Providing Fine-Grained Access Control For Mobile Programs Through Binary Editing *

Raju Pandey Brant Hashii
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{pandey, hashii}@cs.ucdavis.edu

Technical Report TR-98-08

Abstract

With the advent of WWW, there is considerable interest in programs that can migrate from one host to another and execute. For instance, Java 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 the Java program and executes it at the user's site. 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 a host site's protected resources. For instance, they can potentially read a user's private files, access and modify personal information, and steal proprietary information.

In this paper, we present a novel approach for allowing a site to protect and control the local resources that external Java programs can access. In this approach, a site uses a declarative policy language to specify a set of constraints on accesses to local resources and the conditions under which they apply. A set of code transformation tools enforce these constraints on a Java program by integrating the code for checking access constraints into the program and the site's resource definitions. Executions of the resulting modified mobile program and resources satisfy all access constraints, thereby protecting the site's resources. Because this approach does not require resources to make an explicit call to a reference monitor, as implemented in the Java runtime system, the approach does not depend upon a particular runtime system implementation.

*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.

1 Introduction

With the advent of the WWW [3], there is considerable interest in developing runtime infrastructures for the mobile computing model. In the mobile computing model [5, 22, 4], programs, called *mobile programs* in this paper, have the ability to compute at a host, stop their execution, migrate to another host, and restart their executions. The mobile computing model is appealing because it supports efficient utilization of network resources and extensibility of information servers. Also, the model is ideally suited for extensible distributed system structures such as the Internet.

Although appealing from both system design and extensibility points of view, the mobile computing model raises a number of security concerns, namely:

- **Authorization:** Is the mobile program authorized to execute at a host?
- **Safety:** Does the mobile program have the ability to interfere with the executions of other programs (including the operating system and runtime systems) by reading and writing into their name spaces?
- **System resource allocation:** Does the mobile program have the ability to starve programs by consuming too much of system resources (such as cpu, memory and disk)?
- **Access control:** Can the mobile program access local resources that it is not allowed to access?

While a host must protect itself against all of the above security problems, our focus in this paper is primarily on the problem of access control. We emphasize that the access control problem for mobile programs is identical to the access control problem for applets [2, 11]. A solution for the access control problem for mobile programs is, thus, applicable for applets as well.

We describe the access control problem in the mobile computing domain by first examining how mobile programs access resources at a site. In figure 1, we show a program, P, that migrates to a host, H_1. The mobile program runtime system at the host constructs an execution image, P_e, of the program by binding the local resources (such as R_1, R_2, and R_3) with P. Note that the binding involves linking, either statically or dynamically, all code that H_1 provides for accessing the local resources. The runtime system then creates an execution environment for P and start its execution. During the execution, P accesses the local resources by invoking methods on them.

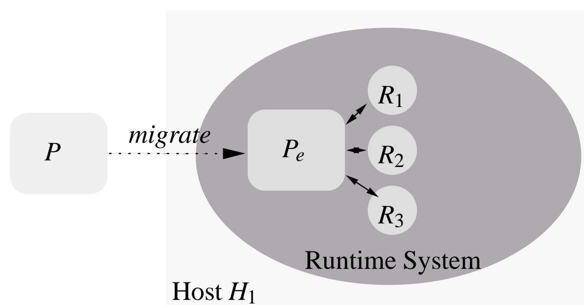


Figure 1: Resource accesses by mobile programs

The access control problem involves allowing a site to control a mobile program's ability to access resources such as R_1, R_2, and R_3. Traditional operating systems [12] implement a notion of access control by limiting accesses to specific resources that the operating systems administer. For instance, they allow users to impose restrictions on accesses to files they own.

We note that the access control problem in the mobile computing domain differs from the traditional access control models in many ways: First, there are no fixed set of resources that a site can administer; different sites may define different resources. An access control mechanism, thus, cannot be based on

controlling accesses to specific resources. It should be applicable to any resource that a host or a user may define and export. Second, the access control model should be customizable in that it should allow access control policies to be changed from one site to another, one mobile program to another and one resource to another. Third, the access control model should support a weaker notion of access control. In traditional access control models, access control involves either allowing an access or completely denying it. In the mobile computing domain, we argue for a *conditional access control* model where accesses to resources can be conditional [17]. In other words, a site may allow a mobile program to access resources if certain conditions are met. These conditions may depend on the state of mobile programs, state of resources, mobile program runtime 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 (number of mobile programs running) and a security state (number of times mobile programs access the database).

Access control specification and enforcement have been studied in great detail. The different approaches can be classified into three categories: *operating system-based*, *runtime system-based*, and *language-based* approaches. In the operating system-based approaches [12, 1], an operating system implements a specific access control model which specifies how system-wide resources such as files, the network and displays can be accessed. A site specifies its security policy within the framework of the security policy model. The operating system enforces the security policy by checking the type of access to resources. In runtime system-based approaches [6, 9], a runtime system enforces specific controls over accesses to various objects. Each method first calls a security manager (or permission controller) which checks to ensure that the method call is permitted. In language-based techniques [7, 24, 21, 14] access control policies are specified along with a program specification. A compiler not only generates code for the program but also for enforcing security policies.

While the OS-based and language-based approaches do support various levels of access control, each has certain limitations. Operating system-based approaches are limited in that the access control model applies only to the resources managed by the operating system. Usually, they cannot be extended to specify and enforce access control over user-defined resources. The language-based approaches are also limited in that they cannot be applied for enforcing constraints on mobile programs. This is because mobile programs have the ability to migrate from one site to another, each with different access control policies. In many cases, the access control policies are unknown. It is, thus, not possible to pre-compile such access control policies in the mobile program code.

The runtime system-based approaches (especially the Java runtime system's security model) are more extensible in that users can define different access policies for different resources. However, enforcement of access control policies require that implementations of resources make explicit calls to a security manager [6, 15] or an access controller [9]. This means that code for every resource must include an explicit access control check call. The approach, thus, will not work for those resources that are not designed with security in mind.

What is needed is a flexible and general mechanism for specifying and enforcing conditional access control. This paper presents such a conditional access control model along with mechanisms for specifying and enforcing access constraints. Specifically, the paper addresses the following:

- *What is the notion of access control in mobile programs?* The access control problem involves al-

lowing a site to control a mobile program's ability to access resources. We propose an access control model in which mobile programs can access specific resources if certain conditions are met. The paper also presents a model of inheritance for access constraints. An important aspect of our model is that access constraints are specified separately from both mobile program and resource definitions.

- *How can access constraints be enforced?* We present a novel mechanism in which access constraints are enforced by integrating access constraint checks directly into mobile program and host resource code. The idea of transforming binary code to enforce security is quite novel in that the access control mechanisms can be used to define and enforce access constraints to systems that were not designed with security in mind. We have implemented a version of the mechanism for mobile programs represented using the Java byte code [16].

- *What is the performance characteristics of our approach?* The performance results show that the overhead of our approach is moderate. Further, our approach performs better than Java's runtime system based approach in certain cases.

This paper is organized as follows: Section 2 contains a brief overview of our approach. Section 3 contains a description of our resource access model and how accesses to various resources can be specified. Section 4 describes an implementation of this model. Section 5 contains an analysis of the performance behavior. Section 6 relates this approach to other related work. Section 7 contains a summary of the approach and discussion of future work.

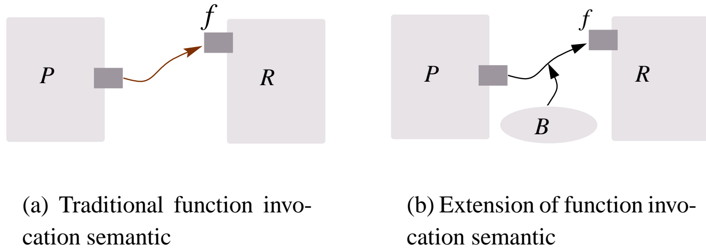
2 Overview

In this section, we give a brief overview of our approach. The approach implements a conditional access control model for mobile programs through two components: an access constraint specification language, and an access constraint enforcement tool.

2.1 Access constraint specification

The access constraint specification language is used to specify constraints over accesses to resources. In this section, we only present the motivation for the language. The details can be found in Section 3.

A mobile program accesses a resource by invoking methods on the resource. For instance, in figure 2(a), we show that program P invokes a method f to access resource R. During the execution, the control jumps to f, executes f, and returns back to P upon termination. This implements a simple access semantics in which there are no constraints on accesses to R through f. Such an implementation allows unrestricted access to R.



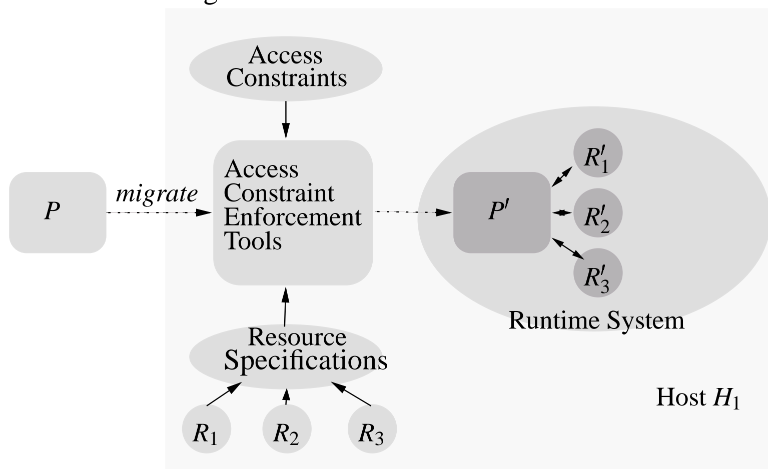
(a) Traditional function invocation semantic

(b) Extension of function invocation semantic

Figure 2: Invocations of functions

Our approach is to allow a host to strengthen the access relationship between P and R by adding a constraint, B (see figure 2(b)), that specifies that function f can be invoked only if constraint B permits

Figure 3: Access control enforcement



it. A site, thus, restricts accesses to its resources by enumerating a set of access constraints. The access constraints form the access control policy at the site.

2.2 Access constraint enforcement

The second component of our approach includes a set of tools that enforce the access constraints specified by a site. We describe the overall approach taken by the tools in this section. An implementation of the tools is described in Section 4.

In our approach, the tools enforce the access constraints by integrating the constraint checking code into the mobile program and resource codes. For instance, in figure 3, we show how the approach works. As P arrives at H_1, the access constraint enforcement tools examine P, local access constraints, and resource definitions to determine various access relationships. The tools then generates a set of constraint checking code and patch the code into P and R_1, R_2 and R_3. The modified program then executes and accesses resources through the constraint checking code. Executions of this code ensures that P can access a resource only if site-specified constraints are true.

We make a number of observations about the approach. In this approach, a site specifies access constraints separately from both mobile program definitions and resource definitions. This has implications on how access control code is managed and enforced at a site:

- Both access constraints and resource definitions can be modified independently from each other. 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 instance, 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 the approach can be used for enforcing security on resources that were not designed with security in the first place. In other words, the security compo-

nent can be added to a resource after it has been designed. In addition, it frees a library designer or resource designer from thinking about security aspects when designing and implementing the library.

The rest of the paper gives the details of the access control mode, the tools, and the performance characteristics of the approach.

3 Access control model

We now describe our access control model that a site can use for specifying access constraints. The access control model includes two elements: a resource model for defining resources that a site wants to control, and an access constraint specification language for expressing access control policies. We describe the two in detail below.

3.1 Resource model

A site provides a number of local resources to a mobile program. These resources include utility libraries, definitions for accessing files, networks and other devices, and interfaces to other resources such as proprietary databases. For instance, a site providing access to a weather database will export a set of interfaces that specify how the database can be accessed. We assume that sites use Java for defining the interfaces of the resources they export. Further, they use Java's inheritance model for defining the resource class hierarchy.

3.2 Access constraint specification language

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

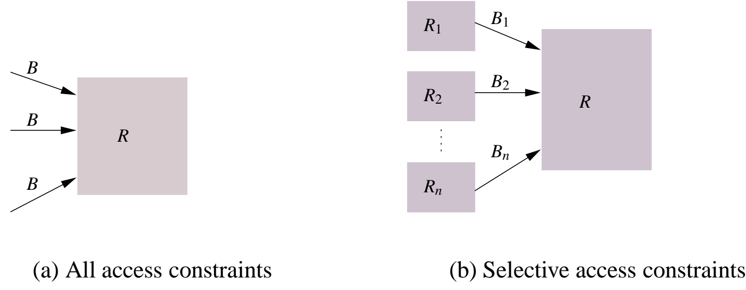
3.2.1 Access constraints

The basis for the language is derived from the observation that control over accesses to specific resources can be defined in terms of a set of constraints over access relationships between entities of a mobile program. The language provides a simple mechanism for specifying the conditions under which relationships among entities (such as classes and methods) of programs are valid. Below, we describe the syntax and semantics of the notation:

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

A site controls accesses to various objects by defining a set of access constraints. We describe the various terms in the grammar informally below:

- `Entity`: An entity denotes objects and method invocations of a mobile program. A `ClassIdentifier`, thus, identifies the set of objects to which a given access relationship applies. Similarly, a `MethodIdentifier` denotes the set of invocations of a method.



(a) All access constraints (b) Selective access constraints

Figure 4: Category of access constraints

- `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 $R_1 \dashv R_2$ is true if an entity R_1 creates an instance of class R_2.
  2. Invoke ($\mapsto$): A relation $R_1 \mapsto R_2$ is true if an entity R_1 invokes an entity R_2.

- `Condition`: The term `Condition` denotes a boolean expression that can be defined in terms of object states, program state (global state), mobile program runtime system state, security state, and parameters of methods.

Semantics: An access constraint of the form `deny` $(R_1\ \sigma\ R_2)$ `when Condition` specifies that if `Condition` is true, entity R_1 cannot access R_2 through relationship σ. Note that R_1 is optional. Hence, there are two kinds of access constraints: *all access constraints* and *selective access constraints*. All access constraints denote those constraints that do not depend on the initiator of the access relationship. For instance, as shown in figure 4(a), no object can access R when B is true. Selective access constraints denote those constraints that depend on the initiator of the access relationship. For instance, as shown figure 4(b), each R_i's access to R is constrained by a separate B_i.
Examples of all access constraints are shown below:

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.

The access constraint enforcement tools implement the two kinds of constraints differently.

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

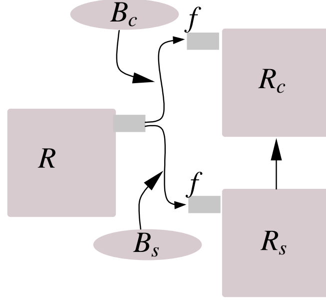


Figure 5: Inheritance of access constraints

unless a site specifically allows them. We call this model the *active permission model*. The active permission model certainly 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 users than allow illegitimate users.

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 (including local methods and library) calls, which can be quite cumbersome. Most approaches deal with this problem by requiring that if a site wants to enforce access control over a method M, they embed calls to an access controller checker within M. The checker enforces an active permission model over calls to M. 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. We note that the active denial model can be used to implement the active permission model by representing the permission conditions through the negation of denial conditions. However, this can be quite cumbersome. We are, therefore, looking at ways of integrating the active permission model in our language.

3.2.2 Inheritance of access constraints

We now present an inheritance model for access constraints. The inheritance model describes what denials to access resources mean in terms of denials to subclasses of resources. In figure 5, we show two classes, R_c and R_s. R_s is a subclass of R_c. Class R_c defines a method f which R_s inherits. The figure also shows the following access constraints on accesses to f from R:

$$\texttt{deny } (R \mapsto R_c.f) \texttt{ when } B_c$$
$$\texttt{deny } (R \mapsto R_s.f) \texttt{ when } B_s$$

In the inheritance model, constraints on access relationships are inherited by subclasses. However, a subclass cannot override the inherited access constraints. Access constraints can only be strengthened through additional constraints. Hence, the resulting access constraint on invocations of f on an object of R_c is:

```
deny (R ↦ R_S.f) when B_s ∨ B_c
```

In other words, method $R_s.f$ cannot be invoked from R if either B_c or B_s is true. This model of inheritance ensures that a mobile program cannot override the access constraints on methods by defining a subclass and by weakening the access constraint. Note that an access constraint defined in a class is not reflected in its super-classes. Note also that the 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.

3.2.3 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 3.1. *(File Access Control).* In this example, we implement the constraints on files that a mobile program can access. Assume that mobile programs access local files by invoking methods on the following class:

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

The following constraint specifies that file "/etc/passwd" cannot be opened by the mobile program:

```
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 the mobile program 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''))
```

■

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 3.2. *(Network access control).* Assume that the following defines the interface for making network connections:

```
Class 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 method `RuntimeSytem.Network.NumConnect` 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 (Runtime.Network.NumConnections() == UPPERBOUND)
```

■

In addition to the runtime system state, a runtime system may wish to store 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. For instance, a security state may capture information such as the number of times method `sqrt` has been called. Specific control over resources can, therefore, be specified by denying services on the basis of the security states of resources. We present an example below that illustrates this:

Example 3.3. *(Control over number of accesses).* Assume that we want to implement the constraint that a program `p` can invoke a function, say `f`, that locks a file at most ten times in order to prevent the effective use of a covert channel.

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 access constraint

```
deny (p ↦ f) when SecurityState.CheckCount(10)
```

specifies that p can invoke f at most 10 times. ■

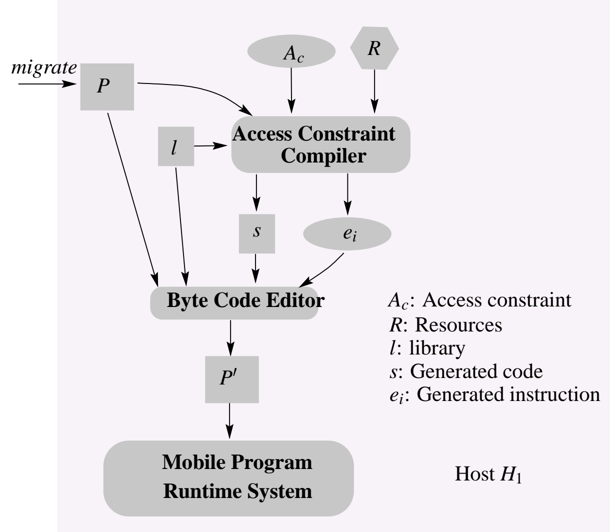


Figure 6: Security policy enforcement of mobile programs

4 Implementation

In this section, we describe an implementation of our approach. The implementation involves enforcing a set of access constraints on mobile programs represented using the Java byte code [16]. While the primary goal of our research project is to develop runtime techniques for supporting secure and efficient executions of mobile programs in general, our initial focus has been on developing support for mobile Java programs only. We have chosen the Java byte code as the intermediate representation language because a Java byte code-based representation of a class contains semantic information, such as types and access relationship, which the tools can use to enforce access constraints.

4.1 Enforcement of constraints

Access constraints are enforced by modifying a mobile Java program in such a way that the execution of the modified program satisfies the access constraints. In figure 6, we show the various steps that a mobile program goes through before it is downloaded in the runtime system. When a program (P) arrives at a site H_1, it is transformed into a program (P') such that an execution of P', in addition to implementing the execution semantics of P, satisfies the access constraints (A_c) imposed by the site (X). In this figure, R specifies interfaces for the resources and libraries that the site makes available to mobile programs. l denotes the local libraries that are linked into P. The transformation is achieved by two tools: an access constraint compiler and a byte code editor.

The primary goal of the access constraint compiler is to determine how the mobile program (P) and resource definitions should be modified in order to implement A_c. It does so by first determining the type information associated with various entities in the mobile program, libraries, and the exported interfaces. It then determines the various `instantiate` and `invoke` relationships between the objects specified in the access constraints. Finally, it generates code fragments (s) which implement specific access constraints, and a set of editing instructions (e_i) for the byte code editor. The binary editor uses e_i to integrate the generated

code (s) within P and libraries (l). The modified program (P') is then loaded in the runtime system and executed. We have separated the notion of editing from code generation in order to make the byte code editor a generic tool for modifying and transforming mobile Java programs.

We now describe how we extract type and access relationships from mobile programs, generate code, and edit the Java class file.

4.1.1 Type extraction

Type extraction involves examining Java class files to determine type definitions declared in the class files. Type definitions are used for constructing a resource model automatically from class files as well as for determining how a mobile program should be modified. This 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* section and the *method definition* section. 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 the 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.

4.1.2 Extraction of access relationships

The second step involves examining the mobile program code to determine invocation and instantiation relationships among different objects. The constraint compiler searches the bodies of methods for method invocation instructions. In the Java byte code, 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.

4.1.3 Code generation and binary editing

We now describe the nature of the code that is generated and its integration within mobile programs. Note that our code generation and editing involves modifying class definitions, in order to add security states and runtime states to mobile programs, and inserting runtime checks into methods. Care must be taken in transforming the code so that the interfaces of classes do not change.

An access constraint of the form

`deny (`R_1` Relation `R_2`) when `B

is implemented by generating the following code:

```
if (B) then error(); // raise exception
else access R2
```

and patching it into the mobile program and resource definitions. The nature of the editing depends on the nature of the access constraints. A constraint of the form

```
deny (Relation R2) when B
```

specifies constraints on accesses to R_2 without any regard to objects or methods that may access the resource. The generated code is, thus, integrated into the method definition of R_2. On the other hand, a constraint of the form

```
deny (R1 Relation R2) when B
```

imposes conditions on accesses to R_2 from R_1. The generated code is, thus, patched into all accesses to R_2 from R_1.

Security state objects are added to a class definition by using the statement

```
add SecurityStateType SecurityStateObject to R
```

In addition, this object is automatically initialized in the constructor for class R. An example of how such an object might be used is given in the performance analysis in Section 5.

4.1.4 Observations

There are a number of small details that must be taken care of whenever code is added into the middle of a method body. First, such additions run the risk of invalidating jump instructions. Java bytecode uses relative offsets for jumps. In other words, the operand for a jump instruction is relative to the current instruction and not to the beginning of the method body. Adding code between a jump instruction and its target requires the offsets to be modified. Our implementation reads the method body as a list of instruction objects. It then creates a second list of pointers to jump instructions and their destinations. After the byte code editor adds code to the method, the distance between these two pointers in the instruction list is recalculated. A similar technique is used whenever a part of the Java class file refers to offsets within the method bodies, such as the exception table.

In addition to fixing jump instruction, the Java virtual machine has two variable length instructions: `tableswitch` and `lookupswitch`. These instructions are used to implement switch statements. The problem is that they contain 0-3 pad bytes so that their first operand begins at an address that is a multiple of four bytes from the start of the current method. Thus, the number of these pad bytes needs to be re-calculated and re-assigned before the jump offsets are re-calculated.

The major problem we encountered in generating and patching security code arises due to conflicts between the Java programming language's support for control of inheritance and our model of inheritance for access constraints. For example, assume that class R_s is a subclass of R_c. class R_c defines a function f which is inherited in R_s. Also, assume that there is an access constraint of the form:

```
deny (↦ Rs.f) when B
```

Even though f is inherited from R_c, it needs to be modified in order to impose the new constraint. However, since policies are inherited down, and not up, the method body of f in R_c should not be modified. One possible solution is to copy the definition of f from R_c into R_s and proceed as before. However, if f is declared to be final, copying is not a solution as it will be rejected by the Java byte code verifier. Although it is possible to modify the class file for R_c to remove the 'final' constraint, such a change may lead to security holes. One solution is to modify $R_c.f()$ to add a runtime check to see whether the current class is an instance of R_s. If it is, then determine B and allow or deny access. If it is not, then proceed as normal. This solution is conceptually not elegant, as it requires a class to know about its subclasses. Another solution is to change all of the calls to f to call a new method f_1 which then conditionally calls f.

4.2 Implementation Details

In this section we describe the code generation and code editing process for different instances of access constraints. We assume, for the purposes of explanation, that condition B is true if the first parameter of R_2 is equal to 5. Note that condition B 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. For our implementation, the action amounts to throwing a security exception, although it could conceivably be any programmable action, such as writing to an audit log, ending the mobile program, or even moving it to another site. These assumptions, thus, does not change the emphasis of our exposition.

4.3 Implementation of all access 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 (↦ R1.f(I)V) when (#2.#1 == 5)
```

Recall that the term `#2` refers to the entity being invoked. Thus, the term `#2.#1` refers to the first parameter of that method. Also note that the `(I)V` following `R1.f` is the Java representation of the signature of that method. This constraint is enforced by generating code of the form shown in Figure 7 and patching the code into the body of `f` defined in the class file associated with `R1`.

In Figure 7, 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 Figures 7 and 8 indicates the `i`th entry in the constant pool. In code segment A of Figure 7, `#67` indexes the integer constant 5, whereas `#65` in code segment B indexes the entry for a `SecurityException` class and `#66` indexes the entry for its constructor.

Code segment A (Figure 7) contains the code for performing the conditional check, 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. Note that 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 (⊣ R2) when B
```

specify that an instance of `R2` cannot be created if B is true. They are implemented by putting constraints on invocations of all constructors of `R2`, which, in the Java byte code, are given a special name `<init>`. This

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

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

```
original code for
method R1.f(I)V
```

Figure 7: The modified method R1.f(I)V

case is, thus, implemented by adding code similar to that shown in figure 7 to all methods with the name `<init>`.

4.4 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 (R1.g()V ↦ R2.f(I)V) when (#2.#1 == 5)
```

Binary editing here involves first searching for all invocations of `R2.f(I)V` within the body of `R1.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 `R2.fIV`, then the generated code is inserted before the invoke opcode.

Note that the access relationship determined in this manner may only be partially correct due to the dynamic binding of methods. It occurs in the following situation. Assume the inheritance hierarchy of figure 5. Also, assume that method `f` is invoked on an object `O` of type *C*:

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

If entity `O` references an object of type *C* or type *S*, and constraint *B* is defined for the method of class *C*, the above describe 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 *S* and object `O` may reference objects of type `C` or type `S`. Note that if it references objects of type `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 `S`, the generated code should be added in order to implement the constraint. Since the reference type of `O` 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 Figure 8.

```
rest of R2.f(I)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
30 if_icmpeq 36                  C
33 goto 44

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

44 invokevirtual #10             E

rest of R2.f(I)V
```

Figure 8: The modified method R2.f(I)V

The first step (code segment *A*) is to access the object reference by popping off the operand stack (containing method parameters and object reference) into local variables. It also pushes them back on the stack (in case the method is called later). Notice that this might 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 `O` operation, and jumping to the method call if the object is not of type `R2`. Term #3 is an index into the constant pool that refers to the class `R2`. 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 `R2`. Other instances of access constraints can be implemented using the above technique.

- The access constraint

  `deny (R1.g()V` $\mapsto$ `R2) when` *B*

  is used to prevent a method from invoking any method of class R2. This can be implemented in the same manner as the previous case, except that it is carried out on all method invocations of `R2`.

- The access constraint

  `deny (R1` $\mapsto$ `R2.f(I)V) when` *B*

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

(a) Security object

```
add SecState SecurityState to R2
deny ↦ R2.f()V when
  #1.SecurityState.check() > 1000000
```

(b) Control access constraints

Figure 9: The binary editing approach

is used to prevent any method of class R1 from executing method f of class R2. This is also similar to case 3. The difference is that the method body of every method defined in R1, is search for invoke opcodes.

- The access constraint

  `deny (R1 ↦ R2) when` B

  is used to prevent any method of class R1 from executing any method of class R2. The algorithm used here is a combination of previous cases, in which every invoke opcode regardless of the method name and signature within every method of R1 is searched for method invocations of class `R2` and replaced with appropriate code.

5 Performance Analysis

In this section, we present the performance analysis of our approach. Specifically, we are interested in analyzing the following:

- What are the time and space overheads associated with the approach?
- How does the approach perform with respect to 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 Java's runtime system in certain cases.

5.1 Performance comparison

We first compare the performance behavior of our approach with the runtime system approach, as implemented by 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

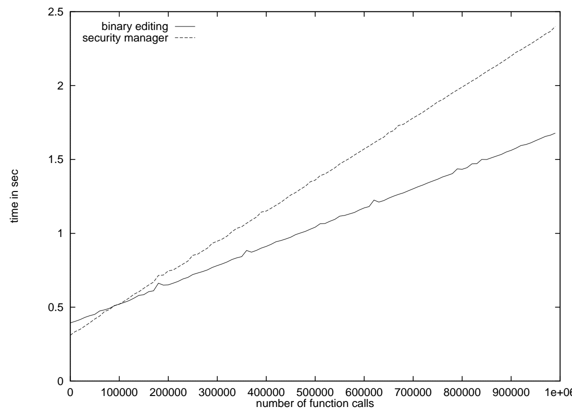
```
class newSecMan extends SecurityManager {
  public newSecMan() {count = 0;}
  public void checkf()
      throws SecurityException {
    count++;
    if (count > 1000000)
      throw new SecurityException();
  }
  int count, n;
}
```

(a) Security Manager

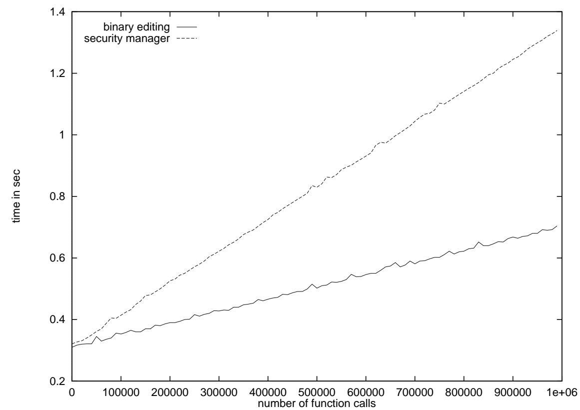
```
class R2 {
  public void f() {
    newSecMan security;
    security = System.getSecurityManager();
    if (security != null)
      security.checkf();
  }
}
```

(b) Resource definition

Figure 10: The Java Runtime System-based Approach



(a) Comparison of execution times with a policy



(b) Comparison of execution times without a policy

Figure 11: Comparison of executions times of our approach and Java runtime system approach

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 3.3. The complete code for our approach is shown in figure 9. We implemented the same policy using Java's security manager as shown in figure 10. 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 11(a) 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 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 Java runtime system approach.) Our approach, in this case, will then always outperform the Java runtime system approach.

In the second experiment, we ran this experiment with no policy implemented. As shown in figure 11(b), the Java runtime system always performs worse 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.

5.2 Overhead measurements

We have measured both the time and space related cost of modifying resources.

There are four factors that affect the execution time associated with access constraint check code generation and editing: cost associated with reading a method, the number of access constraints, types of constraints and the number of occurrences of restricted methods in a program. Note that we do not consider the cost of reading classfiles 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. Note that in ths experiment, only a single method invocation needs to be wrapped. Hence, the cost of editing here is minimally affected by the size 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 need to wrapped changes. We found the cost to be proportional to number of changes required in methods.

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. Again, we do not expect the size of the code to bloat as is the case when compiler inlines large amounts of code repeatedly.

6 Related work

In this section, we will look only at techniques that provide service level access control. Much of the work on mobile program security has dealt with supporting different levels of security for Java programs. We, therefore, first look at Java's security model and various extensions to the model.

The initial security model [6, 15, 8] 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. Once it is set for an application, it cannot be reset. A site then ensures that all protectable resources make an explicit call to the security manager to check if access are 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 main difference between our approach and theirs is that the JVM specifies policies in a procedural form. This allows using 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 declarative policies to include procedural aspects with the security state object.

The approach in [13] 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 [10]. Similarly, the approach in [18] 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 [13] and [18]).

Sun recently redesigned their security model [9] in order to make the security model more flexible. The model replaces the Security Manager with an `AccessController` that checks if mobile programs have the permission to access specific objects. The security model defines a policy language that allows users to state which principles are allowed access to which objects.

Type hiding [23] is a modification of the dynamic linking process in Java in order to hide or replace classes seen by an applet. For example, suppose the policy says that applets should not have access to the local file system. Type hiding allows a class associated with accessing files to be replaces by a proxy class that may throw an exception when accessed. In addition, actual classes can be replaced with proxy classes that check their arguments and conditionally call original methods. This approach is similar to our approach in that it will allow a site to prevent a class from invoking the methods of another class. However, it doesn't have the same level of granularity that we do, nor does it allow the addition of state information to mobile programs.

The approach taken in the Fox Project [20, 19] 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. Unfortunately, this approach is not suited for mobile programs where different sites will have different security policies. Since proof generation is not automated, the proofs will need to be re-generated by hand for each site.

7 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 easier to specify 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. The approach is appealing in that a site specifies access constraints separately from both mobile program definitions and resource definitions. This means that both access constraints and resource definitions can be modified independently from each other. This makes it easier for a site to specify different access constraints for different mobile programs for the same resource. Another important advantage of the separation is that the approach can be used for enforcing security on systems that were not designed with security in the first place.

Our future work first involves generalization of the access control model to implement well-known security policies and constraints. Also, 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 looking at techniques for integrating our technique within the existing operating system and runtime system framework. Integration within the Java class loader is currently underway.

References

[1] Edward 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] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and others. The World-Wide Web. *CACM*, 37(8):76–82, August 1994.

[4] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *ICSE '97. Procceddings of the 1997 international conference on Software engineering*, pages 22–32, Boston, MA, May 1997.

[5] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, IBM T.J. Watson Research Center, 1995.

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

[7] 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.

[8] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, pages 14–19, May/June 1997.

[9] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland 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.

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

[11] M.A. Hamilton. Java and the shift to net-centeric computing. *IEEE Computer*, pages 31–39, August 1996.

[12] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

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

[14] 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.

[15] JavaSoft. *JDK 1.1.1 Documentation*.

[16] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[17] Donald V. Miller and Robert 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.

[18] 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.

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

[20] 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.

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

[22] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[23] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. Technical report, Department of Computer Science, Princeton University, 1997.

[24] 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.