Computer Networks 33 (2000) 77–93

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Supporting reconfigurable security policies for mobile programs

B. Hashii [*,1], S. Malabarba [1], R. Pandey [1], M. Bishop [1]

Parallel and Distributed Computing Laboratory, Computer Science Department, University of California, Davis, CA 95616, USA

Abstract

Programming models that support code migration have gained prominence, mainly due to a widespread shift from stand-alone to distributed applications. Although appealing in terms of system design and extensibility, mobile programs are a security risk and require strong access control. Further, the mobile code environment is fluid, i.e. the programs and resources located on a host may change rapidly, necessitating an extensible security model. In this paper, we present the design and implementation of a security infrastructure. The model is built around an event/response mechanism, in which a response is executed when a security-related event occurs. We support a fine-grained, conditional access control language, and enforce policies by instrumenting the bytecode of protected classes. This method enhances efficiency and promotes separation of concerns between security policy and program specification. This infrastructure also allows security policies to change at runtime, adapting to varying system state, intrusion, and other events. © 2000 Published by Elsevier Science B.V. All rights reserved.

Keywords: Mobile code; Java; Adaptive security policy; Access control; Dynamic classes

1. Introduction

The exponential growth of the Internet has precipitated a shift in popular computing, from stand-alone to distributed applications. In response, programming models that support code migration, such as remote evaluation [31] or mobile programs [5,33], have gained prominence. These models provide runtime systems that can load and execute externally defined user programs. Although appealing in terms of system design and extensibility [6], mobile programs are a security risk. They can maliciously disrupt the execution of programs on a host by unauthorized or improper use of local resources. To maintain security, a host must regulate a mobile program's use of local resources by enforcing an *access control policy* (ACP). The idea is not new; many operating systems limit access to their resources [1]. For example, in the UNIX operating system, users can control access to files they own.

Mobile code environments, however, have two important characteristics: (1) They are *dynamic*, i.e., mobile programs come and go rapidly, and the resources present on a host may change. (2) They are also *unpredictable*, i.e. administrators might not know ahead of time the source, behavior, or requirements of the programs that migrate to their host. There is no fixed set of resources that a host administers. Further, because the different components of resources and mobile programs may require different levels of protection [20], security models must support fine-grained access control.

Several techniques [3,11,13,15,17,19,20,29,35,36]

[*] Corresponding author.
[1] E-mail: {hashii, malabarb, pandey, bishop}@cs.ucdavis.edu

1389-1286/00/$ – see front matter © 2000 Published by Elsevier Science B.V. All rights reserved.
PII: S1389-1286(00)00075-X

have been proposed for defining and enforcing access control for mobile programs. The primary focus in most of these approaches has been on supporting flexibility, expressibility, and efficiency. While the above approaches encompass a wide range of security policy specification and enforcement techniques [18], there is very little or no support for building security environments in which security policies can be changed and reconfigured dynamically in order to adapt to changes in operating conditions.

Dynamic reconfiguration of security policies is needed in several instances, especially in complex and large distributed systems. Consider the following cases.

- Unanticipated changes in the security environment of a system may require that its security policies change. For instance, software bugs may appear that compromise the security of the entire system; exploits exist for recently discovered bugs in such critical components as `imapd` and `ftpd`[2]. Further, spies, covert channels and Trojan horses may lurk in application code. Upon discovering such unanticipated security holes, the system administrator should be able to add policies that revoke a previously trusted program's access rights.
- Security policies may evolve due to the changes in operating conditions and organizational goals. For instance, changes in environmental factors, such as company policy or the law, may result in a different set of access rights. Consider the introduction of privacy laws, which can prohibit the collection and distribution of user-specific information. Also, coalitions are often formed by several companies to pursue common projects. A company may establish security policies that are static while a coalition exists, but change after it dissolves.
- Security policies may vary depending on the state of the system. A computer system under attack may need stricter security polices than during normal operations. For instance, a distributed intrusion detection system may respond to attacks on several sites by establishing new policies based on the attack patterns. Further, in many cases, security policy checks may become unnecessary if trust levels can be established on the basis of a program's past behavior. For example, a system may monitor a program's accesses and, on the basis of past behavior, decide to remove all restrictions on access to specific resources.

It is possible to represent many of these security policies using static security policy mechanisms. However, to do so may require that the user anticipate and specify all possible situations. In addition, the representations may be awkward and incur undesirable overhead. What is needed is support for security policies that can be reconfigured at runtime to adapt to changes in the security needs of a site.

In this paper, we present the design and implementation of a security infrastructure that supports dynamic policies. The infrastructure uses a declarative policy language to specify access constraints. It enforces these constraints by performing binary editing on programs and resources [26]. In addition, the infrastructure provides a runtime meta-interface by representing ACPs as first class objects. The user can inspect, add, delete, and modify security policies at runtime. This mechanism supports dynamic security environments that adapt to unanticipated operating condition changes and system evolution. For example, the meta-interface is useful in large distributed systems, where the local policies in individual clusters must be discovered in order to construct and enforce global policies, and to verify consistency among the different local policies. When policies change, the runtime system instruments the protected classes, using dynamic classes [23], to enforce the new policy.

The remainder of this paper is organized as follows. In Section 2, we describe the declarative security policy language and the meta-policy model. We describe the implementation of the security infrastructure in Section 3. We present a performance analysis of the implementation in Section 4. We compare related work in Section 5. Finally, in Section 6, we discuss future work and conclude.

2. The extensible security infrastructure

We begin discussion of the security infrastructure by motivating our approach. We then present our abstract security model, which defines principals,

[2] For full details see http://www.cert.org/advisories.

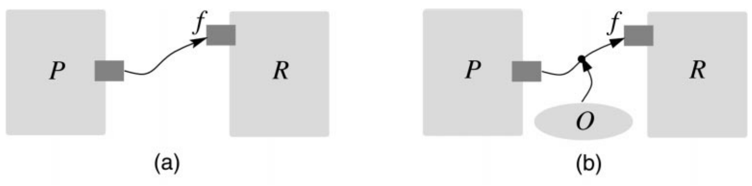

Fig. 1. Method invocation semantics. (a) Default method invocation semantics. (b) Security constraints on method invocations.

resources, and the relationships between them. Finally, we describe the domain-specific language and runtime meta-interface used to specify policies.

A program accesses a resource by invoking resource methods. In Fig. 1a, we show a program P that migrates to a host and accesses R by invoking f. During P's execution, control jumps to f, executes f, and returns back to P once f terminates. The Java compiler implements a simple access semantics in which there are no constraints on access to R through f.

In many cases, a host may wish to impose constraints on P's accesses to R. Our approach is to allow the host to make the access relationship between P and R *conditional*. For instance, in Fig. 1b, the host binds an access constraint, O, over the access relationship between P and R. Thus, P can access R if it satisfies O.

There are two notable aspects of our security mechanism. First, the access constraint, O, is defined separately from both P and f. The infrastructure enforces constraints by integrating interposition code within P and R before they are loaded in the Java virtual machine. Second, O can change at runtime, allowing the security policies of a host to evolve dynamically.

2.1. The event/response model

An access control policy is specified by a three-tuple: (1) specific access relationships that the security infrastructure should monitor, (2) conditions under which security-sensitive accesses warrant a response, and (3) the associated responses.

A security policy defines an access relationship between a principal and a resource. First, we define the notions of resource and principal, and then we show how they are used in our security model to specify policies.

2.1.1. Resource

Both hosts and mobile programs may define services or data structures that they wish to protect from unauthorized access. In keeping with the fine-grained access control model, a resource may represent any software component. Thus, a resource is defined as any method, class, or set of classes that is protected by an ACP. Conceptual resources, such as databases, and hardware resources, such as printers and the disk, must be wrapped by a Java class or method to be protected by an ACP.

2.1.2. Principal

The basis for authorization in a security model is the principal. In traditional systems, a program runs on behalf of a principal who is given certain access rights. Once the program attains these rights, it retains them during its execution. In a mobile code environment, however, a mobile program is typically composed of components that may be loaded from different hosts. A host may, thus, assign different components different rights and privileges, possibly on the basis of their origin. The level of granularity at which access rights must be checked and enforced is much finer in such an environment. The principle of least privilege [30] states that a principal should not have access to resources that are not needed to complete its job. This means that granularity of access control should be at the method level.

A principal in our security infrastructure can, thus, represent a method, an entire class, or a group of classes. For instance, `www.sun.com` denotes a principal comprising all classes loaded from this host; these classes all acquire the privileges assigned to the host. The infrastructure allows a principal to be defined as a group of classes by either enumerating the different classes or providing a filter function that determines if a class belongs to a principal. A site can use the filter function to define principals on the basis of specific characteristics such as signature, code source, or possible behavioral pattern. For example, the following class defines a filter function:

```
class GroupFunction {
 boolean static RogueSite(Class ncl) {
  URL u=ncl.getClassLoader().getURL();
  String name = u.toString();
  return
    name.equals("http://
                   www.roguesite.com");
 }
}
```

This function is executed whenever the system loads a class. The class is given to the function, and if it returns `true` then the class is added to an associated group. We will see in Example 1 how this function is used to define a policy.

It is important that sites define principals carefully so that mobile programs are unable to spoof definitions of principals, and thereby attain privileges that they should not have. For instance, principals based on simple class names can be easily spoofed. Note that we consider the problem of authenticating mobile programs orthogonal to the problem of access control discussed in this paper.

2.1.3. Event

An event occurs when a principal *P* accesses a resource *R*. We use the symbol `->` to denote a principal accessing a resource. Hence, the expression

```
-> File.Open()
```

denotes an event associated with the invocation of `Open` on an instance of a `File` resource.

An event may contain a condition, defined in terms of object, program, global, system or security states as well as the value of method parameters. For instance, the expression

```
-> File.Open() and
     (File.GetName() == "secretfile")
```

denotes only those `File.Open()` events for which the associated boolean condition is true.

2.1.4. Response

A response describes the action performed before and/or after a selected event has occurred. The infrastructure supports several predefined responses, such as `DenyResponse`, `AuditResponse`, and `ChangePolicyResponse`. `DenyResponse` denies access to a principal by throwing a security exception. `AuditResponse` logs any access to a protected resource. `ChangePolicyResponse` responds to an access attempt by changing the security policy. In addition, users can define their own responses and associate them with specific events.

Our security infrastructure provides two mechanisms for specifying policies: a *policy specification language* and a *meta-interface*. The high-level policy language permits rapid and flexible policy specification. Users may write policy files, containing series of statements, and load these files into the system. The statements are then translated into policy objects. The meta-interface provides language support for creation, management, and enforcement of policy objects at runtime.

2.2. The policy language

Fig. 2 lists the grammar for our policy language, which evolved from previous work [26] to include new constructs. Terminals in the EBNF, such as `ClassName` or `MethodName`, correspond to actual Java classes and methods, as described below. The language semantics allow for the use of either built-in event and response classes or user-defined classes. Policies are specified as a list of statements in policy files. The default policy file is loaded when an application starts, and new or modified files may be loaded during runtime via the interface described in Section 2.3.

A host uses this language to specify ACPs. As the grammar illustrates, our language is tailored to this task. It defines event/response relationships and allows entities to be grouped for ease of expression.

In addition to the access constraints, we support an `enable` statement that is used to override access constraints. This is needed when a host wants to override the default principle of least privilege. For example, assume that a security policy specifies that an applet cannot access the file system. The security infrastructure implements the default policy of least privilege, which ensures that the applet cannot access the file system directly or indirectly by calling other methods that access the file system. However,

```
Policy            ::= { PolicyStatements | Definitions }
PolicyStatements  ::= { Constraint | AddStatement | EnableStatement }
Definitions       ::= { PolicyGroup | GroupStatement }
Constraint        ::= before Event do Response | after Event do Response
AddStatement      ::= add Type Name to ClassName
EnableStatement   ::= enable Event
Response          ::= ResponseName ’(’ ParameterList ’)’
Event             ::= [Entity] Invocation Entity [and Condition]
Invocation        ::= ↦
Entity            ::= class ClassName | method MethodName ’(’ ParameterList ’)’ | group GroupName
Condition         ::= BooleanExpression
PolicyGroup       ::= policy PolicyName ’{’ PolicyStatements ’;’ { PolicyStatements } ’}’
GroupStatement    ::= group GroupName ’{’ Entity ’;’ { Entity } ’}’
                          | define group GroupName ’{’ FunctionName ’(’ Parameters ’)’ ’}’
```

Fig. 2. Access control policy language.

in many cases, this may not be desirable [36]. For instance, suppose the applet can write to the screen using the font files stored on the disk. In this instance, we want to enable the display manager to access the font files, regardless of the calling program. The `enable` statement allows one to override the default policy. This is similar to the `enablePrivileged` command in the JDK 1.2 security model [17].

2.3. The meta-policy model

The infrastructure represents security policies and their components as first class objects. Privileged programs can use the meta-interface to examine, add, delete and modify policy objects. Below, we describe policy objects and the meta-interface in detail.

2.3.1. Access control policy objects

Policy objects represent all policy statements, including constraints and groups. The class hierarchy of policy classes corresponds to the non-terminals in the policy language grammar. The security infrastructure represents policies in terms of three kinds of objects: event, response, and constraint.

Event objects. An event object consists of a subject, an invocation target, and, optionally, a boolean condition. An event may be trapped, and any bound responses executed, before or after the invocation. Creating a new event object is similar to specifying it in the policy language. For example, one can create an `Event` object to protect the password file in the following manner:

```
EventObject ev = new Event(
 "FileInputStream.
              FileInputStream(File f)",
 "f.getName() == /etc/passwd");
```

This event is trapped by an invocation of the `FileInputStream` constructor in which the file name is ‘/etc/passwd’. The parameters to the constructor are parsed in the same way as the language to create the same objects.

Response objects. A response object is an abstract class that users extend in order to customize responses. `ResponseObject` contains a `DoResponse` method that is invoked whenever the associated event occurs. A host redefines the `DoResponse` method to define any kind of response. An example of this is the `DenyResponse` class.

```
class DenyResponse
extends ResponseObject {
  public void DoResponse() {
  throw new SecurityException();
  }
}
```

Users may then create response objects by instantiating these classes. For example:

```
ResponseObject response = new
     DenyResponse();
```

Constraint objects. A constraint object represents an access constraint, and includes an event object and an associated response object.

```
class PolicyLoader {
      public PolicyObject get(String policyName);
      public void add(PolicyObject policy, String policyName);
      public void remove(PolicyObject policy);
      public void remove(String policyName);
      public void replace(PolicyObject oldP, PolicyObject newP);
      public void loadFromFile(String fileName);
      public void removeAllPolicies();
       ...
}
```

Fig. 3. PolicyLoader interface.

2.3.2. The policy loader

The core of the meta-interface is a module called the *policy loader*, which manages and enforces policy objects. The interface to the class `PolicyLoader` is shown in Fig. 3. It includes methods for adding, removing and examining policy objects, and loading policy files. Policy files, written in the specification language, may be loaded from disk or other sources, such as the network. `PolicyLoader` parses the policy file, translates the statements into corresponding policy objects, and adds the new objects to the overall system policy.

Users can extend `PolicyLoader` to add functionality. For example, one might redefine `PolicyLoader.loadFromFile()` to load policy files from a URL as part of a distributed security management scheme.

A `PolicyLoader` object is associated with a class loader, and, thus, enforces policies over a given namespace. We discuss this further in Section 3.4.1, and provide more implementation details on policy maintenance and enforcement in Section 3.2.

2.4. Examples

We now present four examples that illustrate how a system administrator can define and modify policies dynamically using the policy language and the meta-interface.

Example 1 (File auditing). In the first example, we illustrate how we can define principals, events and responses, and bind them together to define an access policy.

Assume that we want to audit all file accesses by programs that arrive from `www.roguesite.com`. We use the basic default audit response class, `AuditResponse`, to perform auditing. To enforce the policy, we create a `ReadFile` group that encapsulates the resources that we want to protect, and a `RogueSite` group that defines the principal using the group filter described in Section 2.1.

```
group ReadFile {
  class FileInputStream;
  class FileOutputStream;
  ...
}
group RogueSite {
    GroupFunction.RogueSite(Class
    newclass);
}
after group RogueSite ->
    group ReadFile do AuditResponse()
```

Example 2 (Protecting a resource). In this example, we show how the meta-interface can be used to define and enforce security policies in unanticipated security situations.

Assume that a host allows an external program from `www.roguesite.com` to access a public database server, `DBS`. However, the host discovers that the program is a Trojan horse that is able to exploit a bug in the database server to gain access to a protected section of the database, `PDBS`. As a result, the host wants to change the policy to prevent access to the restricted part of the database without shutting down the site. The host can enforce this new policy by constructing it and loading it using the policy loader:

```
PolicyLoader pl = getClass().getClassLoader().getPolicyLoader();
                                                     // find the policy loader
Group gp = new FilterGroup ("GroupFunction.RogueSite"); // create principal
Event ev = new Event(gp, "PDBS.query"); // specify access
Response newResp = new DenyResponse(); // create a deny response
ConstraintClass newPol = new ConstraintClass(ev, newResp); // create a constraint
pl.add(newPol, "DBSquery"); // prevent access
```

Example 3 (File or network access). In this example, we demonstrate how access control policies can be changed in response to a security-related event. We implement a commonly employed security policy that allows access to either the file system or the network, but not both. This policy could be part of a larger security policy that prevents the flow of information from disk to the rest of the world. Suppose the network is accessed as in Example 4, and the file system is accessed as in Example 1. We provide a basic default change policy response class as shown below:

```
class ChangePolicyResponse extends ResponseObject {
    public void DoResponse(String oldPolicy, String newPolicy) {
        PolicyLoader pl = getClass().getClassLoader().getPolicyLoader();
        if (oldPolicy!= null) pl.remove(oldPolicy);
        if (newPolicy!= null) pl.loadFromFile(String newPolicy);
    }
}
```

This response takes as parameters the name of a policy to remove, and the name of a policy file from which to load a new policy. We then specify four policies with the names "FileChange", "NetworkChange", "DenyFile" and "DenyNetwork". The deny policies prevent access to files and the network, respectively. For example:

```
define policy DenyNetwork {
  before -> Socket.Open()
    do DenyResponse();
}
```

Likewise, the change policies replace themselves with the appropriate deny policy. The "FileChange" policy is:

```
define policy FileChange {
 before ReadFile do
  ChangePolicy("NetworkChange",
               "DenyNetwork")
 before ReadFile do
  ChangePolicy("FileChange",null)
}
```

We can similarly implement "DenyFile" and "NetworkChange". Thus, when a file is read, the file's change policy is invoked, which in turn removes both the file's and network's change policy and adds a policy that prevents access to the network. The new access control policy no longer checks if reads are allowed. This provides a more efficient implementation of the above policy that a similar static policy would.

Example 4 (Control over the number of accesses). Suppose we want to implement the constraint that an object p can open a socket connection using Socket.Open(Host hostId, int Socketid) at most ten times.

We create a new field of p, SecurityState, of type SecState. This class keeps track of the number of times p calls Socket.Open. Let method SecState.CheckCount(int x) be defined in the following manner:

```
public boolean CheckCount(int x) {
  if (count < x) {
    UpdateCount();
               // increment the counter
  return(false);
  } else return(true);
}
```

The following policy statements add the new object to p and specify that p can invoke Open at most ten times.

```
define policy CountSockets {
 add SecState SecurityState to p
 before p -> Socket.Open()
  and p.SecurityState.CheckCount(10) do
  ChangePolicyResponse("CountSockets",
                       "DenySockets")
}

define policy DenySockets {
 before p -> Socket.Open()
 do DenyResponse()
}
```

After a socket has been opened ten times, the condition is always true. When this occurs, the policy is changed to always deny access.

The previous two examples highlight the fact that the reconfigurable policy mechanism can be used to eliminate access checks in several cases. For instance, consider a security policy: an applet can access a resource only if the condition *B* is true. Assume that *B* has the property that once it becomes true, it remains true. Clearly, checks for *B* can be eliminated once *B* becomes true. Using the dynamic security mechanism, a site can specify a policy that dynamically removes checks for *B* once *B* becomes true.

3. Implementation

In this section, we describe the implementation of our security model. We focus on two primary elements: the separation of security policy specification from resource definitions, and the ability to modify security policies during execution. We implement the first by generating binary code for each security policy on the fly, and integrating this code directly into the protected resource. We support the latter using dynamic classes, which allow the system to generate *new* interposition code and add it to previously instrumented classes.

First, we provide some background on dynamic Java classes. We then describe how our implementation enforces security policies, and how they can be modified. Finally, we present an analysis of our model and implementation. We describe potential weaknesses and their solutions, and include a general discussion of our approach's effectiveness.

3.1. Background: dynamic classes

Our implementation relies on *dynamic classes* to change policies. Using dynamic classes, we can instrument classes at runtime, and update their instances if needed. This is necessary to enforce policy changes. A previous paper [23] contains a full account of dynamic classes. In this section, we briefly outline the design and implementation, focusing on semantic and technical issues.

We wished to extend, not replace or weaken, Java's type and dynamic linking systems. We designed the semantics and interface for dynamic classes with this goal in mind. Thus, we define the semantics of a class change as follows: (1) a class change cannot cause any type violations, (2) all subclasses of the target class must change to reflect their new superclass, and (3) all existing instances of the target class must be updated to reflect the new definition. Under these conditions, the runtime state of the system remains consistent across class changes, and Java's type safety characteristics remain intact.

The JVM uses the class loader mechanism [22] for dynamic linking. We extended the class loader to provide a convenient interface for class changes. We use runtime system support for dynamic classes, modifying the JVM in Sun's JDK 1.2 to create a dynamic classes-capable virtual machine. We chose this approach over library-based support for reasons of efficiency and effectiveness.

3.2. Policy enforcement

The security system enforces an ACP by placing interposition code between the code requesting a resource and the resource itself. This interposition code checks if the specific access is allowed. A set of tools generates this interposition code and integrates it into mobile programs and resources before they are loaded.

Fig. 4 provides an overview of the components of the security infrastructure and their interaction. *P* denotes a mobile Java program, which migrates

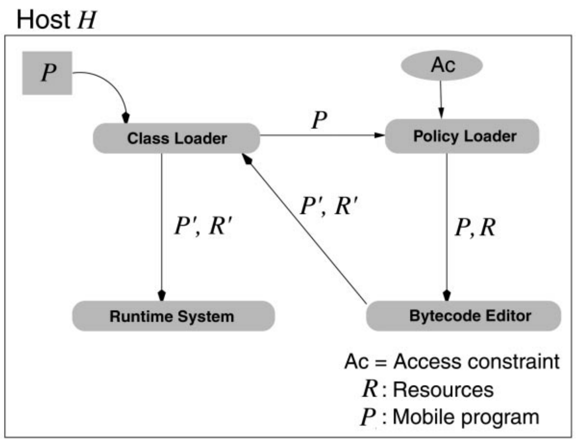

Fig. 4. Security policy enforcement.

to a host denoted H. A PolicyLoader is associated with a dynamic class loader, which defines the namespace that H provides to mobile programs. This PolicyLoader is then responsible for specifying and enforcing ACPs over all classes defined in its namespace. Upon application startup, a dynamic class loader initializes its policy loader. The policy loader parses a policy file and creates the specified policy objects. Then, the dynamic class loader loads application classes. During class name resolution and dynamic linking the class loader retrieves the resources R and passes them to the policy loader. The policy loader then generates the interposition code for enforcing the security policies and integrates it into the resource code.

The nature of the interposition code depends on the type of policy that the policy loader is trying to enforce. For instance, suppose the policy loader is implementing a constraint containing an event/response pair, named `event` and `response`, respectively. In addition, event is a triple defined by <`subject`, `target`, `condition`>. This policy is enforced by inserting the following code into `event.target`:

```
if (caller == event.subject
     && event.condition)
     response.doResponse();
```

The caller is identified by stack inspection, as described in Section 3.4.1. Details of the generated code and how it is integrated into the class definitions are beyond the scope of this paper; see [26].

After receiving the modified mobile programs and resources (P' and R'), the class loader loads the classes into the JVM, replacing any existing versions. In addition, the class loader caches copies, as raw bytecode, of the current and original versions of all loaded classes. The policy loader uses this cached data to re-integrate the interposition code when security policies affecting a class are modified.

3.3. Dynamic policy changes

PolicyLoader maintains a list of all policy objects active within its namespace. The union of the ACPs specified by these objects may be considered the system ACP for that namespace. The host can dynamically change a system ACP by adding or removing policy objects, or by loading a new policy file.

3.3.1. Adding policy objects

A system ACP may by extended via `PolicyLoader.add()` or `PolicyLoader.loadFromFile()`. The policy loader adds the new policy objects to its internal list and enforces them. It first identifies all entities affected by the change. This information is stored within the policy objects. If a group is among these, it expands the policy object to a list of objects, each replicating the policy, but applied to one class within the group. Next, the policy loader retrieves the raw bytecode version of each affected class from the dynamic class loader, and edits the bytecode according to the new policy object. Finally, it invokes the dynamic class loader to replace the existing class definition with the new, modified version. In the event that one of the classes referred to by a new policy object has not yet been loaded into the system, `PolicyLoader` stores the object and edits the class when it is loaded.

3.3.2. Removing policy objects

This requires the removal of any bytecode inserted when enforcing the target policy. First, the policy loader removes the policy object from its policy list. Then, as with addition, all affected classes are identified. For each loaded class, the policy loader retrieves the *original,* unedited, bytecode from the dynamic class loader and re-enforces all remaining policy objects.

3.3.3. Modifying the system policy

The system policy as a whole can be modified by adding and removing policy objects as described above. Alternatively, a host can modify the policy file and use the methods `removeAllPolicies()` and `loadFromFile()`. Both methods empty the policy list and retrieve the original definitions of all affected classes. `loadFromFile()` reloads the policy from the specified location, enforces each policy object, and replaces each class definition with the new version. `removeAllPolicies()` empties the policy list, retrieves the original definitions of all affected classes, and replaces the current definitions with the original versions. Therefore, calling `removeAllPolicies()` and `loadFromFile()` in sequence effectively refreshes the system policy to reflect a new or updated policy file.

3.4. Analysis

In this section, we analyze our security infrastructure and implementation. We identify techniques an attacker might use to circumvent security, and discuss the solutions.

3.4.1. Protecting against unauthorized dynamic modifications

First, we consider the security problems that arise due to the ability to dynamically change the behavior of a system, either directly, through the dynamic class loader, or indirectly, through the policy loader.

As we described in the previous section, the dynamic class loader provides a user or an applet with the ability to modify a class dynamically. A malicious applet, thus, can use this ability to modify a protected resource, and thereby bypass the access control policies associated with the resource. Consider the following security policy: applet A is granted read, but not write, access to a file class, F. So, A can invoke *F.read*, but not *F.write*. This constraint is enforced by inserting checks into *F.write* whenever F is loaded or modified. A could compromise security by adding a new method *F.Awrite*, which is identical to *F.write*, but is *not* protected by an ACP. A may then invoke *F.Awrite* and compromise file system integrity.

The problem arises because both protected resources and external mobile programs reside within the same namespace. Thus, we resolve the problem with strict namespace partitioning, as supported by Java 1.2 [21]. We associate different trust levels with the components of a program. Components with different trust levels are located within different namespaces. That is, they are loaded and managed by a different dynamic class loader and policy loader. Thus, untrusted mobile programs, local resources, and system classes are partitioned into separate namespaces. Now, applets cannot directly modify protected resources, since the dynamic class loader does not allow programs to change classes in different namespaces.

While the above separation does restrict an applet's ability to modify resources directly, the applet can still change policies through policy loaders. Since a policy loader controls all security policies within its namespace, it is vital that the policy loader itself be well protected from untrusted code, which might otherwise circumvent the entire security mechanism. Continuing with the prior example, we have ensured that A cannot modify F. However, A can get a handle to the policy loader in the resource namespace, and simply remove the ACP protecting *F.write*. Therefore, we impose the dynamic class loader's namespace constraint on the policy loader; policy loaders cannot be directly accessed across namespaces. To enforce these constraints, `PolicyLoader` includes native methods for stack inspection [37]. All methods that change policies include code that checks the previous frame on the current thread's call stack. If the corresponding method is not defined within the current namespace, then a security exception is thrown.

However, an applet may invoke the resource policy loader indirectly, via a resource. This should not be completely forbidden, since a user may specify a security policy in which resources initiate changes in the policy when accessed. The solution is to protect all resources and critical components with an ACP. Assuming that these ACPs are correctly specified, all resources are safe from unauthorized access.

3.4.2. Resolving policy conflicts

In Section 2 we described the `enable` policy statement. A policy such as `enable Fonts -> File` allows the `Fonts` class to access the `File` class regardless of what other methods are on the stack. This option introduces the possibility of policy conflicts and security holes. For example, a malicious applet can create a policy `enable BadApp -> File`, which could potentially grant file permissions. We define the semantics of `enable` to resolve conflicts and prevent security violations.

Consider a system using namespace partitioning, as described above. Such a system might have a hierarchy of namespaces, with a System namespace being the lowest and an Applet namespace being the highest. Note also that there is an implicit trust relationship between these levels; high levels need to trust lower levels. Security violations occur when objects in a higher level try to override the security policy of a lower level. Thus, whenever there is a possible conflict between an enable and a constraint, the lower level policy takes precedence.

For instance, a policy conflict would occur if the following two policies were encountered: (1) `before -> File do DenyResponse` and (2) `enable R1 -> File`. If the two policies are defined in the same namespace, then `enable` takes precedence. On the other hand, if (1) were defined in the system namespace, and (2) in the resource namespace, then (1) would take precedence.

Namespace partitioning relies on the assumption that a principal is trusted within its own namespace. Given this assumption, `enable` cannot be used by a malicious applet to circumvent an ACP protecting a resource. An `enable` policy enacted in the applet namespace is overridden by any conflicting policy in the resource namespace.

3.4.3. Reflection attacks

Reflection can be used to defeat some security mechanisms that rely on namespace partitioning [36]. This type of attack assumes that interposition code takes the form of proxy or wrapper classes that hide the protected class. A malicious applet can use reflection to discover the actual name of the protected class and invoke its methods manually, thus bypassing the proxy. Our system is immune to this sort of attack, since there are no proxy classes. Interposition code is placed directly in the protected method, and cannot be circumvented.

3.4.4. Synchronization attacks

Multithreaded systems present the attacker with the opportunity to exploit race conditions. Consider Example 4: if the `CheckCount` method was not atomic, an attacker could potentially violate the access constraint by exploiting a race condition between when the access count is checked, and when it is incremented. To prevent this sort of attack, and any synchronization-related bugs, all interposition code is synchronized. Whenever the bytecode editor encounters an object reference, it places a `monitorenter` instruction that locks the object. It then places all `monitorexit` instructions at the end of the instrumented method.

4. Performance analysis

In this section, we examine the performance behavior of our security infrastructure. The primary goal of the analysis is to evaluate the cost of providing an extensible security infrastructure. In our previous research, we evaluated the performance behavior of the binary-editing-based security infrastructure for static policies [26] and the JVM that supports dynamic classes [23]. To summarize these results:

- In many cases binary-editing-based approaches perform better than reference monitor-based approaches, such as the JDK security model, because interposition code is inlined, as opposed to involving several procedure calls.
- The overhead of implementing dynamic class is moderate, approximately six percent.

Therefore, in this section we focus primarily on the overhead required to implement reconfigurable policies. We performed all experiments on a 266 MHz Pentium II running SunOS 5.6.

One advantage of reconfigurable security is that frequently the number of security checks can be reduced. Recall Example 3. We want to prevent access to either the file system or the network. If a file is accessed, we set a policy to prevent network access. Further file accesses are not monitored. If a static security mechanism is used, a program would have to perform n security checks, where n is the number of times a file is accessed. Each check would consult a database to determine if the network has been accessed, say this takes time c. Our approach only needs to change policies once. Thus, our approach is valid whenever the time to change the policy, p, is less than $n \times c$.

Table 1
Policy modification results, recorded in seconds

Add policy	0.060
Remove policy	0.015
Loading a class (w/o policy)	0.018
Loading a class (w policy)	0.051
Invoking a method (w/o policy)	0.000070
Invoking a method (w policy)	0.00092

4.1. Microbenchmarks

In order to determine *p*, we measured the time to add or remove a simple policy using the meta-interface. The policy we used for these microbenchmarks is:

```
before -> method NoApp. run()V do
    NoResponseObject
```

The NoApp class is a subclass of Thread and redefines the run method. Both the protected method and the response are empty and do nothing. The result of each of these experiments is the average of 100 runs. It takes about 0.060 seconds to add a policy when the protected class has already been loaded. This includes the time to modify and replace the class. It takes about 0.015 seconds to remove a policy, including the time to unmodify and replace the class. Table 1 summarizes the results. For comparison, we have also included the time to load a class, with and without a relevant policy already installed, as well as the time to run the modified and unmodified methods.

4.2. Application benchmarks

We ran a second set of experiments to examine the overhead of enforcing security policies on applications. In particular, we used the SPECjvm ’98 benchmark suite [32]. [3] We specified a simple security policy that prints a warning message when writing more than a given number of bytes. The policy is defined as follows:

```
before -> class spec.io.FileOutputStream
 and spec.io.FileOutputStream.byteCount
  >1000000
 do AlertResponseObject
```

[3] We used a SPECjvm problem size of 100. These results are not SPEC compliant, and are suitable for internal comparison only.

We also did a control run with no policy specified. Table 2 summarizes our results; the performance penalty in most cases was around 7%. The variation in overhead is partly due to the amount of times the restricted resource is used. The `jack` benchmark, for example, is very IO-intensive, resulting in a 25.7% overhead. We expect that in most cases the overhead will be minimal. In the case of `jack`, the higher overhead is due to policy enforcement. We are currently looking at optimization techniques in order to reduce this cost.

5. Related work

We have divided this section into two parts. The first compares approaches for providing access control for mobile code. The second compares security policy languages.

5.1. Access control mechanisms

We evaluate and compare several proposed access control mechanisms with our mechanism. We focus on the power, flexibility, and dynamism of each approach, as compared to our work.

A mechanism for controlling access is a wrapper [34]. A wrapper is a code that encapsulates a resource. It may change the resource’s behavior by running interposition code. Thus, a wrapper could be used to add access control checks to a resource by inlining the interposition code, as in [35], or by calling a wrapper which then calls the original code, as in [13]. A technique operating at the system call level [15] uses a loadable kernel module to intercept and wrap system calls. On the other hand, type hiding [36] modifies the dynamic linking process in Java to hide or replace classes seen by an applet. It allows a class to be replaced by a proxy class that checks the arguments of the invoked method and conditionally throws an exception or call their original methods. The problem with using wrappers to implement extensible security can be seen in Naccio [13]. Naccio provides a framework for specifying resource hooks, state maintenance code, and safety policies. Pro-

Table 2
SpecJVM results, recorded in seconds

Benchmark	Standard JVM (T_o)	Active policy (T_p)	$(T_p - T_o)/T_p$
check	1.06	1.497	0.292
mtrt	1709.399	1845.753	0.0739
jess	1420.888	1532.356	0.0727
compress	7966.578	8219.17	0.0307
db	2675.772	2963.061	0.0970
mpegaudio	6383.705	6746.463	0.0538
jack	2083.441	2804.971	0.257
javac	1682.285	1820.105	0.0757

grams are transformed to use wrappers instead of the original library code. However, in order for Naccio to support extensible security policies, it must recompile policy definitions, recreate library wrappers and re-modify programs to use the new wrappers.

As an alternative mechanism, a resource monitor [2] intercepts and validates resource accesses. Our approach uses an extensible resource manager which is essentially created and modified at runtime. Most approaches, however, use a static resource monitor. While a user can change what a static resource monitor does — denying access or not — fairly easily, it is more difficult to change the set of monitored resources. In order to support policies over any resource, all resources must be monitored, and this can be very inefficient. As a result, most existing systems do not do this. Systems such as Java and Safe-Tcl define a subset of operations which are security-relevant and check only those references. Likewise, the monitors in [20] intercept and authorize all IPC calls.

Another approach that inlines resource monitor code is used in Security Automata SFI Implementation (SASI) developed at Cornell University [12]. The main differences between SASI and our work is the way in which policies are specified, and that SASI has an implementation for x86 assembly code. The advantage of x86 assembly code is limited in that it is difficult to extract application level resources from low-level code. There is also a SASI implementation for Java which is similar to this work. However, it does not support dynamic policy changes.

In Java's security model [17], each resource has a permission class associated with it, for instance java.io.FilePermission. A policy file specifies which principals have which permissions. The java.security.Policy object contains the policies that are currently in effect. A policy can be changed by either setting a new policy object or by calling its refresh method and reloading its policy. However, this form of extensible security is limited. For example, it is possible to write resources that do not call the access controller. Protecting such resources would require rewriting the resource class. Our approach can handle this by automatically modifying the resource. Furthermore, access control decisions are based on the protection domain of the code. It is not clear how to base decisions on environmental or historical conditions.

Deeds [11] is a history-based access control mechanism built on top of Java 1.1's security mechanism. In this system, a security event is a call to the security manager. The security manager will, in turn, call the handlers for the particular event. Policies can be modified by changing these handlers. However, like other static resource monitor approaches, it is unclear how to modify policies over resources that do not use Java's security manager.

Riechmann and Hauck [29] use meta-objects to implement security policies. A number of meta-objects can be attached to an object reference. They are invoked whenever the object is accessed. This is an alternate method of applying extensible security to objects, with the limitation that anyone can attach meta-objects, but only the meta-object can remove itself.

The Distributed Trusted Operating System (DTOS) [4] provides adaptable security by using security servers. When a request for a resource is made to the kernel or other program, a resource monitor checks with a policy server that determines if the access is allowed. Policies can be changed by either reinitializing the server or changing the active server. However, switching between different kinds of policies is difficult. For example, if a Unix style security policy is in place, the context information that the server needs consists of subjects, objects, and actions. Switching to an MLS policy requires changing the context information to labels.

Another alternate mechanism is a capability [9], an unforgeable reference to an object and a set of access rights. Possession of the capability authorizes the holder to perform the associated accesses on the object. Capabilities also have a notion of revocation and delegation. For example, the J-Kernel project [19] extends Java's security model by implementing a capability system within the language. On the other hand, Cherubim's [27,28] active capabilities provide extensible security policies by containing user-defined scripts that are run when received by a host. The limitation with generic capability systems is that they cannot usually prevent a program from leaking a reference to an untrusted object. Our approach solves this problem by protecting the resource itself, and not just its references.

Domain Type Enforcement (DTE) [3] is Trusted Information System's (TIS's) access control project, in which subjects are grouped into domains and objects are grouped into domains. There is also a language (DTEL) that specifies which domains can perform certain operations on which types and how threads can change domains by executing certain specified programs. DTEL operates on the level of files and programs, whereas our language operates at a finer granularity. Fraser and Badge [14] provide a way of maintaining general security properties across policy changes. The idea is that the addition of new policies should not invalidate the access relationships of the previous policies. They can prove that adherence to policy representing predicates during DTE's policy loading process maintains these properties. While we do not provide any such guarantees, we are currently researching the area of policy reasoning and composition.

5.2. Policy languages

Mechanisms for specifying and enforcing security are also a focus in security policy language research. Security policy languages have been considered as the basis for verification of secure systems design. Various considerations have been given to policy languages for doing general enforcement.

In access control matrices (ACM) [1], a two-dimensional matrix captures the access rights of subjects and objects. Entries in a cell determine the list of access rights that a subject has over an object. The ACM is primarily a theoretical tool, and is not used in practice. Its implementations, capability lists and access control lists, are cumbersome to work with when the subjects and objects involved are not known in advance.

Miller and Baldwin [24] describe a method of access control based on boolean expression evaluation. The idea is that each subject and object is given a set of attributes. In addition, there is also a set of rules that link a subject, an object, and an action. These rules can be based on any number of attributes. Since these attributes can be anything, including security level, group membership or time of day, they can be used to implement most security policies. Our approach is similar in that we capture the various attributes in terms of boolean expressions.

Goguen and Meseguer [16] use an algebraic specification approach to specify security policies. Their particular approach expresses security policies as a set of non-interference assertions about a system. Cholvy and Cuppens [7] and Cuppens and Saurel [8] use a form of deontic logic to express policies. In addition to specifying what actions an agent is permitted or forbidden to perform, it allows statements that say what actions an agent is obliged to perform. They use deontic logic to find consistency problems among policies. These policy languages are much more expressive than the one proposed in this paper. We plan to close this gap in the future. Our initial focus has been to develop a simple language for access control that can be implemented easily and efficiently.

The DIAMOND [25] security model provides an alternative model for inheriting security policies in object-oriented systems. It extends the MLS security model described by Denning [10] to object oriented databases. The innovation is that security levels, and hence policies, are not inherited from a class's superclass. Instead, they are derived from its instances. This allows a particular instance of a subclass to have a higher security level than its superclass. DI-

AMOND works strictly with MLS policies, whereas our scheme works for arbitrary models.

6. Conclusion

The mobile code environment is inherently dynamic, unpredictable, and dangerous. We have presented an extensible security infrastructure that supports fine-grained security policies that can be modified dynamically to suit the underlying operating conditions. The security infrastructure enables sites to respond to unanticipated changes in the security environment of a system, and changes in operating conditions and organizational goals. Further, sites can use the dynamic capability to eliminate unnecessary security checks. We have implemented the infrastructure in Java using a dynamic class mechanism.

Our future work involves enhancements and optimizations of our implementation. We are currently using the infrastructure to develop distributed policy discovery and management systems. Further, we plan to integrate the infrastructure into a distributed intrusion detection system.

Acknowledgements

This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government. We thank Jeff Gragg for help and support in implementing the infrastructure. We also thank Fritz Barnes, Earl Barr and the anonymous reviewers for their excellent comments and help in writing this paper.

References

[1] E. Amoroso, Fundamentals of Computer Security Technology, P T R Prentice-Hall, Englewood Cliffs, NJ, 1994.
[2] J.P. Anderson, Computer security technology planning study, Technical Report ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA 01731, October 1972, [NTIS AD-758 206].
[3] L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker and S.A. Haghighat, Practical domain and type enforcement for UNIX, in: Proc. of the 1995 IEEE Symposium on Security and Privacy, Oakland, CA, May 1995, IEEE Comput. Soc. Press, pp. 66–77.
[4] M. Carney and B. Loe, A comparison of methods for implementing adaptive security policies, in: Proc. of the Seventh USENIX UNIX Security Symposium, San Antonio, TX, January 1998, pp. 1–14.
[5] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik, Itinerant agents for mobile computing, IEEE Personal Communications, October 1995, pp. 34–49.
[6] D. Chess, C. Harrison and A. Kershenbaum, Mobile agents: are they a good idea? in: J. Vitek and C. Tschudin (Eds.), Mobile Object Systems. Towards the Programmable Internet, Second International Workshop, MOS ’96, Linz, number 1222 in Lecture Notes in Computer Science, July 1997, Springer, New York, pp. 25–47, Also available at http://www.research.ibm.com/massdist/mobag.ps.
[7] L. Cholvy and F. Cuppens, Analyzing consistency of security policies, in: 1997 IEEE Symposium on Security and Privacy, Oakland, CA, IEEE, pp. 103–112.
[8] F. Cuppens and C. Saurel, Specifying a security policy: a case study, in: 9th IEEE Computer Security Foundations Workshop, Kenmare, June 1996, IEEE Comput. Soc. Press, pp. 123–134.
[9] D. Denning, Cryptography and Data Security, Addison-Wesley, Reading, MA, 1983.
[10] D. Denning and P.J. Denning, Certification of programs for secure information flow, Communications of the ACM 20 (7) (1977) 504–513.
[11] G. Edjlali, A. Acharya and V. Chaudhary, History-based access control for mobile code, in: Proc. of the 5th ACM Conference on Computer and Communications Security, San Francisco, CA, November 1998, pp. 38–48.
[12] Ú. Erlingsson and F. Schneider, SASI enforcement of security policies: a retrospective, in: DISCEX’00, Proc. DARPA Information Survivability Conference and Exposition, Hilton Head, SC, January 2000, pp. 287–295.
[13] D. Evans and A. Twyman, Flexible policy-directed code safety, in: Proc. of the 1999 IEEE Symposium on Security and Privacy, Oakland, CA, May 1999, pp. 32–45.
[14] T. Fraser and L. Badger, Ensuring continuity during dynamic security policy reconfiguration in DTE, in: Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, CA, May 1998, pp. 15–26.
[15] T. Fraser, L. Badger and M. Feldman, Hardening COTS software with generic software wrappers, in: Proc. of the

1999 IEEE Symposium on Security and Privacy, May 1999, pp. 2–16.

[16] J.A. Goguen and J. Meseguer, Security policies and security models, in: Proc. of the 1982 Symposium on Security and Privacy, pp. 11–20.

[17] 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: Proc. of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997, pp. 103–112.

[18] B. Hashii, M. Lal, S. Samorodin and R. Pandey, Securing systems against external programs, IEEE Internet Computing, Nov./Dec. 1998, pp. 35–45.

[19] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu and T. von Eicken, Implementing multiple protection domains in Java, Technical Report 97-1160, Cornell University, 1997.

[20] T. Jaeger, J. Liedtke and N. Islam, Operating system protection for fine-grained programs, in: Proc. of the 7th USENIX Security Symposium, San Antonio, TX, Jan. 1998, pp. 143–157.

[21] JavaSoft, JDK 1.2 Documentation.

[22] S. Liang and G. Bracha, Dynamic class loading in the Java virtual machine, ACM SIGPLAN Notices 33 (10) (1998) 36–44.

[23] S. Malabarba, R. Pandey, J. Gragg, E. Barr and F. Barnes, Runtime support for type-safe dynamic Java classes, in: Proc. of the European Conference on Object-Oriented Programming, Sophia Antipolis and Cannes, June 2000, Springer, To appear, Currently available at http://pdclab.cs.ucdavis.edu.

[24] D.V. Miller and R.W. Baldwin, Access control by boolean expression evaluation, in: Fifth Annual Computer Security Applications Conference, Tucson, AZ, 1990, IEEE Comput. Soc. Press, pp. 131–139.

[25] L.M. Null and J. Wong, The DIAMOND security policy for object-oriented databases, in: 1992 ACM Computer Science Conference, Communications Proceedings, Kansas City, MO, pp. 49–56.

[26] R. Pandey and B. Hashii, Providing fine-grained access control for Java programs, in: 13th Conference on Object-Oriented Programming, ECOOP'99, Lecture Notes in Computer Science, Lisbon, June 1999, Springer, New York.

[27] T. Qian, Active capability: an application specific security and protection model, Technical report, University of Illinois at Urbana-Champaign, December 1996.

[28] T. Qian, Cherubim agent based dynamic security architecture, Technical report, University of Illinois at Urbana-Champaign, June 1998.

[29] T. Riechmann and F.J. Hauck, Meta objects for access control: extending capability-based security, in: New Security Paradigms Workshop, Langdale, 1997, pp. 17–22.

[30] J.H. Saltzer and M.D. Schroeder, The protection of information in computer systems, Proceedings of the IEEE 63 (9) (1975) 1278–1308.

[31] J.W. Stamos and D.K. Gifford, Remote evaluation, ACM Transactions on Programming Languages and Systems 12 (4) (1990) 537–565.

[32] Standard Performance Evaluation Corporation, SPECjvm98 Documentation, 1.01 edition, August 1998, http://www.spec.org/osg/jvm98/.

[33] T. Thorn, Programming languages for mobile code, ACM Computing Surveys 29 (3) (1997) 213–239.

[34] W. Venema, TCP Wrapper: network monitoring, access control, and booby traps, in: UNIX Security Symposium III Proceedings, Baltimore, MD, September 1992, USENIX Assoc., pp. 85–92.

[35] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham, Efficient software-based fault isolation, in: 14th ACM Symposium on Operating Systems Principles, ACM, December 1993, pp. 203–216.

[36] D.S. Wallach, D. Balfanz, D. Dean and E.W. Felten, Extensible security architecture for Java, in: 16th ACM Symposium on Operating Systems Principles, Saint Malo, Oct. 1997, Operating System Review 31 (5), pp. 116–128.

[37] D.S. Wallach and E.W. Felton, Understanding Java stack inspection, in: 1998 IEEE Symposium on Security and Privacy, Oakland, CA, May 1998, IEEE Comput. Soc., pp. 52–63.

Brant Hashii is a PhD student in the Department of Computer Science at UC Davis. His research interests include computer security, security policies, auditing, and mobile programming. Currently, he is exploring ways to apply access constraints to mobile programming systems. He is a member of IEEE and the ACM.

Scott Malabarba is an MS student in the Department of Computer Science at UC Davis. His research interests include dynamic software evolution, extensible runtime systems, and distributed computing.

Raju Pandey is an assistant professor in the Department of Computer Science at UC Davis. His research interests include Web-based computing, parallel and distributed programming, operating systems, and software engineering. He leads the Ariel project, which is developing novel techniques for supporting secure and efficient mobile-program execution. Pandey received a PhD from the University of Texas at Austin. He is a member of the IEEE and the ACM.

Matt Bishop received his PhD in computer science from Purdue University, where he specialized in computer security, in 1984. He was a research scientist at the Research Institute of Advanced Computer Science and was on the faculty at Dartmouth College before joining the Department of Computer Science at the University of California at Davis. His research areas include computer and network security, especially analysis of vulnerabilities, building tools to detect vulnerabilities, and ameliorating or eliminating them. He also teaches software engineering, machine architecture, operating systems, and (of course) computer security. He has chaired sessions and presented talks and tutorials at numerous conferences, organized and chaired the first two UNIX Security Workshops, and has been on numerous program committees. He is a charter member of the National Colloquium on Information System Security Education.