Role-Based Access Control (RBAC) in Java via Proxy Objects using Annotations

Jeff Zarnett
jzarnett@uwaterloo.ca

Mahesh Tripunitara
tripunit@uwaterloo.ca

Patrick Lam
p.lam@ece.uwaterloo.ca

Department of Electrical & Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

We propose a new approach for applying Role-Based Access Control (RBAC) to methods in objects in the Java programming language. In our approach, a policy implementer (usually a developer) annotates methods, interfaces, and classes with roles. Our system automatically creates proxy objects which only contain methods to which a client is authorized access based on the role specifications. Potentially untrusted clients that use Remote Method Invocation (RMI) then receive proxy objects rather than the originals.

We discuss the method annotation process, the semantics of annotations, how we derive proxy objects based on annotations, and how RMI clients invoke methods via proxy objects. We present the advantages to our approach, and distinguish it from existing approaches to method-granularity access control in Java. We demonstrate empirical evidence of the effectiveness of our approach by discussing its application to software projects that range from thousands to hundreds of thousands of lines of code.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*

General Terms

Security, Management

Keywords

Access Control, Java, RMI, Proxy Objects, RBAC

1. INTRODUCTION

Access control regulates accesses to resources by principals. It is one of the most important aspects of the security of a system. A protection state or policy contains all information needed by a reference monitor to enforce access control. The syntax used to represent a policy is called an access control model. Over the past decade and a half, Role-Based Access Control (RBAC) [1] has emerged as the dominant access control model in enterprise settings.

In RBAC, principals are called users. Users get permissions to access resources via membership in roles. We present a new way to realize RBAC in Java. Java is a widely-used programming language, and therefore is an important context in which to realize RBAC. We are not the first to make this observation (see, for example, [2]); however, our approach has several advantages over previous approaches.

Java is an object-oriented programming language. A programmer declares and implements classes; objects are instances of classes. Classes comprise methods and data. A good programming practice is to ensure that all accesses to an object are via methods it exposes, thereby respecting the principles of encapsulation and information hiding.

We provide a mechanism for access control to methods in Java objects. There are built-in mechanisms for program structuring in Java that resemble access control. For example, it is possible to specify that only some methods are public, while others are private. Private methods may be invoked only by other methods within the class. Note that access control based on method visibility is not secure: Java Reflection can circumvent the method visibility rules.

Method visibility mechanisms are coarse-grained: for instance, any class and method may invoke methods that are denoted as public. Several other proposed approaches augment the basic access control features in Java. (See Section 6 for a more comprehensive discussion of related work.) Stack inspection [3], for example, can be used to provide method-specific access control, but does not cope well with remote clients.

We focus on clients that use Remote Method Invocation (RMI). Our approach works as follows. (See Section 2 for details.) We use Java *annotations* [4] in our approach. Annotations enable developers to associate arbitrary metadata—in our case, access control metadata—with code. In particular, developers annotate methods, interfaces, and classes in the Java source code with roles from the developer-specified RBAC policy. We test for the presence of annotations after the source code is compiled.

At compile-time, our system builds interfaces according to the specified policy. At run-time, we create proxies matching these interfaces and give them to clients instead of granting them access to the original objects.

Our approach provides more fine-grained access control than previous approaches: a proxy object exposes only those methods to which the client is authorized. Consequently, an

SACMAT'10, June 9–11, 2010, Pittsburgh, Pennsylvania, USA.

RMI client is unaware of the existence of other methods in the class. This has an additional efficiency side benefit: we preclude the client from even invoking a method to which it is not authorized, and thus, large arguments need not be sent over the network. Other approaches, such as bytecode editing [5], check at the server whether the call is authorized after the client has invoked the method and the arguments have been transmitted.

Furthermore, we bind clients to roles, and clients have access to only those methods whose roles correspond with the ones to which they are authorized. This is a significant improvement over our previous, preliminary work [6], which only supported coarse-grained "accessible" or "not accessible" annotations for methods, interfaces and classes.

The contributions of this paper include:

- the idea of using proxy objects for Role-Based Access Control in Java applications with RMI;
- the use of Java annotations for specifying RBAC policies directly in the code of Java implementations;
- formal semantics for our annotations; and
- our experience using our system to annotate three Java applications.

The remainder of the paper is organized as follows. Section 2 presents a simple example and describes our system in prose and using First Order Logic. Section 3 contains the implementation details of our system. We present an overview of the performance analysis in Section 4 and present a case study of three sample programs in Section 4.1. We present future enhancements in Section 5, related work in Section 6, and conclude in Section 7.

2. OUR APPROACH

In this section, we first present an example, and then discuss our approach to realizing RBAC in Java. We discuss the specification of RBAC policies in our approach and how annotations denote roles. We discuss also the security benefits from using our approach. In Section 2.2, we present precise semantics of our approach in first-order logic.

In our approach, we build proxy objects from the real objects, and allow partially-trusted users to directly access the proxies only. For each role, we derive a proxy object. A client may be bound to more than one role, and may successfully request the proxies that are associated with every role to which the client is bound. A proxy implements only those methods to which the role with which it is associated is authorized. Whether a role is authorized to a method or not is inferred from annotations in the source code.

Annotations in Java are metadata regarding the code they accompany. In the context of this paper, an annotation indicates a role. Annotations may be associated with an interface, a class, or a method (within an interface or a class). We first discuss how we derive the effective annotation on a method informally, and more precisely in Section 2.2. The only clients to which a method is authorized are those that are members of roles in the method's effective annotation.

We also provide a mechanism for specifying the roles to be used in annotations. Developers may associate roles with one another in a role hierarchy [1]. A role hierarchy, denoted as RH, is a partial ordering of the roles. If $\langle r_1, r_2 \rangle \in RH$, then we say that r_1 subsumes r_2. The consequence is that if r_2 is in the effective annotation of a method m, and a client is authorized to r_1, then the client is authorized to m.

Example.

We present an example of an RBAC policy, and a discussion of its use in a snippet of Java code. In this scenario, we wish to assign some rights to one set of users, and different rights to another set of users.

Figure 1 presents a role hierarchy. We have five roles, **IT Management**, **Accounting**, **IT Employees**, **Human Resources** and **Everyone**. The arrows indicate role *subsumption*. That is, everyone in **IT Management** is an IT Employee (and therefore in **IT Employees**), and every member of **Accounting**, **IT Employees** and **Human Resources** is a member of **Everyone**. (We use the term "subsumption" rather than "inheritance" as is customary in RBAC. We reserve the term "inheritance" for Java's notion of inheritance.)

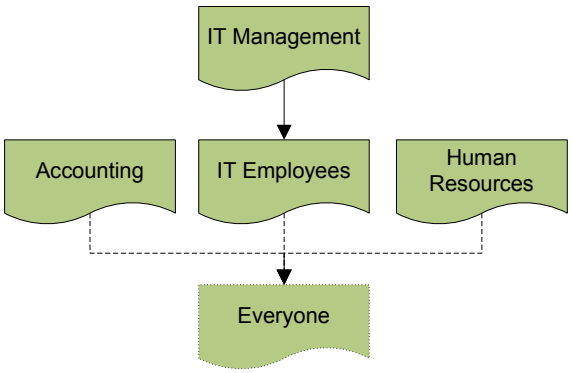

Figure 1: Roles in the Separation of Privilege Scenario. Arrows indicate a subsumption relationship.

The Java code we intend to protect follows.

```
struct Item {
    String name;
    double quantity;
    String unit;
    double price;
}
class Order {
    // Constructor
    Order(List<Items> items) { ... }

    void approve() { ... }
}
```

The code includes a class `Order`, which has two methods, `Order()` and `approve()`. (Following convention, we write method names with parentheses.) The method `Order()` is called a constructor, and executes upon creation of an object of the class `Order`. The method `approve()` is invoked by a client that wants to approve an order that exists.

We require that only members of **ITEmployees** may create orders, and only members of the role **Accounting** may approve orders. We assume that an object of type `Order` on which all clients operate resides on a server, and all clients invoke methods on it via RMI. We next discuss how our system supports and enforces this policy.

Effective annotation.

As we mention above, we allow a method, interface or class to be associated with an annotation. We infer the effective annotation on a method as follows. We have adopted the approach of "most specific annotation applies" in how we infer the effective annotation on a method. We present these rules precisely in Section 2.2.

If a method has an explicit annotation when it is defined in a class, then that explicit annotation is the effective annotation of the method. If a method is inherited from a superclass, but is not explicitly defined in the subclass, then the annotation in the superclass is its effective annotation. The reason is that in Java, when a method that is defined only in the superclass is invoked on an object of the subclass, it is the method from the superclass that is invoked.

If a method is defined in a class but has no annotation on it, and the class has an annotation, then the effective annotation on the method is the annotation on the class. We point out that a class may have no annotation. In this case, we do not infer its annotation from a superclass, if that class indeed inherits another class. The reason is that, as a design choice, we have chosen to disassociate the semantics of security from the intended semantics of class inheritance.

We allow interfaces to be annotated as well. An interface as a whole may be annotated, as may individual methods that are declared within it. If the interface is annotated, then that annotation affects methods declared within the interface only if a method does not have an annotation of its own within the interface. Annotations on an interface do not carry over to subinterfaces. If a method is re-declared in a subinterface, then its annotation in the subinterface affects classes that implement the subinterface.

Annotations on interfaces have different semantics than annotations on classes and methods within classes. Because Java interfaces constrain the contents of classes which choose to implement them, we decided to interpret annotations on interfaces as constraints on annotations for methods defined within classes that implement those interfaces.

Suppose we infer an effective annotation of the set of roles R on a method m that is defined in class c. Suppose also that we infer an annotation R' on all interfaces that c implements that declare m. Then, we require that every role in R' subsumes some role in R. Consequently, a client that is authorized to a role in R' has access to the method. R may include other roles, and consequently authorize other clients as well; R' is a lower bound on the clients that may access the method. We impose this constraint at compile-time—if some role in R' does not subsume any role in R, then we declare a compile-time error.

Security.

As we mention in Section 1, we target our access control approach to RMI-accessible objects only. There exist some design approaches that allow granularity at the level of instances of the class, but in our design, all instances of a class are treated equally. Our threat agents are remote clients. We assume that the potential attacker is a client in a different Java Virtual Machine (JVM) [7] whose only way of accessing objects is via RMI. It is possible to adapt our approach to local clients to provide more limited access control. However, other approaches such as stack inspection [3] combined with resistance to code-modification may be more effective in thwarting local attackers.

By default, our policy is "deny all." Annotations can be seen as selective "allow" rules. As only the methods to which a client is authorized are implemented in the proxy's interface (see the next section for details), our approach hides even the existence of other methods in an object. We see this as both a security and a performance benefit. From the standpoint of security, we have some measure of confidentiality in addition to access control. From the standpoint of performance, the server needs no run-time check to verify whether a client's method invocation is authorized.

We point out also that our solution is impervious to attacks that use Java reflection. Reflection allows a program to observe and modify its own state at run-time, and can be used to bypass security features (e.g., to read the internal state of an object). With reflection, methods marked private are discoverable and invocable, and an attacker may manipulate the internal state of objects.

To justify our assertion, we consider a proxy object that is accessed remotely using RMI. Although the proxy object keeps a reference to the original, the original remains inaccessible from the standpoint of reflection, because reflection cannot be used on remote objects [7]. RMI hides all fields of the object at the server from remote clients; fields do not appear on the client-side stub. As we mention above, proxies implement only those methods to which a role is authorized. As long as a client is able to access only the proxies that are associated with roles to which it is authorized, it is unable to glean information even about the existence of other methods or fields using reflection.

2.1 Annotations

In this section, we discuss how the developer defines roles and the role hierarchy, and how developers insert annotations into the program's code.

Roles and role hierarchy.

We declare a role using what we call a meta-annotation. The following code illustrates how to declare the role **ITEmployees** from Figure 1.

```
@Role
@Retention(RetentionPolicy.RUNTIME)
public @interface ITEmployees { }
```

This code segment is an example of a basic role annotation. `@Role` is a standard Java annotation, used as metadata. Annotations are syntactically denoted by the `@` symbol, and thus the first line applies the `Role` annotation to the current class. The second line is an annotation we add for the compiler that specifies that the `ITEmployees` annotation's presence should be observable at run-time, and the compiler will therefore propagate that annotation to the output class file. Thus, when executing, the program could check if this annotation is present on a method or class. Finally, the syntax for declaring an empty annotation appears on line three. Empty annotations serve solely as markers attached to a method, interface, or class.

We choose to annotate a role declaration with the roles it subsumes as our method of declaring the subsumption relationships. This method is simple and concise, and at compile-time, we can automatically compute the role hierarchy based on the subsumption annotations. The following snippet of code expresses how we declare a role subsumption relationship; **ITManagement** subsumes **ITEmploy-**

ees. This means a user in IT management has all of the rights of a user in IT employees. Note that the following annotation declares a new role and re-uses an existing role in a subsumption relationship.

```
@Role
@ITEmployees
public @interface ITManagement { }
```

To declare that two roles r_1 and r_2 subsume r_3, we annotate the declarations of r_1 and r_2 with r_3—we write "@r_3" above "public @interface r_1 { }" and above "public @interface r_2 { }". To declare that r_4 subsumes r_5 and r_6, we write @r_5 @r_6" above "public @interface r_4 { }". Subsumption is transitive, so in a system where r_7 subsumes r_8 and r_9, and a role r_{10} is introduced which should subsume roles r_7, r_8, and r_9, it suffices to declare that r_{10} subsumes r_7.

Annotations on classes, methods and interfaces.

The following snippet of code illustrates how we annotate the class `Order` and its method `approve()`. `Order` is annotated with the role **ITEmployees** and `approve()` is annotated with the role **Accounting**. We refer the reader to our earlier discussion under "Effective annotation" and the following section for the semantics of such annotations.

```
@ITEmployees
public class Order {
    // Constructor
    Order(List<Items> items) { ... }

    @Accounting
    void approve() { ... }
}
```

It is possible to annotate a class or method with more than one role. Simply list the names of the roles in sequence, each preceded by `@`, above the definition for the class or method.

As we mention above under "Effective annotation," we may annotate interfaces and method declarations within interfaces as well. The following snippets show an example of a method, `getSalary()`, that is declared within an interface `IHiringRequest`, annotated with the role **HumanResources**. We show also a class, `HiringRequest`, that implements `IHiringRequest`. As we clarify in our earlier discussions under "Effective annotation," and more precisely in the following section, we require, at compile-time, the effective annotation on `getSalary()` to include **HumanResources**. In this example, we meet this requirement by explicitly annotating `getSalary()` in its definition in the class `HiringRequest` with **HumanResources**.

```
interface IHiringRequest {
   @HumanResources
   public Money getSalary();
   ...
}

class HiringRequest implements IHiringRequest {
   @HumanResources
   public Money getSalary() { ... }
   ...
}
```

The expected annotation on `HiringRequest.getSalary()` is `@HumanResources`; that is, users who are members of the human resources group must be able to invoke `getSalary()`. The annotation's presence on the interface's method definition implies that any implementations of `getSalary()` must also be annotated with `@HumanResources` or any role that `@HumanResources` subsumes; if any implementation is not so annotated, that is a compile-time error.

2.2 Semantics

To express the semantics of our approach precisely, we use First Order Logic [8].

Annotations in class and interface definitions.

Our annotation inference rules use the following predicates: (1) $\mathsf{effectiveAnnotation}(m, c, r)$ is used to express that the role r is in the effective annotation of method m in class c; (2) $\mathsf{subsumes}(r_1, r_2)$ is used to express that the role r_1 subsumes r_2. We note that the subsumption predicate is reflexive, so $\mathsf{subsumes}(r_1, r_1)$ is always true. (3) The predicate $\mathsf{annotatedCorI}(ci, r)$ is used to express that the class or interface ci is annotated with r—the annotation may be explicit or inferred; (4) $\mathsf{annotatedMethod}(m, ci, r)$ is used to express that the method m is explicitly annotated with the role r in the class or interface ci; (5) $\mathsf{definedIn}(ci, m)$ is used to express that the method m is defined or declared in the class or interface ci respectively; and (6) $\mathsf{extends}(ci_2, ci_1)$ is used to express that the class or interface ci_2 extends (or inherits) the class or interface ci_1.

Note that our predicate $\mathsf{definedIn}(ci, m)$ is true for at most one ci per m. For example, consider a scenario where c_3 extends c_2 and c_2 in turn extends c_1; both c_1 and c_2 define method m. We say that $\mathsf{definedIn}(c_3, m)$ is false because m is not explicitly defined in c_3. $\mathsf{definedIn}(c_2, c_3.m)$ is true, as there is an explicit definition for m in that class; however, $\mathsf{definedIn}(c_1, c_3.m)$ is false. This follows the Java language semantics; a call to $c_3.m()$ results in an effective invocation of the method $c_2.m()$ and not $c_1.m()$.

Having defined the predicates, we next use them to state the rules defining effective annotations.

$$\mathsf{subsumes}(r_3, r_1) \leftarrow \mathsf{subsumes}(r_3, r_2) \wedge \mathsf{subsumes}(r_2, r_1) \quad (1)$$

$$\mathsf{annotatedCorI}(ci, r) \leftarrow \mathsf{annotatedCorI}(ci, r') \wedge \mathsf{subsumes}(r, r') \quad (2)$$

$$\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{annotatedMethod}(m, ci, r) \quad (3)$$

$$\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{effectiveAnnotation}(m, ci, r') \wedge \mathsf{subsumes}(r, r') \quad (4)$$

$$(\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{annotatedCorI}(ci, r)) \leftarrow \mathsf{definedIn}(ci, m) \wedge \forall r \neg \mathsf{annotatedMethod}(m, ci, r) \quad (5)$$

$$\mathsf{effectiveAnnotation}(m, ci_2, r) \longleftarrow \neg\mathsf{definedIn}(ci_2, m) \wedge \mathsf{effectiveAnnotation}(m, ci_1, r) \wedge \mathsf{extends}(ci_2, ci_1) \quad (6)$$

Rule (1) expresses that role subsumption is transitive. Rule (2) implements the role subsumption relation on class and interface annotations: if r subsumes r' and a class or interface ci is annotated with r', then we infer that ci is annotated with r as well. Rule (3) ensures that explicit annotations are also effective annotations: role r is in the effective annotation of method m in class c if it is an explicit annotation on m's definition in c. Rule (4) implements role subsumption for effective annotations: if r subsumes r' and r' is in the effective annotation of m in c, then so is r. Rule

(5) causes class annotations to affect unannotated contained methods: if a method m, defined in class c, has no annotations, then the annotations on c are the effective annotation on m. Finally, Rule (6) computes effective annotations for inherited methods: if method m is not defined in class or interface ci_2, but has an effective annotation in a class or interface ci_1 that ci_2 extends (or inherits), then that effective annotation is also an effective annotation for m in ci_2.

Constraints from interfaces.

As mentioned earlier, annotations on an interface and on method declarations in interfaces constrain the effective annotations on their implementing methods in classes. We specify the rule pertaining to annotations on interfaces below. We introduce the predicate $\mathsf{expectedAnnotation}(c, m, r)$ to express the constraint that r is required to be in the effective annotation of m in c. We adopt the predicate $\mathsf{implements}(c, i)$ to express that class c implements interface i.

$$\mathsf{expectedAnnotation}(c, m, r) \leftarrow \mathsf{implements}(c, i) \wedge \mathsf{effectiveAnnotation}(m, i, r) \quad (7)$$

The rule expresses that if r is in the effective annotation of method m in interface i, and class c implements i, then r is required to be in the effective annotation of m in c. Note that m has an effective annotation in i only if (1) m is defined in i, or (2) i extends an interface that defines m. Also, because c implements i and m is defined in i, Java ensures that m must exist in c as a method, either by inheritance or by explicit definition.

Model.

Rules (1)–(7) have a well-founded semantics [9]. We use negation; however, the negation is stratified.

For a semantics, we specify a model, $\mathcal{M}$, and an environment or look-up table, l [8]. The set of concrete values, A, that we associate with $\mathcal{M}$ is $A = A_i \cup A_c \cup A_m \cup A_r$, where A_i is the set of interfaces, A_c is the set of classes, A_m is the set of methods and A_r is the set of roles. We assume that the sets A_i, A_c, A_m and A_r are disjoint. We consider only those environments in which our variables have the following mappings: the variables r, r', r_1, r_2 and r_3 map to elements of A_r, ci, ci_1 and ci_2 map to elements of $A_c \cup A_i$, c maps to an element of A_c, i maps to an element of A_i and m maps to an element of A_m.

To compute $\mathcal{M}$, we begin with a bare model $\mathcal{M}_0$, with A as its universe of concrete values. We populate the bare model $\mathcal{M}_0$ with relations that make our predicates concrete, by extracting values directly from the Java code. For example, when the class or interface ci_2 extends (in the Java code) ci_1, we add $\langle ci_2, ci_1 \rangle$ to $\mathsf{extends}^{\mathcal{M}_0}$. Similarly, we instantiate $\mathsf{annotatedMethod}^{\mathcal{M}_0}$ to those $\langle m, ci, r \rangle$ tuples such that the method m has the annotation r in its definition in the class or interface ci.

We define $\mathcal{M}$ to be the least fixed point from applying Rules (1)–(7) starting with $\mathcal{M}_0$. The following algorithm α computes $\mathcal{M}$. First, populate the sets $\mathsf{definedIn}^{\mathcal{M}_0}$, $\mathsf{subsumes}^{\mathcal{M}_0}$, $\mathsf{annotatedMethod}^{\mathcal{M}_0}$, $\mathsf{annotatedCorI}^{\mathcal{M}_0}$ and $\mathsf{extends}^{\mathcal{M}_0}$ from the Java code. Observe that $\mathsf{definedIn}^{\mathcal{M}_0} = \mathsf{definedIn}^{\mathcal{M}}$ and $\mathsf{annotatedMethod}^{\mathcal{M}_0} = \mathsf{annotatedMethod}^{\mathcal{M}}$.

Apply Rule (1) repeatedly to compute the transitive closure of $\mathsf{subsumes}^{\mathcal{M}_0}$, which gives $\mathsf{subsumes}^{\mathcal{M}}$. Then, repeatedly apply Rule (2) to get $\mathsf{annotatedCorI}^{\mathcal{M}}$. Next, repeatedly apply Rules (3) and (5), followed by Rule (6). Finally, compute $\mathsf{effectiveAnnotation}^{\mathcal{M}}$ by repeatedly applying Rule (4) and $\mathsf{expectedAnnotation}^{\mathcal{M}}$ by repeatedly applying Rule (7).

We assert that α indeed computes the least fixed point. This is because it exploits a topological ordering of the inference rules. The algorithm α runs in worst-case time quadratic in the input size, which is the number of classes, interfaces, methods and roles.

Consistency and correctness.

The next definition states the consistency property that we enforce at compile-time.

DEFINITION 1. *Our system is* consistent *if*

$$\mathsf{expectedAnnotation}^{\mathcal{M}} \subseteq \mathsf{effectiveAnnotation}^{\mathcal{M}}.$$

As we discuss in Section 2.1, annotations on interfaces impose constraints on the effective annotations on methods defined in classes. If the effective set of annotations on a method in an interface i is E_i, and the effective set of annotations on the corresponding method in a class c that implements i is E_c, the consistency property requires that E_i is a lower bound for E_c, that is, $E_i \subseteq E_c$.

We assert via the following proposition that $\mathcal{M}$ is correct. Correctness is characterized relative to $\mathcal{M}_0$; $\mathcal{M}_0$ expresses exactly what is specified in the Java code. A method in a class has in its effective annotation every role that it should (completeness) and no roles that it should not (soundness).

PROPOSITION 1. $(m, c, r) \in \mathsf{effectiveAnnotation}^{\mathcal{M}}$ *if and only if exactly one of the following three cases is true. Furthermore, if* (m, c, r_1) *and* (m, c, r_2) *are two elements of* $\mathsf{effectiveAnnotation}^{\mathcal{M}}$*, then the same case is true for both elements.*

1. *If m is defined in c, then:*
   (a) *m is annotated with some role r' in its definition such that r subsumes r'; or,*
   (b) *m has no annotations in its definition, and c is annotated with r', where r subsumes r'.*
2. *Otherwise, let $c_1, c_2, \ldots c_n$ be the superclasses of c, and let c_i be the superclass which explicitly defines m. Then m is annotated with r in its definition in c_i via one the above cases.*

Zarnett [10] provides a proof for this proposition.

Semantics of Invocation.

We express our semantics of invocation via the following rules. The predicate $\mathsf{invokes}(m_1, c_1, m_2, c_2)$ indicates that method m_1 in class c_1 invokes method m_2 in c_2. The predicate $\mathsf{canInvoke}(m, c, u)$ indicates that the user u is allowed to invoke method m in class c, and $\mathsf{member}(u, r)$ indicates that u is a member of the role r.

The predicate canInvoke is not restricted to the methods that a user is directly authorized to invoke; it includes methods invoked transitively. Thus, canInvoke includes all methods the user can effectively run. Consider a scenario where role r is not authorized to invoke `b()`, but may invoke `a()`,

and `a()` invokes `b()`. Users who are members of r can invoke `a()`, but our semantics also reflect that members of r can, in effect, invoke `b()`.

$$\mathsf{invokes}(m_1, c_1, m_3, c_3) \leftarrow \mathsf{invokes}(m_1, c_1, m_2, c_2) \wedge \mathsf{invokes}(m_2, c_2, m_3, c_3) \quad (8)$$

$$\mathsf{canInvoke}(m, c, u) \leftarrow \mathsf{effectiveAnnotation}(m, c, r) \wedge \mathsf{member}(u, r) \quad (9)$$

$$\mathsf{canInvoke}(m_2, c_2, u) \leftarrow \mathsf{canInvoke}(m_1, c_1, u) \wedge \mathsf{invokes}(m_1, c_1, m_2, c_2) \quad (10)$$

Rule (8) indicates that invokes is transitive: if m_1 invokes m_2 and m_2 invokes m_3, then m_1 invokes m_3. Rule (9) indicates that a user is allowed to invoke a method to which he is authorized via his role memberships. When we say "user," we mean the customary RBAC meaning of a user [11]; in our context, an RMI client maps to a user.

Finally, Rule (10) indicates that if a user may invoke a method m_1, and m_1 invokes m_2, then the user can (indirectly) invoke m_2 as well.

As the rules above express, a client may of course invoke a method to which it is authorized. In addition, a client may invoke other methods, but only via methods to which he is authorized. That is, if a client is authorized to invoke m_1, and in its execution, m_1 invokes m_2, then this is allowed even if the client is not authorized to m_2 via annotations. The reason we allow this is that we assume that m_1's invocation of m_2 is controlled—the client cannot directly control the parameters and the manner in which m_2 is invoked.

3. IMPLEMENTATION

In this section, we discuss the implementation aspects of our approach. We separate our discussions into a compile-time (Section 3.1) and a run-time component (Section 3.2).

Bytecode generation and modification are at the heart of our implementation. We modify RMI-enabled classes at compile-time and generate class files (compiled output) for interfaces we derive. Our routine for building interfaces is based on prior work [6]. We use a compile-time preparation step that takes place between compiling the source files and running the RMI Compiler (`rmic`). In this step, we employ an *interface builder* (see Section 3.1), which examines classes, derives an interface, and modifies that interface according to the RBAC policy the developer has specified.

3.1 Compile-Time Component

Our program, the Proxy Object RBAC Compiler (`porc`) automatically performs all the compile-time work. It first builds the role hierarchy, based on the role annotations (see Section 2 under "Roles and role hierarchy"). It then expands annotations according to the inference rules (e.g., associates class annotations with contained methods). Next, it examines the classes that implement the interfaces. It reports an error if an implementation lacks an annotation that an interface implies it should have. Finally, it infers the annotations on method implementations from the annotations on the classes in which they are defined.

Once the information about roles and annotations is computed, `porc` searches the code base and finds RMI-accessible objects, examines each of these objects and derives its remote interface. This derived interface enforces the rules described by the annotations (by hiding inaccessible methods). `porc` saves the interface as a new class file, and modifies the original object's class file to implement the derived interface. At run-time, the server simply enters a dynamically-created proxy object of the correct type into the RMI server. No modifications to the Java compiler or the RMI compiler (`rmic`) are necessary.

Figure 2 depicts this process on a sample class `Order`, in a version of our example from Section 2. We show selected roles from Figure 1 in Figure 2. In the figure, `javac`, `porc` and `rmic` are the three stages of compilation. The files `Accounting.java`, `Accounting.class`, `ITEmployees.java`, and `ITEmployees.class` are the roles, before and after compilation. All other boxes are files that contain source code or an intermediate representation. In the figure, we show only new output files and modified input files; unchanged input files do not appear as output of the processing steps.

In Figure 2, we assume that a developer has written the `Order.java` file, and compiled it and the roles into a class file using `javac`. The `porc` examines all the roles in our system and defines the role hierarchy. Next, `porc` identifies `Order` as being a remotely-accessible object, and processes it further by deriving interfaces corresponding to each role. In the figure, we show processing of the `Order` class for the roles of `ITEmployees` and `Accounting` only.

Each generated interface (e.g., `IOrder_Accounting`) contains only the methods authorized to its role. The `porc` modifies `Order.class` to implement each generated interface. It also outputs to `roles.txt` a summary of the role hierarchy from in the given application. In addition, if there are any RMI-accessible classes without any annotations, the `porc` produces a warning to help the developer ensure that he or she has applied the security policy to all classes.

Once our modifications are complete, we invoke `rmic` on the resultant class files using the our provided common intermediary interface `Intermediary.class`, and the RMI compiler produces the intermediary stub classes `Order_ITEmployees_Intermediary_Stub.class` and `Order_Accounting_Intermediary_Stub.class`.

3.2 Run-Time Component

Our authentication mechanism uses the Java Authentication and Authorization Service. JAAS enables developers to specify their own methods for authentication, such as password-based authentication. Upon successful authentication, we issue credentials to the client that describe the client's role(s) in the system. These credentials will be used later when the client attempts to access a proxy. We note that we are not interested in the mapping of clients to roles, as there are many administrative models for doing so. The important information (for us) in the credentials is the list of roles that the client may access.

Authorization relies on the credentials the server has issued to the client. To request a proxy object, the client presents the credentials to the server. Once the server verifies these credentials, it grants access to the proxy object.

To access the proxy, the client performs an RMI lookup of an intermediary object. Intermediary objects are accessible to everyone using RMI; they guard the proxy objects that correspond to roles, by only making proxies available upon receipt of credentials. Intermediaries are necessary because it is not possible to require authentication for access to RMI objects. RMI predates JAAS and has not been updated to enforce the JAAS capabilities [12].

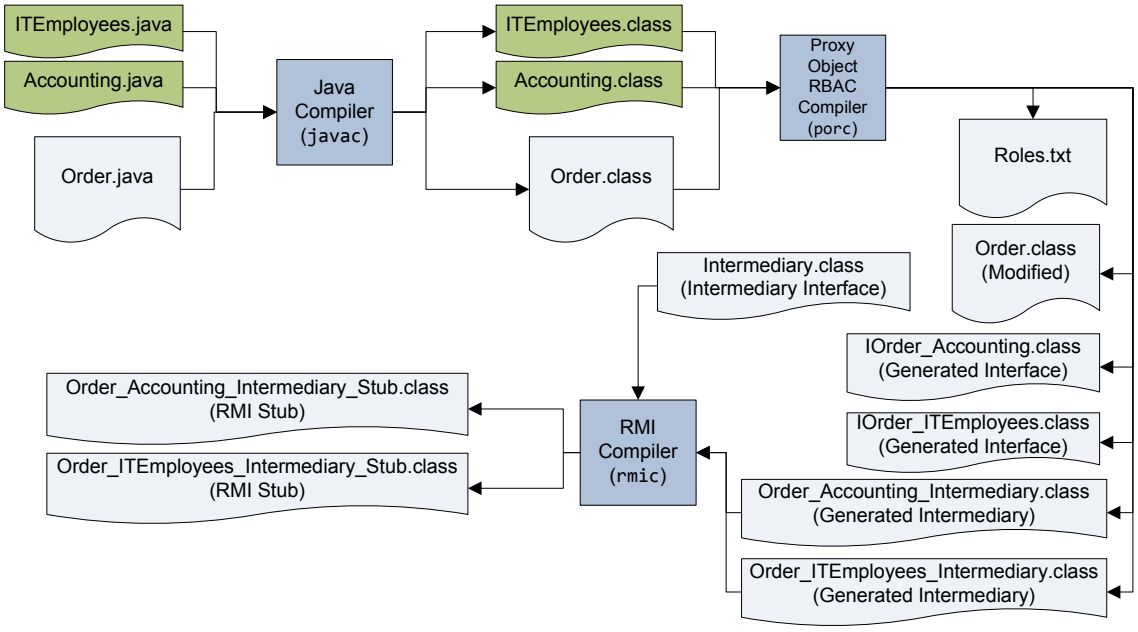

Figure 2: Proxy Object Compiler Processing the `Order` class. The Java Compiler, Proxy Object RBAC Compiler, and RMI Compiler boxes are processing programs; the boxes labelled ITEmployees.java/class and Accounting.java/class are role files. The other boxes are files. Arrows represent input/output from processing steps.

```
public interface Order_Accounting_Intermediary
      extends java.rmi.Remote {
 public Order_Accounting login
                      (Credentials credentials)
 throws LoginException, java.rmi.RemoteException;
}
```

Consider a simple example of a client that connects to a server. The server has an object that can be used to verify its identity [13]. The server's signed identity object is available via RMI. Figure 3 shows the sequence of events starting from when a client wishes to use the first object.

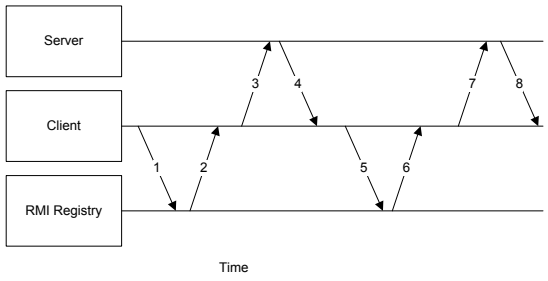

Figure 3: Communication Flow for a Client Accessing the First Proxy Object. Arrows correspond to messages and time proceeds from left to right.

A directed edge in Figure 3 depicts an interaction between two elements of the system, and these edges are numbered from left to right. Before client-server communication begins, the server instantiates the authentication object as well as any proxy objects, and registers these objects so that they are available via RMI (not shown). When a client wishes to access a proxy object, the steps are:

1. The client requests the server's authentication object from the RMI registry.
2. The RMI registry returns to the client a reference to the server's authentication object.
3. The client may verify the authentication object using the server's public key. If the verification of the server's identity succeeds, the client sends login information (using JAAS) to the server.
4. If the server accepts the client's attempt to log in, it returns to the client the set of credentials to which the client is entitled (according to the user-to-role mappings of the system).
5. When the client wants to access an object, it asks the RMI server for the location of the intermediary.
6. The RMI server returns that location to the client.
7. The client presents the server-issued credentials to the intermediary.
8. If the credentials are valid, the intermediary returns to the client a reference to the proxy object appropriate to the request.

A proxy object is an instance of the `Proxy` class. This class handles (in the sense of a Java Invocation Handler) all method invocations on the proxy object. If the method exists in the interface associated with a specific role, then the proxy passes the invocation on to the original object.

The original object executes the method as requested. If a method is not permitted for that role, it does not appear in the interface, and hence is not available for invocation over RMI. Therefore, only the methods that are allowed for a given role may be invoked by users with that role.

4. EMPIRICAL ASSESSMENT

In this section, we present a brief analysis of the performance of our system. The run-time principles of operation of our system are practically identical to those in [6], and the performance is therefore comparable.

To create an instance of a proxy of an object with no defined methods, except those inherited from `Object`, takes less than a millisecond, on average [6]. Thereafter, each method to be processed adds a small penalty. The penalty is roughly constant, so we observe linear behaviour with the number of methods.

Deriving an interface is a significant component of the run-time performance analysis in [6]; however, we instead shift all the interface derivation work to compile-time and need not derive any interfaces at run-time.

Invoking a proxy in place of the original has negligible overhead. Creation of a proxy object takes an order of magnitude less than the RMI lookup. Deriving the interface is on the same order as the RMI lookup.

4.1 Case Study

We have applied our technique to three sample programs that range from almost 9 000 to nearly 300 000 lines of code (LoC). Our approach is applicable to programs that allow clients to use RMI to invoke methods on objects. Many software systems are suitable, possibly after some restructuring. Any Model-Value-Controller (MVC) system is a candidate, if the communication between the model and view/controller are made to use RMI. When the program is structured around persistent domain objects (objects representing program entities), the domain objects can be server-side objects and accessed remotely by the client. A more general scenario is remote administration. In remote administration, the management objects (objects exposing administrative functions) are published via RMI.

There is a performance penalty when developers convert a program from entirely local execution to involving RMI. However, the benchmarks in [6] indicate that our technique adds no significant additional cost over the base penalty for using Java RMI. If the intent is to convert the application to use RMI, then a performance penalty is expected. Our approach allows the application to operate via RMI where it may not have been feasible before, because we offer security.

Our three sample programs are: **JBoss-Messaging**, a Java Message Service (JMS) implementation that ships with the JBoss application server; **jGnash2**, a personal finance application; and **Hospital RMS**, a medical record management system that was a course project for one author. For each program, we developed a role hierarchy intended to represent a possible real-world security policy. Figure 4 shows the roles in the case studies and the subsumption relationships between these roles.

We applied these role hierarchies to their respective programs and examined the process of annotation to evaluate the applicability of our approach to real-world software. We summarize our results in Table 1 (in which "Ann." is the total number of annotations we inserted into the code).

Program	LoC	Roles	Relations	Ann.
JBoss-M.	294 388	5	6	144
jGnash2	69 555	2	1	149
HospitalRMS	8 965	6	5	20

Table 1: Case Study Results for Sample Programs

JBoss-Messaging has many software components; we focus on the "core" package, and more specifically, the server management components. The server management objects are used for administrative functions in JMS, such as creation of topics and queues, setting the dead letter queue, and monitoring the system. The management interface is a relatively small subset of the program, but controlling access to it is vital to maintaining program security. Consider the following example of an annotation we apply in JBoss-Messaging.

```
@QueueAdministrator
public void setDeadLetterAddress
     (final String deadLetterAddress)
     throws Exception { ... }
```

The Queue Administrator and the Server Administrator (by subsumption) now have rights to invoke this method when published via RMI. If the queue control object were published without protection, anyone in the system could set the dead letter address (the place where undeliverable messages end up). In that case, an attacker could then receive messages intended for other users.

In the case of jGnash2, we apply our policy to the domain objects (e.g., `Transaction`) and can therefore implement our policy with a relatively small number of annotations, when compared to the size of the code base. Controlling access to the domain objects prevents an unauthorized user from making changes to the data stored in the system. We may then publish the domain objects via RMI. There are only two roles in this system (read-only and full access) and only a few domain objects, so the number of annotations to implement the security policy is small.

The Hospital RMS application uses a strict model-view-controller (MVC) architecture; we apply our security policy by enforcing access control on the models. Accordingly, using role subsumption, we need only 20 annotations to enforce the basic security policy of the system. This is because each model (e.g., patient record model) is wholly accessible (all methods available) or inaccessible (no methods available) to various roles.

Overall, we can see that the number of lines of code in the program do not necessarily indicate the number of annotations required to implement the security policy, nor do more roles mean more annotations to insert. Instead, what matters most is the design pattern of the application (e.g., MVC) and the security policy to be implemented. The security policy can often be succinctly stated, so that few annotations are needed to implement the policy. Finally, a program typically has only a small number of objects that are intended to be accessible via RMI. Consequently, the total number of classes to annotate is small. As the proxy object compiler issues a warning if there are RMI objects without any annotation, it is easy to ensure all classes that should have a security policy have annotations.

Modifying the code to insert the annotations is easy and requires minimal time. We were able to annotate each of

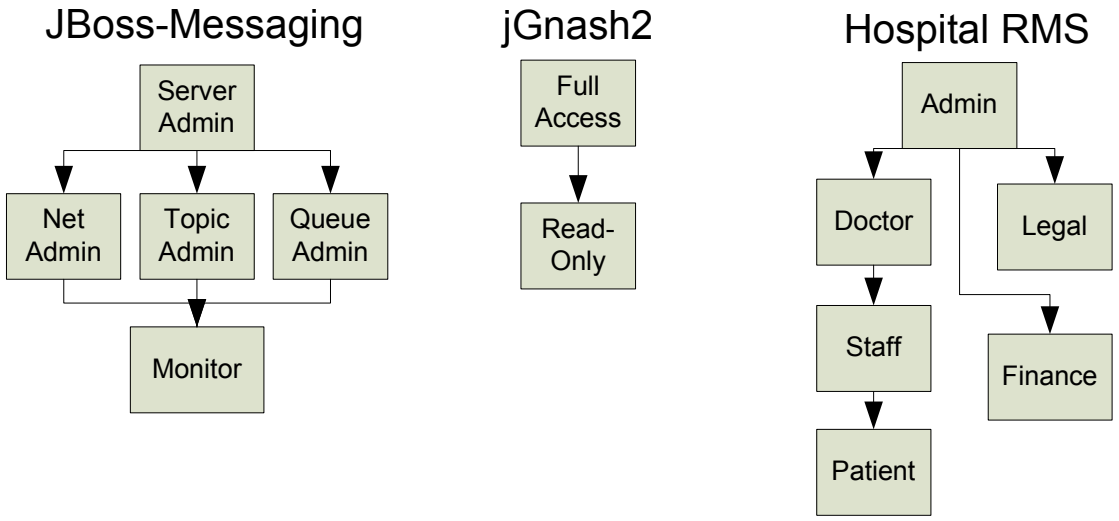

Figure 4: Role Hierarchies for Sample Applications. Boxes denote roles and arrows indicate subsumption relationships.

the sample applications in less than a day. As properly applying the security policy to all RMI-accessible objects ensures the enforcement of that policy, a developer need only locate all RMI-accessible objects, which is easy to do. He or she can then apply annotations to specify the desired security policy. Our system enables developers to succinctly and rapidly encode RBAC security policies without having to enforce them manually.

5. FUTURE WORK

To complement the concept of role-based security, we may wish to introduce logging to our system, to track invocations. If implemented, we could output a log indicating the role that invoked a certain method at a particular time. Such an audit log could be used to catch instances of a System Administrator user performing an `Order.create()` and an `Order.approve()` on the same `Order`, in violation of separation of privilege principles. Such information could also be used for other purposes, such as gathering usage statistics.

Role Administration is a related area that has received much attention in conjunction with role-based access control systems. Our administration procedure, described in the implementation, is configured only at compile time. Although configuration (addition and removal of annotations) is simple enough that a security administrator who is not a programmer could do it, code manipulation is still required. In the future, we could improve role administration to be outside of the code base of the system being protected.

6. RELATED WORK

Our solution is related to numerous areas in Java security. We discuss bytecode editing, stack inspection, interface derivation, proxy object generation, and, of course, role-based access controls.

While type safety obviates many security concerns, access control remains a key issue. Pandey and Hashii [5] investigate bytecode editing to enforce access controls, but do not discuss RMI. Wallach et al [14] enforce access controls using Abadi, Burrows, Lampson, and Plotkin (ABLP) Logic, where authorization is granted or denied on the basis of a statement from a principal allowing or denying access. However, their approach does not work with RMI, and, as acknowledged by the authors, does not handle a dynamic system with classloading well. Although Li, Mitchell, and Tong [13] provide a technique for securing a Java RMI application, their work does not use roles.

Giuri began with a basic overview of Role-Based Access Control [2] and extended this work to a Web context [15]. This work is foundational in Java Role-Based Access Control, and we build upon it by involving RMI and dynamic object generation from proxy objects. Ahn and Hu [16] implemented a system for generating code from language-independent security models; their approach enables separate consistency verification for the model. Like many code generation approaches, however, their system does not support round-tripping and therefore cannot smoothly support concurrent changes to code and model, unlike our system, which weaves the security policy directly into the code, helping ensure that the code and the policy both remain up-to-date.

Stack inspection can provide effective access control, but in an RMI context, the client call stack is unavailable to the server; even if it were available, it would be untrustworthy. A stack inspection scheme would therefore have to consider all remote accesses untrusted, whereas proxies can differentiate between trusted and untrusted RMI calls. Furthermore, the time to perform a stack inspection increases linearly with the depth of the stack [14], while the proxy object overhead is constant. Stack inspection suffers from difficulties with results returned by untrusted code, inheritance, and side effects [3]. Proxy objects are more resistant to these difficulties, because they do not trust any results from untrusted code, are designed with inheritance in mind, and are intended as a tool to avoid harmful side effects. Proxy objects and stack inspection have different principles

of trust. In proxies, a caller is trusted if it receives a reference to the original object. In stack inspection, the callee verifies its caller and all transitive callers.

Interface derivation is already in use in practice. For instance, Bryce and Razafimahefa [17] generate dynamic proxies to go between objects, and restrict access to methods. We permit role-based access controls rather than partitioning into trusted and untrusted clients.

Myers *et al* created JIF (Java Information Flow) [18] to restrict the leakage of information between objects within a Java program. We focus on method invocation by various roles, and do not assign ownership of data to particular principals. Finally, our system makes no changes to the Java language, meaning all semantics of the language remain consistent with the Java language specification.

7. CONCLUSIONS

We have presented a technique for method-level role-based access control for RMI-using Java programs. This work expands on previous work implementing method-level access control using only safe and unsafe annotations. Our technique computes the access rights to a given method based on program annotations. We have described the semantics of our system using First Order Logic. To investigate the behaviour of our system, we implemented a proxy object RBAC compiler. Using this compiler, we conducted a case study, which demonstrated the viability of our system in real software applications of various sizes.

8. REFERENCES

[1] Sandhu, R., Coyne, E., Feinstein H., and Youman, C., "Role-Based Access Control Models," *Computer*, vol. 29, pp. 38–47, Feb 1996.

[2] Giuri, L., "Role-Based Access Control in Java," *Proceedings of the third ACM workshop on Role-based access control*, pp. 91–100, 1998.

[3] Fournet, C. and Gordon, A., "Stack Inspection: Theory and Variants," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, pp. 360–399, May 2003.

[4] Gosling, J., Joy, B., Steele, G., and Bracha, G., *Java Language Specification, 3rd Edition*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[5] Pandey, R. and Hashii, B., "Providing Fine-Grained Access Control for Java Programs," *Proceedings of the 13th European Conference on Object-Oriented Programming*, vol. LNCS 1628, pp. 449–473, 1999.

[6] Zarnett, J., Lam, P., and Tripunitara, M., "Method-Specific Java Access Control via Proxy Objects using Annotations (Short Paper)," *Proceedings of the 5th International Conference on Information Systems Security*, vol. LNCS 5905, pp. 301–309, 2009.

[7] Richmond, M. and Noble, J., "Reflections on Remote Reflection," *Proceedings of the 24th Australasian Conference on Computer Science*, vol. 11, pp. 163–170, 2001.

[8] Hugh, M. and Ryan, M., *Logic in Computer Science*. Cambridge, UK: Cambridge University Press, 2nd ed., 2004.

[9] A. V. Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," *Journal of the ACM*, vol. 38, pp. 620–650, July 1991.

[10] Jeff Zarnett, "Method-Specific Access Control in Java via Proxy Objects using Annotations," Master's thesis, University of Waterloo, 2010.

[11] Ferraiolo, D.F., Kuhn, D.R., and Chandramouli, R., *Role-Based Access Control*. Norwood, MA, USA: Artech House, 2007.

[12] Kumar, P., *J2EE Security for Servlets, EJBs and Web Services: Applying Theory and Standards to Practice*. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[13] Li, N., Mitchell, J. C., and Tong, D., "Securing Java RMI-Based Distributed Applications," *Proceedings of the 20th Annual Computer Security Applications Conference*, pp. 262–271, 2004.

[14] Wallach, D., Appel, A., and Felten, E., "SAFKASI: A Security Mechanism for Language-based Systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, pp. 341–378, 2000.

[15] Giuri, L., "Role-based access control on the Web using Java," *Proceedings of the fourth ACM workshop on Role-based access control*, pp. 11–18, 1999.

[16] Ahn, G-J and Hu, H., "Towards realizing a formal RBAC model in real systems," *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 215–224, 2007.

[17] Bryce, C. and Razafimahefa, C., "An Approach to Safe Object Sharing," *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 367–381, 2000.

[18] Myers, A. C., Nystrom, N., Zheng, L., and Zdancewic, S., "Jif: Java information flow," July 2001. Software release. `http://www.cs.cornell.edu/jif`.