Providing Fine-Grained Access Control for Java Programs *

Raju Pandey and Brant Hashii

Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{pandey, hashii}@cs.ucdavis.edu
http://pdclab.cs.ucdavis.edu/

Abstract. There is considerable interest in programs that can migrate from one host to another and execute. Mobile programs are appealing because they support efficient utilization of network resources and extensibility of information servers. However, since they cross administrative domains, they have the ability to access and possibly misuse a host's protected resources. In this paper, we present a novel approach for controlling and protecting a site's resources. In this approach, a site uses a declarative policy language to specify a set of constraints on accesses to resources. A set of code transformation tools enforces these constraints on mobile programs by integrating the access constraint checking code directly into the mobile program and resource definitions. Because our approach does not require resources to make explicit calls to a reference monitor, it does not depend upon a specific runtime system implementation.

1 Introduction

There is increasing interest in computing models that support migration of programs. In these models, a program migrates to a remote host, executes there, and accesses the site's resources. For instance, Java [2] programs are increasingly being used to add dynamic content to a web page. When a user accesses the web page through a browser, the browser migrates Java programs associated with the page and executes them at the user's site. There are many other computing models that support mobility of programs. For example, the remote evaluation [34] model supports program migration by allowing one to upload a

* 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.

program to a remote site. The mobile programming model [3, 35] supports general purpose mobility that also allows programs to migrate to other sites during their executions. The common element in all of these models is the ability of a runtime system to load externally defined user programs and execute them within the local name space of the runtime system.

Although appealing [4] from both system design and extensibility points of view, mobile programs have serious security implications. Mobile programs have the ability to maliciously disrupt the execution of programs at a site by reading and writing into their name spaces, by using unauthorized resources, by over-using resources, and by denying resources to other programs. For instance, the "Ghost of Zealand" Java applet misuses the ability to write to the screen: It turns areas of the desktop white, making the machine practically useless until it is rebooted.[1] Another example is Hamburg's Chaos Computer Club[2] demonstration of the dangers of using ActiveX [6]. ActiveX is Microsoft's mobile program technology which allows components to be dynamically installed on a user's desktop. The victim uses Internet Explorer to visit a web page that downloads an ActiveX control. The ActiveX control checks to see if Quicken, a financial management software, is installed. If it is, the control adds a monetary transfer order to Quicken's batch of transfer orders. When the victim next pays the bills, the additional transfer order is performed. All of this goes unnoticed by the victim, until she receives her statement.

In this paper, we focus primarily on a specific security problem associated with mobile programs, namely the access control problem. The access control problem involves allowing a site to control a mobile program's ability to access local resources. Many operating systems [17] implement a notion of access control by limiting accesses to specific resources that the operating systems administer. For instance, in the UNIX operating system, the owners of files can control the accessibility of their files.

The access control problem in the mobile programming domain differs from the traditional access control models in many ways. First, there is no fixed set of resources that a site can administer; different sites may define different resources. An access control mechanism cannot be based on controlling accesses to specific resources. The mechanism should be applicable to any resource that a host may define. Second, the access control model should allow the customization of access control policies from one site to another, one mobile program to another, and one resource to another. Third, the access control model should support a fine-grained access control specification. In many access control models, access control involves either allowing an access or completely denying it. In the mobile programming domain, we argue for a *conditional access control* model where accesses to resources can be based on a boolean expression [26]. In other words, a site may allow a mobile program to access resources if certain conditions are met.

[1] For full details see http://www.finjan.com/applet_alert.cfm or http://www.internetworld.com/print/1998/05/11/webdev/19980511-hostile.html.

[2] For full details see http://www.iks-jena.de/mitarb/lutz/security/activex.hip97.html or http://www.iks-jena.de/mitarb/lutz/security/activex.en.html.

These conditions may depend on the state of mobile programs, state of resources, runtime system state and/or security state. For instance, a database vendor may specify that if there are more than 20 mobile programs in the system, each mobile program can only access its database up to ten times. In this example, a mobile program's ability to access the database depends on a runtime system state, such as the number of mobile programs running, and a security state, i.e. the number of times mobile programs access the database.

Access control specification and enforcement have been studied in great detail. The different approaches can be broadly classified into three categories: *operating system-based*, *runtime system-based*, and *language-based*. In the operating system-based approaches [17, 1], an operating system implements a specific access control model which specifies how system-wide resources such as the network, files, and displays can be accessed. The operating system enforces the security policy by checking whether the type of access is allowed. In runtime system-based approaches [10, 13], a runtime system enforces specific controls over accesses to various objects. Each method first calls a reference monitor which checks to ensure that the method call is permitted. In language-based techniques [11, 37, 31, 21] access control policies are specified along with a program specification. A compiler not only generates code for the program but also code to enforce security policies.

In this paper, we present an *alternate* approach for specifying and enforcing access control over mobile programs written in Java. Specifically, the paper describes the following:

– We present an access control model for specifying how accesses to resources can be controlled. In this model, a site defines a set of access constraints, each specifying the condition under which a specific resource can be accessed.
– We present a novel access constraint enforcement mechanism in which access constraints are enforced by integrating access constraint checks directly into mobile program code and resource code before they are loaded into the runtime system.

Separating the specification of access constraints from the specification of Java programs and resources has the following implications:

– Resource developers do not need to manually insert calls to security checking code inside each resource that a host may want to protect. Further, the access control mechanism can be used to define and enforce access constraints to systems that were not designed with security in mind, such as legacy systems.
– Both resource definitions and access constraints can be modified independently without affecting each other's implementation.

We have implemented a version of this mechanism for programs represented using Java bytecode [25]. The performance results show that the overhead of this approach is moderate. Further, it performs better than the approach implemented in the Java runtime system in many cases.

This paper is organized as follows: Section 2 contains a description of our resource access model and how accesses to various resources can be specified.

Section 3 describes an implementation of this model. Section 4 contains an analysis of the approach, including its performance behavior. Section 5 contains a brief survey of related work. Section 6 contains a summary of the approach and discussion of future work.

2 Access Control Model

The access control model contains two parts: a resource model for representing resources and an access constraint specification language. We describe the two in detail below.

2.1 Resource Model

A site provides many resources to a mobile program. These resources include classes for utility libraries, accessing files, networks, and interfaces to other resources such as a proprietary database. For instance, a site providing access to a weather database exports a set of interfaces that specify how the database can be accessed. In our security model, each Java class or method represents a resource and, thus, is a unit of protection. Our access control mechanism does not differentiate between system classes and user-defined classes, or between locally defined classes and classes down-loaded from remote hosts. The model also allows the definition of class-subclass relationships among resources using the Java's inheritance model.

2.2 Access Constraint Specification Language

The access constraint specification language contains two parts: a notation for specifying constraints over accesses to resources and an inheritance model for access constraints.

Access Constraints: We first describe the motivation behind our access control language. A Java program uses a resource by invoking its methods. In Fig. 1(a), we show that program P invokes a method f to access resource R. During an execution of P, the control jumps to f, executes f, and returns back to P upon termination. The Java compiler implements a simple access semantics in which there are no constraints on accesses to R through f.

Our approach is to allow a host to make the access relationship between P and R *conditional* by adding a constraint, B (see Fig. 1(b)). The access constraint is specified *separately* from both P and R and has the effect of imposing the constraint that P can invoke f on R only if condition B is true. A site, thus, restricts accesses to specific resources by enumerating a set of access constraints, which forms a site's access control policy.

Below, we present only the core aspects of the language. For brevity we have omitted the details regarding specification of security constraints over groups of classes, methods and objects. The following EBNF shows how a site can specify access constraints:

(a) Default method invocation semantic

(b) Security constraints on method invocations

Fig. 1. Method invocation semantics

```
Constraints          ::= { AccessConstraint }
AccessConstraint     ::= deny '(' [Entity] Relationship Entity ')'
                             [when Condition]
Relationship         ::= ↦ | ⊣
Entity               ::= ClassIdentifier | MethodIdentifier
Condition            ::= BooleanExpression
```

A site controls accesses to different resources (Java objects) by defining a set of AccessConstraints. We describe the various terms in the grammar informally below:

- Entity: An entity denotes objects and method invocations of Java programs. A ClassIdentifier, thus, identifies the set of objects to which a given access relationship applies. Similarly, a MethodIdentifier denotes a set of invocations of a method. The current implementation defines an entity based on its name. However, this can be extended to define an entity on the basis of its source, signature, or behavior pattern.
- Relationship: The composition mechanisms of a programming language allow one to define various relationships (data composition through aggregation and inheritance, and program composition through method invocations) among the entities of a program. We are primarily interested in the following two access relationships here:
  1. Instantiate ($\dashv$): A relation $E \dashv R$ exists if an entity E creates an instance of class R.
  2. Invoke ($\mapsto$): A relation $E \mapsto R$ exists if an entity E invokes an entity R.
- Condition: The term Condition denotes a boolean expression that can be defined in terms of object states, program state (global state), runtime system state, security state, and parameters of methods.

Semantics: An access constraint of the form

deny (E σ R) when Condition

specifies that entity E cannot access R through relationship σ if Condition is true. E is optional. Hence, there are two kinds of access constraints: *all access*

(a) Global constraints (b) Selective access constraints

Fig. 2. Category of access constraints

constraints and *selective access constraints*. Global constraints denote those constraints that do not depend on the initiator of the access relationship. For instance, as shown in Fig. 2(a), no program can access R when B is true. A host may specify the constraint that no Java applet can access a set of proprietary files.

Selective access constraints denote those constraints that depend on the initiator of the access relationship. For instance, as shown Fig. 2(b), each entity E_i's access to R is constrained by a separate and possibly different B_i. A site can use selective access constraints to associate different security policies with different Java programs that come from different sites.

Examples of all access constraints are:

Constraint	Semantics
deny ($\dashv$ C_2) when B	No instances of C_2 can be created if B is true
deny ($\mapsto$ $C_2.M_2$) when B	Method M_2 of class C_2 cannot be invoked if B is true.

Examples of selective access constraints are:

Constraint	Semantics
deny ($C_1.M$ $\dashv$ C_2) when B	Method M of class C_1 cannot create an object of C_2 if B is true.
deny ($C_1.M_1$ $\mapsto$ $C_2.M_2$) when B	Method M_1 of class C_1 cannot invoke M_2 of C_2 if B is true.

In our approach, the default is to allow all accesses unless a site specifically denies them. We call this model the *active denial model*. This is unlike most approaches in which the default is to deny all requests unless a site specifically allows them. We call this model the *active permission model*. The active permission model provides better guarantees about system security in cases when a site makes mistakes about specifying access control policy, the reasoning being that it is better to deny legitimate accesses than allow illegitimate accesses [33].

We chose to use the active denial model because we want to construct a unified access control framework for all method invocations. In other words, every action (every method call, object creation, deletion, etc.) is conceivably a security relevant event which a site may want to control. For instance, we want to be able to specify constraints such as users can invoke a function, say sqrt, only 10 times. Implementation of this access control model using the active permission model would require that a site define permissions for every method call, which can be quite cumbersome. Runtime system-based approaches [25] deal with this problem by embedding calls to an access controller checker within all methods that the site might want to control The checker enforces an active permission model over these calls. All resources that do not embed calls are not checked and hence can be accessed by anyone. Such models, thus, differentiate between resources that must be protected, through embedded calls, and those that need not. Our approach uses a single mechanism for handling both. The active denial model can be used to implement the active permission model by representing the permission conditions through the negation of denial conditions. We are, therefore, looking at ways of integrating the active permission model in our language.

Examples: We now present three examples. The first example implements a simple file access control mechanism. The second example shows how we can use the state of the runtime system to control accesses to resources. Finally, the last example shows how we can associate specific security states with program components and use these states to specify access control.

Example 1. File access control. In this example, we specify access constraints for controlling the file resources that mobile programs can access. Assume that the file resource is defined using the following Java class:

```
class File {
    public File(String Name);
    public char Read();
    public void Write(char data);
    public final String GetFileName();
}
```

The following constraint specifies that no mobile program can read "/etc/passwd" file:

```
deny ( ↦  File.Read) when (#2.GetFileName() == "/etc/passwd")
```

Here we introduce a new notation within the boolean expression. The terms #1 and #2 refer to the entities before and after the relationship, respectively. Thus, in the above expression the term #2.GetFileName() can be read File.GetFileName().

The access constraint that mobile programs can only read files A and B can be specified by expressions of the form:

```
deny ( ↦  File.Read) when
    ((#2.GetFileName() != "A") && (#2.GetFileName() != "B"))
```

The constraint that mobile programs cannot write to the local disk is specified by the following constraint:

```
deny ( ↦ File.Write)
```

As we can see from the above example, an access constraint can control executions of methods on the basis of program states. In certain cases, a site may wish to impose constraints on the basis of the state associated with the runtime system or the underlying operating system. The policy language allows specification of such constraints. We show this through an example:

Example 2. Network access control. Assume that the following defines the socket resource for making network connections:

```
Class Socket {
    Socket();
    void Open(Host hostId, int SocketId);
    void Write(Bytes data);
    Bytes Read();
}
```

Also, assume that the runtime system keeps track of the number of network connections that have already been opened. This forms the state associated with the runtime system. Let the method RuntimeSystem.Network.NumConnections() return the number of open connections. A constraint that limits the number of network connections to a specific upper-bound can be specified in the following manner:

```
deny ( ⊣ Socket) when
    (RuntimeSystem.Network.NumConnections() == UPPERBOUND)
```

In addition to runtime system state, a site may wish to store additional information for implementing access control. We call this kind of information *security state.* A site may associate a security state with a method, object, or a group of objects, and may define constraints over accesses to methods on the basis of the security state. We present an example below that illustrates this:

Example 3. Control over number of accesses. Assume that we want to implement the constraint that a program p can invoke a method, say f, on a resource R at most ten times.

This can be implemented by associating an object, say SecurityState, with p. The object keeps track of the number of times p calls f. Let method SecurityState.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 policy statements

```
add SecState SecurityState to R
deny (p ↦ R.f) when R.SecurityState.CheckCount(10)
```

adds the new object to R and specifies that p can invoke f at most 10 times.

Inheritance of access constraints: We now present an inheritance model for access constraints. The inheritance model describes what denials to resource accesses mean in terms of denials of accesses to subclasses of resources.

Assume that a site defines two resources, R_c and R_s:

```
class Rc {
    public void f();
    public void g();
    public void h();
}
class Rs extends Rc {
}
```

R_s is a subclass of R_c: R_s inherits methods f, g, and h from R_c. Assume that the site defines the following constraints on the resources:

deny $(E \mapsto R_c.f)$ when B_{cf}
deny $(E \mapsto R_c.g)$ when B_{cg}
deny $(E \mapsto R_s.f)$ when B_{sf}
deny $(E \mapsto R_s.h)$ when B_{sh}

There are two components to the inheritance model:

- Inheritance of access constraints: A subclass inherits all access constraints from its superclasses. Hence, the resulting access constraint on invocations of g on an instance of R_s is defined by the following expression:

  deny $(E \mapsto R_s.g)$ when B_{cg}

  Access constraints are not inherited from subclasses to superclasses. Hence, although the access constraint on h in R_s is B_{sh}, there are no access constraints on h in R_c.
- Strengthening of access constraints: A subclass cannot override its inherited constraints. Specification of additional constraints in the subclass only strengthen the constraints defined in its superclasses. Hence, the resulting access constraint on invocations of f on an instance of R_s is:

  deny $(E \mapsto R_s.f)$ when $B_{cf} \vee B_{sf}$

  In other words, method $R_s.f$ cannot be invoked from E if either B_{cf} or B_{sf} is true.

This model of inheritance ensures that a mobile program cannot override access constraints on methods by defining a subclass and by weakening the access constraints. Also, the above inheritance model applies for access constraints on $\dashv$ as well. That is, if a class R_c cannot be instantiated, none of its subclasses can be instantiated.

3 Access Constraint Enforcement

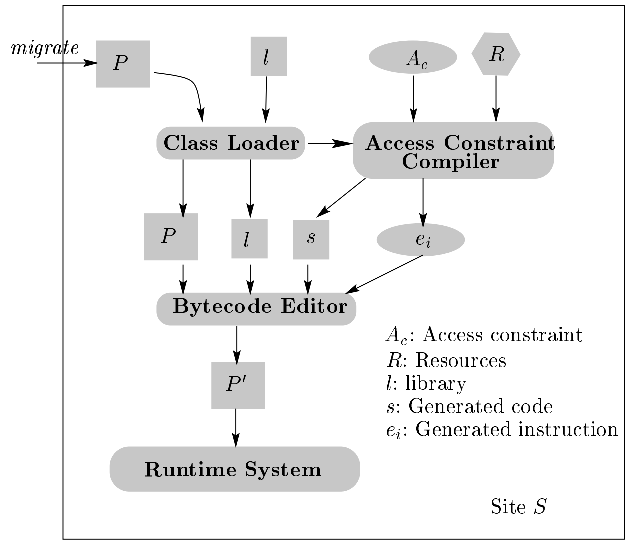

Fig. 3. Security policy enforcement of mobile programs

An enforcement of access constraints on a resource involves placing interposition code between the resource access code and resource definition code. The interposition code checks if a specific resource access is allowed. It can be inserted *manually* by site managers, generated by the compiler, or defined by the runtime systems or operating systems through special system calls. For instance, in the Java runtime system [12, 13], resource developers manually insert calls to a reference monitor in the resources they want to protect. The reference monitor consults access control policies to check if a specific resource access is allowed.

We use an alternate approach for generating interposition code. In this approach, a set of tools generates the interposition code and integrates them within mobile programs and resources before they are loaded in the JVM. In this approach, there are no reference monitors. In essence, the approach generates reference monitors on the fly and integrates them within the relevant Java programs and resources. The approach, thus, eliminates the need to manually include calls to reference monitors in resource definitions.

In Fig. 3, we describe our implementation for enforcing access control policies on Java programs. We show a Java program P that migrates to a site S. R denotes resources that the site makes available to mobile programs; and l denotes local libraries linked into P.

During class name resolution and dynamic linking, the Java class loader [24] retrieves R and l and passes them to a tool, called the *access constraint compiler*. The access constraint compiler examines P, R, and l to determine the resource access relationships that must be constrained in order to implement the access constraint A_c. It then generates interposition code s that implement the specific access constraints. It also generates a set of editing instructions e_i for the bytecode editor. The bytecode editor uses e_i to integrate s within P, R and l. The transformed programs and resources are then loaded into the JVM and executed.

We now describe in detail how we determine access relationships in Java programs, generate code, and edit Java class files.

3.1 Type Extraction

Type extraction involves examining Java class files to determine type definitions declared in the class files. Type definitions are used for automatically constructing a resource model from class files as well as for determining how Java classes should be modified. Type extraction can be done easily since Java class files maintain complete symbolic information about a class. Our type extraction technique makes use of two entities within the Java class file: the *constant pool* and the *method definition* sections. The constant pool is similar to a symbol table in that it contains all of the information needed to dynamically link classes. It is an index to the symbolic references of fields, classes, interfaces and methods, as well as their names. It also contains all literals, both string and numeric, used throughout a class. For example, a methodref entry in the constant pool includes all the symbolic information associated with a method. It contains two constant pool indexes: one for the class name and one for the name and type of the method. The method definitions section defines each method and identifies them by name and signature.

3.2 Extraction of Access Relationships

The extraction of access relationships involves searching the bodies of the methods for method invocation instructions. In the JVM, four opcodes (invokevirtual, invokespecial, invokestatic, and invokeinterface) are used for method invocation. Each method invocation instruction has an operand which indexes into the constant pool. Since this index is either a methodref entry or an interfaceref entry, the class name, method name, and signature of the method being invoked is immediately available. Both instantiate and invoke relationships are, thus, determined by searching the method bodies for one of the four invoke opcodes and matching it with the object's class name, method name, and signature. Note that this information may not be entirely valid due to the dynamic binding of methods. This problem is discussed in detail in the following sections.

3.3 Code Generation and Binary Editing

We now describe the nature of the code that is generated and its integration within mobile programs. Our code generation and editing involves modifying class definitions in order to add runtime state to classes and to insert runtime checks into methods.

An access constraint of the form

$$\texttt{deny}\ (E\ \sigma\ R)\ \texttt{when}\ B$$

is implemented by generating the following code:

```
if (B)
    then error(); // raise exception
else
    access R
```

and patching it into classes and methods. The nature of the editing depends on the nature of the access constraints. Global constraints of the form

```
deny (σ R) when B
```

specify constraints on accesses to R without any regard to objects or methods that may access R. The generated code is, thus, integrated into the methods of R. On the other hand, selective access constraints of the form

```
deny (E σ R) when B
```

imposes conditions on accesses to R from E. The generated code is, thus, integrated into the methods of E which explicitly access R.

We also support addition of security states to specific Java classes in order to monitor site-specific behavior. This mechanism allows a site to customize its security policies, especially if the policies cannot be represented directly by the policy language. Security state objects are added to a class definition by using the statement:

```
add SecurityStateType SecurityStateObject to R
```

The constraint compiler generates code for initializing external objects. Example 3 shows how such objects can be used to specify access control policies.

3.4 Implementation Details

In this section we describe the code generation and code editing process for different instances of access constraints. For the purposes of explanation we restrict access to R when the first parameter is 5. Note that the boolean condition only affects the nature of code that is generated for B; it does not affect the general pattern of the access check code or the method of editing. Also, the following technique is independent of the action that should be taken in the event that an access is denied. Our implementation throws a security exception. Alternatively, one could take any conceivable programmable action, such as writing to an audit log, ending the mobile program, or even moving the mobile program to another site.

Implementation of Global Constraints: The first set of cases involve performing editing within the definition of a called method. We first consider a constraint of the form

```
deny ( ↦ R.f(I)V) when (#2.(1) == 5)
```

Recall that the term #2 refers to the entity being invoked. The term #2.(1) refers to the first parameter of that method. Also note that (I)V following R.f is the Java bytecode representation of the signature of that method. The above access constraint is enforced by generating code of the form shown in Fig. 4 and patching the code into the body of f.

In Fig. 4, the number to the left of an instruction indicates the byte offset for the instruction from the beginning of the method body. Further, a term #i in Fig. 4 and Fig. 5 indicates the ith entry in the constant pool. In code segment A of Fig. 4, #67 indexes the integer constant 5, whereas #65 in code segment B indexes the entry for a security exception class and #66 indexes the entry for its constructor.

```
0 iload 1
2 ldc #67                   A
4 if_icmpeq 10
7 goto 21

10 new #65
13 dup                      B
14 invokespecial #66
17 athrow

original code for
method R.f(I)V
```

Fig. 4. The modified method R.f(I)V

Code segment A (Fig. 4) contains the code for checking the conditional, whereas code segment B contains code for throwing an exception if the boolean condition is true. This code is inserted into the beginning of the method. Care must be taken to ensure that the security exception object and its constructors are defined in the constant pool. If they are not, then these entries are added.

Constraints of the form

```
deny (⊣ R) when B
```

specify that an instance of R cannot be created if B is true. They are implemented by putting constraints on invocations of all constructors of R, which, in the JVM, are given a special name <init>. This case is, thus, implemented by adding code similar to that shown in Fig. 4 to all methods of R with the name <init>.

Implementation of Selective Access Constraints: We now consider the cases in which methods are modified within the calling method. The most specific case involves denying access to a method from a specific method:

```
deny (E.g()V ↦ R.f(I)V) when (#2.(1) == 5)
```

Binary editing here involves first searching for all invocations of R.f(I)V within the body of E.g(). This involves examining the operands of all the invoke opcodes. Since the operand references a methodref entry in the constant pool, we can read the signature, method name, and class name of the method being called. If these match R.f(I)V, then the generated code is inserted before the invoke opcode.

The access relationship determined in this manner may only be partially correct due to the dynamic binding of methods. Assume the inheritance hierarchy of Sect. 2.2. Also, assume that method f is invoked on an object O of type R_C:

```
O.f();
```

If entity O references an object of type R_C or type R_S, and constraint B is defined for the method of class R_C, the above approach works because the constraint is inherited in the subclass as well. The problem arises when the constraint is defined over invocations to method f of R_S and object O may reference objects of type R_C or type R_S. Note that if it references objects of type R_C, the generated code should not be added because constraints are inherited from superclasses to subclasses, and not vice-versa. However, if O references an object of type R_S, the generated code should be added in order to implement the constraint. Since the reference type cannot be determined statically, additional code must be generated that checks for the type of object at runtime and performs access constraint checks on the basis of the type of the object. Thus, in cases where dynamic binding may play a role, an instanceof instruction is added to dynamically check the type of the object. The generated code for this case is shown in Fig. 5.

```
rest of E.g()V

10 istore 2
12 astore 3                    A
14 aload 3
16 iload 2

18 aload 3
20 instanceof #3               B
23 ifeq 44

26 iload 2
28 ldc #35                     C
30 if_icmpeq 36
33 goto 44

36 new #33
39 dup                         D
40 inokespecial #34
43 athrow

44 invokevirtual #10           E

rest of E.g()V
```

Fig. 5. The modified method E.g()V

The first step (code segment A) is to access the object reference by popping the operand stack, which contains method parameters and the object reference, into local variables. The method parameters and object reference are then pushed back on the stack in case the method is called later. This also need to be done if the constraint refers to the parameters of the called method. The second step (B) involves pushing the object reference onto the stack, performing an instanceof operation, and jumping to the method call if the object is not of type R. Term #3 is an index into the constant pool that refers to the class R. As in the first case, code segment C performs the conditional check, and section D throws the security exception. Section E contains the original invoke command. Term #10 is a constant pool index that refers to the method f with signature (I)V and class R. Other instances of access constraints can be implemented using the above technique.

Implementation of Inheritance Model: An implementation of the inheritance model requires care because of the possible conflicts between the Java language mechanism for controlling extensibility and our inheritance model. We illustrate the problem with a simple example.

Assume that class R_s is a subclass of R_c. Class R_c defines a method f:

```
public void f();
```

Assume that R_s inherits f. Also, assume that the site specifies the following access constraint:

deny ($\mapsto$ $R_s.f$) when B

Since R_s inherits f, f needs to be modified in order to impose the above access constraint. However, since policies are inherited down and not up, the method body of f in R_c cannot be modified. A possible solution is, then, to redefine f in R_s:

```
public void f() {
    <interposition code for checking access>
    super.f();
}
```

The above solution works if f is not declared final in R_c. However, if f is declared to be final, we cannot redefine f in R_s as the Java bytecode verifier will reject the redefinition of a final method. Although we can edit the class file for R_c to remove the 'final' constraint, such a change may lead to security holes.

Our solution, therefore, relies on modifying class R_c as follows:

```
class Rc {
    final public void f() {
        _F_CheckMethod();
        <code for f>
    }
    private void _F_CheckMethod() { ; }
}
```

We now redefine _F_CheckMethod() in R_s in order to implement access constraint checks that are specific to R_s:

```
class Rs extends Rc {
            :
    private void _F_CheckMethod() {
        <interposition code for checking F>
    }
}
```

4 Discussions

In this section, we analyze the proposed technique for its suitability as an access constraint enforcement mechanism and for its performance behavior.

4.1 Characteristics of the Approach

In our approach, a site specifies access constraints separately from mobile programs, resources, and other class definitions. Further, the access constraint enforcement mechanism is not part of either the Java runtime system or the compiler. This impacts how access control code is managed and enforced at a site:

- Both access constraints and resource definitions can be modified independently. This makes it easy for a site to specify different access constraints for different mobile programs for the same resource. For instance, a site may specify that mobile program P can access R under condition B_p whereas mobile program Q can access R under condition B_q.
- The same set of access constraints can be applied to different resources without requiring one to copy it from one resource to another. For example, if a single access constraint B applies to multiple resources, it can be defined once and used for all resources.
- An important advantage of the separation is that our approach can be used for enforcing security on resources that were not designed with security in the first place. In other words, the security component can be added to a resource after it has been designed and implemented. Thus, it frees a library or resource designer from worrying about security concerns when designing and implementing the library.

A limitation of our approach is that it may end up building data structures that mirror some of the data structures that runtime systems build. This limitation arises because of the static nature of code enforcement. In many cases, access control policies depend on the history of execution as well as the dynamic state of an executing program. For instance, an access control policy may require that a program access a resource only if all methods currently on the stack are permitted to access the resource. This means that the interposition code must check the permission of all methods currently on the execution stack. Since our enforcement mechanism is completely separated from the runtime system, it needs to build and maintain a separate runtime infra-structure, involving an execution stack, in order to implement such policies. In runtime systems which exports the necessary state, these policies can be easily implemented.

4.2 Performance Analysis

In this section, we describe the performance behavior of the access constraint enforcement mechanism. Specifically, we analyze the following:

- What are the time and space overheads associated with our approach?
- How does our approach perform with respect to the Java runtime system's approach for enforcing access control?

We performed our experiments on a 266 MHz Pentium II running Red Hat Linux 5.0. The results show that both the time and space overheads of the approach are moderate. Further, the approach performs better than the Java runtime system in certain cases.

Overhead Measurements: We measured both the time and space costs of modifying resources.

There are four factors that affect the execution time associated with access constraint check code generation and editing:

- the cost associated with reading a method
- the number of access constraints
- the types of constraints
- the number of occurrences of restricted methods in a program

We do not consider the cost of reading class files in our measurements since the run-time system must perform this operation anyway.

In the first experiment, we looked at how the size of the method being modified affects the cost of editing. In this experiment, only a single method invocation must be wrapped. The cost of editing here is minimally affected by the size of the method. The cost varied between 0.08 and 0.16 seconds for methods ranging from 0 to 3200 instructions. In the second experiment, we looked at how the cost of editing changes when the number of method calls that needs to be wrapped changes. We found the cost to be proportional to number of methods that are wrapped.

We have also calculated the increase in size caused by adding code to class definitions. While the amount of code that is added to a class is independent of the size of the class, it depends on the number of method invocations that need to be wrapped and the complexity of the boolean portion of the constraint. For one wrapper, the minimum addition size (for a true boolean constraint), is 56 bytes. For two simple boolean expressions, it is about 206 bytes.

Performance Comparison: We now compare the performance behavior of our approach with the runtime system approach, as implemented in the JDK 1.1.3.

For this experiment we created a small program to test the performance of implementing security checks around one method invocation. Since the actual amount of work a particular site must perform depends on both the complexity of the access control policy and the number of restricted method invocations in a program, implementing a single policy statement once forms a good basis for comparison. We based our comparisons on the access control policy and classes from Example 3. The complete code for our approach is shown in Fig. 6. We implemented the same policy using Java's security manager as shown in Fig. 7. The test program calls the constrained method variable number of times. The access policy is that the method cannot be called more than 1000000 times.

Figure 8 shows the execution times of our approach and the Java's runtime system approach. In our approach, there is an initial overhead of about 0.08 seconds for code editing, which does not occur in the Java runtime system. However, after about 100000 method calls, our approach performs better than the Java runtime system. This is because our approach inlines the access control check code, whereas in case of the Java runtime system approach, each access

```
class SecState {
  public SecState() {count = 0;}
  public int check()
    { count++; return count; }
  private int count;
}
```

(a) Security object

```
add SecState SecurityState to R
deny  ↦  R.f()V when
  #1.SecurityState.check() > 1000000
```

(b) Control access constraints

Fig. 6. The binary editing approach

```
class newSecMan
  extends SecurityManager {
  public newSecMan() {count = 0;}
  public void checkf()
      throws SecurityException {
    count++;
    if (count > 1000000)
      throw new SecurityException();
  }
  int count;
}
```

(a) Security Manager

```
class R {
  public void f() {
    newSecMan security;
    security =
      System.getSecurityManager();
    if (security != null)
      security.checkf();
  }
}
```

(b) Resource definition

Fig. 7. The Java Runtime System-based approach

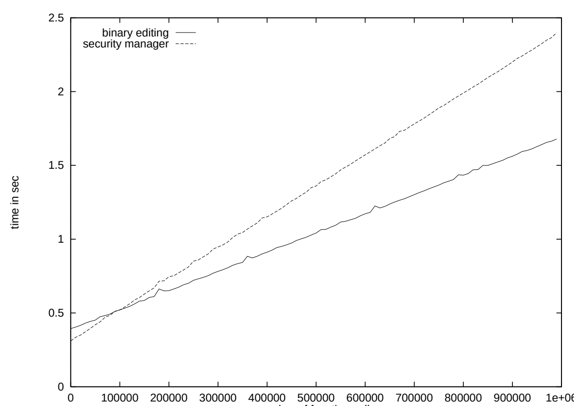

Fig. 8. Comparison of execution times with a policy

constraint check involves making two method calls: one to the system, to get the security manager, and another to the security manager itself. We can reduce our cost even further by pre-editing the methods if we know that only a single access constraint will be applied to the method, as is the case in the Java runtime system approach. Our approach, in this case, will then always outperform the Java runtime system approach.

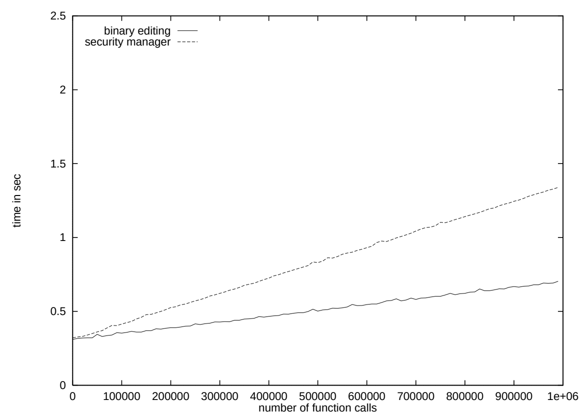

Fig. 9. Comparison of execution times without a policy

In the second experiment, we ran the same program with no policy implemented. As shown in Fig. 9, the Java runtime system is always less efficient that our approach. This is because in the Java runtime system approach, a method must always call the runtime system to check if there is a security manager installed, incurring the overhead of this call. Our approach does not incur any overhead since it does not add any code to methods that do not need to be constrained.

5 Related Work

In this section, we look at techniques that provide resource level access control. Much of the work on mobile program security has dealt with supporting different levels of security for Java programs. Therefore, we first consider Java's security model and various extensions to the model. We then turn to Safe-Tcl, an interpreter based security model. Finally, we discuss Proof Carrying Code, a language based approach.

Java: The initial security model [10, 22, 12] proposed by Sun for Java implements access control policies using a security manager. An access control policy is created by subclassing the SecurityManager class and setting this as the system's security manager. A site then ensures that all protectable resources make an explicit call to the security manager to check if access is allowed. If the check is not allowed, the security manager throws a security exception. Otherwise, the control returns to the calling method. This decision is based on whether the code is trusted, i.e. from the local file system, or untrusted, i.e. an applet downloaded from the net.

The primary difference between our approach and this approach is that the JVM specifies policies in a procedural form. This allows the use of the full range of Java's language to specify any type of policy. In our approach policies are specified in a declarative form. This allows for easier expression and analysis of policies. We also allow policies to include procedural aspects with the security state object.

However, the extensibility of the security manager is limited. Suppose there are other services that the system is providing which needs to be restricted. While it is possible to add methods to a subclass of the SecurityManager class that will do the necessary checks, adding the code to call these checks might not be easy, especially if the programmer did not design these services to do so. This problem is further exacerbated if the software is proprietary code provided by a third party. In contrast, our approach allows us to add security information to mobile programs that might not been designed with security in mind. Further, the security models can be customized on the basis of program, security and runtime states, and method parameters.

The approach in [20] extends the Java security model to implement a domain-based access model. In this model, Java programs are given an unforgeable SecurityToken used to identify their domain. An AppletSecurity object plays the role of the Security Manager. It uses the SecurityToken of the applet to determine the capabilities of that applet, throwing a security exception if the needed capability is not there. Other capability systems have been proposed by JavaSoft, Electric Communities, and [16]. Similarly, the approach in [28] provides a more flexible mechanisms for controlling accesses to resources. Our approach differs from these works in that we propose a framework for implementing various security models and policies, including the ones implemented in [20] and [28].

Sun redesigned their security model [13] in order to provide the security infrastructure for supporting fine-grained access control and configurable security policies. The new model augments the SecurityManager with an AccessController that checks if mobile programs have permission to access specific resources. Permissions are stated in a policy language that allows users to define protection domains based on what URL they came from and on who has signed them. Each protection domain is associated with a set of actions that they are allowed to do. Unfortunately, for old resources to take advantage of the new model, these resources must be re-implemented.

The J-Kernel project [19] extends the JVM security model by implementing multiple protection domains within a single Java virtual machine. It provides access to resources by passing capabilities for them to a system-wide repository. Domains can then look up capabilities from this repository. Capabilities are implemented as wrappers which provide the bookkeeping associated with changing protection domains.

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.

Naccio [9] provides a framework for specifying resource hooks, state maintenance code, and safety policies. State maintenance and access checks are performed by adding wrappers. Programs are transformed to use these wrappers instead of the original library code.

Grimm and Bershad [14] describe an access control mechanism consisting of an enforcement manager and a security policy manager. The system is divided into protection domains. The mechanism examines the system and redirects invocations to access control checks. The security model is based on DTE.

Interpreter-Based Approaches: Safe-Tcl [23, 32, 15] requires at least two interpreters: a regular (or master) for trusted code and a limited (or safe) one for untrusted code. The designers of Safe-Tcl classified a set of instructions as being unsafe and then disabled those instructions in the safe interpreter. When untrusted code needs to access a system resource, the safe interpreter traps into the master one. The regular interpreter then decides whether or not to allow the access. A security policy is specified by aliasing the disabled instructions in the safe interpreter to procedures in the master interpreter. These procedures can then check arguments and, if the security policy allows, call the the masked instruction in the master interpreter. Furthermore, Safe-Tcl allows a program to request a policy which the interpreter can grant to the program as appropriate.

Language-Based Approach: The approach taken in Proof-Carrying Code (PCC) [30, 29] is to associate a site specific security policy with a program by constructing a compiler that takes user programs and site specific policies and generates both the binary code and proof of the program's safety with respect to the specified policies. As an external program is migrated for execution at the kernel, the proof is validated, within the context of the site specific safety policy, at the kernel site. One advantage of this approach is that it is tamper proof. If either the program or the proof has been modified in transit, then there will either be a validation error, or the resulting PCC binary will still validate the policy. Also, since PCC makes the decision on whether a program is secure on properties of the code rather than properties of the code's origin, cryptography is not needed. Further, PCC proof checks are similar to type checkers. They are simple to implement, easy to trust, and very efficient. Unfortunately, this approach is not practical for enforcing host dependent policies. In this case, the host must communicate its policy to the site manufacturing the program and the manufacturing site must create separate proofs for each host. This is especially server for mobile programs which may visit many different sites each with a different security policy.

Security Policy Languages: The area of security policy languages has also focused on mechanisms for specifying and enforcing security. Security policy languages have been considered as the basis for verifying designs of secure systems. Various considerations have been given to policy languages for doing general enforcement.

Access control matrices (ACMs) [1] are a traditional means for specifying what is and is not allowed on a system. With ACMs, a two-dimensional matrix is given with the active entities, called subjects, in the rows and all the entities, or objects, in the columns. A list of access rights that a subject has over an object is given in the corresponding matrix cell. The language described in this paper can be used to describe an access control matrix, as well as the conditional state transitions described in [18].

Miller and Baldwin [27] 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 which 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, it 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 [11] 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. Cuppens, Saurel, and Cholvy [7, 5] use a form of deontic logic to express policies. In addition to specifying what actions an agent is permitted or forbidden to perform, it also allows statements that say what actions an agent is obliged to perform. They use deontic logic to find consistency problems between several 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 which can be implemented easily and efficiently.

The DIAMOND [31] security model provides an alternative model for inheriting security policies in object-oriented systems. This extends the MLS security model described by Denning [8] 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.

6 Summary

We have described a mechanism for implementing general security policies on mobile programs. There are two components of our approach. The first is a simple declarative access constraint language that allows a site to restrict accesses to the objects and methods of the system. The declarative nature of the language makes it easy to specify policies while still allowing a hook to express procedural policies if necessary. The second is a set of tools that enforce the specified constraints by editing mobile programs and resources. Our approach's appeal is that a site can specify access constraints separately from both mobile program definitions and resource definitions. This separation of concerns has a number of benefits. Both access constraints and resource definitions can be modified independently. Sites can easily specify different access constraints for different mobile programs for the same resource. Finally, our approach can enforce security on systems that were not originally designed with security in mind.

Our future work first involves generalizing our access control model to implement well-known security policies and constraints. We are developing mechanisms for facilitating the process of building security models using our approach. As part of our research in system software extensibility, we are considering various approaches for integrating our technique within the existing operating system and runtime system framework. Integration within the Java class loader is currently underway.

7 Acknowledgments

We thank Jeff Gragg and Raja Mukhopadhyay for help and support in implementing the system. We also thank Fritz Barnes, Earl Barr, Matt Bishop, Prem Devanbu, David Evans, Karl Levitt, Scott Malabarba, Ron Olsson, 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, 1994.

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

[3] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, pages 34–49, October 1995.

[4] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MOS '96*, number 1222 in Lecture Notes in Computer Science, pages 25–47, Linz, Austria, July 1997. Springer-Verlag. Also available at http://www.research.ibm.com/massdist/mobag.ps.

[5] Laurence Cholvy and Frédéric Cuppens. Analyzing consistency of security policies. In *1997 IEEE Symposium on Security and Privacy*, pages 103–112, Oakland, California, 1997. IEEE.

[6] T. Coombs, J. Coombs, and D. Brewer. *ActiveX Sourcebook: Build an ActiveX-Based Web Site.* John Wiley & Sons, Inc., 1996.

[7] Frédéric Cuppens and Claire Saurel. Specifying a security policy: A case study. In *9th IEEE Compuer Security Foundations Workshop*, pages 123–134, Kenmare, Ireland, June 1996. IEEE, IEEE Comput. Soc. Press.

[8] D. Denning and P.J. Denning. Certification of Programs for Secure Information Flow. In *Communcations of the ACM*, volume 20(7), pages 504–513. ACM, 1977.

[9] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1999.

[10] J.S. Fritzinger and M. Mueller. Java Security. JavaSoft White Paper, 1996. http://www.javasoft.com/security/whitepaper.ps.

[11] J.A. Goguen and J. Meseguer. Security policies and security models. In *In Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20, 1982.

[12] L. Gong. Java security: Present and near future. *IEEE Micro*, pages 14–19, May/June 1997.

[13] L. Gong, M. Mueller., H. Prafullchandra, and R. Schemers. Going beyong 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.

[14] R. Grimm and B.N. Bershad. Providing policy-neutral and transparent access control in extensible systems. Technical Report UW-CSE-98-02-02, Dept. of Computer Science and Engineering, University of Washington, 1998.

[15] S. Gritzalis and G. Aggelis. Security issues surrounding programming languages for mobile code: Java vs. Safe-Tcl. *Operating Systems Review*, 32(2):16–32, April 1998.

[16] D. Hagimont and L. Ismail. A protection scheme for mobile agents on Java. In *Mobicom '97*, pages 215–222, Budapest, Hungary, 1997. ACM.

[17] M.A. Harrison, W.L. R., and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[18] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[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] N. Islam, R. Anand, T. Jaeger, and J.R. Rao. A flexible security model for using internet content. *IEEE Software*, 14(5):52–59, Sept.-Oct. 1997.

[21] S. Jajodia, S. Pierangela, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 Symposium on Security and Privacy*, pages 31–42, 1997.

[22] JavaSoft. *JDK 1.1.1 Documentation*.

[23] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July-August 1997.

[24] S. Liang and G. Brach. Dynamic Class Loading in the Java Virtual Machine. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, Vancouver, October 1998. ACM.

[25] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

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

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

[28] N. Nagaratnam and S.B. Byrne. Resource access control for an internet user agent. In *Third USENIX Conference on Object-Oriented Technologies and Systems*. USENIX, June 1997.

[29] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages*. ACM SIGPLAN-SIGACT, Jan. 1997.

[30] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating System Design and Implementations*. Usenix, Oct. 1996.

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

[32] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl security model. Technical Report TR-97-60, Sun Microsystem Laboratories, 1997. Available at http://research.sun.com/technical-reports/1997/abstract-60.html.

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

[34] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

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

[36] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible security architecture for Java. Technical report, Department of Computer Science, Princeton University, 1997.

[37] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 33–50, 1992.