RC23836 (W0411-166) November 18, 2004
Computer Science

IBM Research Report

Static Evaluation of Role-Based Access Control Policies in Distributed Component-Based Systems

Marco Pistoia, Robert J. Flynn*, Vugranam C. Sreedhar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201

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

Static Evaluation of Role-Based Access Control Policies in Distributed Component-Based Systems

Marco Pistoia* Robert J. Flynn† Vugranam C. Sreedhar*

Abstract

Understanding and configuring security requirements for large distributed component-based applications is a complex process. Java 2, Enterprise Edition (J2EE) and Microsoft .NET have adopted a form of Role-Based Access Control (RBAC) for controlling accesses to security-sensitive resources. Access to resources is controlled through "security roles" rather than through the user identity. We have found a number of fundamental problems in the way RBAC is often defined and implemented. For instance, in J2EE a user initiating a transaction must have been granted not just the roles necessary to invoke the transaction entry point, but all the roles needed to access each resource traversed by the transaction, taking into account that a J2EE container performs authorization checks only when an access-restricted resource is accessed from another component, and that J2EE allows a "principal-delegation policy" to override the roles granted on subsequent component calls. For a typical application deployer or system administrator, configuring the security requirements for an application and assigning roles to users is quite a challenge. Making any sensible security configuration may require reading the source code of the entire application, if available. For this reason, deployers either turn off security or grant broad authorizations, which in both cases compromises the security of the application.

In this paper, we present a static-analysis tool, called Enterprise Security Policy Evaluator (ESPE), which helps application deployers and system administrators to correctly configure the security of a J2EE application. ESPE determines potential security flaws in J2EE applications. Using the role information computed statically, ESPE can identify whether too many or too few roles have been granted, and detect security policy inconsistencies. ESPE has been used to configure the security requirements of large J2EE applications.

1 Introduction

Role-Based Access Control (RBAC) [14] is gaining popularity in managing the security of large enterprise applications. In RBAC, authorization to access restricted resources is controlled through "security roles" rather than through the user identity. Java 2, Enterprise Edition (J2EE) and Microsoft .NET have adopted a form of RBAC for defining and managing security of enterprise applications. Although RBAC provides a flexible mechanism for managing security of networked applications, it may still be misconfigured. For instance, a system administrator may not diligently and consistently assign roles to users. If authorization fails too often, a system administrator may modify the role assignments so as to compromise the security of the enterprise applications.

A *security role* (*role* for short) is a semantic grouping of access rights [26]. Roles can be assigned to users of an enterprise application. While users and groups are defined at the J2EE server level, roles are application-specific; each application defines its own security roles. In a J2EE application, for example, it is possible to define the role of `Employee` and specify that method `m1()` in enterprise bean `Bean1` can be accessed only by those principals who have previously been assigned the role of `Employee`. If user Bob Smith successfully logs on to the J2EE server as `bob` and attempts to execute—directly or indirectly through

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA. {pistoia,vugranam}@us.ibm.com
†Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201, USA. flynn@poly.edu

a sequence of other method calls—`m1()` on `Bean1`, the method invocation will succeed only if `bob` was previously granted the role of `Employee`.

J2EE and .NET promote the concept of "declarative security". This means that it is not necessary to embed authentication and authorization code within an application. Rather, security information is stored along with other deployment information in configuration files that are external to the application code. In J2EE, these configuration files are called *deployment descriptors* and are defined using eXtensible Markup Language (XML). The following example shows a deployment descriptor fragment defining the `Employee` role and restricting access to method `m1()` in enterprise bean `Bean1`.

```
<assembly-descriptor>
    <security-role>
        <role-name>Employee</role-name>
    </security-role>
    <method-permission>
        <role-name>Employee</role-name>
        <method>
            <ejb-name>Bean1</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>m1</method-name>
        </method>
    </method-permission>
</assembly-descriptor>
```

A system administrator deploying `Bean1` must be aware that its method `m1()` can only be accessed by principals assigned the role of Employee, and will have to assign that role to users accordingly.

Configuring the security of a J2EE application is complicated by several factors:

- The J2EE specification [17] 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. However, the container will not perform authorization checks if the same resource is accessed from its own component. We will show later that lack of such intracomponent authorization checks can lead to security compromises or unnecessary authorization failures.

- Access to any J2EE enterprise resource can be restricted with one or more roles. When multiple roles are specified for a resource, the operation to apply to those roles is a logical `OR`. In order for a user to succeed in accessing that resource, that user must have been granted at least one of those roles. However, if multiple enterprise resources access each other, forming a chain of calls, and access to each of those resources has been restricted with different roles, then the user accessing the first of those resources will need to be granted all the roles necessary to access all the resources that have been invoked through intercomponent calls. In this case, the operation to apply to those roles to compute the set of roles needed by the user is a logical `AND`. This shows that in order to configure a J2EE application correctly so to not grant neither too few roles (which would make the application unstable) nor too many (which would violate the Principle of Least Privilege [32]), system administrators must be able to evaluate potentially complex logical expressions of roles.

- In J2EE, by default, the identity of the principal who initiated a transaction on the client is propagated to the downstream calls. However, some enterprise resources may need to be executed as though they were 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 is 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. We will show that principal-delegation policies can easily lead to security misconfigurations, violations of the Principle of Least Privilege, and stability problems.

- In J2EE, an enterprise resource can be marked as *inaccessible* by explicitly configuring the deployment descriptor of the resource's component. 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.

In this paper, we present a static analysis framework with customizable precision for identifying incorrect and inconsistent security configurations in J2EE applications. We implemented the framework in a tool called Enterprise Security Policy Evaluator (ESPE). ESPE enhances the security and stability of an application by detecting and reporting the following problems and inconsistencies:

1. The roles required at a program point according to the deployment descriptors are insufficient to run the application. If a user were granted only those roles, the container would generate an authorization failure during the execution of the application. (See Case 1 in Figure 1).

2. Some of the roles required at a program point according to the security policy are redundant. A user who is not granted those roles would still be able to access the application. Therefore, granting those roles would constitute a violation of the Principle of Least Privilege. (See Case 2 in Figure 1).

3. The role set by a principal-delegation policy is not sufficient to cover the role requirements for subsequent execution points. The container will generate an authorization failure during the execution of the application (See Case 3 in Figure 1).

4. A component's principal-delegation policy is unnecessary; subsequent execution points do not specify any role requirements. Such principal-delegation policy might constitute a violation of the Principal of Least Privilege. (See Case 4 in Figure 1).

5. An enterprise resource that was configured as inaccessible can be reached from an application entry point through a sequence of invocations, the last of which is an intercomponent call. This is a stability problem because when the J2EE container performs an authorization check, an authorization failure will occur. This will interrupt the application's execution. (See Case 5 in Figure 1).

6. An enterprise resource that was configured as inaccessible can be reached from an application entry point through a sequence of invocations, the last of which is an intracomponent call. The container will not perform any authorization check when the resource is accessed at run time and the application execution will not be stopped. However, this is a potential access-control violation since there exists a possibly undetected path reaching a resource that was intended to be inaccessible. (See Case 6 in Figure 1).

7. A particular execution point can be reached from two or more components each of which sets its own principal delegation policy. If the roles set by the principal delegation policies are different, authorization checks performed at subsequent execution points may fail, causing the container to stop the execution of the application. (See Case 7 in Figure 1).

The rest of the paper is organized as follows: Section 2 describes the J2EE RBAC model and shows how configuring the security of a J2EE application can lead to security flaws and stability problems. In Section 3, we present our interprocedural analysis algorithm for security and stability problem detection and security-policy validation. In particular, we describe two algorithms: one for determining role requirements and another for detecting security and stability flaws in the presence of principal-delegation policies. We conclude this paper with a section on our experience with ESPE and a section on other work that has been done in the area of RBAC and program analysis for security.

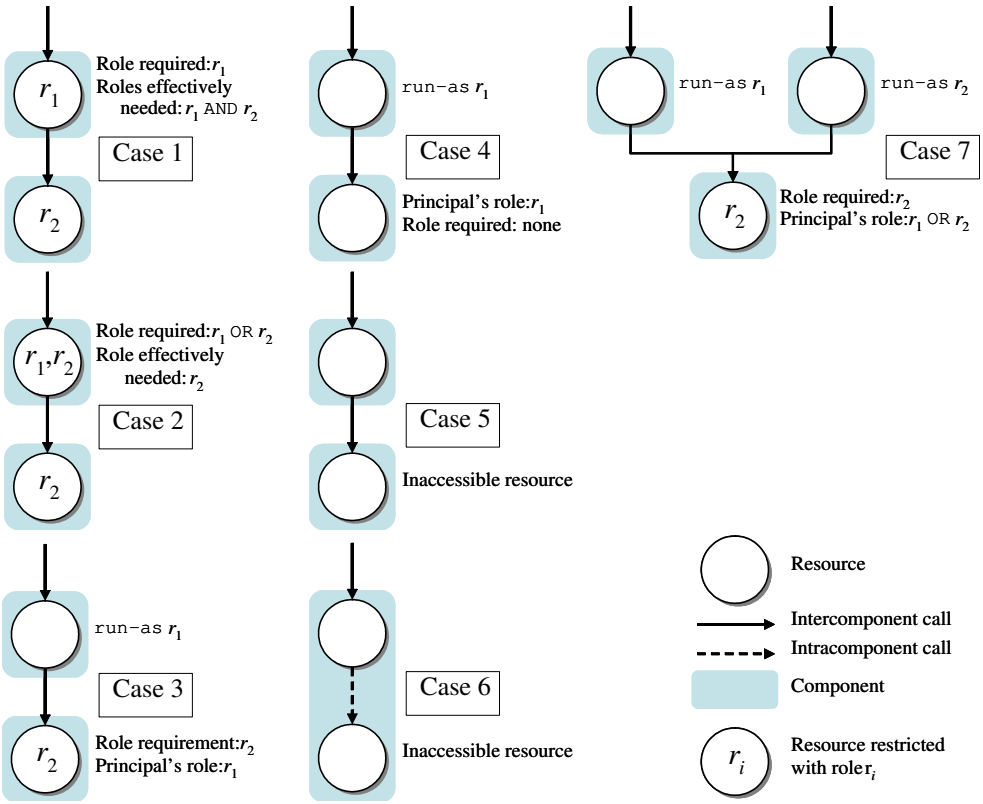

Figure 1: Security and Stability Problems Detected by ESPE

2 A Model for RBAC in J2EE

In this section, we define a simple model for RBAC on which we perform static analysis to identify security flaws and inconsistencies in security configurations. We focus on the J2EE component model [17], in particular the Enterprise JavaBeans (EJB) [12], Java Servlet [35], and JavaServer Pages (JSP) [21] models. Using our analysis framework, we highlight some of the security flaws and inconsistencies that can occur in J2EE security configurations.

2.1 The J2EE Component Model

A *component* is a software element with a contractually specified interface, conforms to a standard model, and can be independently deployed and composed without modifications according to a composition standard. *Composed software systems* are comprised of multiple, possibly heterogeneous, and potentially distributed components. As such, these systems present special security requirements. In particular, security requirements may vary by component, depending on the resources that each component contains. J2EE is a component-based system. Components include enterprise beans [12], servlets [35], JSP programs [21], and J2EE clients. A component can contain security-sensitive resources that need to be access-restricted. For example, if some of the business methods of an enterprise-bean component perform security-sensitive operations, it should be possible to restrict access to those methods.

The methods of an enterprise bean are implemented in a class known as the *enterprise bean class*. However, for an enterprise bean method to be invoked by a client program (such as another enterprise bean, a servlet, or a stand-alone application), that method must be declared in a specific enterprise bean interface. There are four interfaces that an enterprise bean can implement: *remote*, *remote home*, *local*, and *local home*.

Methods implemented in the remote and remote home 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 local home interfaces can be invoked by client programs co-located in the same container (same address space). *Helper methods*—those implemented by the enterprise bean class and not declared in any enterprise bean interface—are inaccessible to clients and can only be invoked from the component to which they belong. According to the J2EE specification, access restrictions on an enterprise bean method is enforced only when that method is accessed from a different component. If that method is invoked from another method within the same component, the J2EE container will not enforce access control restrictions. Since helper methods can only be invoked from the component to which they belong, it follows that helper methods are not subjected to access control restrictions.

Access to a servlet or JSP application can be restricted by restricting access to its Uniform Resource Locator (URL) or Uniform Resource Identifier (URI). Besides restricting access to a URL or URI, J2EE allows restricting access based on the HyperText Transfer Protocol (HTTP) methods `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, and `TRACE`. If a servlet or JSP application's URL or URI has been access-restricted and one or more HTTP methods have been specified in the access-restriction policy, the methods in the `HttpServlet` object corresponding to those HTTP methods will become access-restricted. For example, if the URL matching a servlet has been access-restricted and, along with that URL, the HTTP methods `GET` and `POST` have been restricted too, the servlet container will perform access-control restrictions when the servlet's `doGet()` and `doPost()` methods are invoked through the `service()` method as a result of a request from a client. However, if `doGet()` or `doPost()` are invoked from other methods within the same servlet, no authorization check will be performed.

When an application does not execute due to an authorization failure in a component, the J2EE container prevents the execution of that component. However, in a distributed component-based system, the container often does not provide the exact execution point at which the problem occurred. For large applications, manually tracking the error back across the distributed call stack is difficult, if not impossible. The algorithm described in this paper automates the process of call stack inspection for program understanding and security analysis also in the presence of intra- and intercomponent calls.

2.2 Scenarios

In this section, we present two scenarios highlighting security flaws. The first scenario demonstrates how roles attached to different resources generate logical expressions of role requirements, which may be difficult to evaluate. Complications are also introduced by principal-delegation policies. In the second scenario, we try to capture points of authorization failure. Both these scenarios emphasize the need for a tool that can not only detect security flaws, but also identify those points in the code where a security fault may occur.

2.2.1 Role Expressions

In the scenario of Figure 2, the application's entry point has been restricted so that it can only be accessed by users who have been granted role r_1. However, if a user were only granted role r_1, the application would not execute successfully. In fact, a more accurate analysis reveals that the roles required to access the application through that entry point are r_1 `AND` (r_2 `OR` r_3) `AND` (r_1 `OR` r_5), which evaluates to r_1 `AND` (r_2 `OR` r_3). Notice that granting roles r_4 or r_5 would constitute a violation of the Principle of Least Privilege:

- The resource restricted with r_4 is only accessed from its own component. Therefore, the container will never check that the user has been granted role r_4.
- Since r_1 is explicitly required, role r_1 is sufficient for the user to pass the authorization test for r_1 `OR` r_5.

The evaluation of logical expressions of roles for an application may be further complicated by the presence of principal-delegation policies. When a component sets a principal-delegation policy, all the downstream

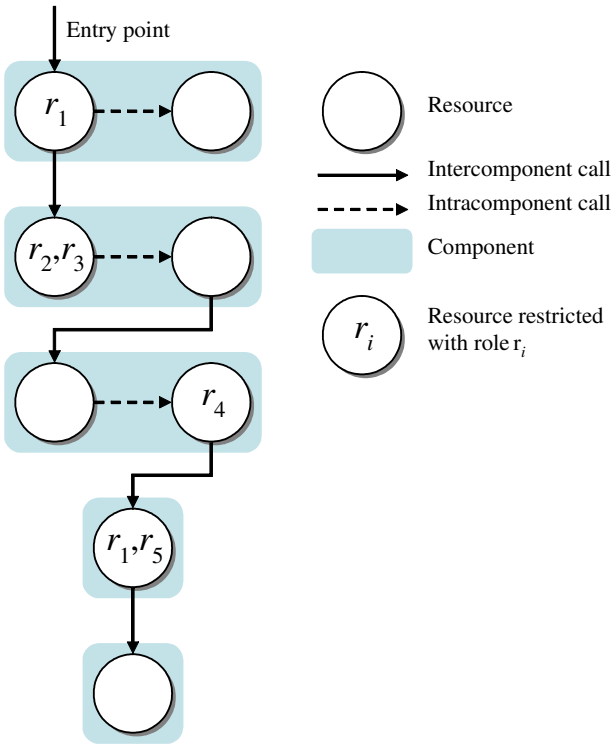

Figure 2: Generating Logical Expressions of Roles

calls will be performed as if the user had only been granted the role specified in the principal-delegation policy's `run-as` deployment descriptor entry. It should be noted that:

- Even though a component entry point may require more than one role to be executed, as in the scenario of Figure 2, only one role can be specified as part of a principal-delegation policy.
- If a user has been granted multiple roles, traversing a component that uses a principal-delegation policy will override all those roles with the single role specified in the component's `run-as` deployment descriptor entry.
- When a component uses a principal-delegation policy, *all* the downstream calls from that point onward will be performed with the role specified in the `run-as` deployment descriptor entry. The J2EE container does not revert the role back after the first downstream call has completed.

Using a principal-delegation policy can easily create security misconfigurations or violations of the Principle of the Least Privilege. The algorithm described in this paper automatically detects all the security problems and inconsistencies that could be generated as a result of using principal-delegation policies.

2.2.2 Points of Failure

The application depicted in the call graph of Figure 3 is destined to fail with an error message that, unfortunately, does not reveal details of where the failure occurred:

```
Application threw an exception: java.rmi.ServerException:
Nested exception is: java.rmi.AccessException: CORBA NO_PERMISSION 9998
```

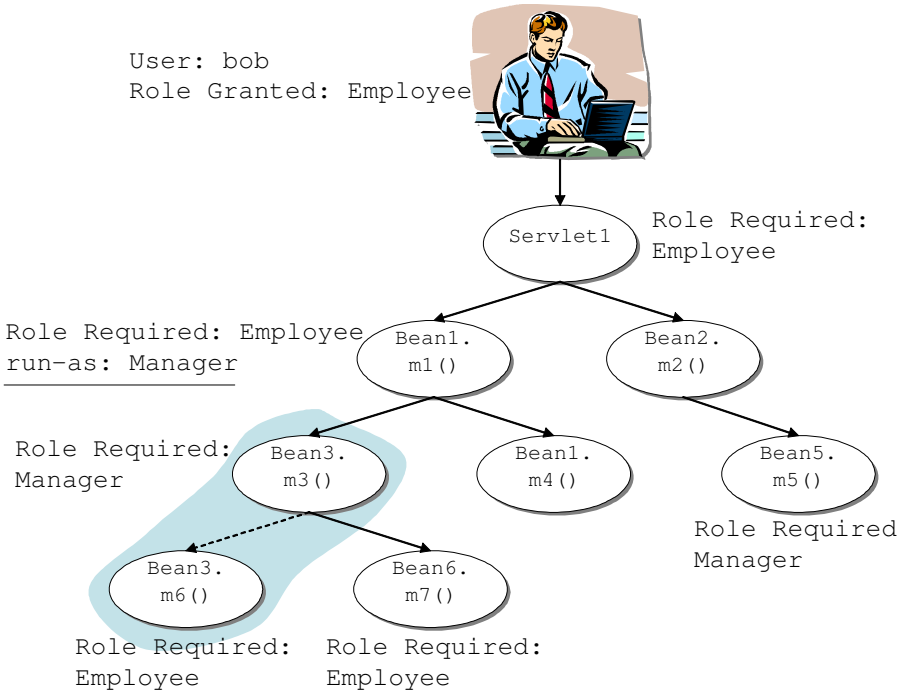

Figure 3: J2EE Authorization Scenario

Understanding the reason of the failure without an automated tool may be difficult and time consuming because it may require performing inspection of code and deployment descriptors. Dynamic analysis is limited too because it relies on having a complete set of test cases covering all possible paths through the set of component resources in an application. In the absence of a complete set of test cases, authorization failures may remain undiscovered until the code is deployed in the enterprise.

A first pass through the code and the deployment descriptors shows that user `bob` has been granted the `Employee` role, which is exactly the role required to invoke `Servlet1`. What may not be so evident is that `Servlet1` invokes `m2()` on enteprise bean `Bean2`, which is not an access-restricted resource, and `m2()` invokes `m5()` on `Bean5`, which has been access-restricted with the `Manager` role. Since the identity of the user, by default, is propagated with no changes, this invocation will fail because user `bob` does not have the required role.

Another point of failure is the invocation of `m7()` on `Bean6`. When `Servlet1` invokes `m1()` on `Bean1`, the authorization check succeeds because access to `m1()` is restricted with the `Employee` role, which is exactly the role possessed by `bob` and propagated by the container so far. Next, `m1()` invokes `m3()` on `Bean3`. This resource requires the user to be a `Manager`. However, `Bean1` uses a delegation policy that overrides the roles of the user and forces all the subsequent downstream calls to be performed under the `Manager` role. Therefore, the authorization check for `m3()` succeeds. The problem is that `Bean3` does not set the principal-delegation policy back to `Employee`, which is the role required to invoke `m7()` on `Bean6`. Therefore, user `bob`, who was granted the role of `Employee` at the beginning, is now denied access to `m7()` for not having the role of `Employee`. Interestingly, the invocation of `m6()` on `Bean3` does not cause an authorization failure even though access to `m6()` also requires the `Employee` role. The reason is that both `m6()` and its predecessor `m3()` belong to the same component, `Bean3`, and the container does not perform authorization checks on intracomponent resource accesses.

3 Interprocedural Analysis for RBAC

In this section, we present a simple and effective interprocedural analysis algorithm for detecting security flaws and inconsistencies in security configurations. We implemented our analysis in the ESPE tool. First, we describe the analysis framework, called DOMO, upon which ESPE is implemnented. Then, we present a basic role analysis that ignores principal-delegation policies. Finally, we show how to handle principal-delegation policies.

3.1 Analysis Framework

ESPE is implemented on top of the DOMO framework, a sophisticated bytecode interprocedural analysis framework whose precision can be customized [15]. DOMO supports a number of object-oriented call-graph-construction and pointer-analysis algorithms, which, in order of precision, are:

1. Class Hierarchy Analysis (CHA) [10]
2. Rapid Type Analysis (RTA) [2]
3. Context-insensitive Control-Flow Analysis disambiguating between heap objects according to concrete types (0-CFA) [36]
4. Context-insensitive Control-Flow Analysis disambiguating between heap objects according to allocation sites (0-1-CFA) [16], as in Andersens analysis [1]
5. Context-insensitive Control-Flow Analysis disambiguating between heap objects according to allocation sites, enhanced with extra context for Container objects (0-1-C-CFA) [15]
6. Context-sensitive Control-Flow Analysis (1-CFA) [16]

As a result, ESPE supports a range of cost/precision analysis trade-offs. If necessary, to minimize the number of false positives, precision can be enhanced by selecting a more context-sensitive analysis algorithm.

We also capitalize on another important characteristic of the DOMO framework—DOMO supports modelling the execution of J2EE applications. Consider for example an enterprise bean having remote interface `Bean2`, remote home interface `Bean2Home`, and enterprise bean class `Bean2Bean`. Suppose that method `m1()` in enterprise bean `Bean1Bean` calls remote method `m2()` on enterprise bean `Bean2Bean`. For this to be possible, `m2()` must be a method declared in `Bean2` and implemented in `Bean2Bean`, and a code similar to the following must be embedded in `m1()`:

```
Context initial = new InitialContext();
Object objref = initial.lookup("java:comp/env/ejb/Bean2");
Bean2Home bean2Home = (Bean2Home) PortableRemoteObject.narrow(objref, Bean2Home.class);
Bean2 bean2Objetc = bean2Home.create();
bean2Object.m2();
```

A traditional static analysis engine would report an edge from the node representing `m1()` to the node representing the invocation of `m2()` on an object of type `Bean2`, but there would not be any edge leading to the actual implementation of `m2()` in `Bean2Bean`. Since `Bean2` does not implement `m2()`, the control-flow graph would be truncated. Conversely, DOMO creates an additional edge that links the declaration of `m2()` in the remote interface `Bean2` to the actual implementation of `m2()` in the enterprise bean class `Bean2Bean`, as shown in Figure 4. Calls to methods declared in the remote home, local, and local home interfaces are modelled in a very similar way.

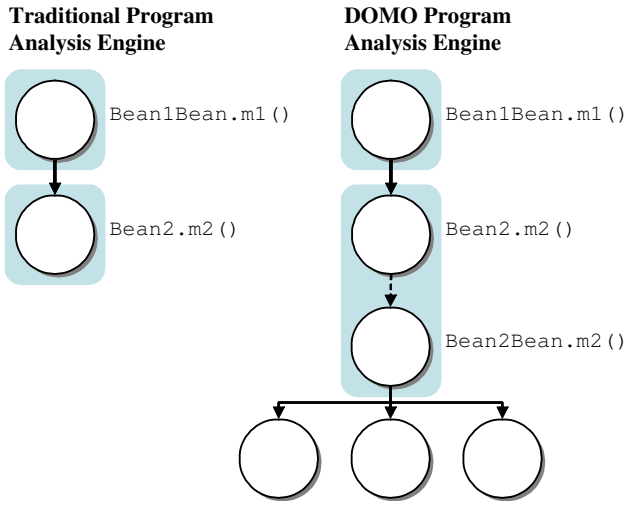

Figure 4: Traditional Program Analysis Engine vs. DOMO

3.2 Call Graph Construction

The first step of the algorithm is to build the interprocedural call graph for the entire J2EE application. In a J2EE application, we identify the following sets of entry points:

- For each servlet or JSP program, the entry points are the life-cycle methods, which are called by the servlet container as a result of invocations from client programs. These methods are `init()`, `service()`, and `destroy()`.
- For each enterprise bean in the application, the entry points are all the methods declared in the remote, remote home, local, and local home interfaces.
- For each J2EE client, the only entry point is the `main()` method.

It should be noted that in the call graph, the method entry points can have parameters, which may include the receiver object. During analysis, the values and object sources of these parameters are unknown, since they are part of the client application, which will only be available at run time. For each parameter object, DOMO uses CHA to build the class hierarchy rooted at the parameter's declared type. When a call back from that parameter object is encountered, DOMO models it by looking for possible implementations of the invoked method in the class hierarchy. Additionally, DOMO can distinguish between *intercomponent edges*, which are those representing intercomponent accesses, and *intracomponent edges*, those representing intracomponent accesses.

3.3 RBAC Analysis

In this section we present a RBAC analysis for computing the effective access-control requirements for all the resources within an application. The first step of the algorithm is to identify which resources in the application are access-restricted and the roles attached to them. The J2EE access-control model is fine grained because it enables restricting access to specific Web resources. For example, rather than restricting access to an enterprise bean as a whole, it is possible to restrict access to one or more of its methods, and different methods of the same enterprise bean can be access-restricted with different roles. In Section 1, we showed a deployment descriptor fragment defining a security role for a J2EE application and using that role to restrict access to an EJB method. Similarly, it is possible to restrict access to servlet and JSP HTTP methods, HTML pages, and any other resource that could be accessed through a URL or URI.

Mapping nodes in the call graph to roles is a straightforward operation that requires scanning the deployment descriptors of all the application components. Once those resources have been identified along with the roles necessary to access them (or no roles, in the case of inaccessible resources), a simple top-down traversal allows locating the nodes in the call graph that represent those resources. Depending on the level of context sensitivity chosen to construct the call graph, a single access-restricted resource may correspond to more than one node in the call graph. For example, there can be multiple nodes in the call graph corresponding to a single EJB method, each of those nodes representing an invocation of that method with a different receiver and/or different parameter objects.

The next step in the algorithm is to identify which roles are effectively required to invoke a resource. The scenario depicted in Figure 2 shows that when multiple access-restricted resources access each other, forming an invocation sequence, then the user accessing the first of those resources will have to be granted not just the role needed to access the first of those resources, but all the roles necessary to invoke all the resources that are accessed through intercomponent calls. Understanding the roles effectively required to access a resource a in an invocation sequence requires detecting all the roles needed to access all the resources directly and indirectly reachable from a through intercomponent accesses. This computation can be performed through a *reverse role propagation* across the call graph edges.

The reverse role propagation in the call graph is initialized by mapping each intercomponent edge leading to an access-restricted resource to the roles required to access that resource. For example, the edge connecting the two nodes in Case 2 of Figure 1 would be mapped to role requirement r_2, while the entry edge would be mapped to role requirement r_1 `OR` r_2. Next, each of these edges propagates the role requirements to its predecessor edges. The operation performed on the role requirements associated with each edge is a logical `AND`. Every time the role requirements associated with an edge change as a result of a propagation, that edge must in turn propagate its role requirements to its predecessor edges. If cycles are present, they can be identified prior to the propagation phase and logically condensed into a single node whose role requirement is the `AND` of the role requirements of all the nodes in the cycle accessed through intercomponent edges. Notice that excluding intracomponent edges from the initialization phase—even when these lead to access-restricted resources, such as the resource restricted with role r_4 in Figure 2—correctly models the lack of intracomponent authorization checks at run time.

The reverse role-propagation process is monotonic, and since the universe of role requirements associated with any application is finite, this algorithm is always guaranteed to converge [23]. It is simple to show that the complexity of reverse role propagation algorithm is $\mathcal{O}(|E| \times |R|)$ where E is the set of edges in the call graph and R is the set of all the roles defined in the application's deployment descriptors.

Upon completion, the reverse role propagation algorithm yields a mapping of edges to role requirements. In particular, the mapping of intercomponent edges to role requirements can be used to understand which roles are necessary to invoke an access-restricted resource, and this information can in turn be used to detect stability problems and security flaws.

3.3.1 Stability Problems Due to Insufficient Role Assignments

Figure 5 shows the initialization and termination of the reverse role-propagation algorithm applied to the scenario of Figure 2. It can be seen that the roles effectively needed by a user to access the application through the entry point are r_1 `AND` (r_2 `OR` r_3) `AND` (r_1 `OR` r_5) = r_1 `AND` (r_2 `OR` r_3). If only role r_1 were granted, like a simple parsing of the deployment descriptor would seem to suggest, the container would generate an authorization failure. (See Case 1 in Figure 1).

3.3.2 Security Flaws Due to Redundant Role Assignments

In other cases, after the reverse role propagation algorithm has terminated, comparing the roles necessary to invoke the resource represented by a node to the union of the roles associated with the outgoing edges can help preventing violations of the Principle of Least Privilege. For example, in the scenario of Case 2 in Figure 1, the roles necessary to invoke the entry point are r_1 `OR` r_2. However, at the end of the reverse role propagation, the only edge leaving the root node will be mapped to role r_2. The only role effectively needed

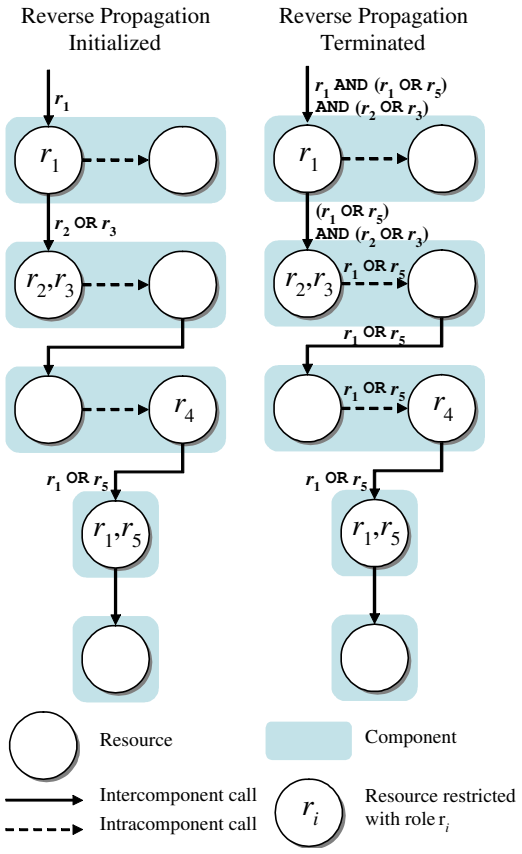

Figure 5: Reverse-Role Propagation Algorithm

by a user to access the application through the entry point is r_2. Granting the user role r_1 alone would cause an authorization failure, while granting the user r_1 `AND` r_2 would constitute a violation of the Principle of Least Privilege because r_1 is redundant.

3.3.3 Security Flaws Due to Inaccessible Resources

Recall that in J2EE, an enterprise resource can be marked as inaccessible by explicitly configuring the deployment descriptor of the resource's component and specifying that the resource should not be accessed by any user, regardless of the user's roles. Every time that resource is accessed through an intercomponent call, the container will perform an authorization check and will prevent the component from executing. We encode such inaccessible resources as if they were available only to users with an `Inaccessible` role. If a node in the call graph is mapped to the `Inaccessible` role, then:

1. All the intercomponent edges incident into that node are indications of potential stability problems (see Case 5 in Figure 1). In fact, an intercomponent incident edge represents an intercomponent access on which the container will perform an authorization check. That check is destined to fail regardless of the roles granted to the user initiating the transaction. As a result, the application will have a run-time authorization failure.

2. All the intracomponent edges incident into that node are indications of potential security inconsistencies (see Case 6 in Figure 1). Even though the container does not perform any authorization check on an intracomponent call, there exists a path leading to a resource that was intended to be inaccessible.

Using the reverse role-propagation algorithm, an `Inaccessible` role requirement can be propagated in the call graph from an inaccessible resource, just like any other role requirement. This allows detecting unintended paths leading to inaccessible resources.

3.4 RBAC Analysis in the Presence of Principal-Delegation Policies

If principal-delegation policies in an application's security configuration are detected, ESPE performs an additional RBAC analysis to detect security misconfigurations due to the use of principal delegation. Such misconfigurations could lead to violations of the Principle of Least Privilege and/or stability problems.

By parsing of the application deployment descriptors, ESPE maps each application component to the role set by the component's principal-delegation policy, if any. Subsequently, ESPE performs a *principal-delegation role propagation* which, as we shall see, is similar to the algorithm that computes reaching definitions [23]. At the end of the principal-delegation role propagation, each intercomponent edge e in the call graph will be mapped to a set containing all the roles set by principal-delegation policies that can reach e in the call graph.

The principal-delegation role propagation is a fixed-point iteration [23], initialized with those intercomponent edges whose tail node represents a resource in a component setting a principal-delegation policy. In the propagation process, each intra- and intercomponent edge propagates the roles it has accumulated to its successor edges. The operation performed on the principal-delegation roles associated with each edge is a logical `OR`, which is equivalent to considering each role as a singleton and then performing set unions. Every time the role set associated with an edge changes as a result of a propagation, that edge must in turn propagate its role requirements to its successor edges. When an intercomponent edge e is encountered as a result of a principal-delegation role propagation from a predecessor, the incoming principal-delegation role set is unioned with the role set already computed for e up to that point. However, if the tail node of e belongs to a component setting a principal-delegation policy to role r, then the singleton $\{r\}$ overrides, or *kills* [23], whatever set was computed up to that point for e.

The principal-delegation role propagation algorithm is monotonic, and since the universe of principal-delegation roles associated with any application is finite, this algorithm is always guaranteed to converge [23]. It is simple to show that the complexity of the principal-delegation role propagation algorithm also is $\mathcal{O}(|E| \times |R|)$, where E is the set of edges in the call graph and R is the set of all the principal-delegation roles defined in the application's deployment descriptors. One could improve the time complexity to $\mathcal{O}(|E| \times \log(|R|))$ using sophisticated elimination-style data-flow analysis algorithms [31].

Upon termination, the principal-delegation role propagation yields a mapping of edges to role sets, each set representing the principal-delegation roles that can reach the associated edge. In particular, the mapping of intercomponent edges to role sets can be used to understand which principal-delegation roles can propagate when an access-restricted resource is invoked. This information, in turn, can be used to detect stability problems and security inconsistencies.

3.4.1 Stability Problems Due to Incorrect Principal-Delegation Policies

Let us assume that, upon completion of the principal-delegation role-propagation algorithm, an intercomponent edge e is associated with role set $\{r_1, r_2, \ldots, r_k\}$. From a logical point of view, this means that the role expression associated with e is r_1 `OR` r_2 `OR` $\ldots$ `OR` r_k, or that when e is traversed, the user initiating the transaction will have exactly one of the roles in the set $\{r_1, r_2, \ldots, r_k\}$, depending on the execution path prior to traversing e. Let n be e's head node. If n is access-restricted with role requirement q_1 `OR` q_2 `OR` $\ldots$ `OR` q_h, this security configuration will not generate an authorization failure and a consequent stability problem for the application if and only if $k \geq 1$ and $\{r_1, r_2, \ldots, r_k\} \subseteq \{q_1, q_2, \ldots, q_h\}$. (See Cases 3 and 7 in Figure 1).

Application	Methods Analyzed	Analysis Time (sec)	Nodes	Edges	Heap Size (MB)
A	477	30	491	1191	65
B	1182	21	1184	8186	62
C	519	25	521	1394	62
D	766	27	771	2142	62
E	1536	30	1743	7760	64

Table 1: Analysis Results (Applications are anonymous)

3.4.2 Security Flaws Due to Redundant Principal-Delegation Policies

As in Section 3.4.1, let e be an intercomponent edge with head node n and let $\{r_1, r_2, \ldots, r_k\}$ be the role set associated with e upon termination of the principal-delegation role propagation algorithm. If $k \geq 1$ and n is not access-restricted at all ($h = 0$, according to the notation of Section 3.4.1), then this is a potential violation of the Principle of Least Privilege, as discussed in Section 1. (See Case 4 in Figure 1).

4 Experience with ESPE

ESPE runs as a stand-alone application and can be launched on top of a Java 2, Standard Edition (J2SE) V1.3 or V1.4 run-time environment from the command line or from within Eclipse V2.1 or V3.0. It can analyze J2EE V1.3 and V1.4 deployed applications packaged in Enterprise ARchive (EAR) files [17] or as separate Java ARchive (JAR) [12] and Web ARchive (WAR) files [35, 21]. These files contain the object code and deployment descriptors of one or more applications. Source code is unnecessary since ESPE analyzes object code. All the library files used by the applications at run time (including `rt.jar` and `j2ee.jar`) are included in the analysis in order for ESPE to construct a complete call graph and provide sound results. For the reports provided by ESPE to be meaningful, the deployment descriptors of the applications being analyzed must contain security configuration information, including the definition of at least one role and an RBAC policy.

Context- and flow-sensitive static analysis has a reputation for requiring significant processing power and memory. We have performed our RBAC analysis on a number of sample programs and commercial J2EE applications. The results reported in Table 1 are from running ESPE on a desktop computer with an Intel Pentium M 1.6 GHz processor and 1 GB of Random Access Memory (RAM). The operating system was Microsoft Windows XP SP1. We ran ESPE from within Eclipse V3.0 using a J2SE V1.4.2_04 run-time environment. Among the applications listed in Table 1, only application *A* makes use of principal-delegation policies. Additionally, application *A* extensively uses RMI-IIOP method invocations. In all those applications, ESPE found several security problems due to insufficient role assignments. Application *A* presented also some stability problems that were due to an incorrect use of principal-delegation policies. DOMO supports multiple levels of call graph precision. While analyzing the applications listed in Table 1, no false positives were detected using DOMO's implementation of RTA, which suggests that using an RTA algorithm is sufficient for ESPE.

Performance is improved by ignoring library method invocations that do not lead to any access-restricted resources. The data-flow analysis to model such calls requires a substantial amount of time and has no security implications. This optimization is a consequence of the fact that with DOMO the analysis can be performed incrementally. Once it has been established that the invocation of a particular library will never lead to an authorization check, the data-flow analysis for that library can be avoided. ESPE is being used by us and others to help application deployers configure the security of J2EE applications. Our preliminary experience has been quite positive.

5 Related Work

Ferraiolo and Kuhn introduced RBAC in 1992 [14]. Subsequently, Sandhu, Coyne, Feinstein, and Youman described mechanisms for constructing and analyzing RMAC models and implementations [33]. More recently, Schaad and Moffett [34] employed the Alloy specification language [18] to model RBAC. In particular, they used the the Alloy constraint analyzer Alcoa [19] to verify key characteristics of the model, such as separation of duties assigned to roles.

Several static and dynamic analysis techniques have been suggested in the area of Web applications, but these works do not provide a significant support for distributed applications, nor do they deal with security issues. For example, Ricca and Tonella [29] propose a Unified Modelling Language (UML) model for Web applications, but their model deals primarily with structural testing of interactive features of Web applications, such as HyperText Markup language (HTML) forms. Brucker and Wolff [5] introduce a mechanism for dynamic analysis of distributed-component systems using the Object Constraint Language (OCL) [37] of the UML standard to formalize component specifications.

An EJB object's confinement can be breached when a direct reference to the EJB object is returned to a client. Such a reference could allow the client program to access security-sensitive fields or invoke enterprise bean business methods without going through the EJB interfaces. In particular, this implies that any access-control restriction on those methods could be bypassed. The purpose of the work of Clarke, Richmond, and Noble [6] is to enforce confinement of EJB objects. They propose simple coding guidelines that, if observed, prevent confinement breaches. Additionally, they describe a straightforward code inspection algorithm that checks for violations of those guidelines in enterprise bean programs. However, their analysis only considers EJB components; servlets and JSP programs are not analyzed. Additionally, RBAC issues are not taken into account.

In addition to RBAC [26], Java offers a low-level access control mechanism to protect static resources, such as the file system, network, and operating system properties [27]. Access control decisions are based on the origin of the code (signers and/or code's network location) and/or the principal running the code. Both static and dynamic analysis techniques are employed in modelling security and authorization. Much of this work has been applied to eliminate or minimize redundant authorization tests or define alternatives to the current approach to defining authorization points within code.

Pottier, Skalka, and Smith [28] extend and formalize Wallach's security passing style [39] via type theory using a λ-calculus, called λ_{sec}. However, their work focuses only on J2SE authorization issues. Additionally, they were unable to perform incomplete-program analyses [30].

Jensen, Métayer, and Thorn [20] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests. They assume all of the code is available for analysis, and that a call graph can be constructed for the code, though they do not do so themselves.

Bartoletti, Degano, and Ferrari [4] are interested in optimizing performance of run-time authorization testing by eliminating redundant tests and relocating others as is needed. The reported results apply operational semantics to model the run-time stack. Similarly, Banerjee and Naumann [3] apply denotational semantics to show the equivalence of "eager" and "lazy" semantics for stack inspection, provide a static analysis of *safety* (the absence of security errors), and identify transformations that can remove unnecessary authorization tests. Significant limitations to this approach are that the analyses are limited to a single thread, and require the whole program; incomplete-program analyses are not supported.

Naumovich [24] proposes a data-flow algorithm for automated analysis of the flow of J2SE `Permission` objects in Java programs. The algorithm produces, for a given instruction in the program, the set of `Permission` objects that are checked on all possible executions up to that instruction. Such information can be used in program understanding tools or directly for checking properties that assert what permissions must always be checked before access to a certain functionality is allowed.

The aforementioned works are specifically designed for J2SE authorization problems and they assume that call graph algorithms are available to translate the theoretical approach into a practical implementation. However, many of the well-known call-graph-construction and data-flow algorithms [30] do not correctly model J2EE intercomponent calls and are too conservative to correctly identify authorization requirements.

The call graph used by ESPE models J2EE intercomponent calls and minimizes the conservativeness to get more accurate authorization information. Additionally, in J2EE, access control is enforced differently from the way it is enforced in J2SE. In J2SE, any security-sensitive resource is by default access-restricted; the Java 2 `SecurityManager` object denies access to the resource unless authorizations have been explicitly granted to the code and/or the user [27]. Conversely, in J2EE, a security-sensitive resource is by default fully accessible and it becomes access-restricted only when an explicit restriction is declared in the component's security policy [26]. All the works described above assume that authorization checks are obtained by passing a `Permission` object to a `checkPermission()` function, but this assumption cannot be made for J2EE because access restrictions are enforced by the container in a vendor-specific way.

Koved, Pistoia, and Kershenbaum [22] describe an algorithm and an actual implementation for correctly identifying the J2SE authorizations needed by a Java program. While their work is specific to J2SE `Permission` objects, it models multi-threading, privileged code, and security policy dynamically enforced by a Java 2 `SecurityManager`, and supports incomplete-program analysis. Their control- and data-flow analysis framework reduces conservativeness by selectively increasing the precision with which `Permission` objects are represented.

Felten, Wallach, Dean, and Balfanz have studied a number of security problems related to mobile code [40, 11, 39, 7, 38, 9, 8]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals currently active in a thread stack at run time (*security state*). In particular, an authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [39]. Each method is modified so that it passes a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, the security passing style explores subgraphs of the comparable invocation graph, and discovers the associated security states and authorizations. Their goal is to optimize the authorization performance, while ours is to discover authorization requirements by analyzing all possible paths through the program, even those that may not be discovered by a limited number of test cases.

Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [13] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. We examine the authorization issue from the perspective of an existing system containing authorization test points. Through static analysis, we discover how the security policy needs to be modified or updated to enable the code to execute.

Finally, Naumovich and Centonze [25] describe how the J2EE authorization model, which is designed to restrict access to EJB methods, can be extended to restrict access to the data handled by those methods. They use points-to analysis [30] to identify which EJB methods access security-sensitive data and the mode of access (read and/or write). Subsequently, RBAC on that data can be achieved by enforcing RBAC on the methods accessing the data. Their work can also be used to find access-control inconsistencies whereby two EJB methods accessing the same data in the same mode should be restricted with the same roles. However, they do not take into account the logical expressions of role requirements that are generated by sequences of intercomponent calls nor the side effects of principal-delegation policies. Since their purpose is to restrict access to the EJB data, they focus on RBAC on EJB methods, but do not deal with RBAC on servlets and JSP programs.

6 Conclusion

In this paper, we presented a simple framework for static analysis of RBAC. We implemented the framework in our ESPE tool, which can be used to help application deployers and system administrators to correctly configure the security of a J2EE application. Our approach to the analysis of RBAC is based on propagating roles over an interprocedural call graph. The interprocedural static analysis used in ESPE is customizable to improve the precision for computing the set of roles required at each program point. Using the role information computed statically, we showed how to identify whether too many or too few roles have been

granted, and detect potential security policy inconsistencies. ESPE is currently being used to configure security requirements of large J2EE applications, and the experience with using the ESPE tool has been very positive. While the analysis techniques described in this paper are in the context of J2EE code, the basic techniques are applicable to RBAC analysis issues in non-Java-based systems as well.

References

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

[2] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, 1996. ACM Press. Also published in ACM SIGPLAN Notices 31(10).

[3] Anindya Banerjee and David A. Naumann. A Simple Semantics and Static Analysis for Java Security. Technical Report CS2001-1, Stevens Institute of Technology, Hoboken, NJ, USA, July 2001.

[4] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54. Elsevier, 2001.

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

[6] 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.

[7] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 18–27, Zurich, Switzerland, 1997. ACM Press.

[8] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and beyond. In IEEE, editor, *1996 IEEE Symposium on Security and Privacy*, pages 190–200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.

[9] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-97, Princeton University, Princeton, NJ, USA, February 1997.

[10] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[11] Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, January 1999.

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

[13] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Oakland, CA, USA, May 2000. IEEE Computer Society.

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

[15] 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.

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

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

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

[19] 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.

[20] Thomas P. Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of Control Flow Based Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 89–103, Oakland, CA, USA, May 1999.

[21] JavaServer Pages™Specification, `http://java.sun.com/products/jsp/`.

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

[24] 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 (ISSTA '02)*, pages 33–43, Rome, Italy, July 2002.

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

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

[27] 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.

[28] François Pottier, Christian Skalka, and Scott F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.

[29] 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.

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

[31] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

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

[33] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.

[34] 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, Monterey, CA, USA, 2002. ACM Press.

[35] Java™Servlet Specification, `http://java.sun.com/products/servlet/`.

[36] Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988.

[37] Object Management Group, Object Constraint Language Specification, `http://www.omg.org/uml/`.

[38] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, 1997. ACM Press.

[39] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

[40] Dan Seth Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Princeton University, Princeton, NJ, USA, January 1999.