Role-Based Access Control Consistency Validation

Paolina Centonze
IBM T. J. Watson Research Center
Hawthorne, NY 10532
paolina@us.ibm.com

Gleb Naumovich
Polytechnic University
CIS Department
Brooklyn, NY 11201
gleb@poly.edu

Stephen J. Fink
IBM T. J. Watson Research Center
Hawthorne, NY 10532
sjfink@us.ibm.com

Marco Pistoia
IBM T. J. Watson Research Center
Hawthorne, NY 10532
pistoia@us.ibm.com

ABSTRACT

Modern enterprise systems support Role-Based Access Control (RBAC). Although RBAC allows restricting access to privileged *operations*, a deployer may actually intend to restrict 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 RBAC policy is equivalent to any data-based RBAC policy. We have built a static analysis tool for Java Platform, Enterprise Edition (Java EE) called Static Analysis for Validation of Enterprise Security (SAVES). Relying on interprocedural pointer analysis and dataflow analysis, SAVES analyzes Java EE 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 Java EE codes have identified several security flaws with no false positive reports.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Reliability, Security, Verification

Keywords

J2EE, Java, Java EE, RBAC, Role-Based Access Control, Security, Static Analysis

1. INTRODUCTION

Modern enterprise systems, such as Java Platform, Enterprise Edition (Java EE)[1] [31] and Microsoft .NET Common Language Runtime (CLR) [10], have adopted Role-Based Access Control (RBAC) [8] 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 set of roles.

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 [26], the security policy *intent* is often 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. In fact, the Java EE Specification [41] explicitly separates these responsibilities. Therefore, a system administrator may not know precisely which data is accessed and modified by each method.

[1]Formerly known as Java 2, Enterprise Edition (J2EE).

In practice, Java EE 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 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 [35] (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 Java EE applications. SAVES relies on a flow-insensitive subset-based pointer analysis [1] and interprocedural mod-ref analysis. This paper discusses some of the implementation challenges in analyzing Java EE 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 introduces motivating examples. Section 3 presents a formal model of RBAC consistency. Section 4 describes the Java EE RBAC model. Section 5 describes the design and implementation of SAVES. Section 6 presents an experimental evaluation of SAVES. Section 7 reviews related work. Section 8 describes future work. Finally, Section 9 summarizes the results and novel contributions of this paper.

2. MOTIVATING EXAMPLE

This section shows a sample Java EE application for which access to security-sensitive operations can be restricted with roles. It then shows a typical RBAC scenario in which access to operations are not restricted consistently with the data held and manipulated by those operations. These examples will be referenced throughout the paper.

As will be explained in Section 4, the Java EE Specification [41] 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 has no access to the source code of the application. Deployment tools used to configure Java EE applications employ introspection on the application bytecode to detect method

```
// 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 code here...
}
```

Figure 1: `StudentBean` Fragment

names and parameter types and to allow a system administrator to map methods to roles. For the `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, indicating a policy flaw that can be automatically detected.

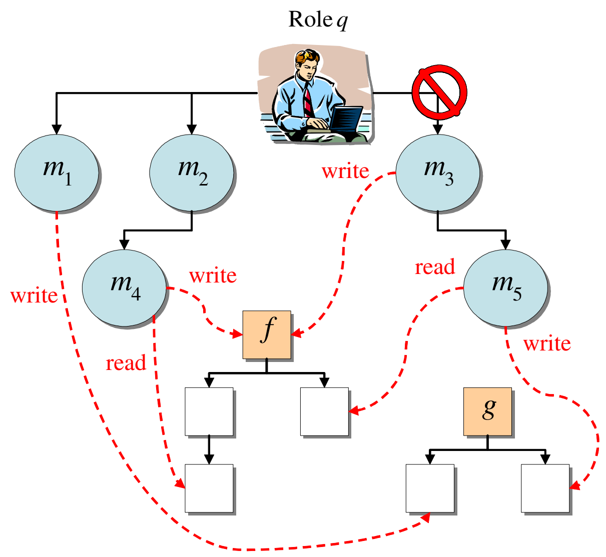

Figure 2: RBAC Policy Scenario

Inconsistencies can be much more difficult to find out than in the example of Figure 1. In general, it may be necessary to examine all the methods of an application, the restrictions imposed on them by the RBAC policy, and the data directly or indirectly referenced or modified by those methods. For example, in the scenario of Figure 2, role q has been denied access to EJB method m_3, but users in role q can still access EJB security-sensitive fields f and g (and the data held by them) directly through methods m_1 and m_2 or indirectly through method m_4. This paper formalizes a concept of consistency for arbitrarily complex method-based RBAC policies and defines an algorithm for automatic policy-flaw detection.

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 X be the set of all possible executions of p and M the set of all methods of p. If $x \in X$, let A_x be the set of activations which arise during x and $M_x \subseteq M$ the set of all the methods executed by x. An element of A_x representing the activation of a method m will be indicated with a_m. During a run, x reads or writes from a set of memory locations, H_x. Let L be a finite set of abstract memory locations partitioning H_x 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 x. Intuitively, L represents the granularity by which an RBAC policy allows control of restricted data.

DEFINITION 3.1. *Given an execution $x \in X$, an activation $a_m \in A_x$, and a memory location $h \in H_x$:*

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

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

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

DEFINITION 3.2. *Given a method $m \in M$ and an abstract location $l \in L$:*

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

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

As an example, `setGrade` in Figure 1 reads `grades`, while `setProfile` writes `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 (not a pointer), the only abstract location reachable from l is l itself.

DEFINITION 3.3. *Given a method $m \in M$ and an abstract location $l \in L$:*

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

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

If l is of a primitive type, the only way l can be partially read or partially written is for it to be read or written, respectively. By definition, a method that reads or writes an abstract location also partially reads or partially writes that abstract location, respectively. The opposite is not necessarily true.

In Figure 1, `setGrade` partially writes `grades`. Table 1 shows the EJB fields read, partially read, written, and partially written by the methods in Figure 2.

Methods	**Fields**			
	Read	Partially Read	Written	Partially Written
m_1				g
m_2		f	f	f
m_3		f	f	f, g
m_4		f	f	f
m_5		f		g

Table 1: Field Accesses for Application in Figure 2

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 $\langle r, \overline{r}, w, \overline{w} \rangle$, 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 powerset of L. T, being the Cartesian product of four lattices, is itself a lattice [13]. Its partial order, $\sqsubseteq$, is obtained by extension from the partial orders of the single Cartesian product components, as follows: If $x = \langle r_1, \overline{r}_1, w_1, \overline{w}_1 \rangle, y = \langle r_2, \overline{r}_2, w_2, \overline{w}_2 \rangle \in T$, then:

$$x \sqsubseteq y \overset{\text{def}}{\Longleftrightarrow} (r_1 \subseteq r_2) \wedge (\overline{r}_1 \subseteq \overline{r}_2) \wedge (w_1 \subseteq w_2) \wedge (\overline{w}_1 \subseteq \overline{w}_2)$$

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

$$x \sqcap y := \langle r_1 \cap r_2, \overline{r}_1 \cap \overline{r}_2, w_1 \cap w_2, \overline{w}_1 \cap \overline{w}_2 \rangle$$
$$x \sqcup y := \langle r_1 \cup r_2, \overline{r}_1 \cup \overline{r}_2, w_1 \cup w_2, \overline{w}_1 \cup \overline{w}_2 \rangle$$

The difference operator on T is obtained by extension from the set-difference operators, \, defined on the single Cartesian product powerset components, as follows:

$$x - y := \langle r_1 \setminus r_2, \overline{r}_1 \setminus \overline{r}_2, w_1 \setminus w_2, \overline{w}_1 \setminus \overline{w}_2 \rangle$$

Being finite, lattice T is complete, contains a top element, $\top := \langle L, L, L, L\rangle$, and a bottom element, $\bot := \langle \varnothing, \varnothing, \varnothing, \varnothing\rangle$, and has finite height [13]. 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 $\alpha : M \to T$ mapping every method to the access tuple of the abstract locations accessed by that method can be defined as follows:

$$\alpha(m) := \langle \rho(m), \overline{\rho}(m), \omega(m), \overline{\omega}(m)\rangle, \forall m \in M$$

In the example of Figure 2, $\alpha(m_1) = \langle \varnothing, \varnothing, \varnothing, \{g\}\rangle$, $\alpha(m_2) = \langle \varnothing, \{f\}, \{f\}, \{f\}\rangle$, $\alpha(m_3) = \langle \varnothing, \{f\}, \{f\}, \{f, g\}\rangle$, $\alpha(m_4) = \langle \varnothing, \{f\}, \{f\}, \{f\}\rangle$, and $\alpha(m_5) = \langle \varnothing, \{f\}, \varnothing, \{g\}\rangle$, as can be seen from Table 1.

3.3 Location Consistency

This section introduces a notion of "location consistency" for a method-based RBAC policy.[2] The location-consistency notion formalizes the property that a method-based RBAC policy embodies a compatible data protection scheme.

3.3.1 Method- and Location-Based RBAC Policies

Given a program p, a function $\beta : X \to T$ mapping each execution $x \in X$ of p to the access tuple of all the abstract locations read, partially read, written, and partially written by p while traversing x can be defined as follows:

$$\beta(x) := \bigsqcup_{m \in M_x} \alpha(m), \forall x \in X$$

Definition 3.4. *Given a program p and a set of roles R:*

- *A* method-based RBAC policy *for p is a function $\mu : R \to \mathcal{P}(M)$.*
- *An execution $x \in X$ is* valid *for a role $q \in R$ with respect to μ, indicated with (x, q, μ), if $M_x \subseteq \mu(q)$, where $M_x \subseteq M$ is the set of methods executed by x.*
- *A* location-based RBAC policy *for p is a function $\Lambda : R \to T$.*
- *An execution $x \in X$ is* valid *for a role $q \in R$ with respect to Λ, indicated with (x, q, Λ), if $\beta(x) \sqsubseteq \Lambda(q)$.*

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 abstract locations that q is allowed to access, respectively. The notion of validity of an execution x for a role q with respect to an RBAC policy formalizes the property that q can execute x 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* induced *with μ is the function $\Lambda_\mu : R \to T$ defined by:*

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

[2]Despite the name, this notion has no relation to the Location Consistency memory model [11].

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, defined on the same set of roles R:*

- *γ is* compatible *with δ if $(x, q, \delta) \Rightarrow (x, q, \gamma), \forall x \in X, \forall q \in R$.*
- *γ is* equivalent *to δ if γ and δ are compatible with each other.*

Function α allows introducing on M an equivalence relation, $\sim$, which partitions M into equivalence classes:

$$\forall m_1, m_2 \in M, m_1 \sim m_2 \overset{\text{def}}{\Longleftrightarrow} \alpha(m_1) = \alpha(m_2)$$

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 $\mu^{-1}(m_1) = \mu^{-1}(m_2)$.

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 "location inconsistent" if $\exists m_1, m_2 \in M, \exists q \in R : \alpha(m_1) \sqsubseteq \alpha(m_2) \wedge m_1 \notin \mu(q) \wedge m_2 \in \mu(q)$. This leads to the more general 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. More precisely:

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 shown in Section 3.3.3, for a method-based RBAC policy, being location consistent represents the existence of an equivalent location-based RBAC policy.

3.3.2 Inconsistency Classification

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 $\exists q \in R, \exists m \in M : m \notin \mu(q) \wedge \alpha(m) \sqsubseteq \Lambda_\mu(q)$, a location inconsistency exists. For example, in the scenario of Figure 2, $m_3 \notin \mu(q) \wedge \alpha(m_3) \sqsubseteq \alpha(m_1) \sqcup \alpha(m_2) = \Lambda_\mu(q)$. A location inconsistency may indicate one or more of the following:

A Access to m should have been granted to q, that is too few permissions are given to q. This can be corrected by changing the RBAC policy μ so that $m \in \mu(q)$. In Figure 2, this would be the case if it could be established that it was a mistake to deny q access to m_3.

B Access to some of the methods in the set $M_{m,q} := \{m' \in \mu(q) : \alpha(m') \sqcap \alpha(m) \neq \varnothing\}$ has been mistakenly granted to q, causing $\alpha(m) \sqsubseteq \Lambda_\mu(q)$. That is, too many permissions are given to q. This can be corrected by changing the RBAC policy μ and denying q access to those methods. The security inconsistency associated with the code in Figure 1 is of type **B**. As a further example, in the scenario of Figure 2, $M_{m_3,q} = \{m_1, m_2\}$, but perhaps it should have been $m_1 \notin \mu(q)$ or $m_2 \notin \mu(q)$, which would have implied $\alpha(m_3) \not\sqsubseteq \Lambda_\mu(q)$.

C Some of the methods in $M_{m,q}$ contain bugs that make them access unintended security-sensitive abstract locations, causing $\alpha(m) \sqsubseteq \Lambda_\mu(q)$. In the scenario of Figure 2, this would be the case if m_1 or m_2 contain bugs that mistakenly cause $\alpha(m_3) \sqsubseteq \alpha(m_1) \sqcup \alpha(m_2)$. For example, m_1 was not intended to partially write g. By correcting this bug, it will immediately follow $\alpha(m_3) \not\sqsubseteq \alpha(m_1) \sqcup \alpha(m_2)$.

D m contains bugs that unintendedly prevent it from accessing one or more security sensitive abstract locations, causing $\alpha(m) \sqsubseteq \Lambda_\mu(q)$. Still in the scenario of Figure 2, this would be the case if it could be established that m_3 contains bugs preventing it from, for example, directly reading g.

Situation **A** is undesirable because users in role 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 users in q can access functionality not intended for them, thereby resulting in a potential violation of the Principle of Least Privilege. Situations **C** and **D** represent bugs in component code or assembly configuration of the components.

3.3.3 Location-Consistency Characterization

This section proves that a method-based RBAC policy μ is equivalent to some location-based RBAC policy if and only if μ is location consistent.

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

PROOF. Let $x \in X$ and $q \in R$ be such that (x, q, μ). By Definition 3.4, $M_x \subseteq \mu(q)$. Therefore:

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

which implies (x, q, Λ_μ). Thus, Λ_μ is compatible with μ.

Now, assume by contradiction that μ is not compatible with Λ_μ. This means that $\exists x \in X, \exists q \in R : (x, q, \Lambda_\mu) \wedge \neg(x, q, \mu)$. From $\neg(x, q, \mu)$, it follows that $\exists m' \in M_x : m' \notin \mu(q)$. On the other hand, from (x, q, Λ_μ), it follows that:

$$\alpha(m') \sqsubseteq \bigsqcup_{m \in M_x} \alpha(m) = \beta(x) \sqsubseteq \Lambda_\mu(q)$$

which contradicts the hypothesis that μ is location consistent. Therefore, μ is compatible with Λ_μ. $\square$

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 $\forall \langle m, q \rangle \in M \times R : m \notin \mu(q), \exists x \in X$ such that x begins execution in method m.*

Intuitively, given an entrypoint policy restricting access to a method m, it is always possible to construct an execution of p starting with m. In other words, m is a potential entrypoint to the application. Assuming that a method-based RBAC policy is an entrypoint policy is actually quite reasonable. For example, both Java EE and CLR support *only* entrypoint policies. In particular, Java EE allows an RBAC policy to restrict access only to servlet HyperText Transfer Protocol (HTTP) methods and EJB interface methods, which can logically begin a new execution (thread of control) for a Java EE application in one or more given roles.

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 $x \in X$ such that $m = m_1$. An execution x with this property must exist since μ is an entrypoint policy. Given $i \in \{1, 2, \ldots, k\}$, it must be $\rho(m_i) \subseteq \rho(m), \overline{\rho}(m_i) \subseteq \overline{\rho}(m), \omega(m_i) \subseteq \omega(m), \overline{\omega}(m_i) \subseteq \overline{\omega}(m)$. 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 $\exists x \in X, \exists q \in R : (x, q, \Lambda) \wedge \neg(x, q, \mu)$, which means that Λ and μ cannot be equivalent. $\square$

Theorems 3.1 and 3.2 provide the foundation for checking consistency of a method-based RBAC entrypoint 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. Section 5 describes the design and implementation of SAVES, an automatic tool built on this principle.

4. RBAC IN JAVA EE

This section describes the Java EE RBAC security system modeled and analyzed by SAVES. The Java EE Specification [41] dictates that when a restricted resource in a component, such as a method in an enterprise bean, is accessed from another component, the Java EE container should perform an authorization check [31]. While users and groups are defined at the system level, roles are application-specific; each Java EE 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

Java EE 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. There are four types of EJB interfaces: 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 co-located 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 Java EE authorization model differs from the formal model of Section 3 in an important detail. Java EE 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 Java EE RBAC policies and can cause security holes if not handled carefully [29]. 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

Java EE 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 Java EE, one such configuration file is called the *deployment descriptor* and is defined using eXtensible Markup Language (XML). The XML code in Figure 3 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 3: Security-Related Deployment Descriptor Fragment

In Java EE, 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`. Access to that resource will be denied to any user, regardless of the user's roles.

If the purpose of an access policy is to restrict access to an EJB 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 [38] 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 invocation.

4.3 Principal Delegation

In Java EE, 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, Java EE 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 [29].

5. STATIC ANALYSIS

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

5.1 Security Analysis

SAVES takes as input a program p, consisting of one or more Java EE applications. Each application is provided as a collection of Enterprise Archive (EAR) files, or a set of Java Archive (JAR) and Web Archive (WAR) files. SAVES retrieves the method-based RBAC policy μ for p from the deployment descriptors embedded in the archive files.

SAVES first performs a flow-insensitive pointer analysis and builds a call graph $G = (N, E)$. SAVES consults the deployment descriptors to determine the EJB interface methods and other Java EE methods which are possible entrypoints for program execution. For each entrypoint method, SAVES extracts from the deployment descriptors the set of roles that are allowed access to that method.

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.

Specifically, SAVES computes the solution to a straightforward dataflow system [19, 23] induced by the call graph and pointer analysis solution. For each $n \in N$ representing the invocation of a method $m_n \in M$, SAVES determines the EJB fields that are directly read and directly written by m_n, and uses the points-to graph to determine the EJB fields that are directly partially read and directly partially written by m_n. $GEN(n)$ is defined as the access tuple computed from n through this computation, while $KILL(n) := \bot, \forall n \in N$. Next, to compute an overapproximation of the access tuple $\alpha(m_n)$ of the EJB fields that are—*directly or indirectly*—read, partially read, written, and partially written by each method $m_n \in M$, SAVES solves the following dataflow equation system for all $n \in N$:

$$\begin{aligned} IN(n) &:= \bigsqcup_{n' \in \Gamma^+(n)} OUT(n') \\ OUT(n) &:= GEN(n) \sqcup (IN(n) - KILL(n)) \end{aligned}$$

where $\Gamma^+ : N \to \mathcal{P}(N)$ is the successor function on the call graph nodes.

As noted in Section 3.2, the height of lattice T is $4|L|$, so the fixed-point iteration involved in solving the dataflow equation system converges after at most $\mathcal{O}(|E||L|)$ iterations [13]. Upon convergence, access tuple $OUT(n)$ overapproximates $\alpha(m_n)$, for all $n \in N$. This allows determining an overapproximation of the location-based RBAC policy Λ_μ induced by μ. Finally, SAVES detects potential location inconsistencies for μ by identifying pairs $\langle q, m\rangle \in R \times M$ that violate the condition of Definition 3.7.[3] If so, SAVES outputs the set of potential inconsistencies. Otherwise, SAVES outputs the overapproximation of Λ_μ.

5.2 Implementation Details

We have developed a general Java bytecode interprocedural analysis framework, called DOMO, which 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 Control Flow Analysis (CFA) call graph construction algorithm [14], which provides a context-insensitive, field-sensitive Andersen's analysis [1], 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 local variables 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 analysis. The analysis models a typical Java EE deployment with three class loaders; one each for application, extension (container run-time), 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 Java EE. 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.[4] 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.[5]
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 Java EE methods, when the method's definition is static and the semantics fixed. In many cases, we substitute synthetic models in place of standard Java Platform, Standard Edition (Java SE) and Java EE 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 [42].

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.

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

[3]In practice, since Java EE allows only RBAC entrypoint policies, it is sufficient to restrict this verification to those pairs $\langle q, m\rangle \in R \times M$ in which m is a Java EE entrypoint method.

[4]More precisely, SAVES requires the bytecode of the application under analysis to have already undergone deployment configuration. During this process, the application's class files are packaged into archives and each archive is assigned a deployment descriptor, which contains information essential for the analysis, including the RBAC policy configuration.

[5]For example, the mapping of an EJB remote method to its actual implementation in an EJB class can be achieved more precisely and efficiently with a summary as compared to analysis of a container's RMI-IIOP implementation from first principles.

For example, suppose the analysis encounters a call to 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` enterprise bean. The Java EE specification defines the semantics of this call; the container consults 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 semantics 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 and notices that this method represents the container-managed relationship of `Person` to `Accounts`. 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 4. 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 were sensitive to this distinction.

Using a similar logic, we have generated models for other aspects of the Java EE specification, including 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 Java EE applications. In addition to core reflective instantiation with `newInstance`, Java EE applications often create objects via invocations to services such as Java Naming and Directory Interface (JNDI) `lookup`, JavaBeans instantiation, RMI `narrow`, 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 [9, 22]. 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 Java EE programs.

6. EMPIRICAL EVALUATION

This section describes the experimental results obtained by using SAVES on the following publicly-available Java EE applications: `Trade3` [16], `ITSOBank` [28], `DukesBank` [39], `Bookstore` [7], `EnrollerApp` [39], `SavingsAcc` [39], `PetStore` [40], and `CartApp` [39]. Of these applications, only `ITSOBank` and `DukesBank` came with predefined roles and method-based RBAC policy configurations, the reason being that defining the RBAC policy of an application is a task that the Java EE Specification [41] 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 configure the deployment information for each of them, define relevant roles, and use those roles to restrict access to security-sensitive resources. For each application, roles were defined based on the application's target deployment environment. For example, the roles defined for an online banking application are different from those defined for an online store. Access to entrypoint methods was restricted with roles based on the introspection performed on the applications by Sun Microsystems' Deployment Tool for Java 2 Platform Enterprise Edition 1.4, which only reveals the fully qualified signature of each entrypoint method. The RBAC policy configurations were not based on any other specific knowledge of the applications. The same situation is typically faced by any system administrator who needs to configure the RBAC policy of an application without any information about the fields accessed by application's methods and without knowing the application's internal calling structure.

Table 2 reports characteristics of the applications and results of the analyses performed by SAVES. 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 Java SE and Java EE functionalities were made part of the analysis scope by adding the Sun Microsystems Java SE V1.4.2_05 and Java EE V1.4 core libraries to the analysis scope, respectively.

For each application, Table 2 reports:

- The overall size of the EAR files comprised by the application (which includes JAR and WAR files, deployment descriptors, and other supporting files) and the size of the EJB bytecode included in the EAR files being analyzed
- The total number of the methods analyzed (all the methods reachable from the application's entrypoints, including methods in the Java core libraries), the number of application methods, and the number of the EJB interface 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

Name	Size (KB)		Methods			Time	Mem.	Roles	Inco.	Classification			
	EAR	EJB	Total	App.	Bus.	(sec)	(MB)			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	60.00	330	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 2: SAVES Experimental Results

- 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 3.3.2

As Table 2 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.

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 [36]. Surprisingly, few approaches for analyzing role-based access control mechanisms have been suggested. Schaad and Moffett [37] used the Alloy specification language [17] for modeling the RBAC96 access model, and the Alloy Constraint Analyzer (Alcoa) [18] 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, 21]. 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 [24] 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 [26] 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 [33] 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 [2] describe a technique for specification based testing of distributed components, such as Common Object Request Broker Architecture (CORBA) [4] and EJB components. This approach uses the Object Constraint Language (OCL) [27] of the UML standard to formalize specifications of such components.

Clarke et. al [3] 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 [34]. 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 Java EE role-based security mechanism considered in this paper, Java also includes lower-level security mechanisms [32, 12] designed to enable users to run untrusted code in a restricted environment, which could potentially damage the system or steal sensitive data. A *permission* signifies the right to perform a security-sensitive operation, and can be granted to a software components and users. Koved, Pistoia, and Kershenbaum [20] proposed an interprocedural analysis algorithm for the complementary problem of determining what permissions have to be granted to a given program or component to execute it without run-time authorization failures. Subsequently, Pistoia, et al. [30] 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 [25] use static analysis for certifying information control flow and avoiding costly run time checks.

8. FUTURE WORK

Currently, a security policy on an RBAC system, such as Java EE, can only be specified in terms of methods. System administrators may want to have the flexibility to be able to specify RBAC policies that restrict access to both locations and methods. Without imposing a change in the underlying container specification, the work presented in this paper can be extended to allow system administrators to restrict access to abstract locations. SAVES can then detect the access tuple associated with each method and identify a method-based RBAC equivalent to the desired location-based RBAC policy, if one exists. Otherwise, SAVES can report the flaws identified in the location-based RBAC policy. Furthermore, the work presented in this paper can be extended to detect whether mixed RBAC policies that restrict access to both data and operations are incompatible.

9. SUMMARY

Unlike other access control systems, RBAC allows restricting access to operations rather than data. This paper has introduced a novel theoretical foundation for correlating an operation-based RBAC policy with a data-based RBAC policy. Relying on the location-consistency property, this paper has shown how to infer whether for an operation-based RBAC policy there exists any equivalent data-based RBAC policy. Furthermore, this paper has described the design and implementation of SAVES, a static analysis tool for Java EE applications. SAVES analyzes Java EE 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 Java EE codes have identified several security flaws with no false positive reports.

10. ACKNOWLEDGMENTS

The authors wish to thank Julian Dolby for his contributions to the DOMO analysis framework; Phyllis Frankl and Stuart Steele for providing useful comments on an earlier version of this paper; and the anonymous reviewers of ISSTA 2006 for their insightful comments.

11. REFERENCES

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

[2] Achim D. Brucker and Burkhart Wolff. Testing Distributed Component Based Systems Using UML/OCL. In K. Bauknecht, W. Brauer, and Th. Mück, editors, *Proceedings of Informatik 2001*, volume 1 of *Tagungsband der GI/ÖCG Jahrestagung*, pages 608–614, Vienna, Austria, November 2001. Österreichische Computer Gesellschaft.

[3] Dave Clarke, Michael Richmond, and James Noble. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 374–387, Anaheim, CA, USA, 2003. ACM Press.

[4] CORBA/IIOP Specification, `http://www.omg.org`.

[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] 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.

[9] Stephen J. Fink, Julian Dolby, and Logan Colby. Semi-Automatic J2EE Transaction Configuration. Technical Report RC23326, IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY, USA, 2004.

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

[11] 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.

[12] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA, USA, second edition, May 2003.

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

[14] David Grove and Craig Chambers. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[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, Portland, OR, USA, May 2003.

[16] IBM Corporation, Trade3 Benchmark, `http://www.ibm.com/software/`.

[17] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[18] 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, Limerick, Ireland, 2000. ACM Press.

[19] 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.

[20] 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.

[21] 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, Athens, Greece, November 2000. ACM Press.

[22] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 139–160, Tsakuba, Japan, November 2005.

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

[24] 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 Communications Security*, pages 73–84, Washington, DC, USA, October 2003. ACM Press.

[25] Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint Malo, France, October 1997. ACM Press.

[26] 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.

[27] Object Management Group. Object Constraint Language Specification, Chapter 6 of OMG Unified Modeling Language Specification (draft). `http://www.omg.org/uml`, February 2001.

[28] Joaquin Picon, Patrizia Genchi, Maneesh Sahu, Martin Weiss, and Alain Dessureault. *Enterprise JavaBeans Development Using VisualAge for Java.* IBM Redbooks. IBM Corporation, International Technical Support Organization, San Jose, CA, USA, June 1999.

[29] Marco Pistoia and Robert J. Flynn. Interprocedural Analysis for Automatic Evaluation of Role-Based Access Control Policies. Technical Report RC23846 (W0511-020), IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY, USA, November 2005.

[30] 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.

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

[32] Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. *Java 2 Network Security.* Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.

[33] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Toronto, ON, Canada, 2001. IEEE Computer Society.

[34] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An Extensible and Highly-modular Software Model Checking Framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[35] 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.

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

[37] Andreas Schaad and Jonathan D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, Monterey, CA, USA, 2002. ACM Press.

[38] Sun Microsystems, Enterprise JavaBeans$^{\text{TM}}$ Specification, `http://java.sun.com/products/ejb/`.

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

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

[41] Sun Microsystems, Java$^{\text{TM}}$ Platform, Enterprise Edition Specification, `http://java.sun.com/j2ee`.

[42] Xiaolan Zhang, Larry Koved, Marco Pistoia, Sam Weber, Trent Jaeger, Guillaume Marceau, and Liangzhao Zeng. The Case for Analysis Preserving Language Transformation. In *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis (ISSTA).* ACM Press, July 2006.