Saving the World from Bad Beans: Deployment-Time Confinement Checking

Dave Clarke
Institute of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
dave@cs.uu.nl

Michael Richmond
Almaden Research Center
International Business Machines
San José, California, USA
mrichmon@acm.org

James Noble
School of Mathematical and Computer Sciences
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

ABSTRACT

The Enterprise JavaBeans (EJB) framework requires developers to preserve architectural integrity constraints when writing EJB components. Breaking these constraints allows components to violate the transaction protocol, bypass security mechanisms, disable object persistence, and be susceptible to malicious attacks from other EJBs. We present an object confinement discipline that allows static verification of components' integrity as they are deployed into an EJB server. The confinement rules are simple for developers to understand, require no annotation to the code of EJB components, and can be efficiently enforced in existing EJB servers.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Languages, Verification, Reliability

Keywords

Enterprise JavaBeans, Confinement, Deployment Tools

1. INTRODUCTION

The Enterprise JavaBeans (EJB) architecture [45, 24, 38] is designed to support enterprise scale software systems. EJB applications are comprised of a collection of EJB components (called *beans*) that are assembled into the desired application and executed on an EJB server.

Internally, an EJB Server wraps developer-supplied beans in a wrapper that provides capabilities such as security, persistence, transactions, and so forth. The integrity of the EJB architecture depends upon every bean being confined within its wrapper, and all access to each bean being mediated via that wrapper.

Unfortunately, malicious or erroneous developers can create bean objects which can escape from the confines of their wrappers. Without the wrappers' confinement, beans can be accessed directly, and the benefits and protection provided by the EJB architecture are lost. For example, methods may be invoked on beans outside the required transaction or security context, so rollbacks will not be performed when transactions fail; privileged methods may be invoked by clients who don't have the necessary security credentials; and beans may mount malicious attacks on other, unconfined, beans.

This is essentially a confinement problem: the EJB architecture must prevent access to the internal objects implementing each bean, and permit access only through the associated wrapper. Large-scale changes to the EJB architecture to remove this vulnerability are not feasible, as the EJB architecture is already well established in the marketplace, and is widely used to implement large scale systems. As beans are written by third-parties, hand-checking, verification, or certification to ensure confinement properties is also impractical, without a large infrastructure to introduce certifying compilation [17] and proof-carrying code [34]. Similarly, changing the EJB specification to require programming language support for confinement, such as Ownership Types [16] or Confined Types [48] is impractical given the investment in the existing architecture.

To solve this problem, we introduce a simple confinement discipline for Enterprise JavaBeans. This discipline can be thought of as a programming convention which can be checked statically whenever a component is deployed into an EJB server, and, by preventing unconfined EJB components from being deployed, preserves the integrity of the EJB architecture.

The confinement discipline is designed to be simple, easy for developers to understand, and ensures that their designs will match the constraints imposed by the EJB architecture without requiring developers to significantly change their work flow. The confinement checking relies solely upon the Java bytecode and deployment descriptor of the bean to be deployed, is efficient to execute, and so could be incorporated into production EJB servers.

The confinement analysis we perform is coarse-grained

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

when compared to previous approaches such as Grothoff *et al.* [19]. As a result, our analysis is relatively cheap, avoiding, for example, any expensive control flow analysis. Nevertheless, our results demonstrate that our approach is appropriate for ensuring confinement in the EJB domain.

Confinement has been proposed in various forms in the literature, however to the best of our knowledge, this is the first application of confinement analysis to realistic systems used in commercial software environments. Ensuring confinement in this setting is a vital part of maintaining the integrity of these systems. As such, we advocate that future versions of the EJB specification require the enforcement of EJB confinement in the manner we discuss.

This paper is structured as follows. Section 2 reprises the basics of the Enterprise JavaBeans architecture. Section 3 describes the confinement problem in more detail with Section 4 presenting the design of our confinement checker. Section 5 outlines the implementation of our confinement tool and presents our experience with applying this tool to a variety of EJB components. Section 6 evaluates our approach and considers some alternatives, Section 7 places our work in the context of related work, and finally Section 8 concludes the paper.

2. ENTERPRISE JAVA BEANS

The Enterprise JavaBeans (EJB) architecture specification [45] defines a component architecture designed to support the implementation of enterprise scale applications. Individual EJB components (or *beans*) are separately developed black box components that are designed to be assembled to form an application. Each bean consists of Java code to implement the behavior of the bean, plus an XML *deployment descriptor* describing its properties and requirements for security, transactions, persistence, and so on. An EJB server provides the necessary runtime environment to host Enterprise JavaBeans. The EJB architecture is designed to be implemented either as a stand-alone application server to which Java-based clients may connect and interact with the hosted applications, or as part of the larger Java 2 Enterprise Edition (J2EE) architecture which is designed to support n-tier web-enabled distributed applications. The EJB architecture specification defines the role and behavior of an EJB server in hosting EJBs.

2.1 EJB Life-cycle

Enterprise JavaBeans have a richer life-cycle than traditional software applications [45, 46]. Figure 1 shows the phases of the life-cycle of an EJB and the possible transitions between different phases in the lifetime of a component, and the EJB specification defines a range of roles for the people involved with each phase of the EJBs life-cycle.

The phases of an EJBs life-cycle are:

1. **development** — during which an EJB is written in Java, and packaged with appropriate deployment descriptors by a *bean developer*.

2. **deployment** — during which additional deployment descriptors may be added or modified by the *application assembler*. After these changes have been made the EJB is actually deployed onto the server.

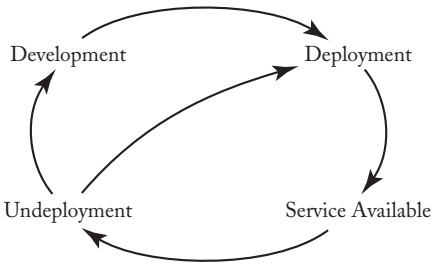

Figure 1: The life-cycle of an EJB component.

3. **service availability** — during which the EJB is available for use on the server. Clients may instantiate EJBs and invoke methods on these instances, and

4. **undeployment** — during which the EJB and all instances of it are removed from the server. This may be the first step in updating a bean with a new version, or may be due to the EJB no longer being necessary.

The deployment phase is the key difference between the EJB life-cycle and that of other software components, and represents the "third-party" nature of EJBs and the separation of roles between a bean developer (writing individual beans) and an application assembler (combining them into an application). During deployment, an application assembler (or separate deployer or system administrator) will adjust properties in the bean's deployment descriptor that specify security roles and permissions, the bean's transaction and persistence needs, and any bean-specific properties that can be used to modify bean behavior. Technically, deployment culminates in the transfer of a bean's compiled code and deployment description into an EJB server, and the automatic generation of wrapper objects used within the EJB server itself.

2.2 Component Structure

Within an EJB server, each EJB instance is comprised of a pair of objects: an actual `EJB` object[1] and an `EJB Interface` object, as shown in Figure 2. The `EJB` object implements the functional or business logic and holds any required state of the component bean [45]. This is the object that the developer implements, that the application assembler uploads to the server, and that provides the behavior required of the component. For example, in a shopping cart bean, the `EJB` object would implement methods such as `addItem()`, `removeItem()`, `getTotalCost()`, and `placeOrder()`.

An EJB object may use a number of other objects internally to implement the business logic of the EJB component, just as any other object-oriented design typically delegates complex behavior to many communicating objects. The objects which help to implement the business logic of an EJB are known as `Helper` objects and are also shown in Figure 2. EJBs often also use additional objects to transfer data in or out: these objects are known as `Transfer` objects [29]. A shopping cart bean could have a helper object

[1]The EJB terms for these objects are `EJB` and `EJBObject` respectively. We have chosen to adjust these terms to avoid the confusing circumlocutions such as "EJB Object object" that result.

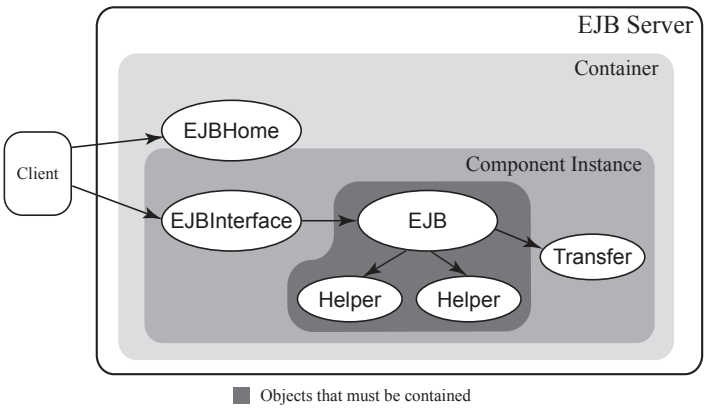

Figure 2: EJB component structure highlighting objects which must be contained.

such as a `SalesTaxCalculator` and a transfer object such as an `OrderLineCollection`. Helper and transfer objects are written by bean developers along with the bean to which they belong, and are generally deployed together in a single component archive.

In contrast, the `EJB Interface` (`EJBI`) object is the reification of the component's external interface as an object in the system. The `EJB Interface` object is typically compliant with the Java Remote Method Invocation (RMI) specification [44]. This means that objects in other Java virtual machines, possibly on different hosts, may hold a reference to the `EJBI` and perform method invocations in a network-transparent way. All method invocations on the EJB component are required to pass through an `EJB Interface` object on the component, irrespective of the location of the caller.

The implementation of an EJB Interface object is generated automatically by the EJB server during deployment, based on a Java interface provided by the bean developer and specified in the deployment descriptor. The `EJBI` object typically implements or controls many of the services that the EJB architecture offers to individual beans, including transaction management, persistence, and security. To provide these services, the EJB server inserts a number of hooks into the call path of the component in the implementation of the `EJB Interface`. These hooks perform up-calls to the server to request various services on behalf of the component [45].

The `EJB container` is an architectural abstraction that acts as the interface between an EJB component instance and the EJB server on which it is hosted [45]: in practice, an EJB Interface object will invoke its container, and the container then invokes the server proper. An EJB server may host any number of containers, with each container potentially hosting any number of components. In most existing EJB servers a new container is created for each type of bean during deployment. This container will then only host EJB instances of the type currently being deployed. That is, if an EJB server hosts `Shopping Cart` and `Stock Item` EJBs, the server will host every `Shopping Cart` instance in one container, and every `Stock Item` in a separate container.

Each component type is also associated with an `EJB Home` object which is shared between all component instances of the same type on a single EJB server. An `EJB Home` object acts as an object factory for instances of the component type with which it is associated. Any client that requires a new component instance must first obtain the correct `EJB Home` object from the system name service then call `create()` on the `EJB Home` object itself. An `EJB Home` object may also implement methods to allow clients to find component instances which have previously been created on the server. In some cases, these instances may have been created in previous system sessions and then persisted to secondary storage for later retrieval.

As with the `EJB Interface`, the `EJB Home` is generated automatically by the EJB server during deployment based on an interface provided by the bean developer. The bean developer, therefore, must create three Java files to specify an Enterprise JavaBean: Java interfaces for the `EJB Home` and `EJB Interface` objects, and a Java class for the `EJB` that implements the business logic for the bean — along with a deployment descriptor. During deployment the EJB server generates implementations of the `EJB Home` and `EJB Interface` objects and instantiates one `EJB Home` object and a container if necessary.

3. A CONFINEMENT PROBLEM

Figure 2 shows how the structure of the EJB server is designed to provide an EJB object with an external interface, the `EJB Interface` object, through which method invocations are performed. Indeed, in the EJB architecture, no object other than the `EJB Interface` object should *ever* receive a direct reference to the `EJB` (or to any of its internal Helper objects). Any such reference would allow direct access to the `EJB`, bypassing the protection provided by the `EJB Interface`. That is, an `EJB Interface` acts as a mandatory wrapper, proxy [41] or decorator [18] surrounding the EJB object. The correct functioning of the EJB server requires that a bean's implementation remains *confined* behind its interface object with all access being performed via the `EJBI`.

Unfortunately, if an internal object (either an `EJB` object or one of its helpers) does become accessible outside the `EJBI` wrapper (*i.e.* becomes *unconfined*) the integrity constraints of the EJB architecture are broken. External clients will then be able to access the bean object or its helpers directly, thus bypassing the control and management that is provided by the EJB Interface wrapper.

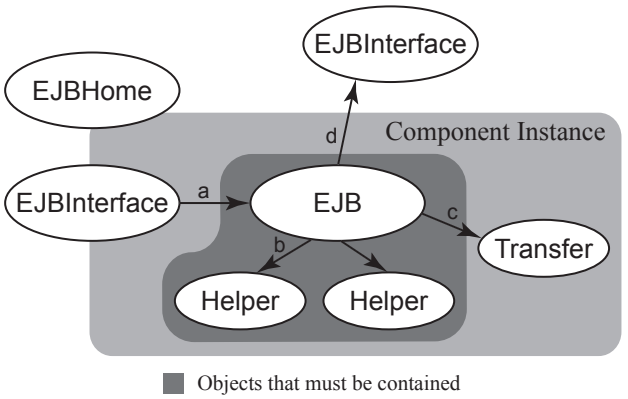

Figure 3: Illustration of references that could potentially escape an EJB component.

For example, the EJB architecture supports method-level security checking implemented in the `EJBI` object; bypassing the security layer may enable unauthorized access to otherwise restricted data. Similarly the `EJB Interface` object implements the EJB's transactions, persistence, and object swapping services. Invoking methods outside the transaction contexts provided by the `EJBI` can introduce inconsistencies into shared backing databases and prevent transactions rollbacks and commits from being executed properly; avoiding the persistence services can allow a bean's state to be changed by a method without this change being mirrored in the local persistent store; finally, attempts to call methods on an `EJB` may cause serious server problems if the actual bean object itself has been pooled by the server to reduce its memory use. Any of these situations can undermine performance or integrity, or result in unexpected exceptional behaviour on the server hosting the unconfined bean.

The EJB architecture is built around a strong notion of confinement. Thus, it is important that future EJB specifications enforce this notion of confinement as emerging application server technologies, such as component migration [38], typically rely heavily on the complete confinement of individual components.

3.1 Breaching Confinement

Figure 3 illustrates the range of inter-object references that may occur within an EJB component. To ensure the integrity of the EJB architecture, our primary concern is to prevent references to confined objects a and b from escaping the `EJB Interface`. Meanwhile, we must allow the return of references to transfer objects c either by copy or reference. The return of references to external beans (or rather their EJB Interfaces) such as d, must also be allowed, irrespective of the type of the `EJB Interface`.

The whole premise of Enterprise JavaBeans is that individual beans will be written by third parties and compiled outside of the server framework, possibly by persons of malicious intent. As Java is an object-oriented language, object references are unrestricted: any object may be accessed from anywhere [35]. Even if a bean were required to store its internal references in private fields, its developer could expose those private references simply by writing public methods that return them. Thus, it is quite possible for malicious or erroneous developers to violate the confinement relationship between the `EJB Interface` and the `EJB` object and its helper objects — by writing methods which return these objects directly; by creating transfer objects that store references to these objects, and then copying the transfer objects outside; or by giving references to these objects to shared external objects which have somehow been passed to the EJB.

This problem is particularly unpleasant as unwitting developers can inadvertently violate containment simply by using the Java keyword "`this`" to pass the current object as an argument, or return it to the caller. This idiom is illustrated in the method `badReturn()` shown below. When developing an EJB component, if the developer wishes to pass or return the current component they are required to use the result of the method `getEJBObject()` in place of "`this`", as illustrated by the method `goodReturn()`.

```
public class CartBean implements SessionBean {
 protected SessionContext context;
 // Called once by container during bean creation
 public void setSessionContext(SessionContext _ctx) {
   this.context = ctx;
 }
 ...
 // incorrect way to return reference to bean
 public CartEJBI badReturn() {
   return(this);
 }
 // correct way to return reference to bean
 public CartEJBI goodReturn() {
   return(context.getEJBObject());
 }
 ...
}
```

3.2 Bean Verification

In spite of the consequences of the server losing its architectural integrity, the EJB specification does *not* require that the confinement of beans be enforced. Existing EJB implementations, such as the EJB Reference Implementation [42], certainly do not enforce confinement and thus allow the deployment and use of unsafe beans.

On the other hand, most EJB implementations (including the EJB Reference Implementation, Oracle9*i* [36], and JBoss [31]) do include stand-alone *verifier* tools that de-

velopers can use to check their beans' Java code and deployment descriptors. Generally, these tools make syntactic checks for consistency between the various code files and the deployment descriptor — for example, that various method names, parameters, and types are consistent across all the files, and that (where appropriate) they match signatures required by the EJB specification.

Amongst other things, the verifiers can raise errors on EJBs which expose the class of the `EJB` object from the EJBI wrapper. The current EJB specification [45] does require that references to the types of `EJB` objects cannot appear in the interfaces of the `EJB Interface` wrappers. Checking this condition will not, however, catch even simple cases where the `EJB` instances are returned via variables of a super type such as `java.lang.Object`, or where a reference to an `EJB` object will escape the component as a field of another object.

```
class BadBean implements SessionBean {
 ...
 public Object exposeMyself() {
   return (Object) this;
 }
 Mole OopsIDidItAgain() {
   return new Mole(this);
 }
}
```

More complex cases, such as ensuring that a bean's helper objects remain within the bean, and that transfer objects do not accidentally expose an EJB objects or its helpers, cannot be handled by these verifiers. Furthermore, these verifiers must be run manually by bean developers: they are not incorporated into the respective EJB servers. Beans that fail the verifier can still be deployed without errors being detected by the server — until, of course, the confinement problems appear.

3.3 Confinement Model

To solve this problem, we first establish a *confinement model* that captures the confinement relationships discussed above. That is, we determine which objects are confined, which are not confined, and which manage the boundary between the confined and unconfined objects.

Every EJB constitutes a single *unit of confinement* defining a single confined space. Objects which exist within this space are confined and no references to them may leak to objects outside of this space. For a bean, the confined objects are the `EJB` and any associated helper objects.

Topologically, every path in the object graph from an unconfined object to a confined object must pass through a boundary object. In our case, any path from an object which is not part of a given bean instance to the `EJB` object or one of its helpers must pass through the `EJBI` object. This implies that if an object has a reference to a confined object, then it too is confined or on the boundary; confinedness is a virus transmitted via reference, blocked only by the prophylactic boundary.

Objects on the boundary thus form the interface to the unit of confinement. These objects exist to allow controlled access to the objects within the unit of confinement. The interface objects are permitted to access the confined objects and may be referenced by objects that exist outside of the confined space. Thus, the objects on the boundary constitute the points at which references to confined objects can leak to the rest of this system.

Interface	Status	Role
EJBHome	Boundary	User extended System provided
EJBLocalHome	Boundary	User extended System provided
EJBObject	Boundary	User extended System provided
EJBLocalObject	Boundary	User extended System provided
EnterpriseBean	Confined	User implemented (via sub-interface)
EntityBean	Confined	User implemented
SessionBean	Confined	User implemented
MessageDrivenBean	Confined	User implemented
SessionSynchronization	Confined	User implemented (optional)
EJBContext	Confined	System provided
EntityContext	Confined	System provided
SessionContext	Confined	System provided
MessageDrivenContext	Confined	System provided
Handle	Unconfined	System provided
HomeHandle	Unconfined	System provided
EJBMetaData	Unconfined	System provided

Figure 4: Status of `javax.ejb` Interfaces

For an EJB component, the objects on the boundary are the `EJBHome` and `EJBI` objects. The interfaces for these objects are supplied by the bean developer, however, as these objects' implementations are generated by the EJB server during deployment, we can fortunately trust that the server safely implements these objects without introducing code that breaches confinement.

Any transfer objects used by the bean are outside the unit of confinement. This means that references to these objects can be passed across the confinement boundary, however to maintain confinement we will have to place restrictions on their use (see Section 4.3 below).

We've analyzed the interfaces provided by the `javax.ejb` API and classified them according to their confinement requirements. This classification is shown above in Figure 4, along with the provenance of each interface.

Any user provided object which implements a subtype of `EnterpriseBean` including `EntityBean`, `SessionBean` and `MessageDrivenBean` is determined to be confined according to our classification. In the EJB architecture, every `EJB` object is required to implement one of these interfaces. We assume all other objects within the same Java package as the `EJB` object are that object's helpers and so must also be confined. (Other schemes are possible.)

In addition, we introduce a new field to the deployment descriptor used to describe the bean to the server. This field is a list of classes which are confined, inclusive of the package containing the `EJB` object. Any objects from packages not listed in this field are unconfined, as far as the given bean is concerned.

This means that using the common EJB development approach of defining all classes for a single bean in one package will result in the safest confinement relationship — assuming that all objects referenced by the `EJB` object are helper objects. Where transfer objects are required, the developer must actively declare their classes in a separate package.[2]

[2] An alternative solution in which the developer must specify

Note that this confinement model can be enforced by checking only confined and boundary classes. Our rules do not constrain unconfined classes, and so such classes need not be checked.

4. CHECKING BEAN CONFINEMENT

Section 3 has defined a confinement model for EJB. To be practically useful, we need to ensure this that model is enforced within the EJB system and architecture. There a number of approaches to addressing confinement problems (e.g. [1, 2, 48, 12, 16, 19, 33, 8]) though none satisfies the constraints imposed by Enterprise JavaBeans. In this section, we examine those constraints, before delving into the particulars of our problem, and ultimately producing a developer discipline for addressing bean confinement.

4.1 Forces on Feasible Solutions

Several forces constrain the approach that may be used to address the problem of bean confinement: checking must be performed on the deployed bytecode, the EJB architecture is effectively unchangeable, and we must make a minimal impact on deployment.

4.1.1 Bytecode Checking

EJB components are developed in a separate environment from the EJB server to which they are submitted as Java bytecode. An EJB is written externally, by an unknown source, and generated by an unknown compiler which may not even have been a Java compiler. Consequently, the characteristics of the source programming language and compiler cannot be specified or modified. This separation of responsibilities for compilation and execution is one of the key benefits of the EJB architecture, so we cannot consider approaches which change this model.

Certifying compilation [17] and proof-carrying code [34] could be used to ensure that an EJB was compiled using a tool that enforces confinement. These approaches, however, would force EJB developers to change their development tools in order to support EJB confinement checking.

As discussed in Section 2.1, the EJB life-cycle involves an additional step, known as deployment, between compilation and execution. Since changes to EJB development tools are not feasible, deployment-time is the earliest point in the life-cycle that confinement checking can be performed.

4.1.2 Unchangeable Architecture

EJB servers from a range of vendors are in widespread use today. These servers are generally built to execute on standard Java Virtual Machines. As a result, any approach to enforcing EJB confinement may only minimally impact EJB servers. Changes to the JVM and the EJB architecture are not feasible, as these technologies are well established and in widespread use. We can (and do) assume that the EJB server implementation does not violate the confinement relationships behind the scenes.

The solution we present requires no changes to the core EJB architecture. Rather, we require the deployment tool perform a series of checks over the bytecode of the bean as it is transfered to the server. If any of these checks fail, then an error message is displayed and the deployment is aborted. To support these checks we propose that the an EJB's deployment descriptor is extended to identify the helper and transfer objects.

4.1.3 Deployment Impact

To have minimal effect on the deployment phase, our confinement checking must be efficient. This precludes the use of complicated program analysis. Additionally, since EJBs may be deployed individually, we do not have access to all of the final application thus preventing the use of full program analysis.

To enable agreement between bean developers and vendors in their understanding of the constraints required, we require that those constraints be simple to understand and implement. Furthermore, we want to be more flexible than existing approaches to confinement by allowing classes to be used by different beans.

These constraints restrict the machinery we can bring to our aid, as well as limiting where we can apply it.

4.2 Developer Requirements

The core of our proposal imposes a small number of syntactic *confinement constraints* onto beans' code. Figure 5 summarizes the confinement constraints which we require EJB components to satisfy in order to be verified by our confinement checking tool and thus ensure that they maintain the integrity of the EJB architecture. This tool can categorize classes based on the breakdown shown in Figure 4. That is, the classes which are on the boundary, those which are confined, and those which are outside.

CB1	All classes implementing an EJB must be confined, such as classes implementing the `EnterpriseBean` interface, any other confined interface in Figure 4, and any helper classes a developer specifies. Interfaces extending boundary interfaces are on the boundary. All other classes are unconfined.
CB2	No confined type can appear in the signature of a boundary method, nor in static fields, nor as an exception.
CB3	No confined type can be cast to an unconfined type.
CB4	No unconfined type can be cast to a confined type.
CB5	No field, method, or static of an unconfined class having a confined type is accessible in confined code. Exceptions cannot be caught at confined types.
CB6	No confined class may extend an unconfined class, except `java.lang.Object`.

Figure 5: Confinement constraints for Enterprise JavaBeans.

To use this tool, the developer specifies (nominally in the deployment descriptor) those objects that are used as helper objects, and thus confined, and those that are used as transfer objects, and thus not confined. For the current version of our tool, the developer specifies a list of the confined pack-

each confined class or package individually is conceivable, however we feel that this may overly complicate the creation of deployment descriptors.

ages for a bean: all non-EJB classes within such a package are also considered to be confined. In most cases, the developer will create two packages: one that is confined and contains all the EJB classes and any associated helper classes, and the other containing any required transfer objects.

This ensures that developers using the common EJB development approach — defining all classes for a single bean in one package — will receive the safest confinement relationship. That is, the assumption that all objects referenced by the `EJB` object are helper objects and thus confined. In the case where a transfer object is required, the developer must actively declare this class in a separate package.

4.3 Developer Discipline for Modular Confinement

To maintain our confinement model, we wish to prevent unconfined code from accessing confined objects. The access could be based on carelessness on the part of the confined code, which may inadvertently pass a reference out, or it could be more elaborate, whereby an unconfined class attempts to appear confined by sub-classing classes which the confined code thinks are confined. Our rules guard against all such accesses (Figure 5).

In this section, we explain how our rules work, and through exhaustive coverage of the different ways which references to confined objects could leak (which are more or less standard across programming languages [26]), we argue that they cover all bases.

To illustrate various points, we have used the following example code. From the perspective of class `Confined`, classes `SomeConfined` and `ConfinedException` are considered confined. The class `Unconfined` is not confined and treats everything it can get its hands on as unconfined.

```
class Confined {
   Unconfined non;
}

class SomeConfined extends Confined { ... }

class Unconfined extends SomeConfined {
 void throwsConfined() throws ConfinedException {
   ...
 }
 SomeConfined meAsConfined() {
   return (SomeConfined) this;
 }
 void takeIt(Confined it) {
   it.stealAndMunge();
 }
}

class ConfinedException {
 Confined data;
}

class UnConfinedException extends ConfinedException {}
```

We also assume that we are working in a scope defining at least the following variables:

```
Confined con;
Unconfined non;
```

4.3.1 Confinement

CB1 states the fundamental property of our model; that certain classes must be considered as confined, others are on the boundary, and the remainder are unconfined, as dictated by the EJB architecture.

4.3.2 Casting

Subtyping (or more specifically typecasting) is troublesome as it permits type information to be forgotten or imprecisely reconstructed. As we use type information to represent whether something is confined or otherwise, rules **CB3** and **CB4** ensure that this information is not lost via widening nor gained via narrowing.

Widening (upcast) can be used to forget that an object has a confined type, enabling it to pass through the boundary from inside to out. The expression `(Object)con`, for example, could enable the confined contents of the variable `con` to leak out of the confinement area. Furthermore, as classes outside can use class `Confined`, they can do the cast in the other direction and access the confined object. We prevent the initial widening using rule **CB3**.

Narrowing (downcast) enables an outside object to be passed in as some unconfined type and be cast to a type which is confined (but not necessarily in the same place) and thus capable of storing confined objects. This form of *spoofing* is possible because one Java type may correspond to different regions of confinement. Rule **CB4** prevents an object of type `Unconfined` being cast to type `Someconfined`, and thus prevents this form of spoofing.

4.3.3 Subclassing

Inherited code that may not know that an object is confined may pass itself (or other confined objects) to outside objects. It may not be possible to subject all such classes to our tests, or such classes may be written in a manner which is safe but too subtle for our tool to check. We trust a few special classes such as `java.lang.Object` to avoid this problem, whereas the majority of classes we simply cannot trust. Untrusted code could pass out instances of itself via static fields, for example. If extended by a confined class, then this behavior could occur through a call to superclass methods. As there is no way to avoid this behavior with the simple approach we take, we simply avoid it via rule **CB6**.

Conversely, code inheriting from a confined class may violate the confinement expectations of the original class, thus creating opportunities for spoofing. This is similar to the issues that rule **CB6** is designed to prevent. Whereas rule **CB6** is designed to prevent super class behavior from violating containment, rule **CB4** is designed to prevent subclass behavior from violating containment. If it were possible to cast an unconfined type down to a confined type, then the confined type may end up calling the unconfined type within the confined space as the result of a call to super. This sequence would then allow the unconfined type to spoof the confined type.

4.3.4 Fields and Methods

There are a number of ways a confined object can be leaked through fields and methods. The majority of our confinement rules are designed to tackle these leaks.

First, an unconfined object may obtain a confined object from a field, or, similarly, a confined object could be returned from a method to an outside object. These cases are prevented by a combination of rules **CB2**, **CB3**, and **CB6**. By prohibiting confined types from appearing on the boundary via rule **CB2**, we prevent their direct escape. Addition-

ally, by preventing casts from confined types to unconfined types with rule **CB3**, we prevent their escape as types that are permitted on the boundary. We also rely on the super classes not doing anything untoward, so we only allow subclassing of confined classes and `java.lang.Object` in rule **CB6**.

Second, a confined object could be assigned to the field of an object that is not confined, or, similarly, a confined object could be passed as an argument to a method of an unconfined object. Since we do not check unconfined code, such classes may extend confined classes, though the objects will (at least initially) be separate. Such objects, as they are unconfined, can be accessed in confined code. Rule **CB4** prevents these objects from being cast to a confined type, which would lead to confusion over whether the object was confined or not, as mentioned above. The interface of the unconfined type may include fields and methods which have confined types in their signatures. An object implementing such an interface could leak confined objects to the outside, so rule **CB5** prohibits this. For example, if the code `non.takeIt(this)` could appear within confined code, it would result in a reference to a confined object leaking to an unconfined object.

4.3.5 Statics

Static fields carrying confined values are trouble. A confined object could be assigned to a static field that is accessible to objects outside the confinement boundary or by different instances of the same confined type. Additionally, static fields defined in unconfined code that contain values of a confined type are too freely accessible to trust with confined objects. Thus, they must not be accessed from within confined code. For similar reasons, we exclude and protect against accessing static methods from within confined code. Although these restrictions on statics may seem constricting, the EJB specification actively discourages bean developers from using static fields. "An EJB is not allowed to use read/write static fields. Using read-only static fields is allowed. Therefore, all static fields must be declared as final."[45] — although the current EJB specification does not require EJB servers to enforce this requirement.

4.3.6 Exceptions

Exceptions cross boundaries between objects, bypassing the usual return mechanism. Once again, this opens two possible channels by which leaks may occur: A confined object could be thrown as an exception; or an unconfined exception may be caught at a confined type. The second case is illustrated in the following example code, which occurs in a confined context:

```
try { non.throwsConfined() }
catch (ConfinedException ce)
{  // thinks ce is confined, but it's not }
```

Thinking the exception is confined could lead to leaks, because the unconfined originator of the exception may have retained a reference. Catching this exception would be excluded by rule **CB5**, though in practice it is sufficient to never consider an exception class as confined.

4.3.7 Native Methods

A native method can pass objects around under the hood, violating confinement. Fortunately, classes with native methods cannot be deployed by the EJB server.

4.3.8 Reflection

Java's reflection mechanism enables traversal of the object graph irrespective of any desired containment model. As such, code using reflection may violate the confinement of any bean. In fact, code using reflection may also violate the internals of the server and thereby subvert the server's access to deployed beans.

An EJB server relies heavily on the Java reflection mechanism to perform the code analysis required to generate the appropriate wrapper implementations (see section 2.2). It would be possible for our confinement checker to prevent the deployment of beans that use reflection. There are however, legitimate situations where it is appropriate for an EJB component to use the Java Reflection API [39].

We are focused primarily on preventing developers from inadvertently violating confinement. Any use of reflection is necessarily intentional and thus, we choose to trust developers who use it rather than impose constraints that would hinder the use of the EJB platform.

Figure 6 summarizes the effects of our constraints from the perspective of confined code. All combinations are covered: whether an object is moved into or out of an area of confinement; and whether the static (as it appears in code) and dynamic (actual class) types of the object are confined or unconfined, *as far as the confined code is concerned.*

5. IMPLEMENTATION AND EXPERIENCE

For experimental purposes, we implemented our deployment checker as a standalone tool. Using this tool, we analyzed a number of existing beans that we obtained from the web, together with some beans we crafted ourselves. Our results show that our confinement model is compatible with existing EJB development approaches. Additionally, the use of our tool can prevent developers from using conventional and otherwise safe Java idioms which would result in errors in the resulting beans, for example, returning `this` from a business method on the bean.

5.1 Deployment Checking Tool

Our confinement checking tool implements the tests described above by performing a simple traversal of the contents of a class file. The tool uses the SOOT framework for manipulating Java bytecode from McGill University [47]. Among a wealth of other things, this framework provides `jimple`, a simple typed representation of Java Bytecode, and the appropriate library support to manipulate it. The deployment tool was consequently straightforward to implement, requiring only 700 lines of Java.

For each class submitted to the tool we determine from its type and from the deployment descriptor whether it is confined, on the boundary, or unconfined. We do not subject unconfined classes to any checks. The only boundary classes permitted are those extending one of the EJBI interfaces defined by the EJB specification and shown in Figure 4. These must be interfaces, and we traverse their declared methods to ensure that they do not contain any confined types. A confined class is checked in a single pass which mainly searches for places where widening or narrowing could occur, as well as checking various interfaces, static fields and methods, and exceptions. If one of the tests fails, an error

Dynamic type Static type	Confined Confined	Confined Unconfined	Unconfined Confined	Unconfined Unconfined
Inside to out	blocked at boundary (**CB2**)	no widening from confined to unconfined (**CB3**)	malformed class (**CB6**)	"Don't believe his lies"† (**CB5**)
Outside to In	blocked at boundary (**CB2**)	no narrowing from unconfined to confined (**CB4**)	blocked at boundary (**CB2**)	"Don't believe his lies"† (**CB5**)

CB1 merely tells us which types are definitely confined or otherwise

† Confined objects which an unconfined object has are most likely not *our* confined objects.

Figure 6: Confinement Scenarios Summarized (from the perspective of confined code)

message is shown to the client and the bean is rejected. Figure 7 shows an example of the error messages generated by our confinement checking tool.

```
...
[dc] Processing class: mar.basicfail.SampleEJBI
[dc]   Class is on boundary - proceeding with boundary
       checks
[dc] Boundary class has confined in interface (CB2).
[dc]   Offending Method (in return type):
       returnAsSessionBean
[dc] Boundary class has confined in interface (CB2).
[dc]   Offending Method (in return type):
       returnAsSampleEJB
...
[dc] Return statement violates CB3/4
[dc]   Value type = mar.basicfail.SampleEJB
[dc]   Return type = java.lang.Object
[dc]   Offending statement: return r0
[dc]
[dc] Deployment failed!!!
...
```

Figure 7: Snippet of confinement tool output generated in case of error.

5.2 Testing Existing Beans

To test our tool, we used it to analyze a number of beans. We developed a few simple test beans, some of which intentionally violate the confinement model (in one case also violating the EJB specification — although that bean was subsequently accepted for deployment by the standard J2EE RI server). In addition, we analyzed a number of publically available beans that we obtained from web. These beans are from from the Pet Store application [43] used in Javasoft's J2EE tutorial for their J2EE Reference Implementation and from IBM's WebSphere Trade application [22] as used in IBM's benchmark suite. Of these downloaded beans, three from the Pet Store application required minor restructuring of their package layout. In each of these, the required changes involved modifying the package name of the transfer classes and amending the appropriate import statements to reflect this package change. Our findings are summarized in Figure 8.

We took each example bean and inspected the code to determine which classes were helpers and which were transfer objects. Only three beans used transfer objects: these were rewritten to move the transfer objects' classes into a separate package. We compiled the code to Java Bytecode, which we submitted our deployment checker. The results are given in Figure 8. Only three beans failed to be accepted by the deployment checker, though two of these were written specifically to fail to test our checker. The remainder of the beans were accepted with no further changes to their source code. This suggests that our approach is compatible with existing approaches to writing EJBs.

It is worth investigating in detail why the third failing bean was not passed by the checker as we had expected. When processing the bean `TradeBean`, the tool reported the following cryptic error (reformatted):

```
Invocation which violates CB3/4.
Offending expression: staticinvoke <java.util.Collections:
void sort(java.util.List,java.util.Comparator)>(r1, $r13)
argument number 2
```

After tracking down the error in the source code, we found that it corresponds to the second of the following lines of code.

```
ArrayList sortedQuotes = new ArrayList(quotes);
java.util.Collections.sort(sortedQuotes,
                    new quotePriceComparator());
```

The class `quotePriceComparator` is an inner class defined in the `TradeBean`. Since the `TradeBean` is confined, then its inner classes must also be confined. However, the instance of the confined class is passed to a static method thereby violating confinement rule `CB3`.

Here is the implementation of the offending class:

```
class quotePriceComparator
      implements java.util.Comparator {
 public int compare(Object quote1, Object quote2) {
   double change1 = ((LocalQuote) quote1).getChange();
   double change2 = ((LocalQuote) quote2).getChange();
   return
    new Double(change2).compareTo(new Double(change1));
 }
}
```

This is pretty innocuous, since it not only does not refer to the instance of `TradeBean`, its interface provides no way for the `TradeBean` instance to leak. A more sophisticated analysis could have detected this. Instead, we believe that this class need not be nested, and certainly need not be confined, since the objects which it compares (of type `LocalQuote`) are not confined. Moving this class out of the confined package resolves the problem.

(As an interesting aside, the WebSphere application had many classes named with the suffix `Bean`, although many of them were actually transfer objects rather than `EJB` objects. This unusual naming practice is likely to cause difficulties for those trying to maintain confinement without our help.)

The execution times of our tool on various beans are included in Figure 8. These times represent a fraction of the

Name	Description	Source	Size[1]	Time[2]	Comments
Simple `int` return	Returns `int` from method	authors	52	1.33	Accepted
Simple `EJB` return	Returns `EJB` object as various types	authors	64	1.42	Failed (CB3/4)[‡]
Simple Helper	Returns helper object from method	authors	119	1.94	Failed (CB3/4)[‡]
Simple Transfer	Returns transfer object from method	authors	119	1.53	Accepted
PS Inventory	Inventory EJB from Pet Store application	Javasoft	215	1.22	Accepted
PS Address	Address EJB from Pet Store application	Javasoft	373	1.22	Accepted*
PS CreditCard	CreditCard EJB from Pet Store application	Javasoft	318	1.27	Accepted*
PS ContactInfo	ContactInfo EJB from Pet Store application	Javasoft	431	4.83	Accepted*
PS CustomerAccount	CustomerAccount EJB from Pet Store application	Javasoft	196	3.55	Accepted
PS CustomerProfile	CustomerProfile EJB from Pet Store application	Javasoft	263	1.73	Accepted
PS Customer	Customer EJB from Pet Store application	Javasoft	179	3.52	Accepted
WS Account	Account EJB from WebSphere Trade application	IBM	503	5.30	Accepted
WS AccountProfile	AccountProfile EJB from WebSphere Trade application	IBM	349	1.18	Accepted
WS Holding	Holding EJB from WebSphere Trade application	IBM	336	1.25	Accepted
WS KeyGen	KeyGen EJB from WebSphere Trade application	IBM	132	1.22	Accepted
WS KeySequence	KeySequence EJB from WebSphere Trade application	IBM	156	6.46	Accepted
WS Order	Order EJB from WebSphere Trade application	IBM	465	1.24	Accepted
WS Quote	Quote EJB from WebSphere Trade application	IBM	700	1.25	Accepted
WS Trade	Trade EJB from WebSphere Trade application	IBM	1205	5.88	Failed (CB3/4)[†]

[1] size given in lines of code (including comments)
[2] analysis timings in seconds on a iBook 800MHz PowerPC G3, running Mac OS X Version 10.2.4
* after transfer objects were moved to an unconfined package
† accepted after minor modification (see text) ‡ written to fail (see text)

Figure 8: Summary of results of checking various simple test beans and several 'real-world' EJB components.

time taken to deploy a bean using existing deployment tools. For comparison, our tool takes 1.42 seconds to check the confinement properties of our SimpleEJB bean, whereas deployment of this bean, on the J2EE Reference Implementation server, takes from 16 to 24 seconds depending on the server state. As such, the deployment time overhead due to confinement checking is not significant compared to existing deployment times.

As the SOOT framework loads and type checks classes lazily, and thus interacts with our deployment phase, we consider these times to be an upper bound which could be significantly improved by a little optimization, by using a more specialized implementation, and by caching the results of previously computed analyses. The timings are nonetheless encouraging as they are already small. Each bean class is checked only once at deployment time and thus have no effect on the subsequent use of a bean.

6. DISCUSSION

The approach we propose for ensuring bean confinement is simple for all parties: simple for a developer to understand and abide by and simple for an EJB Server vendor to implement correctly and efficiently, imposing little overhead on the deployment phase. Furthermore, it requires virtually no change to existing EJB servers, and very little change to user code. In any case, these changes primarily solidify and enforce relationships which are implicit in the current specification. Last but not least, our approach introduces no run-time overhead.

In addition to these benefits, our approach has some less obvious ones which came about because our rules are tailored to our particular application. By adopting an asymmetric approach, whereby we check only that confined code defends itself against confinement leaks, rather than checking potentially malicious code, we need not check unconfined code and gain obvious efficiency benefits. Furthermore, our approach is modular. Confinement is specified per unit of deployment and can be different in different units. Unconfined code is unrestrained and different beans can share the same classes. A class can be considered confined with respect to two different beans, or confined in one and not in another, or even used freely within unconfined code. This plurality is enabled by the defensive and asymmetric approach: rather than globally stating that a class is confined, confinement is relative to a particular unit of deployment. Parametric classes, as in Ownership Types [16], would have achieved the same effect with rather more effort.

Unfortunately our approach also has a number of weaknesses due both to Java's weak type system and the fact that we must interface with existing library code. The main place this shows up is with collections. As a simple example, a bean may wish to use a vector to store a collection of helper objects. Unfortunately, rule **CB5** prevents the fruitful use of `java.util.Vector`, as the element type `java.lang.Object` is not confined. There are a number of ways around this problem:

- We could provide additional analysis for collections (much of `java.util`) to ensure that the collection provides the appropriate degree of confinement, that the collection is used within confined code in a confined manner, and that the developer never exploits the small gap between where an object in the collection is of type `java.lang.Object` and when it is recast to the appropriate confined type. The required analysis is relatively simple and is equivalent to showing that the class is generic [23] and sensibly treats elements with the type parameter as type (for example, by not casting them to `java.lang.Object`).
- We could invent a class `ConfinedObject` which can safely be treated as confined (this could replace or complement `java.lang.Object`). All confined classes

could extend this class. We could then supply a library of collections which take confined objects and themselves can be confined. Because of our constraints, these can be treated as confined and not appear in the interface, or be safely used in unconfined code. But this suggests another alternative.

- Rather than introduce a special object class, we can just specify `java.lang.Object` as being confined and enable a whole lot of `java.lang` and `java.util` to be confined (those classes which satisfy our conditions). Then we can use collections freely in confined code. The only problem then is that neither the collection nor `java.lang.Object` can appear in boundary interfaces. In relation to rule **CB5**, this could be a severe problem.

The generic version of Confined Types goes some way to addressing this problem, though it lacks the required support at the bytecode level [48].

In any of these cases, it would be sensible to cache (or even pre-compute) which library classes can be treated as confined. We could even check which static methods, such as the one we encountered in Section 5.2, do not affect confinement — technically, those which borrow their arguments [20]. As an aside, note that arrays can store confined objects, so long as they are not cast to `Object[]`, which rule **CB3** prevents.

A number of alternative approaches could have been applied to address EJB confinement but were not suitable for a number of reasons.

A programming language that supported confinement via Ownership Types [16] or Confined Types [48] (with some extension) could have been used. Apart from requiring that the developer change languages, the checking is performed too early, as the server accepts only Java bytecode. This suggests that the fact that the desired confinement constraints are satisfied could be recorded in the bytecode, via a sort of proof-carrying code [34]. The proof of confinement could then be checked at deployment time. Not only does this approach require appropriate programming language and compiler support, we feel that the underlying technology has not yet reached the degree of maturity to be placed in the hands of EJB developers.

A stronger module system which was designed with confinement in mind would alleviate the difficulty discussed with collection objects. While stronger module systems for Java exist [3, 6], none as far as we are aware support a confinement discipline. In any case, this approach is again not feasible because it would force the developer to change languages or compiler. Under our scheme, the developer can continue using Java and her favorite compiler.

One alternative which would place no demands on the developer nor the compiler is to leave the entire bytecode analysis up to the deployment checker, perhaps using escape analysis [7], without any support from a programming discipline. Relying on program analysis suffers from a number of potential problems. A sophisticated program analysis is not likely to be as efficient as our approach, because it would need to analyze both confined and unconfined code. Sophisticated analysis tend to be sensitive to small changes in code, so (especially if different vendors implemented different algorithms) a bean may run on one vendor's EJB server but not on another. If the constraints that are checked are hard to understand, vendors would be likely to abandon confinement checking altogether.

Other approaches we considered include sand-boxing [32] to completely separate beans from each other, perhaps using Java's proposed Isolate API [30], and bytecode rewriting [40] or program monitoring [28] to ensure that no references to beans leak. All of these approaches would be difficult to implement correctly, result in a loss of efficiency to ensure confinement, and require significant changes to the EJB server. Additionally, a runtime solution would require special exceptions to be raised or having the offending code crash. Both of these unpleasant outcomes are avoided by finding confinement violations in advance of execution.

To summarize: these approaches, we feel, throw too much technology at what turns out to be a relatively simple problem. The solution we propose is lightweight and well-suited to the problem at hand, and, as checking is done before the beans are executed, developers can readily understand any failing checks and there is no impact on the EJB server's run-time performance.

7. RELATED WORK

Pointer confinement is an issue of increasing importance in object-oriented software, touted not only as beneficial for software engineering, but also promising to make reasoning simpler, opening more opportunities for optimization, and closing security holes.

The work reported in this paper specializes the confinement of pointers to the real-world setting of Enterprise JavaBeans. The resulting confinement scheme has a different character from the existing and more general approaches to confinement and related problems that exist in the literature.

The Geneva Convention on the Treatment of Object Aliasing [21] precipitated a stream of proposals addressing aliasing, beginning with the overly strict disciplines of Islands [20] and Balloons [2]. Other approaches followed that are based on notions of ownership, read-only, anonymity, borrowing and similar ideas. Boyland, Noble and Retert present a capability calculus unifying many of these ideas [11], however, this is primarily descriptive and not intended for practical application. Clarke and Wrigstad [15] describe a more type-theoretic perspective, expressing many of the same concepts from Boyland *et al.* in a system based on ownership types. Both papers provide good surveys of the area.

Ownership types can express that an object has an owner and is not accessible outside that owner [16, 13, 12]. Ownership types originate from Flexible Alias Protection [35], and have been applied in modular reasoning about programs [33], expressing architectural constraints [1], in preventing data races [9], deadlocks [8], and safe region-based memory management in Real-time Java [10]. Ownership types are capable of expressing the pointer confinement desired here, at the cost of additional type annotations, and hence modifications to languages, compilers, and most likely, run-time systems.

Vitek and Bokowski take a more lightweight approach than ownership types with their Confined Types [48]. Using a few simple annotations, Confined Types enable objects to be confined within their defining Java package. Follow-up work by Grothoff, Palsberg and Vitek [19] demonstrated that program analysis performed by their `kacheck` bytecode

analysis framework obviates the need for annotations. As the rules underlying Confined Types and the approach taken closely resemble our EJB confinement model and its checking, a detailed comparison is warranted.

Firstly, the granularity of the confinement specified between the two systems is different. Confined Types offers package level confinement, whereas the confinement we offer is per object (or more precisely, per bean). A relatively small change to the rules of confinement — forbidding confined objects access to the package scope — is required to bridge this gap [14].

By having confinedness specified per package, Confined Types do not permit a class to be used as confined in one package (bean) and unconfined elsewhere — confined classes are in essence sealed within the package. Not using packages as confinement boundaries, and checking only confined code, has enabled us to overcome this limitation. We can use the same class in different beans and within unconfined code.

The aspect which is present in Confined Types but lacking in our system is *anonymous methods*. These are methods which do not pass `this` to another object, and they determine which methods can be safely inherited and used within confined code. This enables a degree of flexibility, though it slices class interfaces into parts which can and cannot be accessed. Rather than use the notion of an anonymous method, we allow only existing confined code and `java.lang.Object` to be subclasses. We actually permit methods which would not be considered anonymous, because our other rules guarantee that they behave sensibly. Unfortunately, neither our work nor Confined Types deals very well with collections and static methods such as `java.util.Collection.sort`, discussed in Section 6. We provide a number of choices for addressing collections, but statics require further analysis.

Compared to Grothoff *et al.*'s `kacheck` framework [19], our approach uses a significantly less sophisticated approach to analysis. As such, our analysis does significantly less work than the `kacheck` framework. Specifically, we do not do a full confinement analysis — performing no flow and no super-type analysis. On the other hand, our analysis finds places where the boundary is crossed and checks the types that may cross the boundary. As these checks are performed purely by inspecting (typed) bytecode, they are relatively inexpensive to implement and execute. In addition, we analyze only confined classes and boundary interfaces, and thus the amount of code we analyze is also smaller.

In light of `kacheck`'s results, our results are perhaps somewhat surprising. We are able to ensure confinement in a large proportion of our sample cases by performing a very crude and semantically shallow analysis. We feel that this is due to the domain in which we are working. The Enterprise JavaBeans model is naturally confinement oriented as it is based on the use of wrapper objects as interfaces to the EJB components. As such, our results naturally follow from the properties of the domain. The `kacheck` authors noted themselves that they expect their "numbers [representing the amount of confined classes] will rise even further once programmers start to write code with confinement in mind" [19]. In addition, since we are not aiming for a general solution, we have been able to carefully select confinement rules that work well in our domain. In contrast, Confined Types may be more widely applicable, although we also expect to be able to generalize our approach.

In private communication, Jan Vitek claims that Confined Types can be encoded using existing Java mechanisms such as interfaces and false exceptions. Unlike our system, this would not permit a class to be used as confined in one part of the code and unconfined in another. More importantly, existing confined type regimes can only confine objects within a static scope, while our system confines objects within other objects. Hogg's Islands [20] are the only other ad-hoc alias scheme to provide this level of protection, but Islands require significantly more programming-level support than the scheme we present here.

On another front, Banerjee and Naumann prove a representation independence theorem for Java programs which exhibit a certain form of pointer confinement [4, 5]. This theorem means that one can replace the implementation of the confined entity by an equivalent one, irrespective of the remainder of the program. The notion of confinement they employ is slightly different from ours, in that it also prohibits, in our terminology, different boundary objects from referring to each other and confined objects from referring to any external objects. Such a discipline would severely cripple the EJB model. Furthermore, their analysis is performed over all the code in a system, rather than just analyzing the confined code. In any case, we stress that their main contribution is the representation independence theorem, which is to a large degree parametric in the confinement regime.

As a rough estimate, we place our proposal somewhere between the the Confined Types analysis work and that of Banerjee and Naumann, with the important result that our analysis, though coarser than these previous approaches, has provided good results without the costs associated with more complex analysis.

In an abstract setting, Leroy and Rouaix confirm the folklore that strongly-typed applets are more secure than untyped ones [27]. Applets are prevented from modifying certain sensitive locations using lexical scoping, procedural abstraction and type abstraction. Their work gives confidence that ours is even possible — in a language such as C [25], one would lack such confidence. An aspect which parallels the work presented here is that they verify that checks between sensitive and insensitive data need to be applied at the boundary. Leroy and Rouaix insert run-time checks at these places in a type-based manner, whereas we perform a static check.

As we mentioned in section 6, there are many other possible approaches to this problem including program analysis, sandboxing, program monitoring via bytecode rewriting, and proof-carrying code. Each of these approaches have drawbacks that we deemed inappropriate for the EJB domain.

Along with our experience with our deployment checker, a number of experimental studies have indicated that a significant amount of confinement exists in real-world programs, through both examination of bytecode [19], or direct examination of object graph dumps [37]. These results suggest that confinement is indeed prolific enough to warrant language mechanisms to provide programmers a means of expressing it.

8. CONCLUSION

The EJB architecture depends upon (but does not enforce) a containment constraint to protect its architectural integrity. Without proper enforcement of this constraint, de-

veloper errors and malicious code can violate the integrity of the EJB architecture with dire consequences: database inconsistency, transaction failures, and security holes.

Our solution requires bean developers to specify for each unit of deployment which objects are to be confined (ostensibly using package names) and to follow a disciplined programming style to ensure that no confined object crosses that boundary. The scheme is enforced during deployment time as beans may be developed by unknown sources, although compile time checks could easily be added to development environments.

The changes we propose are relatively inexpensive to incorporate: we merely require that the confinement constraints implied by the EJB specification are enforced by EJB Servers. Given the existing overhead of bean deployment — parsing XML deployment files, generating and compiling classes, and the frequent restarting of new JVMs — our experimental results indicate that our analysis will have negligible effect on deployment times. Furthermore, our approach does not require any changes to the JVM that is hosting the EJB Server.

Although confinement has been proposed in various forms in the literature, this is, to the best of our knowledge, the first application of confinement analysis to realistic systems that are used in commercial software environments. Because of the importance of maintaining confinement in this setting, we advocate that future versions of the EJB specification require that confinement be specified in the manner we have discussed *and* that compliant EJB servers enforce this constraint.

From a broader perspective, we have demonstrated that the work which we and others have been doing in the abstract to address problems of confinement has a concrete, practical application. A relatively simple confinement model with natural, easily specified boundaries can protect the integrity of the Enterprise JavaBeans architecture, while making very low demands on the developers and insignificant demands on the run-time system.

9. ACKNOWLEDGMENTS

The authors thank Jens Palsberg, Eelco Dolstra and Tobias Wrigstad for their comments on a draft of this paper, and Adrian Mos for directing us to some useful sample EJB code. Jan Vitek helpfully took the trouble to explain some of the technicalities of Confined Types. The last author would like thank the staff of Ward 18, Wellington Hospital. This work was supported in part by the Department of Computing Sciences at Purdue University, DARPA grant contract F33615-01-C-1894, and the Royal Society of New Zealand Marsden Fund.

10. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 311–330. ACM Press, 2002.

[2] P. S. Almeida. Balloon Types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*, June 1997.

[3] D. Ancona and E. Zucca. True modules for Java classes. In *European Conference on Object-Oriented Programming*, June 2001.

[4] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence in object-oriented languages. Submitted for publication, December 2002.

[5] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[6] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in Java. *Software–Practice and Experience*, 2003.

[7] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*. ACM Press, October 1999.

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 211–230. ACM Press, 2002.

[9] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 56–69. ACM Press, 2001.

[10] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 118–125, San Diego, California, June 2003.

[11] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming*, June 2001.

[12] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[13] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 292–310. ACM Press, 2002.

[14] D. Clarke, J. M. Fox, J. Noble, and J. Vitek. Scopedjava: Ownership for real-time Java. Under revision, Nov. 2002.

[15] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming*, July 2003.

[16] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 48–64. ACM Press, October 1998.

[17] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *PLDI 2000. ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, January 1995.

[19] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating

objects with confined types. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 241–255. ACM Press, 2001.

[20] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM SIGPLAN Conference on Object-Oriented Pogramming, Systems, Languages and Applications*, pages 271–285. ACM Press, 1991.

[21] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[22] IBM. *WebSphere end-to-end benchmark and performance sample application: Trade 3*. `http://www-3.ibm.com/software/webservers/appserv/benchmark3.html`.

[23] A. Igarashi, B. C. Pierce, and P. Wadler. A recipe for raw types. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001.

[24] N. Kassem and the Enterprise Team. *Designing Enterprise Applications with the Java^{TM}Platform, Enterprise Edition*. Addison-Wesley, June 2000.

[25] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, March 1988.

[26] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[27] X. Leroy and F. Rouaix. Security properies of typed applets. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*. Springer, 1999. Also appears in 25th ACM Conference on Principles of Programming Languages, 1998.

[28] J. L. Lujo Bauer and D. Walker. Types and effects for non-interfering runtime monitors. In *International Symposium on Software Security*, November 2002.

[29] F. Marinescu. *EJB Design Patterns*. John Wiley, February 2002.

[30] Java Community Process. Application Isolation API Specification, 2003. `http://jcp.org/jsr/detail/121.jsp`.

[31] JBoss Group. JBoss Application Server. `http://www.jboss.org`.

[32] G. McGraw and E. Felten. *Securing Java*. John Wiley and Sons, January 1999.

[33] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[34] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[35] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.

[36] Oracle Corporation. *Oracle9i Application Server*, September 2002.

[37] A. Potanin. The Fox – a tool for object graph analysis, 2002. Honours Report, Computer Science, Victoria University of Wellington, New Zealand.

[38] M. Richmond. *Flexible Migration Support for Component Frameworks*. PhD thesis, Macquarie University, Sydney, Australia, 2003.

[39] M. Richmond and J. Noble. Reflections on remote reflection. In *Proceedings of the 24th Australasian Computer Science Conference (ACSC-01)*, volume 23.1 of *Australian Computer Science Communications*, pages 163–170. IEEE Computer Society, January 2001.

[40] A. Rudys and D. S. Wallach. Enforcing Java run-time properties using bytecode rewriting. In *International Symposium on Software Security*, November 2002.

[41] M. Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204. IEEE, May 1986.

[42] Sun Microsystems. *Java 2 Enterprise Edition Reference Implementation*. `http://java.sun.com/j2ee/`.

[43] Sun Microsystems. *Java Pet Store Sample Application*. `http://java.sun.com/blueprints/code/`.

[44] Sun Microsystems. *Java^{TM}Remote Method Invocation Specification revision 1.8*.

[45] Sun Microsystems. *Enterprise JavaBeans Specification version 2.3*, Aug 2002.

[46] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman, Harlow, Essex, 1997.

[47] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. SOOT - A Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, November 1999.

[48] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.