RC23876 (W0602-110) February 13, 2006
Computer Science

IBM Research Report

Role-Based Access Control Consistency Validation

Paolina Centonze, Gleb Naumovich*, Stephen J. Fink, Marco Pistoia
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Polytechnic University
5 MetroTech Center
Brooklyn, NY 11201

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Role-Based Access Control Consistency Validation

Paolina Centonze
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
paolina@us.ibm.com

Gleb Naumovich
Polytechnic University
5 MetroTech Center
Brooklyn, NY 11201
gleb@poly.edu

Stephen J. Fink
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
sjfink@us.ibm.com

Marco Pistoia
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
pistoia@us.ibm.com

ABSTRACT

Modern enterprise systems support Role-Based Access Control (RBAC), allowing for flexible access control on sensitive operations. Although these systems specify RBAC based on privileged *operations*, in many cases the deployer actually intends to control access to privileged *data*. This paper presents a theoretical foundation for correlating an operation-based RBAC policy with a data-based RBAC policy. Relying on a *location-consistency* property, this paper shows how to infer whether an operation-based policy is equivalent to any data-based RBAC policy.

We have built a static analysis tool for J2EE called Static Analysis for Validation of Enterprise Security (SAVES). Relying on interprocedural pointer analysis and dataflow analysis, SAVES analyzes J2EE bytecode to determine if the associated RBAC policy is location-consistent, and reports potential security flaws where location-consistency does not hold. The experimental results obtained by using SAVES on a number of production-level J2EE codes have identified several security flaws with no false positive reports.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Role-Based Access Control, Interprocedural Analysis

Keywords

Security, J2EE, Java, Static Analysis

1. INTRODUCTION

Modern enterprise systems, such as Java 2, Enterprise Edition (J2EE) [34] and Microsoft .NET Common Language Runtime (CLR) [11], have adopted Role-Based Access Control (RBAC) [9] to restrict access to enterprise applications and their functionalities. A *role* is a set of access rights that can be assigned to users and groups of a computer system. A client can access role-restricted functionality only if an administrator has granted the client the appropriate role(s).

To configure the RBAC policy for an enterprise system or application, an administrator must identify the operations required by the various users and groups. If the administrator fails to correctly identify the data and functionality that should be available to a role, the system or application can be vulnerable to unauthorized access.

At present, application developers and deployers define the roles that make sense for an application and then identify which methods each role should be allowed to call. Therefore, access is defined in terms of **operations** on components. However, as already emphasized in previous work on RBAC [29], the security policy *intent* is to protect privileged **data**, as opposed to operations.

For example, for a Web application that allows professors to assign grades to the students enrolled in a class and those students to check their grades, it is natural to specify that users in role `Professor` should have write access to the data representing student grades, while users in role `Student` should have only read access to that data. Specifying access in terms of methods could be more cumbersome, since there will likely be multiple methods for writing and reading grades, and without access to the source code it may not be clear which methods read and write that security-sensitive information. In such cases, defining access control on the basis of data is more straightforward and convenient (and therefore less error-prone) than defining access on the basis of operations.

Several factors further complicate RBAC administration, including:

- In systems in which access control can be specified both on the functions performed by applications and the data accessed by those applications, security inconsistencies can easily arise. For example, a role may be denied access to an object but granted the permission to invoke a function that allows accessing that object.

- A system administrator deploying an enterprise application is often not the same person who developed that application. For example, the J2EE Specification [43] explicitly separates these responsibilities. Therefore, a system administrator may not know precisely which data is accessed and modified by each method.

ISSTA 2006 Portland, Maine USA

In practice, J2EE and CLR currently support *only* operation-based RBAC. This paper presents a method to help administrators reason about an operation-based RBAC in terms of an equivalent data-based RBAC.

The novel contributions of this paper are:

1. **Theoretical Foundation for RBAC Consistency.** This paper defines "location-consistency," a property indicating whether a given operation-based RBAC policy is equivalent to any data-based RBAC policy. If so, the theory provides a straightforward method to construct the equivalent data-based policy.

2. **Static Analysis for RBAC Consistency Validation.** This paper describes a whole-program [37] static analysis technique to determine whether an operation-based RBAC policy is location-consistent and to construct an equivalent data-based RBAC policy, if one exists. If no such equivalent policy exists, the analysis reports flaws in the operation-based RBAC policy. A flaw may indicate a violation of the Principle of Least Privilege [38] (the policy is too permissive), a possible authorization run-time failure (the policy is too restrictive), or an implementation bug.

3. **SAVES.** This paper presents the design and implementation of a static analysis tool called Static Analysis for Validation of Enterprise Security (SAVES), which implements the RBAC consistency validation analysis for J2EE applications. SAVES relies on a flow-insensitive subset-based pointer analysis [2] and interprocedural mod-ref analysis. This paper discusses some of the implementation challenges in analyzing J2EE applications, and presents an empirical study on a number of production-level applications. The experimental results uncover a number of flaws with method-based RBAC policies, with no false positive reports.

The remainder of this paper proceeds as follows. Section 2 describes RBAC via a motivating example. Section 3 presents a formal model of RBAC consistency. Section 4 describes the J2EE RBAC model. Section 5 describes the design and implementation of a static analysis tool to verify RBAC consistency. Section 6 presents an experimental evaluation of the tool. Finally, Section 7 reviews related work.

2. MOTIVATING EXAMPLE

This section shows a fragment of a sample J2EE application for which access to security-sensitive operations can be restricted with roles. This code will be used as an example throughout the paper.

As will be explained in Section 4, the J2EE Specification [43] allows deployers to restrict access to methods of Enterprise JavaBeans (EJB) components. The code in Figure 1 represents a code fragment of an enterprise bean, `StudentBean`, with two methods, `setGrade` and `setProfile`. A system administrator configuring `StudentBean` usually would not have access to the source code of the application. Deployment tools used to configure J2EE applications employ introspection on the application bytecode to detect method names and parameter types and to allow a system administrator to map methods to roles. For the

```
// package and import declarations here...
public class StudentBean implements SessionBean {
    private String name, address;
    private Map grades = new HashMap();
    public void setGrade(String c, Character g) {
        grades.put(c, g);
    }
    public void setProfile(String n, String a,
            Map m) {
        this.name = n;
        this.address = a;
        this.grades = m;
    }
    // other methods here...
}
```

Figure 1: `StudentBean` Fragment

`StudentBean` application, it makes sense for `setGrade` to be restricted with role `Professor`, since that method, as its name suggests, allows write access to the `grades` field and the data held by it. It may appear natural to restrict `setProfile` with role `Student` since each student should be allowed to update his or her own profile. What a system administrator would probably not know is that granting `Student` access to `setProfile` implicitly allows `Student` to modify the `grades` security-sensitive field. It will be shown that this faulty RBAC policy violates a well-defined consistency property that can be automatically detected, indicating a flaw with the policy.

This example overly simplifies more complex situations in which detecting an inconsistency may require examining all the methods of an application, their access-control policies, and the data directly and indirectly accessed by those methods. This paper formalizes a concept of consistency for arbitrarily complex method-based RBAC policies and defines an algorithm for automatic detection of policy flaws.

3. RBAC CONSISTENCY MODEL

This section formalizes the different modes in which a program can access data, and introduces a lattice based on operations performed by program methods on program locations. This allows partitioning the set of methods of a program into equivalence classes. Furthermore, this section formalizes and characterizes the notion of location-based consistency for a method-based RBAC policy.

3.1 Notation

Given a program p, let E be the set of all possible executions of p and M the set of all methods of p. If $e \in E$, let A_e be the set of activations which arise during e and $M_e \subseteq M$ the set of all the methods executed by e. An element of A_e representing the activation of a method m will be indicated with a_m. During a run, e reads or writes from a set of memory locations, H_e. L represents a finite set of abstract locations, which partitions H_e into disjoint sets. For example, in Java, for a class `C`, `C.f` denotes an abstract location corresponding to the `f` field of all objects of type `C` that arise in an execution e. Intuitively, L represents the granularity by which an RBAC policy allows control of restricted data.

DEFINITION 3.1. *Given $e \in E$, $a_m \in A_e$ and $h \in H_e$:*

- a_m directly reads h *if* a_m *executes a statement that reads* h

- a_m indirectly reads h *if:*
  1. a_m *directly reads* h*, or*
  2. $\exists a_{m'} \in A_e$ *such that* a_m *calls* $a_{m'}$*, and* $a_{m'}$ *indirectly reads* h

The corresponding predicates *directly writes* and *indirectly writes* are defined analogously.

Definition 3.1 can be extended to methods and abstract locations as follows:

DEFINITION 3.2. *Given* $m \in M$ *and* $l \in L$*:*

- m directly reads l *if* $\exists e \in E$ *and* $\exists h \in l$ *such that* e *contains an activation* a_m *and* a_m *directly reads* h
- m indirectly reads l *if:*
  1. m *directly reads* l*, or*
  2. $\exists m' \in M$ *such that* m *calls* m'*, and* m' *indirectly reads* l

The corresponding predicates *directly writes* and *indirectly writes* are defined analogously. Abstract location $l \in L$ is *read* (*written*) by m if l is directly or indirectly read (written) by m. As an example, method `setGrade` in Figure 1 directly reads field `grades` and method `setProfile` directly writes fields `name`, `address`, and `grades`.

It is also important to reason about reads and writes of abstract locations that may not appear as security-sensitive, but they are so because they are referenced by other abstract locations that are security-sensitive. An abstract location $l' \in L$ is *reachable* from $l \in L$ if l' can be obtained from l through a series of zero or more dereferences. Clearly, if $l \in L$ is of a primitive type, the only abstract location reachable from l is l itself.

DEFINITION 3.3. *Given* $m \in M$ *and* $l \in L$*:*

- m directly partially reads l *if* $\exists l' \in L$ *such that* l' *is reachable from* l *and* m *reads* l'
- m indirectly partially reads l *if:*
  1. m *partially reads* l*, or*
  2. $\exists m' \in M$ *such that* m *calls* m'*, and* m' *indirectly partially reads* l

The corresponding predicates *directly partially writes* and *indirectly partially writes* are defined analogously. A method $m \in M$ *partially reads* (*writes*) an abstract location $l \in L$ if m directly or indirectly partially reads (writes) l. As an example, method `setGrade` in Figure 1 indirectly partially writes field `grades`. If l is of primitive type (not a pointer), the only way l can be partially read (written) is for it to be read (written).

3.2 Access Lattice

An *access tuple* provides an abstract representation of the behavior of a method with respect to memory access. An access tuple is defined in the general form $(R, \overline{R}, W, \overline{W})$, where $R, \overline{R}, W, \overline{W} \subseteq L$. Let $T = \mathcal{P}(L) \times \mathcal{P}(L) \times \mathcal{P}(L) \times \mathcal{P}(L)$ be the set of all access tuples, where $\mathcal{P}(L)$ indicates the power set of L. T, being the Cartesian product of four lattices, is itself a lattice [14]. Its partial order is obtained by extension from the partial orders of the single Cartesian product components, as follows: If $x = (R_1, \overline{R}_1, W_1, \overline{W}_1), y = (R_2, \overline{R}_2, W_2, \overline{W}_2) \in T$, then

$$x \sqsubseteq y \Leftrightarrow (R_1 \subseteq R_2) \wedge (\overline{R}_1 \subseteq \overline{R}_2) \wedge (W_1 \subseteq W_2) \wedge (\overline{W}_1 \subseteq \overline{W}_2)$$

Similarly, the meet and join lattice operations induced by the Cartesian product are derived, respectively, as follows:

$$x \sqcup y = (R_1 \cup R_2, \overline{R}_1 \cup \overline{R}_2, W_1 \cup W_2, \overline{W}_1 \cup \overline{W}_2),$$
$$x \sqcap y = (R_1 \cap R_2, \overline{R}_1 \cap \overline{R}_2, W_1 \cap W_2, \overline{W}_1 \cap \overline{W}_2)$$

Being finite, the T lattice is complete, has finite height, and contains a top element, $\top = (L, L, L, L)$, and a bottom element, $\bot = (\varnothing, \varnothing, \varnothing, \varnothing)$ [14]. Specifically, since the height of $\mathcal{P}(L)$ is $|L|$, the height of T is $4|L|$.

The sets of abstract locations read, partially read, written, and partially written by a method $m \in M$ are indicated with $\rho(m)$, $\overline{\rho}(m)$, $\omega(m)$, and $\overline{\omega}(m)$, respectively. By definition, $\rho(m) \subseteq \overline{\rho}(m)$ and $\omega(m) \subseteq \overline{\omega}(m)$. Sets $\rho(m)$, $\overline{\rho}(m)$, $\omega(m)$, and $\overline{\omega}(m)$ are all bounded by the universe L of all the abstract locations in p. Therefore, they are all finite sets. A function $\alpha : M \rightarrow T$ mapping every method to the access tuple accessed by that method can be defined as follows: $\alpha(m) = (\rho(m), \overline{\rho}(m), \omega(m), \overline{\omega}(m)), \forall m \in M$. Through function α, access tuples can be used to describe how methods access abstract locations.

3.3 Location Consistency

This section introduces a notion of "location-consistency" for a method-based RBAC policy.[1] The location-consistency property formalizes the notion that a method-based RBAC policy embodies a consistent data protection scheme.

Let R be the set of roles defined by an RBAC policy for program p. For $q \in R$, an execution $e \in E$ is said to be *in* q if the user executing e was granted role q.

For every $e \in E$, a function $\beta_e : R \rightarrow T$ mapping each role $q \in R$ to the access tuple of all the abstract locations read, partially read, written, and partially written by q while traversing e can be defined as follows:

$$\beta_e(q) = \bigsqcup_{m \in M_e} \alpha(m), \forall q \in R$$

DEFINITION 3.4. *Let* $e \in E$ *be an execution of* p*.*

- *A* method-based RBAC policy *for* p *is a function* $\mu : R \rightarrow \mathcal{P}(M)$
- e *is* valid *with respect to* μ*, indicated with* (e, μ)*, if, for every* $q \in R$*, when* e *is in* q*,* $m \in \mu(q), \forall m \in M_e$
- *A* location-based RBAC policy *for* p *is a function* $\Lambda : R \rightarrow T$*.*
- e *is* valid *with respect to* Λ*, indicated with* (e, Λ)*, if, for every* $q \in R$*, when* e *is in* q*,* $\beta_e(q) \sqsubseteq \Lambda(q)$

A method-based RBAC policy μ and a location-based RBAC policy Λ map each role $q \in R$ to the set of methods that q is allowed to execute and to the access tuple representing the

[1]Despite the name, the location-consistency property defined in this paper has no relation to the Location Consistency memory model [12].

abstract locations that q is allowed to access, respectively. The notion of validity of an execution e with respect to an RBAC policy formalizes the notion that e executes without run-time authorization failures.

Every method-based RBAC policy has a corresponding location-based RBAC policy, naturally defined as follows:

DEFINITION 3.5. *If μ is a method-based RBAC policy for p, the location-based RBAC policy* associated *with μ is the function $\Lambda_\mu : R \to T$ defined by:*

$$\Lambda_\mu(q) = \bigsqcup_{m \in \mu(q)} \alpha(m), \forall q \in R$$

It is also possible to compare method- and/or location-based RBAC policies as follows:

DEFINITION 3.6. *Given two RBAC policies γ and δ for p:*

- *γ is* consistent *with δ if $(e, \delta) \Rightarrow (e, \gamma), \forall e \in E$*
- *γ is* equivalent *to δ if γ and δ are consistent with each other*

Function α allows introducing an equivalence relation $\sim$ on M, defined as follows: Given $m_1, m_2 \in M$, $m_1 \sim m_2 \Leftrightarrow \alpha(m_1) = \alpha(m_2)$. Intuitively, if $\exists m_1, m_2 \in M : m_1 \sim m_2$, it makes sense for a method-based RBAC policy μ to restrict access to m_1, m_2 with the same roles, which means $\mu(m_1) = \mu(m_2)$. Function α allows also comparing the access levels of two methods by comparing the access tuples corresponding to those methods. Intuitively, a method-based RBAC policy μ is "inconsistent" if $\exists m_1, m_2 \in M, \exists q \in R \mid \mu(m_1) \sqsubseteq \mu(m_2) \wedge m_1 \notin \mu(q) \wedge m_2 \in \mu(q)$. This formalizes the notion that if μ denies a role q access to a method m_1 to prevent q from accessing certain abstract locations in certain modes, then μ should deny q access to all the methods having a level of access grater than or equal to that of m_1; otherwise, q could use those other methods to bypass the intended data restrictions. This leads to the following, more general definition:

DEFINITION 3.7. *A method-based RBAC policy μ is said to be* location-consistent *if:*

$$\forall q \in R, \forall m \in M, m \notin \mu(q) \Rightarrow \alpha(m) \not\sqsubseteq \Lambda_\mu(q).$$

As will be proved shortly, the notion of location consistency represents the fact that a method-based RBAC μ is equivalent to some location-based RBAC. If μ is *not* location-consistent, μ embodies a security policy that does not correspond to any protection scheme based solely on data protection. If $\exists q \in R, m \in M : m \notin \mu(q) \wedge \alpha(m) \sqsubseteq \Lambda_\mu(q)$, a location inconsistency exists, which may indicate that:

A Access to m should have been granted to q, that is too few permissions are given to q

B Access to some of the methods in the set $M_{m,q} = \{m' \in M : m' \in \mu(q) \wedge \alpha(m') \sqcap \alpha(m) \neq \varnothing\}$ has been mistakenly granted to q, that is too many permissions are given to q

C Some of the methods in $M_{m,q}$ contain bugs that make them access security-sensitive fields that were not intended to be accessed by them

D m contains bugs that make it not access one or more security sensitive fields that it was supposed to access

Situation **A** is undesirable because q may not be able to access the required functionality and run-time authorization failures may occur, making the application unstable. Situation **B** is undesirable too because q can access functionality not intended for it, thereby resulting in a potential violation of the Principle of Least Privilege [38]. Situations **C** and **D** represent bugs in component code or assembly configuration of the components. The security inconsistency described in the motivating example of Section 2 is of type **B**.

THEOREM 3.1. *If μ is a location-consistent RBAC policy on p, then Λ_μ is equivalent to μ.*

PROOF. If $e \in E$ is valid with respect to μ, $m \in \mu(q), \forall m \in M_e$. Therefore:

$$\beta_e(q) = \bigsqcup_{m \in M_e} \alpha(m) \sqsubseteq \bigsqcup_{m \in \mu(q)} \alpha(m) = \Lambda_\mu(q), \forall q \in R$$

which proves that Λ_μ is consistent with μ.

Now, assume by contradiction that $\exists e \in E$ such that e is valid with respect to Λ_μ but not valid with respect to μ. Since e is not valid with respect to μ, $\exists q \in R, \exists m' \in M_e : m' \notin \mu(q)$. However, since e is valid with respect to Λ_μ:

$$\alpha(m') \sqsubseteq \bigsqcup_{m \in M_e} \alpha(m) = \beta_e(q) \sqsubseteq \Lambda_\mu(q)$$

which contradicts the hypothesis that μ is location-consistent. Therefore, μ is consistent with Λ_μ.

This proves that μ and Λ_μ are equivalent. □

Theorem 3.1 proves that if a method-based RBAC policy μ is location-consistent, there exists always a location-based RBAC policy that is equivalent to μ, and that policy is Λ_μ.

Next, it will be shown that with one further assumption, if μ is *not* location-consistent, then it corresponds to *no* location-based policy.

DEFINITION 3.8. *A method-based RBAC policy μ is said to be an* entrypoint policy *if for $m \in M, q \in R, m \notin \mu(q) \Rightarrow \exists e \in E$ such that e begins execution in method m.*

Intuitively, given an entrypoint policy restricting access to a method m, then one can construct an execution with method m. In other words, m is a potential entrypoint to the application. J2EE and CLR support *only* entrypoint policies. For example, J2EE allows an RBAC policy to restrict access only to EJB interface methods, which can logically begin a new execution (thread of control) for a J2EE application in a given role.

THEOREM 3.2. *If a method-based RBAC entrypoint policy μ for p is not location-consistent, there exists no location-based RBAC policy Λ for p such that Λ is equivalent to μ.*

PROOF. Assume by contradiction that a location-based RBAC policy Λ for p exists such that Λ is equivalent to μ. Since μ is not location-consistent, $\exists q \in R, \exists m \in M : m \notin \mu(q) \wedge \alpha(m) \sqsubseteq \Lambda(q)$. Let $m_1, m_2, \ldots, m_k \in M$ be the methods traversed by an execution e such that $m = m_1$. An execution e with this property must exist since μ is an entrypoint policy. If $i \in \{1, 2, \ldots, k\}$, $\rho(m_i) \subseteq \rho(m), \overline{\rho}(m_i) \subseteq \overline{\rho}(m), \omega(m_i) \subseteq \omega(m), \overline{\omega}(m_i) \subseteq \overline{\omega}(m)$. Therefore, $\alpha(m_i) \sqsubseteq \alpha(m)$, and since $\alpha(m) \sqsubseteq \Lambda(q)$, this proves

that $\alpha(m_i) \sqsubseteq \Lambda(q), \forall i = 1, 2, \ldots, k$. This implies that e is valid for Λ but not for μ, which means that Λ and μ cannot be equivalent. □

Theorems 3.1 and 3.2 provide the foundation for checking consistency of a method-based RBAC policy μ. By determining whether or not μ is location-consistent and computing Λ_μ, it is possible to provide a great deal of information about how μ controls access to sensitive data. In later sections, we describe the design and implementation of an automatic tool built on this principle.

4. RBAC IN J2EE

This section describes the J2EE RBAC security system modeled and analyzed by SAVES. The J2EE Specification [43] dictates that when a restricted resource in a component, such as a method in an enterprise bean, is accessed from another component, the J2EE container should perform an authorization check [34]. While users and groups are defined at the system level, roles are application-specific; each J2EE application defines its own security roles. At run time, when the program attempts to access a role-restricted resource, the security system verifies that the user initiating the access was granted the required role.

4.1 Component Model

J2EE provides access control for EJB components. The methods of an enterprise bean are implemented in a class known as the *EJB class*. For a client program (such as another enterprise bean, a servlet, or a stand-alone application) to access an EJB method, that EJB method must be declared in a specific *EJB interface*. An EJB may implement one of four types of interface: *Remote*, *RemoteHome*, *Local*, and *LocalHome*. Methods implemented in the Remote and RemoteHome interfaces can be invoked by client programs located in different containers (for example, different processes or systems) through Remote Method Invocation (RMI) over Internet Inter-Object Request Broker (ORB) Protocol (IIOP). Methods implemented in the Local and LocalHome interfaces can be invoked by client programs colocated in the same container (same address space). *Helper methods*—those implemented by the enterprise bean classes and not declared in any EJB interface—are not directly accessible by clients and can only be invoked from the component to which they belong.

The J2EE authorization model differs from the formal model of Section 3 in an important detail. J2EE subjects *only* intercomponent calls to authorization checks. Access control restrictions do not apply to helper methods, or calls within the same EJB component. This exemption, driven by implementation concerns, complicates reasoning of J2EE RBAC policies and can cause security holes if not handled carefully. For clarity of exposition, the formal RBAC consistency model presented in Section 3 does not deal with the differences between inter- and intracomponent calls, but the model can be easily extended to faithfully represent those differences as well. The implementation described in Section 5 correctly models both inter- and intracomponent calls.

4.2 Declarative Security

J2EE and CLR promote the concept of *declarative security*; it is not necessary to embed authentication and authorization code within an application. Rather, security information appears, along with other deployment information, in configuration files external to the application code. In J2EE, one such configuration file is called the *deployment descriptor* and is defined using eXtensible Markup Language (XML). The XML code in Figure 2 shows a deployment descriptor fragment defining the `Professor` role and restricting access to the `setGrade` Remote interface method in enterprise bean `StudentBean` shown in Figure 1. By default, all roles are allowed access to all methods that are not explicitly restricted by the application deployment descriptor. Such methods are called *unchecked*.

```
<assembly-descriptor>
   <security-role>
      <role-name>Professor</role-name>
   </security-role>
   <method-permission>
      <role-name>Professor</role-name>
         <method>
            <ejb-name>StudentBean</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>setGrade</method-name>
         </method>
   </method-permission>
</assembly-descriptor>
```

Figure 2: Security-Related Deployment Descriptor Fragment

If the purpose of an access policy is to restrict access to a field to users with a specific role, one might assume it would be sufficient to restrict access to the field's getter and setter accessor methods. After all, the EJB Specification [42] mandates that reads and writes to EJB fields should always go through accessor methods. Therefore, to restrict access to a field, should not it be sufficient to restrict access to its accessor methods? The answer is no, for three reasons. First of all, the container does not enforce the EJB Specification rule above. Second, a field may very well be accessed by a non-accessor method; for example, `setProfile` in Figure 1 is not a setter accessor method, and yet it assigns a value to field `grades`. Third, as observed in Section 4.1, the container may not even perform authorization checks on all the invocations of an accessor method; the container will not check intra-component calls. A user lacking the required role may still be able to write to a security-sensitive field by simply going through an internal method.

4.3 Principal Delegation

In J2EE, by default, the identity of the principal who initiates a transaction on the client propagates to the downstream calls. However, some enterprise resources may need to be executed as though called by a principal with specific roles. For this purpose, J2EE allows associating "principal-delegation policies" with components. A *principal-delegation policy* consists of a `run-as` entry in the component's deployment descriptor. The entry's value holds the name of a role specific for the application to which that component belongs. The effect of a principal-delegation policy is that all the downstream calls from that component onward will be performed as if the caller had been granted only the role specified in the `run-as` entry. Detecting the security inconsistencies that can be generated by principal-delegation policies falls beyond the scope of this paper. We described

an interprocedural analysis for detection of such inconsistencies as part of a separate work [32].

4.4 Inaccessible Resources

In J2EE, an enterprise resource can be marked as *inaccessible* by explicitly configuring the deployment descriptor of the resource's component and listing that resource in an `exclude-list`. The resource should not be accessed by any user, regardless of the user's roles. When that resource is accessed through an intercomponent call, the J2EE container will perform an authorization check and will prevent access to that resource. However, if the resource is accessed through an intracomponent call, the container will not perform any authorization check and the resource will be accessed in spite of the inaccessibility rule. The interprocedural analysis described in Section 5 correctly identifies inaccessible EJB methods and models the propagation of security information accordingly.

5. STATIC ANALYSIS

We have implemented a tool called SAVES that checks location consistency for a method-based J2EE RBAC. This section describes SAVES and presents some implementation details.

5.1 Overview

SAVES takes as input a collection of J2EE applications provided as a set of Enterprise Archive (EAR) files, or a collection of Java Archive (JAR) and Web Archive (WAR) files. SAVES retrieves the method-based RBAC policy μ about the application under analysis from the deployment descriptors of the EJB JAR files.

SAVES first performs a flow-insensitive pointer analysis and builds a call graph. SAVES consults the deployment descriptor to determine the EJB interface methods and other J2EE methods which are possible entrypoints for program execution. For each entrypoint method, SAVES extracts from the application's deployment descriptor the set of roles with which that method has been protected. More details on the pointer analysis and call graph construction appear below.

SAVES uses the pointer analysis abstractions to partition the memory locations into abstract locations; each abstract object in the pointer analysis corresponds to an abstract location described in Section 3.1. With the call graph and pointer analysis in hand, SAVES performs a context-insensitive interprocedural mod-ref analysis to determine the access tuple solution for each method. This analysis computes the solution to a straightforward *GEN*/*KILL* dataflow system induced by the call graph and pointer analysis solution [21, 25]. As noted earlier in Section 3.2, the lattice height in this problem is $4|L|$, where $|L|$ is the number of abstract locations, so the fixed-point iteration converges after at most $\mathcal{O}(|E||L|)$ iterations, where E is the set of edges in the call graph.

Having computed the access tuples, SAVES determines if μ is location-consistent, and if so, outputs the the field-based RBAC policy λ_μ induced by μ. If not, SAVES outputs a set of potential inconsistencies.

5.2 Implementation Details

We have developed a general Java bytecode interprocedural analysis framework, called DOMO. DOMO supports a range of object-oriented call graph construction and pointer analysis algorithms, focusing primarily on flow-insensitive algorithms. SAVES relies on DOMO's 0-1-CFA call graph construction algorithm, which provides a context-insensitive, field-sensitive Andersen's analysis [2], building the call graph on-the-fly. The pointer analysis filters by declared type on-the-fly, uses field-sensitive models, and tracks flow through locals with flow-sensitive def-use information from a register-based Static Single Assignment (SSA) representation [5].

In most cases, the analysis safely models the Java Virtual Machine (JVM) specification, including exceptional control flow. We do not consider potential side effects from concurrent operations on shared data structures; however, since enterprise beans are forbidden to manipulate threads, that should not affect the correctness of the target analyses. The analysis models a typical J2EE deployment with three class loaders; one each for application code, extension (container run-time) code, and primordial (core-library) code.

5.3 Modeling Enterprise Beans

A major challenge was to design the analysis architecture with enough flexibility to effectively analyze higher-level semantics of J2EE.

An immediate design question is whether to analyze the application *before* deployment or *after* deployment. During deployment, EJB applications pass through an extensive source-to-source code generation step, which introduces implementation details of the target EJB container implementation.

We chose to analyze the program *before* deployment. Instead of analyzing the deployed code, we explicitly model many aspects of how the program interacts with the EJB container. This choice has three advantages: (1) Scalability – it reduces the body of code to analyze, (2) Precision – it is likely that a human-generated summary of container semantics is more precise than could be inferred practically from the raw container implementation, and (3) Portability – the analysis results do not vary depending on the container implementation.

Modeling simple library methods, such as most native methods in the Java standard libraries, can be done concisely with a straightforward specification. For simple flow-insensitive models, we use an XML language to represent a method's semantics. This approach also suffices for some J2EE methods, when the method's definition is static and the semantics fixed. In many cases, we substitute synthetic models in place of standard Java 2, Standard Edition (J2SE) and J2EE methods that we assert will not have side effects affecting the properties of interest, such as I/O. These models improve performance by reducing the scope of the analysis, and in many cases increase precision by eliminating opportunities for dataflow pollution. In particular, a model of native methods is essential when analyzing applications for security since many security functions are implemented as native methods.

Modeling enterprise beans presents more engineering challenges, since the set of methods and their behavior is determined by the application's deployment descriptor. For these cases, we have implemented as part of DOMO a simple *EJB pseudo-compiler* that takes as input the application code and the deployment descriptor, and produces analyzable artifacts that represent method behavior.

For example, suppose the analysis encounters a call to

```
java.util.Collection getAccounts() {
    AccountEntHome h = ContainerModel.
            getPooledAccountHomeInstance();
    AccountEnt B = H.findByPrimaryKey( 0 );
    HashSet S = new HashSet();
    S.add(B);
    if (condition) {
        throw new RemoteException();
    }
    if (condition) {
        throw new EJBException();
    }
    return S;
}
```

Figure 3: Pseudo-code Showing the Analyzable Artifact Generated for `PersonEnt.getAccounts`

a method `PersonEnt.getAccounts`, where `Person` is an entity bean using Container-Managed Persistence (CMP), and `accounts` is specified to be a *Container-Managed Relation* (CMR) with the `Account` EJB. The J2EE specification defines the semantics of this call; the container will consult a backing persistence manager (usually a database) to determine the return value of `getAccounts`. DOMO does not analyze the thousands of methods in the container implementation that will perform this function; instead DOMO bypasses the container and recognizes special semantices for this call. Before attempting to resolve this call with standard Java semantics, the analysis checks for registered special semantics for this call. A registered EJB pseudo-compiler consults the deployment descriptor, notices that this method represents the container-managed relationship of `Person` to `Account`s. To model these semantics, the system will generate an analyzable artifact representing the semantics of a call to `getAccounts`, and model the call as dispatching to this artifact. To generate the artifact for this CMR access, the EJB pseudo-compiler consults the deployment descriptor to deduce bean `Account`'s primary key type, remote interface, and home interface. Based on these types, it generates an artifact similar to that shown in Figure 3. The simple semantics there suffice to construct a correct call graph incorporating the call to `getAccounts`. Note in particular the call to a class called `ContainerModel`. The `ContainerModel` is a distinguished analyzable artifact which simulates pooling of bean instances, along with other global container artifacts. Note also that this model for `getAccounts` will not suffice for all possible client analyses. For example, the returned collection is modeled as always containing one element. In reality, it may have zero or many. We would have to further refine the generated model in order to support a client analysis that was sensitive to this distinction.

Using a similar logic, we have generated models for many aspects of the J2EE specification, including many functions for servlets and JavaServer Pages (JSP) applications, most CMP-related methods, much of Java DataBase Connectivity (JDBC), some Simple Object Access Protocol (SOAP) functions, and some Apache Struts functions.

5.4 Dealing with Reflection

Reflection and introspective services arise often in J2EE applications. In addition to core reflective instantiation with `newInstance`, J2EE applications will often create objects via invocations to services such as JNDI lookup, JavaBeans instantiation, RMI `narrow` services, serialization, return values from objects such as `java.sql.ResultSet`, various flavors of servlet and JSP contexts and sessions, and message arguments to message-driven beans.

It is impractical to expect a tool user to specify the behavior of calls to each of these services. While it may be possible to statically divine the behavior of some opaque services from configuration data, in other cases, we must fall back to conservative static estimates.

The analysis deals with reflection by tracking objects to casts, as in [10, 24] . When an object is created by reflective instantiation, the analysis assumes (unsoundly) that the object will be cast to a declared type before being accessed. So, the analysis tracks these flows, and infers the type of object created based on the declared type of relevant casts. While technically unsound, we believe that this approximation is accurate for the vast majority of reflective factory methods in J2EE programs.

6. EXPERIMENTAL RESULTS

This section describes the experimental results obtained by using SAVES on the following publicly-available J2EE applications: `Trade3` [44], `ITSOBank` [17], `DukesBank` [18], `Bookstore` [7], `EnrollerApp` [18], `SavingsAcc` [18], `PetStore` [31], and `CartApp` [18].

Of these applications, only `ITSOBank` and `DukesBank` came with predefined roles and method-based RBAC configurations, the reason being that assigning the RBAC configuration to an application is a task that the J2EE Specification [43] delegates to the deployer, and is supposed to be performed based on the system on which the application will run. Before analyzing the other applications, it was therefore necessary to deploy them, define relevant roles, and use those roles to restrict access to security-sensitive resources. Roles were assigned based on the introspection performed on the applications by Sun Microsystems' Deployment Tool for Java 2 Platform Enterprise Edition 1.4, without any specific knowledge of the applications' source code—similar to what a system administrator would do.

All experiments ran on an IBM X40 laptop having a 1.20 GHz Intel Pentium M processor and 1.5 GB of RAM. The operating system was Microsoft Windows XP Professional, Version 2002, Service Pack 2. SAVES is implemented in Java, and ran on the Sun Microsystems' Java 2 Runtime Environment, Standard Edition, V1.4.2_05. The J2SE and J2EE functionalities were made part of the analysis scope by adding the Sun Microsystems J2SE V1.4.2_05 and J2EE V1.4 core libraries to the analysis scope, respectively.

Table 1 reports characteristics of the applications and results of the SAVES analysis. For each application, the table shows:

- The overall size of the EAR files comprised by the application—this includes JAR and WAR files, deployment descriptors, and other supporting files
- The size of the EJB bytecode included in the EAR files being analyzed
- The total number of methods analyzed—these are all the methods reachable from the application's entry-points, including methods in the Java core libraries
- The number of methods analyzed from application code (not libraries)

Name	Size (KB)		Methods			Time (sec)	Mem. (MB)	Roles	Inco.	Classification			
	EAR	EJB	Total	App.	Bus.					A	B	C	D
Trade3	1,076	136	5,438	677	48	152.11	218	3	15	6	9	0	0
ITSOBank	302	205	2,323	191	22	64.97	255	2	2	2	0	0	0
DukesBank	128	34	1,579	188	21	118.67	160	2	2	0	2	0	0
Bookstore	359	33	12,178	256	13	3416.38	820	3	3	3	0	0	0
EnrollerApp	15	12	2184	55	12	65.71	252	3	9	5	4	0	0
SavingsAcc	10	7	2154	19	5	64.50	250	3	4	3	1	0	0
PetStore	1,282	133	6,411	339	36	255.22	300	2	0	0	0	0	0
CartApp	7	5	27	9	3	20.66	133	2	2	2	0	0	0

Table 1: SAVES Experimental Results

- The number of EJB interface methods (business methods)
- The wall-clock time to perform the analysis
- The total amount of memory (Java heap size) required to perform the analysis
- The total number of roles defined in the application
- The total number of inconsistencies found by SAVES
- A classification of those inconsistencies partitioned in groups **A**, **B**, **C**, and **D** as explained in Section 3.

As the Table shows, SAVES detected a fair number of policy inconsistencies across these applications. For `Trade3` and `EnrollerApp`, two of the inconsistencies of Type **B** could have been easily interpreted as bugs of types **C** and **D**, respectively. SAVES did not report any false positive inconsistencies; the precision of the underlying pointer analysis and call graph construction proved sufficient to accurately abstract these applications' behaviors. Given the effort that it would have taken to find these inconsistencies by performing code inspection or testing, these results are very encouraging and demonstrate the value of this approach.

7. RELATED WORK

Mechanisms for role-based access control in the networking environment have been proposed more than a decade ago [8]. Work on building and analyzing models and implementations for role-based access control has concentrated on complex architectures [39]. Surprisingly, few approaches for analyzing role-based access control mechanisms have been suggested. Schaad and Moffett [19] used the Alloy specification language for modeling the RBAC96 access model [40]. They use the Alloy Analyzer [20] to check the desirable properties, such as separation of duties assigned to roles, of such models. XML documents are often used by Web applications. Several mechanisms and frameworks for specification and enforcement of access policies for XML documents have been proposed [6, 23]. Such mechanisms are flexible in the sense that they prohibit or allow access to specific individual elements in XML documents. Recently, Murata, Tozawa, Kudo, and Satoshi [26] proposed a static analysis approach based on finite state automata that alleviates the burden of enforcement of such specifications at run time. Another positive side effect of this work is faster execution of queries over XML documents in some situations. Naumovich and Centonze [29] first identified the need for a consistency validation analysis for method-based RBAC policies. That preliminary work was purely theoretical and did not introduce a formal model for RBAC policy consistency validation. The algorithm described had not been implemented and for this reason its usefulness could not be validated through significant experimental results.

In the area of Web applications, a number of testing and static analysis techniques have been proposed, but they have concentrated primarily on the problem of control and information flow between static and dynamic HTML pages utilized by Web applications. For example, Ricca and Tonella [35] introduced a Unified Modeling Language (UML) model for Web applications that is useful for structural testing. However, this model concentrates on links between Web pages and interactive features of Web applications, such as HTML forms, and does not provide support for distributed object components.

Several works appeared in the area of quality assurance of distributed components. Brucker and Wolff [3] describe a technique for specification based testing of distributed components, such as CORBA [1] and EJB components. This approach uses the Object Constraint Language (OCL) [30] of the UML standard to formalize specifications of such components.

Clarke et. al [4] address the confinement problem of EJB objects. This problem arises in situations where direct references to EJB objects or other server-side distributed objects are returned to clients. Such references allow clients to use EJB objects directly, without going through the indirection of EJB interface objects. As a result, the EJB RBAC model can be circumvented. Clarke et. al define the possible ways in which confinement of EJB objects can be breached and define simple programming conventions which, if observed, support inexpensive static analysis able to detect confinement breaches or verify that no confinement breach is possible for a given set of enterprise beans.

Cadena [15] is an integrated development environment for building, modeling, and analyzing distributed components based on the CORBA standard. The formal underpinnings of Cadena allow extensive model checking support [36]. As a result, architectural properties about event-based inter-component communications can be checked. No analysis of RBAC policies for CORBA was done in the Cadena work.

In addition to the J2EE role-based security mechanism considered in this paper, Java also includes lower-level security mechanisms based on the notion of codebases and permissions [41, 13]. This mechanism is designed to enable users to run untrusted code in a security-restricted environ-

ment, which could potentially damage the system or steal sensitive data. A *codebase* for a software component identifies the location of this component. In Java, components can be fetched from an arbitrary URL and loaded in a running system. A *permission* signifies the ability to perform a sensitive operation (e.g. read a file on the local disk) and can be granted to a software component. Naumovich described a static analysis technique for automated analysis of the flow of permissions in Java programs [28]. This technique, based on data flow analysis [16], produces, for a given instruction in the program, a set of permissions that are checked on all possible executions up to this instruction. Koved, Pistoia, and Kershenbaum [22] proposed a static dataflow algorithm for a complementary problem of determining what permissions have to be granted to a given program or component to run it on a client machine. Subsequently, Pistoia et al. [33] described an interprocedural analysis algorithm to detect which portions of library code would be good candidates for becoming privileged without introducing tainted variables inside trusted code. These types of permission analysis are orthogonal to the analysis we describe in this paper.

Secure information flow is important in the context of Web applications. A number of approaches for reasoning about flow of information in systems with mutual distrust have been proposed. For example, Myers and Liskov [27] use static analysis for certifying information control flow and avoiding costly run time checks .

8. REFERENCES

[1] CORBA/IIOP 2.2 specification. `ftp://ftp.omg.org/pub/docs/formal/98-02-01.pdf`, February 1998.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[3] Achim D. Brucker and Burkhart Wolff. Testing distributed component based systems using UML/OCL. In *Informatik 2001*, volume 1, pages 608–614, November 2001.

[4] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)*, pages 374–387. ACM Press, 2003.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[6] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for xml documents. *ACM Transactions on Information Systems Security*, 5(2):169–202, 2002.

[7] Harvey M. Deitel, Paul J. Deitel, and Sean E. Santry. *Advanced Java 2 Platform: How to Program*. Prentice Hall, Upper Saddle River, NJ, USA, September 2001.

[8] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[9] David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992.

[10] Stephen Fink, Julian Dolby, and Logan Colby. Semi-automatic J2EE transaction configuration. Technical Report RC23326, IBM, 2004.

[11] Adam Freeman and Allen Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.

[12] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.

[13] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.

[14] George Grätzer. *General Lattice Theory*. Birkhäuser, Boston, MA, USA, second edition, January 2003.

[15] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th international conference on Software engineering*, pages 160–173, 2003.

[16] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[17] IBM Corporation, IBM Redbooks, `ftp://www.redbooks.ibm.com/redbooks/SG245429/`.

[18] Sun Microsystems, J2EE 1.4 Tutorial, `http://java.sun.com/j2ee/1.4/download.html#tutorial/`.

[19] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.

[20] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering*, pages 730–733. ACM Press, 2000.

[21] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, USA, 1973. ACM Press.

[22] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.

[23] Michiharu Kudo and Satoshi Hada. Xml document security based on provisional authorization. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 87–96. ACM Press, 2000.

[24] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of Programming Languages and Systems: Third Asian Symposium, APLAS 2005*, November 2005.

[25] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1997.

[26] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML Access Control Using Static Analysis. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 73–84. ACM Press, 2003.

[27] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142. ACM Press, 1997.

[28] Gleb Naumovich. A conservative algorithm for computing the flow of permissions in Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 33–43, July 2002.

[29] Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, September 2004.

[30] Object Management Group. Object constraint language specification, chapter 6 of omg unified modeling language specification (draft). `http://www.omg.org/uml`, February 2001.

[31] Sun Microsystems, Java PetStore, `http://java.sun.com/developer/releases/petstore/`.

[32] Marco Pistoia and Robert J. Flynn. Interprocedural Analysis for Automatic Evaluation of Role-Based Access Control Policies. Technical Report RC23846, IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY, USA, 2006.

[33] Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[34] Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.

[35] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd international conference on Software engineering*, pages 25–34. IEEE Computer Society, 2001.

[36] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276. ACM Press, 2003.

[37] Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.

[38] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.

[39] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[40] Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22. ACM Press, 2002.

[41] Sun Microsystems. Java security architecture. `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.fm.html`, 1998.

[42] Enterprise JavaBeans™Specification, `http://java.sun.com/products/ejb/`.

[43] Java™ 2 Platform Enterprise Edition Specification, `http://java.sun.com/j2ee`.

[44] IBM Corporation, Trade3 Benchmark, `http://www.ibm.com/software/appserv/benchmark3.html`.