Validating Structural Properties of Nested Objects

Darrell Reimer*, Edith Schonberg*, Kavitha Srinivas*, Harini Srinivasan*,
Julian Dolby, Aaron Kershenbaum, Larry Koved
IBM Research, 19 Skyline Drive, Hawthorne, NY, USA 10532
ediths@us.ibm.com

Abstract
Frameworks are widely used to facilitate software reuse and accelerate development time. However, there are currently no systematic mechanisms to enforce the explicit and implicit rules of these frameworks. This paper focuses on a class of framework rules that place restrictions on the properties of data structures in framework applications. We present a mechanism to enforce these rules by the use of a generic "bad store template" which can be customized for different rule instances. We demonstrate the use of this template to validate specific bad store rules within J2EE framework applications. Violations of these rules cause subtle defects which manifest themselves at runtime as data loss, data corruption, or race conditions. Our algorithm to detect "bad stores" is implemented in the Smart Analysis-Based Error Reduction (SABER) validation tool, where we pay special attention to facilitating problem understanding and remediation, by providing detailed problem explanations. We present experimental results on four commercially deployed e-commerce applications that show over 200 "bad stores".

Categories and Subject Descriptors
Software [**Software Engineering**]: Coding Tools and Techniques

General Terms
Design, Management, Performance, Verification

Keywords
Code validation, frameworks, context sensitive analysis

1. Introduction
Today's e-commerce applications make extensive use of frameworks, which provide commonly needed services, such as security, availability, session management, and resource pooling. In fact, many e-commerce applications consist of several different frameworks integrated by only a small amount of glue code. While this development represents a major success for software reuse, these framework-intensive applications pose unique challenges when runtime problems arise. Frameworks are replete with implicit and explicit assumptions and usage rules, and it is very easy for a programmer to inadvertently violate one of these rules, either from misunderstanding or oversight. Even simple coding errors can cause subtle, difficult-to-diagnose runtime problems. If undetected before a system goes into production, one of these errors can result in significant revenue loss.[1]

Based on five years of experience solving high impact problems in J2EE™ applications, we have discovered that many common framework-related errors in these applications conform to a small set of illegal code patterns that can be detected by static analysis. Among these, we have identified a new pattern of errors, which we call ***bad stores***. Bad store errors result in data structures that violate framework rules. In our J2EE examples, they manifest themselves at runtime as data loss, data corruption, and race conditions, causing intermittent failures, performance degradation, and system outages. They often evade early detection during testing and cause havoc later in production.

Contributions. This paper makes the following novel contributions:

- We introduce a general, customizable bad store template for describing a range of framework-related data structure rules. The bad store template states that an object ***A*** cannot be stored into a data structure rooted at another object ***B***. Individual rules are formed by instantiating the template and specifying the properties of ***A*** and ***B***, thereby defining the precise conditions for violating the rule.
- We give specific examples of bad store rules, which can be used to detect problems such as data loss, data corruption, and race conditions in applications that use the J2EE framework.
- We present a validation algorithm for detecting bad store rule violations.
- We present results of our analysis applied to 4 commercially deployed applications, with over 600,000 lines of code.

* Authorship order is alphabetical.
[1] A single system outage can cost a business over $100,000.

™ Java 2 Enterprise Edition (J2EE) is a trademark of Sun Microsystems.

The bad store template is one of the small set of customizable rule templates supported by the SABER validation tool [23]. SABER is targeting a wide audience with varying levels of skill, so that ease-of-use is an important goal for both rule customization and for problem understanding and remediation. We believe that customizable templates, like design patterns, are easy to use and retain flexibility and generality. Additionally, we have accumulated enough empirical data that we feel confident about selecting broadly useful templates.

In our experiments, we have defined bad store rules for applications containing JSPs, Servlets, Enterprise Java Beans, HttpSessions, Web Services and Struts [17]. We conjecture that bad store rules can be used for finding errors in other types of applications using frameworks as well.

2. Examples

Our examples of bad stores are distilled from five years of experience working with commercial J2EE applications. In a typical scenario, a problem does not appear during unit testing and even under load testing, since the precise conditions which trigger the problem occur infrequently. However, after a system has been in production for several months and the overall load has increased significantly, intermittent transaction failures start to appear. To diagnose the problem, it takes days of effort to isolate the right set of events in a complex production environment.

Persistence. An HTTP request is stateless. An application typically uses an *HttpSession* object to cache state across multiple HTTP requests, creating one *HttpSession* object for each user session. If a session object cache grows too large or it is write-through, it is written to a persistent store.

Subtle errors arise when a session is persisted and a cached object does not implement the *Serializable* interface. *Serializable* methods implement the conversion of in-memory object representations to flattened external representations. Classes must implement the *Serializable* interface to assume responsibility for saving and restoring their own object state. If an *HttpSession* which contains non-"serializable" data is written to a persistent store, the non-serializable data is silently lost, and the next access to this session object results in the retrieval of corrupt or missing data from the persistent store. Therefore, it is unsafe to store a non-serializable object in an *HttpSession*. For example, in Figure 1, a single problematic leaf node of a cached object is non-serializable, which violates the *HttpSession* rule.

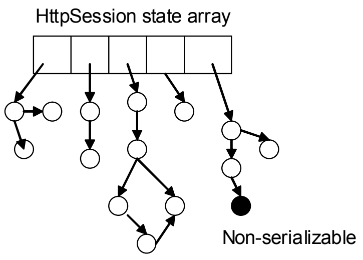

Figure 1.

Multiple Address Spaces. J2EE applications can be configured with multiple application servers, each of which spawns its own Java Virtual Machine, resulting in multiple address spaces. Separate requests from the same user session can be processed by different application servers. These requests can share the same *HttpSession* object, through the persistence mechanism described in the prior section. Problems arise when address-space-specific data is stored into an *HttpSession* object, and is subsequently shared by multiple application servers. In one application we studied, resource handles to data in a backend system, specific to a given address space, were being stored into *HttpSession* objects. Transactions failed when an application server with a different address space retrieved a back end handle from a session that it had not created. Resource handles are one example of address-space-specific data.

Shared Fields. In J2EE applications, there is a single *Servlet* object for each type of request, and the multiple server-worker threads that process *Servlet* requests share the *Servlet* objects. A major source of production problems results from developers not realizing that *Servlet*s are multi-threaded and that *Servlet* instance fields are shared state.

In the example in Figure 2, which was taken and cleansed from a commercial system in production, the field *user* is actually shared across all invocations of *MyServlet.* If two or more users of the system make requests to *MyServlet* at the same time, a race condition occurs involving the field *user.userId*. This race condition, in fact, is also a potential security violation, since it is possible for one user to be validated with someone else's userid. The user may receive incorrect data or unauthorized access to parts of the system.

```
MyServlet extends HttpServlet {

 private User user = new User();

 public void doGet(HttpServletRequest req,
          HttpServletResponse resp) {
 getInfo(req);
 if (validate())
  < do something >
 }
 void getInfo(HttpServletRequest req) {
 Session session = req.getSession();
 user.userId =
    session.getAttribute("userid");
 }
 private boolean validate(){
  checkAuthorization(user.userId);
 }
}
```

Figure 2.

We found that this simple type of race condition is so common in e-commerce systems, that it is useful to detect all store operations to *Servlet* instance fields in code reachable from *Servlet* methods. A related rule detects objects referenced by static fields, which are also shared state in multi-threaded code. We use an additional simple filtering heuristic to reduce the false positive rate by filtering out stores in synchronized methods, as described in Section 3.

In general, static race detection algorithms must identify which state is shared, which code is multi-threaded, and which code blocks are synchronized [11]. We use domain knowledge to customize our rules. We know that *Servlet* interface methods are multi-threaded and that multi-object data structures are sometimes shared, thus requiring object reference analysis to identify shared state. We also know that synchronized methods and blocks are used rarely in the user-written part of e-commerce applications – it is best practice in these applications to rely on frameworks for common resource management, and avoid user synchronization. These observations are verified in our experimental results.

3. Bad Store Template

The philosophy of SABER is to support a few general customizable templates for defining runtime problems. Customization is accomplished by instantiating template parameters. We prefer templates because we are targeting application developers, and templates are easy to understand and use. Customization is necessary because application developers will invariably want to validate rules which are different from the predefined rule set that a tool provides. Generally, a rule set encodes domain knowledge, addressing problems which are the result of framework misuse, rather than Java errors *per se*. Development groups often have their own rules they wish to enforce for their custom frameworks, and additional components are always being added to any platform.

The bad store template is one of the key general templates in SABER.[2] The runtime problems in Section 2 can be described by instantiating this template.

3.1 Bad Store Template Parameters

The basic form of the bad store template is as follows:

> *Do not transitively store references to objects with properties O into fields with properties F in methods transitively called from M.*

Individual *bad store rules* are defined from this template by providing specific properties for parameters *O, F*, and *M*. If there are no properties specified for a parameter, the default is "all". A bad store rule is applied transitively. That is, if an object X is stored in object Y, and Y is stored in object Z, then we must test whether it is legal to store X in Y, Y in Z, and X in Z.

Supported object properties for parameter *O* are:

- A package, class, or interface name.
- All subclasses of a class or interface.

For example, if the parameter *O* is instantiated with the property "class Foo", then any object which has the concrete type *Foo* satisfies the property. If *O* is instantiated with the property "all subclasses of Foo", then any object which is an instance of either *Foo* or any subclass of *Foo* satisfies the property.

Supported field properties for parameter *F* are:

- A field modifier: *static, transient, final, private, public, protected, volatile*.
- A field name.
- A class, package, or interface name.
- A class loader name, which serves as a namespace.
- A field in all subclasses of a class or interface.

For example, if *F* is instantiated with the property *static*, then any field which is static satisfies the property**.** If *F* is instantiated with the property "field x in Foo", then only the field *x* in objects of concrete type *Foo* satisfies the property. If *F* is instantiated with the property "Foo", then all fields in objects of concrete type *Foo* satisfy the property. Finally, if *F* is instantiated with the property "field x in all subclasses of Foo", then the field *x* in any object which is an instance of either *Foo* or any subclass of *Foo* satisfies the property.

Multiple properties can be specified for each parameter. Each property can be specified as either a *match* property or *filter* property. Match properties specify which fields or objects satisfy the rule, and filter properties specify which fields or objects do not match, and should be ignored. When a rule is evaluated, a field (object) matches a set of field (object) properties if the field (object) satisfies all match properties and is not filtered out by any filter property. Additionally, field filter objects are applied transitively.

For example, if a set of field properties includes the match property "class Foo", and the filter property "field x in Foo", then all fields except *x* in objects of type *Foo* satisfy the set of field properties.

There is one filter that is always applied: any bad store detected must be in a method transitively called from a root method. This eliminates from consideration potential stores in methods which are never used by the application. By default, the root methods are the application entry points. For J2EE, the entry points are JSP, *Servlet*, and EJB methods. We provide a "root method" parameter *M*, making it possible to override the default root methods, thereby further restricting the set of potential bad stores.

More formally, let *P* be a set of field (object) properties. Then, $P = P_m \cup P_t$, where P_m are match properties, and P_t are filter properties. If *p* is a property in *P* and *x* is a field (object), then the predicate *p(x)* is true if *x* satisfies *p*.

A field (object) *x satisfies* the property set *P* iff:

$$P = \phi \text{ or}$$
$$(\forall p_m \in P_m)\ p_m(x) \text{ and } (\forall p_t \in P_t)\ \neg p_t(x) .$$

[2] The other SABER templates are: Must call X, Do not call X from Y, Must call X after Y, Do not extend X, and Implement method pairs X and Y consistently.

Bad Store Rule Don't Store ….	***Object Parameter O***		***Field Parameter F***	
	Match	**Filter**	**Match**	**Filter**
Non-Serializable Data In HttpSession		Subclasses of java.io.Serializable	Field javax.servlet.HttpSession.attributeValue s	transient
Address-Space Data In HttpSession	Subclasses of java.lang. Thread		Field javax.servlet.HttpSession.attributeValue s	
In Servlet Field			Subclasses of javax.servlet.Servlet	static
				Subclasses of SingleThread Model
In Static Field			Static	

Table 1.

A field *x survives* the property set *P* iff:

$$P = \phi \text{ or } (\forall p_t \in P_t) \neg p_t(x)$$

Bad Store Definition. Let *R* be a rule, where parameter *F* is instantiated with a set of field properties *FP*, parameter *O* is instantiated with a set of object properties *OP* and *M* is instantiated with a set of root methods *MP*.

A field-object pair *(f, o)* is a *bad store* with respect to rule *R* iff:

- *f* satisfies the field properties *FP*,
- *o* satisfies the object properties *OP,*
- *o* is transitively referenced by *f,* such that each field in the reference sequence from *f* to *o* survives the field properties *FP*, and
- the bad store of *o* occurs in a method transitively called from a root method in *MP*.

3.2 Examples

Specific bad store rules for the examples in Section 2 are shown in Table 1. (The parameter *M* is not shown.) The first two rules match stores to *HttpSessions*. The next two rules match stores into shared state. Both *Servlet* member fields and static fields are shared state.

The first rule matches all non-serializable objects which are stored in an *HttpSession* object, as follows. The object parameter *O* is instantiated with a filter property that filters out all objects which are subclasses of the interface `java.io.Serializable`. Therefore, only objects which are not serializable need to be examined. The field parameter *F* is instantiated with a match property which is a field name *attributeValues* in *HttpSession*. The field *attributeValues* is the name of the state array shown in Figure 1. Additionally, the field properties also include a filter property *transient*. If a non-serializable object is stored into a transient field, then we don't have to worry about persistent data consistency, by definition.

Note that filtering is transitive. If an object X is stored in a transient field y1 of object Y, and Y is stored in *attributeValues* in an *HttpSession* object, then all objects stored in y1 are filtered out, including X.

The second rule is like the first rule, except it supplies a match property for the parameter *O* instead of a filter property, which matches all objects which are instances of `java.lang.Thread`. This class contains address-space specific data, so this rule detects problems where address-space data is stored into *HttpSessions.*

In the third rule, the parameter *F* is instantiated with a match property which matches all fields that are members of subclasses of the J2EE class `javax.servlet.Servlet.` Static fields are filtered out, since these are handled in the next rule. The rule also filters out fields in *SingleThreadModel* subclasses since these fields are never shared. The parameter *O* is not instantiated, so all objects match the rule. We also supply for the parameter *M* the root methods of *Servlet*: *service, doGet, doPut*. All stores not reachable from these methods are ignored.

The last rule is like the third, except the field property *F* is instantiated with the match property *static*, and *SingleThreadModel* is not a filter.

3.3 More Filtering

It is well known that reporting too many false positives renders a validation tool unusable. Filtering is critical for reducing false positives. We have been able to reduce false positives to an average of 22% using the following techniques.

Callback filters. Some filtering cannot be expressed in a simple way as filter properties, and it is necessary to write code to achieve precise results. We provide a callback filter parameter as part of the bad store template. During evaluation, all user-supplied filter methods are called to prune the set of potential bad store operations.

For the first two rules which detect stores to shared state, we provide a callback filter in SABER. The filter eliminates store operations that occur in synchronized methods. This is a heuristic which eliminates false positives in some cases when accesses to shared state are properly synchronized. However, this filtering is not conservative, and so we can miss some real race condition errors.

Framework code. While there may be runtime problems in the underlying framework used by an application such as J2EE, it is possible to suppress reporting these problems. Optionally, we only report problems in the application code, and filter out all bad stores which are completely contained within the framework code.

4. Algorithm

This section presents our algorithm for finding occurrences of bad stores in a program. The algorithm, based on static analysis, computes an approximation of the bad store problems in a program. Because we also use filtering heuristics to minimize false positives, the algorithm is not sound. We believe that soundness is less important than scalability for the types of applications we are looking at, and it is unclear that both can be achieved without introducing large numbers of false positives.

4.1 Inputs From Static Analysis

Our algorithm uses a call graph *CG*, a field-sensitive *points-to* graph *PTG* [10] which represents object referencing, a mapping *storeOps* from object fields to nodes in the *CG* where stores to the given field occur[3], and a mapping *storeValues* from *CG* node and field to the set of possible objects (i.e. nodes in the *PTG*) that can be stored in the given field by the given node.

We refer to an allocated instance of a class as an *object,* and a specific field of an allocated object as an *object field*. Each object corresponds to an allocation site (`new` operation) in a Java program. A *PTG* contains two types of nodes, *object nodes* and *object field nodes*, which represent objects and object fields respectively. Similarly, a *PTG* contains two types of edges. There is an edge from an object node to an object field node if the field is a member of the object, and there is an edge from an object field node to an object node if the field may hold a reference to the object. Figure 3 illustrates the *PTG* and *CG* for Figure 2. The root of the *PTG* graph is the object field node *MyServlet.user*. Since there is an object of type *User* referenced by this field, there is a corresponding object node *User1* in the *PTG* which is a successor node of *MyServlet.user.* Similarly, the field *userId* is a member of *User1,* so there is an object field node labeled *User1.userId* which is a successor of *User1*. Finally, the field *userId* is assigned a reference to a String object, which is represented by the object node labeled *String1*. This object node is a successor of *User1.userId.*

[3] We actually analyze individual stores within methods, but we ignore that here to simplify exposition.

The call graph *CG* has two subgraphs, rooted at the *Constructor* node and the *doGet* node. *Constructor* is the default constructor method generated by the compiler to initialize *User* objects. The dashed arrows from *PTG* to *CG* represent *storeOps*, showing the code locations of field stores. Specifically, there is a store to field *MyServlet.user* in *Constructor*, and to *User1.userId* in *getInfo*. The arrows from *CG* to *PTG* represent *storeValues*, showing the objects stored by each store operation.

4.2 Bad Store Algorithm

We make the following definitions. We call an object field that satisfies the field properties *FP* in a rule a *root field* and an object which satisfies the object properties *OP* a *bad object*. If the store of a reference to a bad object *o* into a root field *f* is indirect, then there is a direct store of *o* into another object field reachable from *f* in the *PTG*. We call this reachable object field a *leaf field* of the bad store. We call a store operation where *o* is stored into a leaf field a *leaf operation*. In Figure 3, *MyServlet.user* is a root field, *String1* is a bad object, and *User1.userId* is a leaf field. The leaf operation is an assignment statement in the method *getInfo*. (Note that the root field and the leaf field can be the same object field. Also, there may be more than one leaf field and leaf operations for the same bad object.)

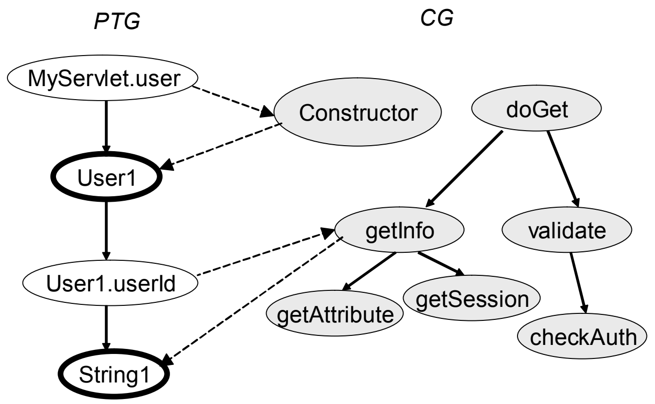

Figure 3.

The result of the algorithm is a set of store operations in the program code, which potentially result in an illegal data structure, according to a set of bad store rules. It computes the map *Problems*, which associates each root field to bad store information. For each potential bad store, the information includes the bad object, the leaf fields storing the bad object, and all of the leaf operations assigning the bad object to the leaf field. All of this information is important for providing enough context information to explain the problem to the user. The algorithm is executed for each bad store rule by traversing the *PTG*. The algorithm is shown in Figure 4.

4.3 Example

We apply the bad store algorithm to Figure 3, using the “Don’t Store in Servlet Field” rule described in Section 3.

Bad Store Algorithm. Let R be a bad store rule with field properties FP and object properties OP and root nodes RN.

```
1. $reachableCallGraphNodes \longleftarrow CG^*(RN)$;      /* all nodes reachable from the root nodes in $CG$ */
2. $rootFields \longleftarrow \{f \mid f \in PTG$ & $f$ is an object field node & $f$ satisfies $FP\}$; /* fields that satisfy the field properties of $R$ */
3. forall $f \in rootFields$ {
     $Reachable_f \longleftarrow \phi$;
     $Problems_f \longleftarrow \phi$;
4.   $filteredClosure(Reachable_f, f)$;       /* $Reachable_f$= nodes reachable from $f$ which survive filters */

     /* objects that satisfy the object properties of $R$ */
5.   $badObjects \longleftarrow \{o \mid o \in Reachable_f$ & $o$ is an object node & $o$ satisfies $OP\}$;
6.   forall $o \in badObjects$ {
7.       $leafFields \longleftarrow \{ lf \mid lf \in PTG$ & $lf$ is a predecessor of $o$ in $PTG$ & $lf \in Reachable_f\}$;
8.     forall $lf \in leafFields$ {
9.       $leafOperations \longleftarrow \{n \mid n \in storeOps(lf)$ & $o \in storeValues(n, lf)$
                                                  & $n \in reachableCallGraphNodes\}$;
10.      Apply the callback filters to prune $leafOperations$;
11.      if $leafOperations \neq \phi$ {
           $Problems_f.add([o, lf, leafOperations])$ ;
         }
       }
     }
   }

 /* $closure$ = the set of nodes reachable from $n$ which are not filtered. */
12. $filteredClosure(closure, n)$ {
13.   if $n$ is an object field node & $\neg survives(n, FP)$ return;
15.   for each child $cn$ of $n$ in the $PTG$ {
16.     if $cn \notin closure$ {
17.       $closure.add(cn)$;
18.       $filteredClosure(closure, cn)$;
        }
      }
    }
```

Figure 4.

After step 1, the set *reachableCallGraphNodes* consists of all nodes in the call graph which are reachable from *doGet* since *doGet* is a root method. The set *reachableCallGraphNodes* does not include *Constructor,* since *Constructor* is not a root method.

In step 2, object field *MyServlet.user* is identified as a root field, since it is a member of *MyServlet*, which is a subclass of `javax.servlet.Servlet`, and hence satisfies the field match property of the rule.

In step 4, all the nodes *User1*, *User1.userId*, and *String1* are assigned to $Reachable_{MyServlet.user}$. In our implementation, computation of closure is cached.

In step 5, both *User1* and *String1* are identified as bad objects, since they are both reachable from the root field *MyServlet.user* and satisfy the object properties in the rule. (Actually, there are no object properties, so all reachable objects match).

In step 7, both *User1.userId* and *MyServlet.user* are identified as leaf fields.

In step 9, the leaf operations in methods *Constructor* and *getInfo* are obtained, using *storeOps*. However, since *Constructor* is not in *reachableCallGraphNodes*, its leaf operation is discarded. There are no remaining leaf operations for *MyServlet.user*, so this leaf field is also discarded. The only interesting bad object is *String1* stored in leaf field *User1.userId.*

In step 11, $Problems_{MyServlet.user}$ is assigned the triplet [*String1, User1.userId,* { *getInfo*}].

4.4 Context Sensitivity

Context sensitivity distinguishes different runtime uses of a given static construct. SABER provides *method sensitivity* to distinguish different calls to a given method, and *class sensitivity* to distinguish different allocations of the same class.

The *method sensitivity* of the call graph construction algorithm used in SABER is similar to the Cartesian Product Algorithm (CPA) [1] in that it uses the sets of types of each argument to determine what context of the target method to use. It is distinguished from CPA because its contexts represent sets of types for each argument rather than individual types, as in CPA. This was designed to improve the scalability of CPA, but it introduces issues about whether the analysis is monotonic [19]. In terms of [21], SABER uses a parameterized object-sensitive analysis, with a varying context depth. For most allocation sites, the site itself is used as the object name, corresponding to depth of 1. For instance methods of collection classes, the allocation site is combined with the allocation site of the receiver object, corresponding to a depth of 2.

Early experiments analyzing J2EE applications for bad stores indicated that *method sensitivity* alone was insufficient for meaningful results and for scalability. We needed to be able to vary *class sensitivity* as well. SABER provides three levels, which are statically defined by type name at configuration:

- *Class-based sensitivity*, where a single allocation is used for all objects of a given class.
- *Allocation-based sensitivity*, where instances are differentiated based on the byte code offset within a method where the allocation occurred
- *Receiver-based sensitivity* is a refinement of allocation-based sensitivity for the instance. Allocation sites are augmented with the allocation site of the receiver of the method where the allocation occurs. Note that this only makes sense when receiver types are always precise, as in CPA. Our modified CPA specialized receiver types to have this added level of precision only where it is needed.

In SABER, the most commonly used method of *class sensitivity* is *allocation-based sensitivity* which is appropriate for our analyses because it makes the *PTG* more precise. There are two exceptions to *allocation-based* sensitivity. First, there are some classes (e.g., Strings) that have many allocations for which precision is not helpful. For these classes, we use *class-based sensitivity.* Second, *receiver-based sensitivity* is used to handle a pattern of code found in the collection classes in the J2SE framework. These container classes contain an internal data structure to store the elements of a collection. Without *receiver-based sensitivity*, the analysis identifies only a single allocation instance of the internal data structure for the entire application, because allocation of the internal data structure occurs at a single byte code offset within the program. Consequently, all stores to a collection of a given type appear to be a store to this single internal structure, resulting in a large number of false positives. We refer to this type of coding pattern as a *container pattern*.

```
public class Htable {

 private Entry table[];

 public Htable() {
 table = new Entry[];
 }

 /* Appears to put everything ever stored
       into e.value. */
 public void put(Object key, Object value) {
  // E1 declaration
  Entry e = new Entry(key, value);
       table[index] = e;

 }

 /* Appears to get everything ever stored
       into e.value */
 public Object get(Object key) {
  ….
 }
}

public class A {
 public void A'() {
 Object value = new Object();  //V1
 Htable h = new Htable();   //H1
 h.put(key, value);
 }
}

public class B {
 static Htable h = new Htable();  //H2

 public void B'() {
 Object value = new Object();  //V2
 h.put(key, value);
 }
}
```

Figure 5.

Figure 5 shows an example from the *container pattern*. The stores of objects *V1* and *V2* to different hash tables in the application *H1*and *H2* appear to be assignments into a single data structure *E1*. Consider the rule "Don't Store in Static Field" in Table 1. Without receiver context, we would incorrectly assume that stores occurred to a static field both in `B'` and in `A'`, when the store in `A'` is clearly not a bad store. By splitting the allocation instances of *E1* based on where the receiver hash tables *H1* and *H2* are allocated, we can correctly associate the appropriate stores with the appropriate objects. In our experiments, we found that such addition of context for collection classes reduced the number of false positives in one application from 61% to 13%. Interestingly, the remaining 13% false positives in that specific application occurred because of the occurrence of the *container pattern* in a different J2SE class (`java.text.DecimalFormat`).

Container patterns are also commonly used within the application code. It would be desirable for the call graph constructor to automatically select the appropriate level of context for these classes to eliminate false positives from the *container pattern*. This is however a difficult issue to address because precise knowledge of object use requires the construction of a call graph to identify locations that can benefit from selective addition of *receiver-based sensitivity*.

Iterative construction of call graphs could address this difficulty, but discovering where additional precision is required is challenging. Further, this approach might not scale for large scale J2EE applications. Therefore, we chose to pre-configure SABER to run with *receiver-based sensitivity* for collection classes alone, and tolerate false positives from the *container pattern* more generally. The percentage of false positives from the *container pattern* varies between 0% and 39% in the applications we studied.

5. Evaluation

To evaluate the effectiveness of our technique, we analyzed four commercial real-world J2EE applications. In this section, we briefly describe the benchmarks, their characteristics, and then present the results.

Benchmarks. We have applied the bad store template analysis against four commercially deployed J2EE applications checking for non-serializable data stored in HttpSession objects and for race conditions due to inappropriate stores to static fields and Servlet instance fields (see Table 1). Each of the applications analyzed has been in production for over a year. The four applications analyzed are:

(1) Human resources skills tracking tool
(2) Internet order status checking application
(3) Internet based financial application
(4) A large commercial framework based on J2EE

Benchmark Characteristics. To perform the bad store analysis, analyzing the application alone is not sufficient because object allocations occur in and control flow paths pass through other code that the application depends on. This other code includes the J2EE framework, any third party libraries, and the JDK itself, all of which must be included in the analysis. However, while all of the additional code must be included in the analysis scope, only code which is reachable from one of the root methods will be included in call graph construction.

	App.1	App.2	App.3	App.4
App. Classes	974	135	730	7874
App. Classes Analyzed	730	48	694	1942
Synthesized	15	4	33	111
Framework Classes	1767	855	869	1762
Frameworks Classes Analyzed	87	31	66	63
JDK Classes	6432	5722	5620	7215
JDK Classes Analyzed	437	403	419	244
Total Classes	9188	6716	7252	16962
Total Classes Analyzed	1254	482	1179	2249

Table 2

Table 2 shows the overall number of classes in application, J2EE framework or JDK code for the four applications we analyzed, and contrasts it with the number of classes that were included in the call graph construction. Synthesized classes reflect classes that were missing from the scope of the analysis due to missing byte code.

The number of classes and lines of code in an application provide only an approximate measure for judging the scalability of an analysis. Specific coding patterns within an application can have a large impact on the performance and scalability as well. Table 3 provides a more precise measure of scalability. The size of the *CG* is estimated by showing the number of call graph nodes and edges. The size of the *PTG* is estimated by the number of object nodes, and the number of object field nodes. The scalability of the algorithm is affected by the number of root fields and the maximum nesting depth of objects in these applications. By maximum nesting depth, we mean the depth at which the bad store can occur within an object's nested structure, where a store to *X.Y.Z* has a nesting depth of 3. As shown in Table 3, objects in J2EE applications can have a nesting depth of up to 78, which highlights how difficult it is to find bad stores without automation.

	App. 1	App.2	App.3	App.4
CG Nodes	51584	7043	29756	56552
CG Edges	161480	15438	203214	252090
PTG roots	696	578	695	3917
PTG object nodes	7415	1855	5085	10058
PTG object field nodes	27234	5583	14611	56349
Max. Nesting Depth	21	10	9	78

Table 3

Results. Bad stores involving non-serializable objects were found in three of the four applications analyzed, and race conditions were found in all four applications analyzed. Table 4 shows a summary of the problems found and the false positive rate for each of the applications analyzed.

To validate false positives we presented the results of each problem to the code owner. If they agreed that the problem needed to be fixed, the problem is reported as a "real" problem. Otherwise it is reported as a false positive. The four applications analyzed resulted in false positives due to:

(1) inadequate analysis of synchronization for detecting races conditions,
(2) a lack of context sensitivity to detect instances of the *container pattern,*
(3) application usage knowledge not available to static analysis

Table 5 breaks down the causes of the false positives found. Of the 55 false positives due to lack of context sensitivity, adding selective context sensitivity for one additional class, *java.math.BigDecimal,* would eliminate 53. The 4 false positives due to synchronization approximation can be eliminated with better analysis that models synchronization monitors, added as a user-supplied callback filter. For the two remaining false positives from application usage knowledge, one resulted from the use of a *Hashtable*, which is thread safe, keyed by thread id where each thread only accessed entries corresponding to its thread id. Another false positive which SABER could not remove was due to the fact that one *Servlet* in the application was only used by a single administrator so the developers were not concerned in ensuring that it was thread safe.

Bad Store Results	App. 1	App. 2	App. 3	App. 4
Non-Serializable in HttpSession Total	0	1	78	18
Non-Serializable in HttpSession False Positives	0	0	0	0
Non-Serializable in HttpSession False Positive %	0%	0%	0%	0%
Race Total	34	12	15	127
Race False Positives	8	1	2	51
Race False Positive %	24%	8%	13%	40%

Table 4

Different problems found with the bad store template can have differing importance to developers. For storing non-serializable objects into an HttpSession, problems found are roughly of equal significance. However, with the race detection analysis, the likelihood of a race occurring combined with its potential impact determines the importance of the problem reported. Table 6 shows a breakdown of the types of race conditions detected in the four applications.

Race Condition False Positive Causes	App. 1	App. 2	App. 3	App. 4
Lack of context sensitivity	3	0	2	50
Synchronization approximation	4	0	0	1
Application usage knowledge	1	1	0	0
Totals	8	1	2	51

Table 5

Race conditions due to problems with initialization such as lazy initialization are typically of the least importance because they are typically found at system startup and do not co-mingle data from different users. One particular style of initialization race conditions called double check locking (DCL) [3] is less likely to occur so they have even less importance than general initialization race condition problems. The remaining causes of race conditions are classified based on whether any synchronization was present or if synchronization was present but was problematic. In race conditions that are not in initialization code, 80% of them occurred in code with no synchronization at all.

Race Condition Cause	App. 1	App. 2	App. 3	App. 4	Total
Initialization	5	5	11	36	57
Initialization (DCL)	15	1	0	0	16
No Synch.	4	5	2	32	43
Incorrect Synch.	2	0	0	8	10
Total	26	11	13	76	126

Table 6

6. Related Work

There are many program static validation tools that can be classified based on underlying analysis technology. Specialized checkers such as Lint[18], ITS4 [26], JTest [22] and FindBugs [15] provide useful information about code violations, but they are limited in the types of violations they can detect because they only do syntactic analysis. Annotation checkers such as LCLint[13], Aspect[16], Extended Static Checker[9] employ program verification techniques to identify defects. Adding annotations is tedious for a programmer. Other tools, including SLAM [2], MOPS [5], and Bandera [7], perform model checking. By exploring large state spaces, model checking is very accurate in detecting problems. However, scalable model checking of programs with complex heap data structures, such as enterprise applications, is an open problem.

Finally, there are static analysis tools, such as xgcc [14][12], ESP [8], and PREfix [4]. Static analysis is conservative, and can result in too many false positives. Techniques for filtering false positives are effective, but can result in unsound analysis. Both xgcc and ESP are validators for C programs, and are designed primarily for analyzing systems code. Framework-intensive Java programs pose unique challenges for a static analysis engine, including polymorphism, long call chains, and deep object nesting. Therefore, it is unclear how these systems will perform on such programs.

The xgcc work [14][12] proposes a rules language *metal* for defining legal and illegal code patterns, with the advantage that the class of runtime problems that can be identified statically is very general. In *metal*, temporal properties specify that a program must perform an action *X* before (or after) an action *Y*, for example, "a storage location must be allocated before it is freed" or "a file which is opened must always be closed". More specifically, temporal properties are expressed as state machine transitions triggered by pattern-matched code snippets. The state machine rules are evaluated during a traversal of the static control flow graph – evaluation failure identifies a program problem.

Bad store rules, which specify structural properties, can not be comfortably expressed as temporal rules, and require a different algorithm approach. A bad store problem can span program statements which are not related by control flow. For example, each request to an e-commerce application is in effect a separate transaction, and problematic data structures can persist across multiple requests. Therefore, an algorithm based only on control flow is not effective in finding bad store errors.

Some static analysis techniques for detecting race conditions [11][25][20] rely on annotations to eliminate false positives, by identifying synchronization primitives, shared state, and which code is multi-threaded, requiring synchronization. Of these systems, RacerX [11] is the most sophisticated in terms of automating discovery of this information, thereby minimizing the need for annotation. RacerX uses domain knowledge about C systems code to improve their results. For programs using monitor synchronization, which is the Java synchronization model, detecting race conditions statically involves lockset analysis [24][6][11], which computes the set of possible locks protecting a shared variable access.

Our bad store rule for race conditions applies to a specific J2EE illegal usage pattern, and does not represent a general solution for race conditions. Rather, our bad store template is a general solution for a class of problems that require the analysis of object relationships. We are not aware of any other static validation tool that detects a broad category of bad store problems.

7. Conclusions

We have presented a generic technique that investigates the structural properties of deeply nested objects. We use this technique to detect over 200 serious defects in four commercial J2EE applications with object nesting depths of up to 78 levels. Although we present results with J2EE applications, we conjecture that our techniques can be applied successfully to other frameworks.

Acknowledgements: We thank Gary Sevitsky, as always, and Robert D. Johnson for supporting this work.

References

[1] O. Agesen. The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. ECOOP '95*, August 1995. Springer-Verlag 1995.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis, In *Proc. of the 29th POPL*, January 2002, 1-3.

[3] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J. Maessen, J.D. Mitchell, K. Nilsen, W. Pugh, E.G. Sirer. The "Double check locking is broken". http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30 (7), June 2000, 775-802.

[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software, In *Proc. of the Ninth ACM Conference on Computer and Communications Security* (Washington, DC, November 2002), 235-244.

[6] J-D Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreded object-oriented programs. In *Proc. Of the ACM SIGPLAN 2002 PLDI,* 2002.

[7] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. Of ICSE, 2000.*

[8] M. Das, S. Lerner, M. Seigle. ESP: Path-Sensitive Program Verification in Linear Time, In *Proc. of PLDI 2002*.

[9] D. L. Detlefs. An overview of the extended static checking system. *SIGSOFT Proceedings of the First Workshop on Formal Methods in Software Practice*, January 1996, 1-9.

[10] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers, In *Proc. of PLDI*, June 1994, 242-257.

[11] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks, In *Proc. of the 19th ACM symposium on Operating systems principles*, 2003, 237-252.

[12] D. Engler, B. Chelf, A. Chou and S. Hallem, Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, In *Proc. of SOSP*, October 2000, 1-16.

[13] D. Evans. Static Detection of Dynamic Memory Errors. In *Proc. of PLDI*, May 1996, 44-53.

[14] S. Hallem, B. Chelf, Y. Xie and D. Engler, A System and Language for Building System-Specific, Static Analyses, In *Proc. of PLDI*, June 2002, 69-82.

[15] D. Hovemeyer and W. Pugh, Finding Bugs is Easy, http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf

[16] D. Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Trans. on Software Engineering and Methodology*, 4 (2), April 1995, 109-145.

[17] Java^{TM} 2 Platform, Enterprise Edition Specification, v1.4 API Specification, Sun Microsystems. 11/24/2003.

[18] S.C. Johnson. Lint, a C program checker. Unix Programmer's Manual, *4.2 Berkeley Software Distribution Supplementary Docs*; U.C. Berkeley, 1984.

[19] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java, In *Proc. of OOPSLA*, Nov 2002.

[20] K.M.Leino, G. Nelson, and J. Saxe. ESC/Java User's Manual. *Technical note 2000-002*, Compaq Systems Research Center, Oct. 2001.

[21] A. Milanova, A. Rountev, B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java, *Proc. ISSTA 2002*, July 2002.

[22] Parasoft Corporation. Automatic Java[TM] software and component testing: using Jtest to automate unit testing and coding standard enforcement, http://www.parasoft.com/jsp/products/article.jsp?articleId=839&product=Jtest.

[23] D. Reimer, K. Srinivas, H. Srinivasan, RD Johnson, and L. Koved. SABER: Smart Analysis Based Error Reduction. *IBM Research Report*, 2004.

[24] S. Savage, M. Burrrows, G. Nelson, P. Sobalvarro and T. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4): 391-411. 1997.

[25] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Proceedings*, Winter Technical Conference, 1993, 97-106.

[26] J. Viega, J. T. Bloch, T. Kohno, G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code, In *Proc. of the 16th Annual Computer Security Applications Conference*, December 2000, 257-269.