive.

JDK1.2 uses "lazy evaluation". Whenever a permission checking is requested, the intersection is computed with regard to the thread current state as represented in the form or the equivalent of the current stack plus its parental history. In fact, the intersection of all permission sets is not literally implemented.

More precisely, suppose the thread traverses `m` callers belonging to `n` different protection domains, in the order of domain 1 to domain 2 to domain `n`. Domain `n` then invokes the `checkPermission()` method. The lazy evaluation implements the follow algorithm:

```
i = n;
while (i > 0) {
    if (domain i does not have permission)
        throw AccessControlException;
    else if (domain i is privileged) return;
    i = i - 1;
} return;
```

One potential downside of this approach is performance penalty at actual time of permission checking, although this penalty would have been incurred anyway in the "eager evaluation" approach, albeit at earlier times and spread out among each cross-domain call. Performance data will show later that lazy evaluation is efficient in practice.

4.4 More on Privileged Code

The correct usage of the privilege primitive is the following:

```
try {
    AccessController.beginPrivileged();
    some sensitive code
} finally {
    AccessController.endPrivileged();
}
```

The concept of being privileged is scoped by the thread within which the call to become privileged is made. The same protection domain in other threads are not affected. Also, the privilege primitive can be nested. Moreover, although the code inside try-finally is privileged, it will lose its privilege if it calls code that is less privileged.

Finally, although it is a good idea to use `beginPrivileged()` and `endPrivileged()` in pairs as this clearly scopes the privileged code, some programmers may call an extra or forget to call `endPrivileged()`. We have provided extra protection to reduce the risk in such a situation. For example, to mount an attack through a forgotten `endPrivileged()` call, the attacking code must be able to recreate the stack history, with the correct frame depth and frame number that is arbitrarily chosen by the Java virtual machine from a pool of available numbers.

4.5 Performance

Because earlier JDK implementations did examine stack context to look for the presence of applets, having the `AccessController` look over the stack history is not intrinsicly slower. The `AccessController.checkPermission()` method is implemented as follows (pseudo code).

```
void checkPermission(Permission perm)
        throws AccessControlException {
    getProtectionDomainContext();
    for each domain in context {
        get the permission_set of the domain;
        if (permission_set.implies(perm) == false)
            throw new AccessControlException();
    }
}
```

Note that an optimization technique, explained later, eliminates the need to deal with privileged code here. Also, the `implies()` method can be expensive because it goes through all permissions (granted to the domain) that are of the appropriate type and checks if any of those implies the requested permission.

We deployed a few optimization techniques in `getProtectionDomainContext()` to make access control go fast, mainly by reducing the size of the context as it impacts the for loop. Obviously some of these techniques need to be changed if the basic algorithm is changed.

- In the protection domain context that is returned to `AccessController`, we return only the domains up to (and include) the first privileged domain, because our algorithm says that if a privileged class belongs to a protection domain that has the requested permission, the check should go through; otherwise, the check should fail.[4]
- We eliminate system domains from the returned protection domain context, because the system domain by default has all permissions.
- We return only *distinct* domains. For example, if there are many classes that all belong to one protection domain, then the returned context contains only one such domain. The order of domains does not matter for security.
- All above optimizations are applied to the current thread context as well as the inherited parent thread context.

We tested our implementation on a Sun Sparc Ultra-1 running Solaris 2.5.1 with the latest in-house build of JDK1.2, but without a JIT compiler. We used loops with 10,000 iterations, and used the simple `System.currentTimeMillis()` method call to obtain time in milliseconds.

We first created call chains of the following structure, which we use $S^2A(BC)^nB$ to denote, where the chain starts with two classes of the system domain (S) and an application domain (A), n iterations of the two applications domains (B and C) calling each other, and ends with domain B invoking the call `AccessController.checkPermission()`. We configured the security policy to grant the requested permission to all application domains, which ensures that all

[4] This optimization is also convenient, because information about privilege status is stored in a C structure. If we have to return domains beyond the first privileged one, we would create additional storage and processing overhead in Java, not only for the extra domains but also their privilege status.

application domains are checked. For comparison, we also measured time without optimization (by removing those techniques from the code). The results are reported in the left half of Table 1.

n	domains (2n+4)	checking time (optimized)	checking time (non-optimized)
0	4	174.6$usec$	177.8$usec$
1	6	252.2$usec$	308.0$usec$
2	8	261.6$usec$	444.3$usec$
3	10	270.7$usec$	580.2$usec$
4	12	283.7$usec$	730.4$usec$
5	14	297.5$usec$	878.7$usec$
6	16	302.9$usec$	1001.8$usec$
7	18	312.2$usec$	1151.6$usec$
8	20	324.9$usec$	1270.4$usec$
9	22	333.7$usec$	1403.2$usec$

Table 1: Performance of checkPermission (08/01/97)

We can see that for every two additional domains, about an extra 10$usec$ is used. We also imagined that, if we have a large number of distinct protection domains, the checking time should increase roughly linearly. For validation, we repeated the same experiments, except that we removed the optimization technique that caches already-checked domains. This means that none of the application domains are collapsed or cached, so that a stack of the form $\mathrm{S}^2\mathrm{A}(\mathrm{BC})^n\mathrm{B}$ is equivalent to having $2n + 2$ distinct application domains. The results are reported in the right half of Table 1 below. We can see that for each additional domain, about 70$usec$ more time is required. Figure 4 illustrates the numbers reported in Table 1.

The above experiments were performed where each protection has 4 or 5 different permissions. When the number of permissions increase, we expect the checking time to increase too, although we use a variety of optimization techniques there. We currently do not have any performance numbers in this regard. We also do not report experiment results with more than 20 distinct application domains, as we do not envision the number being significantly larger in real-world applications.

We faced more problems with the privileged construct. We started off by implementing the privileged mechanism in Java code. This implementation included a hash table indexed on the thread object, and stored information that correspond to `beginPrivileged()` and `endPrivileged()` calls.

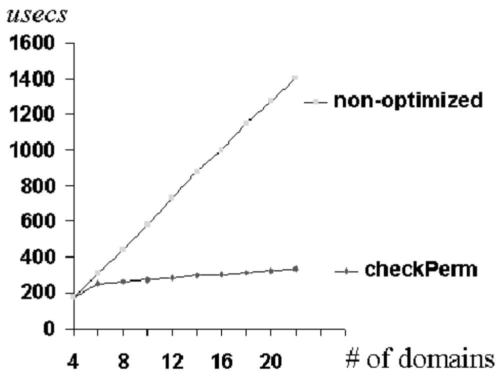

Figure 4: Performance of checkPermission (08/01/97)

This was fairly efficient for the sort of access control checking we envisioned, such as checking for file access.

However, soon the security model became widely used throughout the JDK code base itself, and packages for debugging and reflection (among others) demanded super fast speed. The earlier pure Java implementation became somewhat problematic for these new areas of applications, because they would have to originate calls from C into Java and then back to C, where the round trip cost is too high, and because having to manage a hash table for every privilege call was relatively expensive.

We subsequently changed our implementation to maintain and manipulate a record of privileged frames in a per-thread environment in C. After other various tuning, we obtained the following rough estimate, on the same test platform described earlier, except that we used loops with 100,000 iterations. We ran tests over several method calls (including a void call that does nothing) and the results for comparison, in microseconds ($usec$), are in Table 2.

	plain call	privileged call	increase
m1	0.86$usec$	2.96$usec$	2.10$usec$ (244%)
m2	2.72$usec$	4.92$usec$	2.20$usec$ (81%)
m3	5.83$usec$	7.96$usec$	2.13$usec$ (37)%
m4	31.03$usec$	33.98$usec$	2.95$usec$ (10%)
m5	815.20$usec$	816.94$usec$	1.74$usec$ ($< 0.3\%$)

Table 2: Performance of Privileged Blocks (07/30/97)

This table shows that the extra effort for creating and dismantling a privileged block is consistently at not more than 2 or $3usec$. Here, we use the following notations.

m1 = void method
m2 = File("/tmp/a")
m3 = Date()
m4 = System.getProperty(user.home)
m5 = Date(1, 2, 3)

5 Discussion and Future Work

To provide a rough sense of the scale of the work, we counted the number of lines of Java source code in newly introduced classes that related directly to the protection concept and its usage. We did not count classes that existed in JDK1.1, some of which are extensively modified, and the relatively small amount of new C code. The total number of lines of code in newly created classes excluding the copyright notices comes to about 5000. The total footprint (i.e., size of bytecode) of these classes is about 48.5Kbytes. Numerous existing classes (perhaps with another 5000 lines of code) have been changed, so the total implementation effort went much beyond creating the counted new classes. For example, supporting packages, such as `java.security.cert` and the entirely X.509v3 implementation, have been added for the first time.

It is worth noting that we have been able to evolve the existing security mechanisms in JDK1.0.x and JDK1.1.x to implement the new security architecture without changing the underlying Java virtual machine and without causing any incompatible API changes. Our design and implementation follow strictly the binary compatibility rules given in the Java language specification.

We can imagine several refinements. For example, the privileged primitive as currently designed would enable privilege for all the permissions granted to a domain. We can extend the primitive so that a protection domain can request privilege for only a subset of all its granted permissions. This could further reduce the security impact of making a programming mistake. The pseudo code segment below illustrates how to request privilege of reading one file.

```
FilePermission p =
    new FilePermission("/tmp/x.txt", "read");
try {
    AccessController.beginPrivileged(p);
    some sensitive code
} finally {
    AccessController.endPrivileged(p);
}
```

Moreover, we tend to think of the system domain as a single collection that includes all system code. It is possible to arrange so that system code run in multiple system domains, where each domain protects a particular type of resource and is given a special set of permissions. For example, we can grant file system code only access permissions to file system resources, and network system code only access permissions to network resources. This would be one step closer to the least-privilege principle, and the consequence of an error or security flaw in one system domain is more likely to be confined within its smaller boundary.

Finally, protection domain also serves as a convenient point for grouping and isolation between units of protection within the Java runtime. For example, although the `AppletClassLoader` class used by the `appletviewer` in JDK1.2 will load classes from different domains, it is possible to customize a classloader that will create different domains such that they are separated from interacting with each other. In this case, any permitted interaction must be either through trusted system code or explicitly allowed by the domains concerned.

Note that, in JDK1.2 architecture, the issue of accessibility is orthogonal to security. In the Java virtual machine, a class is distinguished by itself plus the class loader instance that loaded the class. Classes loaded by different class loaders live in different name spaces. Thus a class loader can be used to isolate and protect code within one protection domain if the loader refuses to load code from different domains. On the other hand, if one wishes to allow code from different domains to interact with each other, one could use a classloader, such as the `AppletClassLoader`, that defines classes in different domains.

6 Related Work

The fundamental ideas adopted in the new JDK1.2 security architecture have roots in the last 40 years of computer security research. Significantly, our design has been inspired by the concept of protection domains and the work dealing with mutually suspicious programs in Multics [10, 8], and right amplification in Hydra [6, 12].

One feature that is not present in operating systems such as Unix or MS-DOS, is that we implement the least-privilege principle by *automatically* intersecting the sets of permissions granted to protection domains that are involved in a call sequence. This way, a programming error in system or application software is less likely to be exploitable as a security hole. During the course of JDK1.2 implementation, from time to time, some (non-security) feature would no longer work due to the more restricted security model that has been put in place. Such events have pushed the developers to take a closer look at their code to make sure that necessary security checks are invoked and that the privilege primitive is used only scarcely and only under the appropriate circumstances.

Another character of Java is that its protection mechanisms are language-based, within a single address space. This feature is a major distinction from more traditional operating systems, but is very much related to recent works on software-based protection and safe kernel extensions (e.g., [2, 1, 11]), where various research teams have lately aimed for some of the same goals with different programming techniques.

7 Conclusion

This paper describes in detail the design and implementation of the concept of protection domains, which is an important foundation of the new security architecture delivered in the upcoming Java™ Development Kit (JDK1.2.) from JavaSoft, Sun Microsystems.

Our contribution is to adapt well-established concepts from years of computer security research to a design and implementation that suits particularly well for the Java environment. Our engineering effort has clearly demonstrated the flexibility and evolveability of Java and has proven that policy-based, easily configurable, fine-grained access control can be achieved efficiently on the Java platform.

Acknowledgments

Members of the JavaSoft community, including Josh Bloch, Sheng Liang, Marianne Mueller, Hemma Prafullchandra, Nakul Saraiya, and Bill Shannon, provided valuable assistance in design reviews and implementation.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Colorado, December 1995. Published as ACM Operating System Review 29(5):251–266, 1995.

[2] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[3] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[5] J. Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, California, August 1996.

[6] A.K. Jones. *Protection in Programmed Systems*. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA 15213, June 1973.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1997.

[8] J.H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[9] J.H. Saltzer and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[10] M.D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1972.

[11] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996. Published as *ACM Operating Systems Review*, 30, special winter issue, 1996.

[12] W.A. Wulf, R. Levin, and S.P. Harbison. *HYDRA/C.mmp – An Experimental Computer System*. McGraw-Hill, 1981.