

Static Analysis of Role-Based Access Control in J2EE Applications*

Gleb Naumovich
Department of Computer and Information
Science
Polytechnic University
5 MetroTech Center
Brooklyn, NY 11201
gleb@poly.edu

Paolina Centonze
Department of Computer and Information
Science
Polytechnic University
5 MetroTech Center
Brooklyn, NY 11201
pcento02@utopia.poly.edu

Abstract

This work describes a new technique for analysis of Java 2, Enterprise Edition (J2EE) applications. In such applications, Enterprise Java Beans (EJBs) are commonly used to encapsulate the core computations performed on Web servers. Access to EJBs is protected by application servers, according to role-based access control policies that may be created either at development or deployment time. These policies may prohibit some types of users from accessing specific EJB methods.

We present a static technique for analyzing J2EE access control policies with respect to security-sensitive fields of EJBs and other server-side objects. Our technique uses points-to analysis to determine which object fields are accessed by which EJB methods, directly or indirectly. Based on this information, J2EE access control policies are analyzed to identify potential inconsistencies that may lead to security holes.

1. INTRODUCTION

In recent years, Web applications have become mainstream. Such applications typically run in a client-server fashion. The server side contains repositories of data used by the application, as well as computational logic for processing that data. In addition, the server side may contain modules for interacting with clients. Commonly, this is done by the means of generating HTML pages that are downloaded by clients and viewed in the clients' Web browsers. Since Web applications execute in the untrusted environment of the Inter-

*This research was partially supported by a Capacity Building in Information Security Research and Education grant from Defense Advanced Research Projects Agency (DoD Contract F49620-01-1-0243) and by the National Science Foundation under Grant CCR-0093174. The views, findings, and conclusions presented here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, National Science Foundation, or the U.S. Government.

net, security of such applications is an important issue.

With many Web applications, the functionality and data available to a client depends on whether this client belongs to a pre-defined category of users. For example, in a Web application that supports teaching a course, professors and students are granted different levels of access: professors can post grades for students, while students can only check these grades. Such groups of users are typically called *roles* and the mechanism for controlling the level of data and functionality available to roles is called *role-based access control* [5]. Clearly, it is important that developers and installers of a Web application correctly identify the data and functionality that should be available to a role. A failure to do so can leave Web applications vulnerable to unauthorized access.

With the rise in popularity and penetration of Web applications, enabling technologies and frameworks for such applications have proliferated. In recent years, Java has become a popular platform for implementing Web services and applications. A *Web service* is an interface that describes a collection of network-accessible operations based on open Internet standards [18]. Java 2, Enterprise Edition (J2EE) [27] includes a number of technologies supporting development and deployment of Java-based Web services and applications. Enterprise Java Beans (EJBs) are distributed objects commonly used to encapsulate the core computations performed on Web servers. Access to EJBs is protected by application servers, according to role-based access control policies. These policies define the level of access to the application functionality by restricting access to the EJB methods and servlets that can be called by clients. In this paper, we present a technique for static analysis of the role-based access control model used in J2EE applications. Since Web services are often implemented as EJBs, this work applies in particular to role-based access control for J2EE-based Web services.

At present, application developers and/or deployers determine roles that make sense for the application or its specific deployment configuration and then determine which EJB methods each role should be allowed to call. Therefore, access is defined in terms of operations on components, instead of data encapsulated and used by the components. We believe that basing access control on data could be useful in two ways:

1. In some situations, defining access on the basis of data is more straightforward and convenient (and therefore less error-prone) than defining access on the basis of operations. For

example, for a Web application that allows professors to assign grades for the students enrolled in a class and for 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 it is likely that there will be multiple methods for writing (e.g. `add`, `remove`, `modify`) and reading (e.g. `getAllGrades`, `getHomeworkGrade`) grades.

2. Even in the case where access is specified on the basis of operations, it may be useful to validate these specifications on the basis of data accessed and manipulated by these operations. Intuitively, if two methods that have been granted to two different roles access data in similar ways, it may be an indication that access granted to one of these roles is either too high or too low.

The latter of the two points above is the basis of the approach we describe in this paper. Our approach is based on static analysis, uses the Java code in the EJBs and the access control policy of the J2EE component under analysis, and finds potential problems related to assignment of access permissions to roles defined by the access control policy. Optionally, a user of our technique can reduce the scope of the analysis, specifying which data fields in the EJBs are considered security sensitive.

we define a simple semi-lattice based on operations performed on subsets of data in the component under analysis. Elements of this semi-lattice represent equivalence classes for access of data by individual methods and roles. We then use this semi-lattice for analysis of role-based access control policies on the basis of methods. If a role is given access to one method in an equivalence class but not other methods in this equivalence class or weaker equivalence classes, our analysis reports a potential problem. The problem could be that the role is given either too much or too little access. In this case, the analyst needs to adjust the security policy. The problem could also be that there is a bug in the implementation of one or more methods in the EJBs or their helper classes. In this case, the analyst needs to fix the bugs and re-run the analysis. The analyst may also decide that the level of access specified for the roles is adequate, due to considerations that are application-specific or even configuration-specific.

The rest of this paper is organized as follows. In Section 2, we give the background on J2EE and the role-based access policies for EJBs. Section 3 describes the points-to graph formalism we use for our analysis. Section 4 describes a semi-lattice that captures equivalence classes of accesses to data encapsulated by EJBs. In Section 5 we describe the analysis of this semi-lattice capable of detecting inconsistencies in granting EJB access to roles. Section 6 surveys the related work and Section 7 concludes and describes our future work.

2. ENTERPRISE JAVA BEANS

Customarily, J2EE applications are split into four tiers. The *information systems* tier holds persistent storage technologies, usually database management systems. The *business* tier consists of distributed objects that implement the core functionality, or *business logic*, of the application. Usually, these distributed objects are implemented as EJBs. EJBs run within *EJB containers*, off-the-shelf products that provide a number of important automated services,

such as persistence and transaction management. The *Web* tier is the application part that generates documents that are displayed by the browser on the clients of the application. This tier includes technologies such as servlets. Servlets commonly generate HTML pages that are served to clients by Web servers. The *client* tier includes components that execute on client machines and interact with the server-side tiers. Often, the client tier consists of browsers that display content generated by the Web tier. Client tier can also include applications that directly interact with EJBs in the business tier or (less frequently) with data storage components in the information systems tier.

2.1 EJB Basics

EJBs [24] encapsulate the business logic of J2EE applications. An EJB is responsible for connecting to legacy systems, such as databases, to perform transactions and retrieve information on behalf of the users. Individual EJBs are stand-alone components that can be assembled into an application. The general behavior of each EJB is represented by an EJB class and, optionally, a number of helper classes. This behavior may be modified to fit a specific configuration or even a specific installation of the application. This is done with a *deployment descriptor*, an XML file that describes the properties of EJBs, including their security, persistence, and transactional behavior. This approach reflects the philosophy of component-based development in J2EE: developers of EJBs concentrate on implementing the business logic, application assemblers concentrate on composing the EJBs to form applications, and application deployers concentrate on creating configurations of these applications, including configuring their security aspects [18]. This distribution of responsibilities means that a deployer may not be familiar with details of the implementation of EJBs and still be charged with specifying the access control policy specification for these beans. Our automated analysis is particularly valuable in such situations.

Within an EJB container, each instance of an EJB is represented by two objects. One is the actual EJB object whose methods implement the business logic and that maintains the state of the EJB; this object may be arbitrarily complex and may use other objects. The other is the *EJB interface object* that is used for all communications between the EJB object and clients making calls to its methods¹. For communication with remote clients, EJB interface objects normally employ the RMI protocol [26]. The EJB interface object may not include some of the methods in the EJB object. The EJB methods that are not included in the EJB interface object cannot be called by clients. This mechanism provides a convenient way of making some methods defined by an EJB invisible to clients. When a J2EE application server deploys an EJB object, it automatically generates and deploys the corresponding EJB interface object from an interface produced by the application developer or assembler.

Each EJB class (as opposed to an EJB object) is associated with an *EJB home object* that can be used by clients to obtain references to EJB interface objects corresponding to the EJB objects of this class². Whether a new EJB object is created and deployed

¹In this sense, the EJB interface object is a *proxy* for the EJB object. For this reason, it is important that no references to EJB objects or their helper distributed objects are returned by the methods in the EJB interface object [3]. Note that, despite the terminology, there is no inheritance relationship between the EJB object and the corresponding EJB interface object.

²An EJB home object is thus a factory object from the perspective of clients.

```

public interface Gradebook extends javax.ejb.EJBObject {
    public void addHomework(Homework h) throws RemoteException;
    public void removeHomework(Homework h) throws RemoteException;
    public Set homeworks() throws RemoteException;
    public void setGrade(Grade g, Student s, Homework h) throws RemoteException;
    public Grade getGrade(Student s, Homework h) throws RemoteException;
    public Map getAllGrades(Student s) throws RemoteException;
}

```

Figure 1: EJB interface for the Gradebook example

or an existing EJB object is re-used is determined by the application server and the type of this EJB object. A reference to an EJB home object is obtained by clients via the Java naming services; this requires each EJB class used by an application to be bound to a unique name. Similar to EJB instance objects, EJB home objects are created by the application server based on an interface that is supplied by the application developer or assembler.

In addition to EJB interface and EJB home objects, EJB local interface and EJB local home objects can be used³. These objects are used for method communications within the same EJB container. Therefore, remote clients cannot communicate with EJB local interface and EJB local home objects. From a performance perspective, invoking methods of EJB local interface objects is efficient, because, unlike in remote method invocations, the arguments and results of local method invocations are passed by reference rather than by value. From a security perspective, the usefulness of EJB local interface objects is that they could enable trusted components, such as servlets and EJBs, to call sensitive methods of EJB objects, while these methods cannot be called by remote clients directly. Similar to EJB interface and home objects, EJB local interface and home objects are created by the application servers based on interfaces supplied by application developers or assemblers.

2.2 Role-based Access Control for EJBs

For the purpose of illustration, we use an EJB for storing grades for students taking a course. Figure 1 shows the EJB interface for this EJB. Methods `addHomework` and `removeHomework` add the given homework to and remove the given homework from the set of all homeworks for the course, respectively. Method `homeworks` returns all homeworks for the course as a set. Method `setGrade` sets the given grade for the given homework for the given student. Methods `getGrade` and `getAllGrades` retrieve the existing grades: the former returns an individual grade for the given student and homework, while the latter returns a map from homeworks to grades for those homeworks for the given student.

The J2EE access control mechanism relies on a simple form of role-based access control [5]. It often happens in J2EE applications that, while no code from clients is executed on the server, certain types of clients must not be allowed access to certain functions of the server components. The J2EE access control model identifies *roles*, or types, of different clients and specifies what EJB functionality can be accessed by each role. Figure 2 gives an example J2EE access control policy specification (referred to simply as *access policy* hereafter) for the `Gradebook` example, where two roles, `student` and `professor`, are identified. According to this policy, professors are allowed to call all methods in the EJB interface for EJB `Gradebook`, indicated by symbol `*`, while students are

```

<assembly-descriptor>
  <security-role>
    <description>
      Any student at the university
    </description>
    <role-name>student</role-name>
  </security-role>

  <security-role>
    <description/>
    <role-name>professor</role-name>
  </security-role>

  <method-permission>
    <role-name>professor</role-name>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>

  <method-permission>
    <role-name>student</role-name>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        homeworks
      </method-name>
    </method>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        getGrade
      </method-name>
    </method>
  </method-permission>
</assembly-descriptor>

```

Figure 2: Example J2EE access policy

allowed to call only methods `homeworks` and `getGrade`. Note that by default, all roles are allowed access to all methods that are not explicitly mentioned in the model. For illustration, consider the access policy in Figure 3. This policy is equivalent to the policy in Figure 2. It specifies that only users in role `professor` have access to methods `addHomework`, `removeHomework`, `setGrade`, and `getAllGrades` of the `Gradebook` EJB. Methods `homeworks` and `getGrade` are not explicitly named by this policy, so both roles can call them. In addition to restricting access to methods of EJBs, J2EE access policies may restrict access to servlets and other shared resources (such as distributed databases), but since in this work we concentrate on EJBs, we do not handle such specifications.

³This feature was introduced in the EJB Specification 2.0.

```

<assembly-descriptor>
  <security-role>
    <description>
      Any student at the university
    </description>
    <role-name>student</role-name>
  </security-role>

  <security-role>
    <description/>
    <role-name>professor</role-name>
  </security-role>

  <method-permission>
    <role-name>professor</role-name>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        addHomework
      </method-name>
    </method>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        removeHomework
      </method-name>
    </method>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        setGrade
      </method-name>
    </method>
    <method>
      <ejb-name>Gradebook</ejb-name>
      <method-name>
        getAllGrades
      </method-name>
    </method>
  </method-permission>
</assembly-descriptor>

```

Figure 3: Example J2EE access policy using implicit grants of access to EJB methods

Consider the implementation of several methods of the Gradebook EJB class, shown in Figure 4. Note that method `getGrade` makes a call to method `getAllGrades` of the same EJB. However, while access to method `getGrade` is granted to role `student` by the access policy in Figure 2, access to method `getAllGrades` is not. Consequently, the container will throw a `SecurityException` when a call to `getAllGrades` is made from `getGrade` called by a student. An obvious solution to this problem is to grant access to `getAllGrades` to role `student`. However, this would give role `student` the ability to call method `getAllGrades` directly. If such direct access is undesirable, the `run-as` mechanism of J2EE can be used. Using this mechanism, a deployer can specify that all calls made by methods of a certain bean have to be made with the authority of a specific role. Figure 5 contains a specification (meant to be added to the access policies in Figures 2 and 3) that says that all calls made by methods of EJB `Gradebook` are made with the authority of `professor`. Now, when a student calls `getGrade`, the call to `getAllGrades` made by `getGrade` is made with the authority of role `professor`. Since access to method `getAllGrades` is granted to this role by the access policies in Figures 2 and 3, the

```

public class StoreBean
  implements javax.ejb.EntityBean {
  private Set homeworks;
  private Map studentsToHomeworksToGrades;
  ...
  public Grade getGrade(Student s,
                          Homework h) {
    if (!this.homeworks.contains(h)) {
      throw new NoSuchHomeworkException(h);
    }
    return (Grade)
      ((Map) this.getAllGrades(s)).get(h);
  }
  public Map getAllGrades(Student s) {
    Map result = (Map)
      this.studentsToHomeworksToGrades.
        get(s);
    if (result == null) {
      throw new NoSuchStudentException(s);
    }
    return result;
  }
}

```

Figure 4: Implementation of methods `getGrade` and `getAllGrades` of EJB `Gradebook`

```

<enterprise-beans>
  ...
  <entity>
    <ejb-name>Gradebook</ejb-name>
    ...
    <security-identity>
      <run-as>
        <role-name>professor</role-name>
      </run-as>
    </security-identity>
    ...
  </entity>
  ...
</enterprise-beans>

```

Figure 5: A `run-as` specification for the `Gradebook` example

call succeeds.

In this paper, we do not take the `run-as` mechanism into account. We believe that use of this mechanism is error prone and should be used sparingly. A separate analysis should be performed to ensure that no security hole is introduced by using this mechanism. Since our approach processes all method calls from EJB methods, it essentially assumes that the appropriate `run-as` permissions have been granted. Therefore, our approach is orthogonal to an analysis into secure usage of the `run-as` mechanism.

3. ANNOTATED POINTS-TO ESCAPE GRAPHS

Our analysis requires computation of which EJB fields may be used and modified by a given EJB method. We use points-to graphs produced by points-to analysis [9] for computing this information. A points-to graph representation particularly convenient for our needs

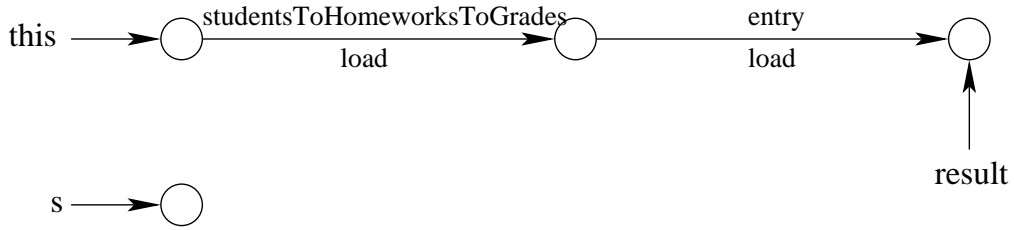


Figure 6: APE graph for method `getAllGrades` from Gradebook example

is *Annotated Points-to Escape (APE)* graphs of Souter and Pollock [23]. The points-to analysis that produces APE graphs is based on the precise compositional points-to analysis of Whaley and Rinard [28] but augments the latter with additional annotations on the edges that represent field references. These annotations list methods that load or store the corresponding fields. The APE graph for method `getAllGrades` from Figure 4 appears in Figure 6. Each circle-shaped node in this graph represents one or more run-time objects. The object node pointed to by variable `this` represents the EJB object on which method `getAllGrades` is called. Since this method uses field `studentsToHomeworksToGrades` of the EJB object, the edge with this field name is labeled as loaded by this method. Similarly, the field of the map object is also marked as loaded by this method⁴.

APE graphs represent information only about object and class fields of reference types. For our approach to be useful, we also have to be able to reason about fields of primitive types, such as `int`. We extend APE graphs to preserve information about uses and definitions of fields of primitive types. The augmented APE graph construction algorithm checks the code of a method for which the APE graph is being constructed for loads and stores of fields of primitive types. For each field of primitive type in the class of a given object node, we create a node of a new kind, *primitive*. When a load (store) to a field f of primitive type of an object is detected, we create a new edge from the object node to the corresponding primitive node, if this edge does not already exist. A label naming the method that performs this load (store) is added to the edge. Due to space constraints, we do not give a detailed algorithm or examples of handling fields of primitive types in this paper.

4. EJB ACCESS SEMI-LATTICE

4.1 Computing field accesses for EJB methods

Let $Classes$ be the set of all classes in the component under analysis (CUA). Let $EJBClasses \subseteq Classes$ be the set of all EJB classes in the CUA and $EJBInterfaces$ be the set of all EJB interfaces describing access to the classes in $EJBClasses$. We use functions $fields$ and $methods$ to return respectively the set of fields and methods in a given class. Let $EJBInterfaceMethods$ be the set of all methods declared in $EJBInterfaces$: $EJBInterfaceMethods = \bigcup_{i \in EJBInterfaces} methods(i)$. Let $EJBFields$ be the set of all fields of the EJB classes in the CUA: $EJBFields = \bigcup_{c \in EJBClasses} fields(c)$.

We say that an EJB field f is *written* by a method m if the value of f is modified by the thread executing m while m is on the

⁴In [23], annotations on edges of APE graphs are more complex, including, for example, information about calling contexts. The reason for this is that APE graphs were originally created for the purpose of creating contextual def-use pairs, used for white-box testing. For our purposes, annotations illustrated in Figure 6 are sufficient.

call stack. Note that, if f is of reference type, it is its reference value that needs to be modified for this field to be considered written. Similarly, an EJB field f is *read* by a method m if the value of f is used by the thread executing m while m is on the call stack. The APE graphs introduced in Section 3 conservatively represent which EJB fields are written and read by which methods. To determine if an EJB field f is written or read by method m , it is sufficient to search for an edge based on f in the APE graph for m . If such an edge exists and contains annotation *store*, then f is written by m . If such an edge exists and contains annotation *load*, then f is read by m . For example, according to the APE graph in Figure 6, method `getAllGrades` reads field `studentsToHomeworksToGrades` of the EJB.

In practice, it may be important to reason about reads and writes of fields of objects that are referenced by EJB fields, in addition to reads and writes of the EJB fields themselves. We define a simple calculus for reasoning about object references. A *field sequence* f_0, f_1, \dots, f_k is interpreted as a series of field dereferences, where f_0 is an EJB field, and for any $i, 1 \leq i \leq k$, f_i is a field in one of the classes whose object f_{i-1} may reference. Formally, let $type(f)$ represent the declared type of field f and let $Classes(t)$ be the set of all classes that can represent type t at run time. (For Java, this set includes all non-abstract subclasses of t if t is a class, plus t itself if t is not abstract, and all non-abstract classes that directly or indirectly implement t if t is an interface.) Then f_0, \dots, f_k is a field sequence if $f_0 \in EJBFields \wedge \forall i, 1 \leq i \leq k, \exists c \in Classes(type(f_{i-1})) : f_i \in fields(c)$. Essentially, f_0, \dots, f_k represents objects that can potentially be reached from an EJB object via a number of field dereferences. Reading and writing values represented by such sequences is defined as follows. To determine if a field sequence f_0, \dots, f_k is written by method m , we determine if a prefix $f_0, \dots, f_j, j \leq k$, of this sequence is present in the APE graph for m and the edge for f_j is labeled *store*. The reason we look for a prefix is that an assignment of a reference can be viewed as assigning not only this reference but also all values that can be reached through this reference. Figure 7 illustrates this point. Figure 7(a) shows the APE graph for a method immediately before statement `t.f2 = u` in this method. In this graph, object `o1` represents an EJB in the component under analysis and field sequence `f1, f2, f3` references object `o4`. Figure 7(b) shows the APE graph for the same method immediately after this statement. Note that now field sequence `f1, f2, f3` references object `o6`. Therefore, an assignment involving prefix `f1, f2` of field sequence `f1, f2, f3` results in the modification of this field sequence.

To determine if a field sequence f_0, \dots, f_k is read by method m , we determine if this sequence is present in the APE graph for m and the edge for f_k is labeled *load*. Figure 8 presents an algorithm for determining if a given method reads or writes a given field sequence.

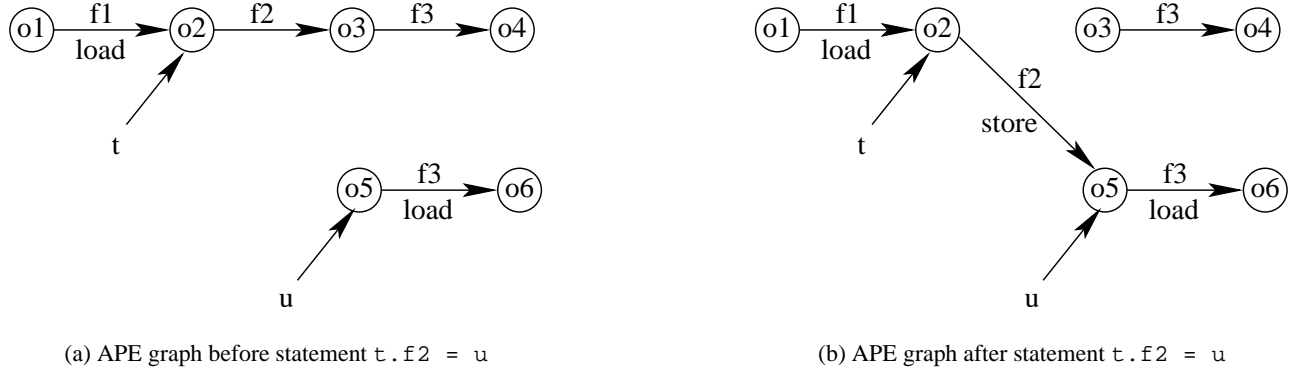


Figure 7: Illustration of methods writing field sequences.

Input: method $m \in EJBInterfaceMethods$, field sequence $f = f_0, \dots, f_k \in FieldSequences$, and the APE graph for m
Output: A set of strings `result`. This set contains string r if m reads $f = f_0, \dots, f_k$ and string w if m writes f . If m neither reads nor writes f , the returned set is empty.
Steps:

```

result =  $\emptyset$ 
let  $c$  be the APE node pointed to by variable this
mark  $c$  as visited
for each  $f_i$  in  $f_0, \dots, f_k$ :
  for each edge from  $c$  labeled with  $f_i$ :
    if this edge is labeled with store
      add  $w$  to result
    if this edge is labeled with load and  $f_i = f_k$ 
      add  $r$  to result
return result

```

Figure 8: An algorithm for determining if a method reads or writes a field sequence

Because points-to graphs contain cycles, field sequences can be infinite. To avoid this, we only consider field sequences that correspond to acyclic paths in the APE graph for method m . While this decision limits our approach theoretically, we do not believe that handling cyclic paths is useful in practice. It is unlikely that in practice analysts will be interested in specifying long and complicated field sequences. Let *FieldSequences* be the set of all acyclic field sequences for the CUA.

It may be important to reason about situations where some objects that are reachable from EJB fields may be modified by a method. We say that a field sequence s is *partially written (partially read)* by method m if there exists a field sequence t for which s is a prefix, such that t is written (read) by method m . For example, if the Map field `studentsToHomeworksToGrades` in Figure 4 is initialized only in the constructor of the EJB and method `setGrade` (listed in Figure 1) uses this map by creating a new entry corresponding to the given grade, this method partially writes field `studentsToHomeworksToGrades`. Note that if a field sequence is written (read) by a method, then it is also partially written (partially read) by this method.

We refer to the sets of field sequences written and read by a method m as $W(m)$ and $R(m)$ respectively. We refer to the sets of field sequences partially written and read by a method m as $PW(m)$ and $PR(m)$. Note that according to our definitions, $W(m) \subseteq PW(m)$ and $R(m) \subseteq PR(m)$. Because we do not allow cyclic field sequences, sets W, R, PW , and PR are finite.

For the purposes of our analysis, not all fields in the CUA may be important. Let $S \subseteq FieldSequences$ be the set of security sensitive field sequences, identified by the analyst. For each of these fields, the analyst can specify whether full or partial access is of interest and whether only reads, only writes, or both reads and writes should be tracked. This information can be represented as tuples of the form (s, r, w) , where $s \in S$, $r \in \{r, pr, -\}$, and $w \in \{w, pw, -\}$ and

- r indicates that we are interested in reads of field s ,
- pr indicates that we are interested in partial reads of field s ,
- w indicates that we are interested in writes of field s ,
- pw indicates that we are interested in partial writes of field s , and
- $-$ indicates that we are not interested in any reads (writes) of field s .

We call the set of all tuples of the form (s, r, w) specified by the user *specification of field accesses of interest* and denote it P . Figure 9 presents an algorithm for computing sets R, W, PR , and PW for a given method, while taking into account only field sequences and accesses specified in P .

By default, if the analyst does not specify security sensitive fields and accesses to them, we work with $P = \{(f, pr, pw) \mid f \in EJBFields\}$. In other words, we take all reads and writes of data reachable from the EJB fields into account.

4.2 Computing methods granted to roles

We denote the set of all roles identified in the access policy for the CUA *Roles*. For the purposes of our analysis, for each role, we need to know all EJB methods that this role is allowed to call. This information is not available directly from the access policy, since all roles are given access to all methods not explicitly mentioned in this

Input: method $m \in EJBInterfaceMethods$, specification of field accesses of interest P , and the APE graph for m
Output: sets $W(m)$, $R(m)$, $PW(m)$, and $PR(m)$.
Steps:

```

for each tuple  $(s, r, w) \in P$ :
  run the algorithm in Figure 8 for  $m$  and  $s$ ,
  obtain result
  if  $r = r$  and result contains  $r$ 
    add  $s$  to  $R(m)$ 
  if  $w = w$  and result contains  $w$ 
    add  $s$  to  $W(m)$ 
  if  $r = pr$ 
    if a forward reachability analysis in the APE
    graph, starting from each node that represents
    the end of  $s$ , finds an edge labeled load
      add  $s$  to  $PR(m)$ 
  if  $w = pw$ 
    if a forward reachability analysis in the APE
    graph, starting from each node that represents
    the end of  $s$ , finds an edge labeled store
      add  $s$  to  $PW(m)$ 

```

Figure 9: An algorithm for computing sets W , R , PW , and PR for a given method

policy. For example, roles `professor` and `student` are granted access to method `getGrade` of EJB `Gradebook` by the access policy in Figure 3, although this method is not explicitly named in this policy. Below we give steps for computing all methods in the CUA to which a given role has access. We start from the set of access permissions AP identified directly by the access policies:

$$AP = \{(q, m) \mid q \in Roles, m \in EJBInterfaceMethods, \text{ and } m \text{ is explicitly granted to } q \text{ by the access policy}\}$$

(Note that one record in an access policy can produce multiple access permission pairs, if a method signature matches multiple methods, such as signature `*` in Figure 2, or zero pairs, if no matching method is found in EJB interfaces.)

We refer to the set of all interface methods in the CUA that a role q is allowed to call $AccessibleMethods(q)$. To compute it from AP , we first compute set $GloballyAccessibleMethods$ of interface methods that can be accessed by all roles by default (these methods are not named by the access policy):

$$GloballyAccessibleMethods = \{m \mid m \in EJBInterfaceMethods \wedge \forall (q, m') \in AP, m \neq m'\}$$

The set of accessible methods for each role q is computed as follows:

$$\forall q \in Roles, AccessibleMethods(q) = GloballyAccessibleMethods \cup \{m \mid m \in EJBInterfaceMethods \wedge \exists (q, m) \in AP\}$$

4.3 Access Semi-lattice

We need to reason about accesses of a method m to the set of fields in the CUA. For this purpose, we define *access tuples* of the general form (rf, prf, wf, pwf) , where

- $rf \subseteq FieldSequences$ is a set of read field sequences,

- $prf \subseteq FieldSequences$ is a set of partially read field sequences,
- $wf \subseteq FieldSequences$ is a set of written field sequences, and
- $pwf \subseteq FieldSequences$ is a set of partially written field sequences.

Let $AccessTuples$ be the set of all access tuples.

We define a partial order on access tuples as follows:

$$\forall x = (rf_1, prf_1, wf_1, pwf_1), y = (rf_2, prf_2, wf_2, pwf_2) \in AccessTuples, x \sqsubseteq y \text{ if } rf_1 \subseteq rf_2 \wedge prf_1 \subseteq prf_2 \wedge wf_1 \subseteq wf_2 \wedge pwf_1 \subseteq pwf_2$$

We define the join operation \sqcup for a semi-lattice with elements from the set of access tuples $AccessTuples$ as follows:

$$\forall x, y \in AccessTuples : x \sqcup y = z \in AccessTuples \text{ such that } x \sqsubseteq z \wedge y \sqsubseteq z \wedge [\forall w \in AccessTuples : x \sqsubseteq w \wedge y \sqsubseteq w \Rightarrow z \sqsubseteq w]$$

The bottom and top values for this semi-lattice are $(\emptyset, \emptyset, \emptyset, \emptyset)$ and $(FieldSequences, FieldSequences, FieldSequences, FieldSequences)$ respectively.

5. SECURITY ANALYSIS

Let function $accesses$ return the tuple describing field accesses for a given method: $\forall m \in EJBInterfaceMethods, accesses(m) = (R(m), W(m), PR(m), PW(m))$. We also extend this function to apply to roles in the following way:

$$\forall q \in Roles, accesses(q) = \bigcup_{m \in AccessibleMethods(q)} accesses(m)$$

In our security analysis, we use the lattice of interface methods defined in Section 4.3. Intuitively, if some role q is given access to method m_1 but not to method m_2 , we expect the level of access of m_1 to be the same or less than that of m_2 : $\forall m_1 \in AccessibleMethods(q), m_2 \notin AccessibleMethods(q) \wedge accesses(m_1) \sqsubseteq accesses(m_2)$. More generally, we require that a role q should be given access to all methods whose access level is not greater than the combined access level for all methods accessible for q :

$$\forall q \in Roles, \forall n \notin AccessibleMethods(q) : \bigwedge_{m \in AccessibleMethods(q)} accesses(m) \sqsubseteq accesses(n)$$

If this condition is violated, we report a potential problem. Figure 10 presents an algorithm for performing our static security analysis.

Complexity

The worst-case complexity of our technique depends on the size of APE graphs for the methods in $EJBInterfaces$, the number of EJB methods, the number of roles, and the number and length of sensitive field sequences in set S .

Theorem 1 (Complexity of the EJB security analysis). *The worst-case complexity of the EJB security analysis is $\mathcal{O}(|EJBFields||S|(FN^{Q+2} + |Roles|))$, where F is the number of fields in all classes in the CUA, N is the number of nodes in the largest APE graph, and Q is the size of the longest field sequence in S .*

Input: EJB class and interfaces specifications, APE graphs for the interface methods *EJBInterfaceMethods*, a J2EE role-based access policy for these EJBs, and the specification of field accesses of interest *P*.

Output: Success or failure.

Steps:

for all methods in *EJBInterfaceMethods*, use algorithm in Figure 9 to compute sets *R*, *W*, *PR*, and *PW*

for all $q \in \text{Roles}$:

compute $R(q) = \bigcup_{m \in \text{AccessibleMethods}(q)} R(m)$

compute $W(q) = \bigcup_{m \in \text{AccessibleMethods}(q)} W(m)$

compute $PR(q) = \bigcup_{m \in \text{AccessibleMethods}(q)} PR(m)$

compute $PW(q) = \bigcup_{m \in \text{AccessibleMethods}(q)} PW(m)$

for all $m \in \text{EJBInterfaceMethods}$,

$m \notin \text{AccessibleMethods}(q)$:

if $\text{accesses}(m) \not\subseteq \text{accesses}(q)$

report a potential security problem, including field sequence accesses exhibited by method *m* that are not exhibited by role *q*.

Figure 10: An algorithm for security analysis of J2EE access policies

Proof. Our technique consists of several steps. First, we compute sets *R*, *W*, *PR*, and *PW* for each method $m \in \text{EJBInterfaceMethods}$. This involves processing each field sequence $s \in S$ and determining if this field sequence appears in the APE graph for *m*. In the worst case, the APE graph contains an edge based on any field *f* between any two nodes in the graph. We need to process all occurrences of sequence *s* in the graph and the number of such sequences in the worst case is $n^{\text{size}(s)}$, where *n* is the number of nodes in the APE graph and $\text{size}(s)$ is the number of fields in sequence *s*. Furthermore, for computation of sets *PW* and *PR*, we may need to navigate all edges in the graph once. The number of such edges is quadratic in the number of nodes in the graph and linear in the number of fields in *EJBFields* in the worst case. Therefore, this step of our analysis takes $\mathcal{O}(|\text{EJBFields}|FN^{Q+2}|S|)$, where *F* is the number of fields in all classes in the CUA, *N* is the number of nodes in the largest APE graph, and *Q* is the size of the longest field sequence in *S*.

For each of sets *R*, *W*, *PR*, and *PW* and each role our algorithm computes the union of the corresponding sets for all methods from *EJBInterfaces* to which this role is given access. Therefore, computation of these sets for each role requires $\mathcal{O}(|\text{EJBFields}||S|)$. In total, this step of the analysis requires $\mathcal{O}(|\text{Roles}||\text{EJBFields}||S|)$.

Finally, for each role *q* and each method *m* not granted to this role, we check if $R(m) \subseteq R(q)$, $W(m) \subseteq W(q)$, $PR(m) \subseteq PR(q)$, and $PW(m) \subseteq PW(q)$. Since these operations are based on the subset relationship, they take $\mathcal{O}(|S|)$ each. In total, this step of the analysis takes $\mathcal{O}(|\text{Roles}||\text{EJBFields}||S|)$.

Combining complexities for the individual steps yields the stated complexity for our technique. \square

Note that although the worst-case complexity of our technique is exponential in the length of field sequences used for identifying sensitive data, we expect that in practice, these field sequences will

be short, bounded by a small constant.

6. RELATED WORK

Mechanisms for role-based access control in the networking environment have been proposed more than a decade ago [5]. Work on building and analyzing models and implementations for role-based access control has concentrated on complex architectures [21]. Surprisingly, few approaches for analyzing role-based access control mechanisms have been suggested. Schaad and Moffett used the Alloy specification language [10] for modeling the RBAC96 access model [22]. They use the Alloy Analyzer [11] 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 [4, 13]. Such mechanisms are flexible in the sense that they prohibit or allow access to specific individual elements in XML documents. Recently, Murata et. al [14] 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.

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 [19] introduced a 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 CORBA [1] and EJB components. This approach uses the Object Constraint Language (OCL) [17] 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 role-based access 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 EJBs.

Cadena [7] 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 [20]. As a result, architectural properties about event-based inter-component communications can be checked. No analysis of role-based access policies for CORBA was done in the Cadena work.

In addition to the J2EE role-based security mechanism considered in this paper, Java also includes lower-level security mechanisms based on the notion of codebases and permissions [6, 25]. This

mechanism is designed to enable users to run untrusted code, which could potentially damage the system or steal sensitive data. A *code-base* for a software component identifies the location of this component. In Java, components can be fetched from an arbitrary URL and loaded in a running system. A *permission* signifies the ability to perform a sensitive operation (e.g. read a file on the local disk) and can be granted to a software component. Naumovich described a static analysis technique for automated analysis of the flow of permissions in Java programs [16]. This technique, based on data flow analysis [8], produces, for a given instruction in the program, a set of permissions that are checked on all possible executions up to this instruction. Koved et. al proposed a static data flow algorithm for a complementary problem of determining what permissions have to be granted to a given program or component to run it on a client machine [12]. 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. E.g., Myers and Liskov use static analysis for certifying information control flow and avoiding costly run time checks [15].

7. CONCLUSION AND FUTURE WORK

In this paper, we described a static analysis technique for validating the standard role-based access control policies used with Enterprise Java Beans in J2EE applications. Our technique allows analysts to mark fields in EJBs or other server-side classes as security sensitive. Then the technique computes read and write accesses to such fields for all methods in EJB interfaces that can be called by untrusted clients. These accesses are then aggregated to obtain field accesses that can be performed by different roles of users that may call methods in EJB interfaces according to the access policy. Finally, a simple check is performed to determine if a method m , access to which has not been granted to a specific role q requires a lower level of field access than the access level already granted to this role. Such an occurrence may indicate that

1. access to m should have been granted to q ,
2. access to some method is mistakenly granted to q ,
3. one of the methods granted to q contains a bug that makes it access security sensitive fields, and
4. one of the methods not granted to q contains a bug of not accessing one or more security sensitive fields.

Situation (1) is undesirable because the role may not be able to access the required functionality. Situation (2) is undesirable because the role can access functionality not intended for it, thereby resulting in a potential security hole. Situations (3) and (4) represent bugs in component code or assembly configuration of the components. Since granting access to methods for roles is application-specific, our analysis is not capable of distinguishing between these four cases, requiring human intervention. It is also possible that the problem report produced by our technique corresponds to none of the two problems above, i.e. is a false positive.

We plan to implement our technique as a tool with a graphical user interface that presents problems found by the technique to the analyst. After examining a problem description, the analyst may “check off” this problem, indicating that it is not a problem for the

deployment configuration at hand. The tool will retain such inputs from the analyst and will not report the same problem after subsequent run.

Once the tool is implemented, we plan to experiment with a variety of EJB-based Web applications to evaluate its usefulness. In particular, the number of false positives reported by the tool has to be investigated. Our hope is that the number of false positives is not large compared to the number of real problems detected.

We also plan to implement a J2EE deployment tool that allows a deployer to specify the role-based access control policy in terms of fields as well as in terms of methods. For example, for the partial EJB in Figure 4, we could specify that only role `professor` is allowed to both read and write field `studentsToHomeworksToGrades`, either directly or indirectly, while role `student` is allowed only to read this field, both directly and indirectly. This tool will convert specifications involving EJB fields into specifications based on methods using the dependency analysis similar to the one described in this paper. As a result, XML deployment descriptors produced by this tool will be in the standard J2EE format and therefore will work with the existing Web application servers. Finally, we plan to design static analysis techniques for validation of the use of the `run-as` mechanism, illustrated in Section 2.2. We will develop techniques for computing redundant `run-as` specifications, as well as situations where the use of `run-as` specifications may lead to giving an unnecessarily high level of access to a role.

8. REFERENCES

- [1] CORBA/IIOP 2.2 specification. <ftp://ftp.omg.org/pub/docs/formal/98-02-01.pdf>, Feb. 1998.
- [2] A. D. Brucker and B. Wolff. Testing distributed component based systems using UML/OCL. In *Informatik 2001*, volume 1, pages 608–614, Nov. 2001.
- [3] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)*, pages 374–387. ACM Press, 2003.
- [4] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for xml documents. *ACM Transactions on Information Systems Security*, 5(2):169–202, 2002.
- [5] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
- [6] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.
- [7] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th international conference on Software engineering*, pages 160–173, 2003.
- [8] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.
- [9] M. Hind. Pointer analysis: Haven’t we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.

- [10] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.
- [11] D. Jackson, I. Schechter, and H. Shlyachter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering*, pages 730–733. ACM Press, 2000.
- [12] L. Koved, M. Pistoia, and A. 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. ACM Press, 2002.
- [13] M. Kudo and S. Hada. Xml document security based on provisional authorization. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 87–96. ACM Press, 2000.
- [14] M. Murata, A. Tozawa, M. Kudo, and S. Hada. Xml access control using static analysis. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 73–84. ACM Press, 2003.
- [15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142. ACM Press, 1997.
- [16] G. Naumovich. A conservative algorithm for computing the flow of permissions in Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 33–43, July 2002.
- [17] Object Management Group. Object constraint language specification, chapter 6 of omg unified modeling language specification (draft). <http://www.omg.org/uml>, Feb. 2001.
- [18] M. Pistoia, N. Nagarathnam, L. Koved, and A. Nadalin. *Enterprise Java Security: Building Secure J2EE Applications*. Addison-Wesley, Reading, MA, 2004.
- [19] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd international conference on Software engineering*, pages 25–34. IEEE Computer Society, 2001.
- [20] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276. ACM Press, 2003.
- [21] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.
- [22] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22. ACM Press, 2002.
- [23] A. L. Souter and L. L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, 29(11):1005–1018, 2003.
- [24] Sun Microsystems. Enterprise javabeans specification, v. 2.1. <http://java.sun.com/products/ejb/docs.html>.
- [25] Sun Microsystems. Java security architecture. <http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.fm.html>, 1998.
- [26] Sun Microsystems. Java remote method invocation specification. <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html>, 2003.
- [27] Sun Microsystems. Java 2 platform, enterprise edition (j2ee). <http://java.sun.com/j2ee/>, 2004.
- [28] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming*, pages 187–206, Oct. 1999.