Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room

Conference Paper · June 2020

DOI: 10.1145/3385412.3386026

6 authors, including:

Anastasios Antoniadis
National and Kapodistrian University of Athens
3 PUBLICATIONS **75** CITATIONS
SEE PROFILE

Yannis Smaragdakis
National and Kapodistrian University of Athens
177 PUBLICATIONS **7,237** CITATIONS
SEE PROFILE

Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room

Anastasios Antoniadis
University of Athens
Greece
anantoni@di.uoa.gr

Nikos Filippakis
CERN
Switzerland
nfil@protonmail.com

Paddy Krishnan
Oracle Labs Australia
Australia
paddy.krishnan@oracle.com

Raghavendra Ramesh
ConsenSys
Australia
raghavendra.ramesh@consensys.net

Nicholas Allen
Oracle Labs Australia
Australia
nicholas.allen@oracle.com

Yannis Smaragdakis
University of Athens
Greece
smaragd@di.uoa.gr

Abstract

Enterprise applications are a major success domain of Java, and Java is the default setting for much modern static analysis research. It would stand to reason that high-quality static analysis of Java enterprise applications would be commonplace, but this is far from true. Major analysis frameworks feature virtually no support for enterprise applications and offer analyses that are woefully incomplete and vastly imprecise, when at all scalable.

In this work, we present two techniques for drastically enhancing the completeness and precision of static analysis for Java enterprise applications. The first technique identifies domain-specific concepts underlying all enterprise application frameworks, captures them in an extensible, declarative form, and achieves modeling of components and entry points in a largely framework-independent way. The second technique offers precision and scalability via a sound-modulo-analysis modeling of standard data structures.

In realistic enterprise applications (an order of magnitude larger than prior benchmarks in the literature) our techniques achieve high degrees of completeness (on average more than 4x higher than conventional techniques) and speedups of about 6x compared to the most precise conventional analysis, with higher precision on multiple metrics. The result is JackEE, an enterprise analysis framework that can offer precise, high-completeness static modeling of realistic enterprise applications.

CCS Concepts: • **Software and its engineering** → **Compilers**; **General programming languages**; • **Theory of computation** → **Program analysis**.

Keywords: static analysis, points-to analysis, Java EE

ACM Reference Format:
Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3386026

1 Introduction

Throughout the 20-plus years since its introduction, Java has been one of the world's most popular programming languages—currently the outright #1 per the TIOBE index [3]. In the desktop and server world, *web* or *enterprise* applications[1] are by far the primary success domain of Java. The majority of server-side software is employing one of many Java-based technologies for its business logic, for interfacing with databases, and for responding to front-end requests, all in a distributed, elastic-capacity setting.

Similarly, Java has been a dominant platform for language research. Researchers have sought to capitalize on a language with high real-world relevance, yet offering ease of experimentation via a standardized bytecode and (at least at first) manageable front-end syntax. In static program analysis research, in particular, Java has been the setting of probably the best-known research frameworks, such as Soot [33], WALA [11, 29], or Doop [8].

[1]Established terminology often makes a distinction between *web applications*, running inside a web server (and using technologies such as servlets or JSP), and *enterprise applications*, running inside an *application server* or *enterprise container* (such as WebSphere, JBoss, WebLogic, and using technologies such as EJBs and JMS). Both kinds of technologies were introduced by Java Platform, Enterprise Edition [2, 20]. In our context, we use the terms "web application" and "enterprise application" interchangeably, to encompass both kinds of technologies, covering the entire scope of Java EE.

Although enterprise applications are such a major use domain of Java, researchers might scoff at their mention: enterprise applications seem like a messy collection of disparate technologies, tuned to the needs of the software practices of-the-day. Yet one can only wonder if such derision isn't simply sour grapes: enterprise applications represent a major failure of applying programming languages research to the real world—a black eye of the research community, and especially of the static analysis community. Essentially none of the published algorithms or successful research frameworks for program analysis achieve acceptable results for enterprise applications on the main quality axes of static analysis research: *completeness*, *precision*, and *scalability*. Running Soot, WALA, or Doop out of the box on a realistic Java enterprise application yields virtually zero coverage of the application code, or fails to scale.

The literature contains very few techniques that specifically target enterprise applications. The best known research is IBM'S TAJ [32]. TAJ leverages points-to analysis, along with JSP, Enterprise Java Beans, and framework modeling to perform high-precision taint analysis. TAJ is the only past work to target web applications close to industry-level-size. However, it is specifically geared towards taint analysis (not modeling of all value-flow in the program), followup work [28] claims that its modeling is incomplete, and it is evaluated on web applications an order of magnitude smaller than our benchmarks. Other well-known research efforts, IBM's F4F [28] and Andromeda [31], also target a specific client analysis (taint analysis). Even so, both F4F and Andromeda are evaluated on systems with just a few hundred classes and computed call-graph sizes in the low thousands—metrics that are two orders of magnitude lower than current realistic enterprise applications.

The technical challenge of enterprise applications for static analysis is their complexity, in terms of both size and dynamic patterns. Enterprise applications heavily leverage frameworks in their development. These frameworks are designed with an emphasis on abstraction: the framework aims to be as general as possible, applying to all different applications. To achieve this goal, frameworks commonly employ dynamic techniques, such as *dependency injection* [1, 14]. Customization of the framework to the specific application logic is performed via annotations, side-documents in XML, and various other techniques that commonly resist static analysis and directly hurt analysis *completeness*.

Furthermore, frameworks aim for transparent performance in a distributed setting. The primary technique for performance is *highly-generic, object-agnostic caching*. Caching of objects in a large number and variety is ubiquitous in various levels of enterprise application frameworks. For instance, the *identity map* pattern is a foundational one in Fowler's "Patterns of Enterprise Application Architecture" [13] for caching database-related objects. At the other end, the Spring framework [16] introduces aggressive caching of view objects, for front-end purposes. However, central caches of highly heterogeneous objects are hostile to static analysis, in terms of both *precision* and *scalability*.

In this work we present JackEE, a static analysis framework for enterprise applications. JackEE shows that scalable, high-completeness, high-precision static analysis of realistic enterprise applications is possible. The work essentially breaks open the domain of enterprise applications, so that rich further research can be conducted and evaluated. JackEE presents several elements of technical value and novelty:

- We introduce techniques and general concepts for identifying and modeling enterprise application entry points. JackEE defines a vocabulary that allows expressing (with small per-framework effort) the behavior of enterprise frameworks, both current and to come (Sections 2 and 3).
- To address precision and scalability challenges, we introduce a *sound-modulo-analysis* model of Java data structures (esp. variants of `HashMap` and `ConcurrentHashMap`) that makes the analysis of enterprise applications more scalable and precise. Unlike other modeling/mocking efforts that throw away relevant information, our sound-modulo-analysis replacements maintain the *full* behavior of the data structure as far as the static analysis is concerned. For instance, this entails modeling all possible aliasing performed and all exceptions possibly thrown by data structure operations. Several specific technical insights are uncovered in the course of this modeling—e.g., we discuss how the implementation of the `TreeNode` class of the standard `java.util.HashMap` is hostile to the very same kind of context-sensitivity (2-object sensitivity) that makes analysis of the rest of `java.util` very precise.

  These techniques interact beneficially with the central data structures (e.g., caches) in enterprise frameworks, resulting in analyses with substantial speedup (up to 15.5x) *and* higher precision (Section 4).
- We evaluate JackEE over realistic benchmarks. We define a benchmark suite containing applications suggested by experts (e.g., two large enterprise applications proposed by Oracle researchers as representative of realistic software), top-popularity representatives of major classes of enterprise applications (CMS, Version Control repository hosting, e-shop/e-commerce and blogging software), as well as smaller benchmarks from prior literature. This also offers a strong suite for future evaluations of static analyses over enterprise applications (Section 5). (All our software and all but one of our benchmarks are open-source, the remaining benchmark is free-binary-only.)

2 Web Application Background

We next give a highly condensed, non-exhaustive description of today's Java enterprise application scene and the most popular web application frameworks, with an eye on how to model application behavior statically. There are three elements we will be examining in the technologies sampled:

- **What**: key concepts, entities and kinds of data/objects exchanged between units of application functionality;
- **Where**: entry points and interconnection points for the application functionality;
- **How**: ways (e.g., subtyping, XML attributes, or Java annotations) used to describe the "what" or the "where".

2.1 Java EE Servlets

Java Platform, Enterprise Edition (Java EE, formerly J2EE) is the standard behind Java web applications and frameworks. The core element of Java EE are Java Servlets and the most fundamental way to build a web application is using the Servlet API. A servlet is a class that accepts requests, handles them, and then sends back a response to the client. Servlets are controlled by another Java application, the servlet container (e.g., Tomcat, Jetty, GlassFish, WildFly).

Generic Servlets. A generic servlet extends the abstract `javax.servlet.GenericServlet` class, overriding appropriate methods. The class provides lifecycle methods, such as `init` and `destroy`, and the `service` method that is called by the servlet container to respond to a request. The `service` method accepts a `ServletRequest` object which contains the request by the client and a `ServletResponse` object that will be used to respond to the client.

Http Servlets. HTTP servlets extend the abstract `HttpServlet` class and need to override methods relating to HTTP requests, such as `doGet`, `doPost`, `doPut`, etc.

Servlet Filters. A servlet filter is an object that can intercept HTTP requests targeting a web application. It supports lifecycle methods (`init`, `destroy`) and the main functionality method `doFilter`, invoked by the web container every time a resource needs to be filtered. `ServletRequest` and `ServletResponse` objects are passed are parameters.

Summary:

- **What**: different kinds of servlets, servlet request/response objects, filters.
- **Where**: lifecycle methods (`init`, `destroy`), `service`, `doFilter`, `doGet`, `doPost`, etc.
- **How**: subtyping abstract classes.

2.2 Enterprise Java Beans

Enterprise Java Beans (EJB) are another major part of the Java EE specification. An EJB is a server-side software element that captures the business logic of an application, including its communication with the front end and its storage component (i.e., interfacing with a database). EJBs are deployed in an EJB container, typically inside an application server, such as JBoss, WebLogic, or WebSphere.

EJBs follow a tedious array of conventions, regarding serialization, construction, access to fields. These conventions can be largely expressed as subtyping of interface and abstract classes and following strict naming conventions. However this is an outdated, low-level view, rarely visible in the application source (or static bytecode) nowadays. The modern way to satisfy these conventions is through *dependency injection*: using Java annotations or XML configuration files to guide a meta-programming layer, which dynamically generates glue code and transforms existing code (by way of *method*, *field*, or *constructor* injection), in order to provide the necessary interfacing with the protocol expected by the EJB container. Java provides a standard API specification for the dependency injection mechanism, which is implemented by web frameworks such as Spring (discussed later).

There are currently two main types of EJBs: *session* and *message-driven* beans.

Session Beans. A session bean encapsulates business logic that can be invoked programmatically by a client. The lifecycle of a session bean instance is managed by the EJB container. For purposes of distributed replication, session beans are classified as *stateless*, *stateful*, or *singletons*, typically distinguished by the Java annotations `@Stateless`, `@Stateful`, and `@Singleton` in the code.

Using a session bean in client code is done by attaching the `@EJB` annotation on the client's field referencing the bean.

Message-Driven Beans. A message-driven bean allows the asynchronous processing of messages. This type of bean normally acts as a Java Message Service listener that receives JMS messages. Message-driven beans are annotated with the `@MessageDriven` annotation. Message-driven beans are not used for dependency injection but their methods act as web application entry points.

Summary:

- **What**: different kinds of beans, bean client classes.
- **Where**: lifecycle methods, methods of beans.
- **How**: Java annotations or XML configuration.

2.3 Spring

Spring is by far the most popular Java web application framework. It builds over the Java EE protocols, providing significant reusable functionality and automation. To use a systems analogy, if Java EE were a Unix OS, Spring would be the most common implementation of C and libc.

Spring is primarily a Web model-view-controller (MVC) framework, designed around a `DispatcherServlet` that dispatches requests to handlers. The default handler is based on the `@Controller` and `@RequestMapping` annotations.

Controllers. The core element of a Spring web application is the Controller. Controllers accept incoming requests, process the payload of the request, send the data to a Model for further processing and then return the processed data to the View for rendering. The `DispatcherServlet` plays the role of the Front Controller in the architecture sending requests to controllers that handle incoming requests at the application level.

Spring MVC Interceptors. Spring MVC interceptors are analogous to Servlet Filters, allowing chaining of request and/or response processors. An interceptor is defined via subtyping (e.g., implements the `HandleInterceptor` interface) and provides methods `preHandle`, `postHandle`, and `afterCompletion`.

Spring Authentication Managers and Providers. Spring Security provides security features such as authentication and authorization to secure Java Enterprise Applications. The main interface for authentication is `AuthenticationManager` which only has one method, `authenticate`, accepting (and returning) an `Authentication` object. `AuthenticationManager` is implemented by `ProviderManager`, a class that controls one or more `AuthenticationProvider` implementations. Spring both provides default `AuthenticationProvider` implementations and supports the creation of custom `AuthenticationProvider` implementations as beans.

Spring Beans. Spring supports constructor, field and method injection using XML configurations as well as Java annotations, such as `@Autowired` or `@Inject`.

Summary:

- **What**: controllers, interceptors, authentication managers, providers, and beans.
- **Where**: methods of controllers and of other handler classes, `preHandle`, `postHandle`, `authenticate`, etc.
- **How**: Java annotations or XML configuration, subtyping relationships.

2.4 Other Technologies

There is a vast array of other technologies in the enterprise application area, as well as a lot more features in the technologies discussed. However, the principles as far as modeling entry points, generated objects, and induced inter-connectivity are similar, with a different concrete instantiation. To illustrate, we consider two examples.

Apache Struts 2. Apache Struts 2 is another popular open-source web application framework. Although it has several differences from Spring, they are both based on an MVC architecture and allow dependency injection and component interconnection through XML configuration files, or Java annotations. As a representative example, Struts 2 handles requests using classes that implement `com.opensymphony.xwork2.Action` (or extend an `ActionSupport` class). On these classes, the `execute` method needs to be overridden to provide custom behavior. This method can be annotated with `@Action` and `@Result`.

JAX-RS-based REST. REST is a software architecture style for Web services. In RESTful Web Services the web services are resources and can be identified by their URIs. REST client applications can use HTTP GET/POST methods to invoke services. JAX-RS is a specification that provides support for the creation of RESTful web services.

JAX-RS uses annotations to indicate the mapping of a resource class to a web resource. Annotations include `@Path` (the relative path for a resource), `@GET`/`@PUT`/`@POST` (used on methods that implement a corresponding HTTP request), `@QueryParam`/`@HeaderParam` (to bind a method argument to information from an HTTP request).

3 Analysis Completeness

Popular static analysis frameworks for Java [8, 11, 33] provide no support for the lifecycle or injected semantics of enterprise applications. Instead, analyses expect their users to provide customization for web applications, which is a significant burden, virtually never overcome in practice. The very same analysis frameworks, however, provide special-purpose support for modeling, e.g., the Android system lifecycle [4, 15], with its numerous entry points determined by subtyping conventions and XML configuration files.

This raises the question: is the domain of enterprise applications so much more complex and tedious than other domains, or could a flexible but relatively straightforward modeling, with modest customization, suffice for a large variety of realistic enterprise applications?

3.1 Overview

JackEE's modeling of enterprise applications attempts such a general-yet-customizable modeling of frameworks. We provide a vocabulary, based on well-recognized enterprise application concepts and mechanisms, for capturing the elements of application behavior that a static analysis would normally miss. Such elements include objects with special roles, methods with implicit semantics (especially entry points), injected fields or methods invisible in the code, relationships between objects that are not visible in the code, etc. Customization based on this vocabulary is done on a *per-application-framework*, and not per-application basis: each application framework's modeling captures its conventions for entry points, object inter-relationships, injection hooks and resulting code, etc.

Conceptually, JackEE is similar to well-known work in the literature on static analysis of enterprise applications: IBM's F4F [28] (framework for frameworks), which seeks to offer a way to encode framework behavior in a generic way. There is no publicly available version of F4F but the published description suggests a mechanism with significant differences.[2]

[2]We contacted the first author of F4F who informed us that the last remnants of public entry point logic in the open-source IBM WALA framework were removed in 2012 [27] because of bitrot. Although we cannot directly investigate the differences with F4F, it is worth noting that its published results evaluate it on systems with a maximum of 790 classes and computed call-graph size of at most 5,300 edges. In our own evaluation (Section 5)

Inputs:
`Class_Annotation`(*c : ClassType*)
`Method_Annotation`(*m : Method*)
`Field_Annotation`(*f : Field*)

`XMLNode`(*f : file, nodeId : int, parentId : int, namespaceURI : symbol, name : symbol*)
`XMLNodeAttr`(*f : file, nodeId : int, index : symbol, name : symbol, value : symbol*)

Outputs:
`Servlet`(*c : ClassType*)
`Controller`(*c : ClassType*)
`RESTResource`(*c : ClassType*)
`Interceptor`(*c : ClassType*)
`Bean`(*c : ClassType*)
`BeanFieldInjection`(*c: ClassType, f : Field, o : Value*)

`GeneratedObject`(*o : Value, c : ClassType*)
`EntryPointClass`(*c : ClassType*)
`ExercisedEntryPoint`(*m : Method*)

Figure 1. Base relations, over which a framework is modeled.

Foremostly, F4F is a framework for modeling frameworks by writing analysis-based *generators*. Each "model" of an application framework in F4F is a generator (in Java), based on a WALA program analysis and other accompanying information (e.g., XML files) of the analyzed application, that will produce a specification of the joint application's and framework's behavior (in a standardized language, WAFL). Writing such a generator is far from a trivial task, even with support from the F4F framework and with a standardized output language.

Instead, JackEE leverages the context of the Doop[3] analysis framework [8] to offer high-level, rule-based specifications of web application framework semantics. We chose Doop because, among the main Java analysis frameworks, it is distinguished by its use of declarative specifications, in the Datalog language. Modeling a new framework in this setting is an easy task, once the main, framework-agnostic vocabulary has been defined. Our model of a new enterprise framework is a collection of logic rules, most of which appeal to simple configuration predicates.

3.2 Vocabulary

Figure 1 shows a sample of key concepts of our framework-modeling vocabulary, pertaining both to program text and configuration files (e.g., `Class_Annotation`, `XMLNode`) and to the program semantics (e.g., `Interceptor`, `Servlet`) or to the analysis semantics (e.g., `GeneratedObject`, `EntryPointClass`). The former are framework-specification inputs, while the latter are outputs and directly inform the analysis.

these metrics are larger by some-two orders of magnitude—e.g., over 9,000 classes in the application as well as another 28,000 in libraries, yielding 1.1M call-graph edges for alfresco.

[3]Publicly available at https://bitbucket.org/yanniss/doop .

Based on this vocabulary, the semantics of a framework can be described in terms closely following the scheme of **what**, **where**, and **how** of the previous section. It is easy, for instance, to express as rules (for a given framework):

- that a method with a certain annotation is an entry point that should be exercised with a specific type;
- that a class named in a certain XML node is an interceptor;
- that a class with a given annotation should have an object generated for it.

The per-framework rules can directly appeal to program structure (and even analysis-level) concepts as defined in the Doop framework. Furthermore, the rules vocabulary leverages some pre-defined, framework-agnostic functionality.

3.3 Framework-Independent Support

The "output" concepts and entities of Figure 1 abstract away from the specifics of each framework, in order to allow us to pre-define common functionality. When a rule produces analysis-level concepts (e.g., `EntryPointClass`) or enterprise-domain concepts (e.g., `Controller`), it triggers a lot of further inferences. Most of them concern the derivation of entry points, as well as a policy for how to exercise them inside the analysis (i.e., with abstract objects of what type and with what interconnectivity).

Once entry points are identified, the framework-independent policy for creating abstract objects (subsequently called *mock* objects) for use at entry points can be described by the following rules:

- Given an entry point method m and its declaring type C, we create a receiver mock object of type C for the entry point method. The static analysis considers the receiver variable (`this`) to point to the mock object.
- Given an entry point method m and its argument with index i and type T:
  - If T has concrete subtypes in the application, we create a mock object for each subtype of T in the application
  - If T does not have subtypes in the application, we find all the casts in the entry-point method that are to subtypes S of T. We create one mock object for each S.
  - We follow the rule of one mock object per-type to ensure that the analysis will remain scalable regardless of the number of entry points.
  - The static analysis considers argument i to point to the above type-compatible mock objects.
- For each mock object type T we mark the constructors of T as entry points to ensure that the mock object of T has acquired the required state for the points-to analysis to fully analyze the entry point method and its callees. The constructor arguments are analyzed under the same mocking policy, recursively, i.e., they get assigned mock objects of compatible types.

This reusable strategy can be customized per-analysis (e.g., to create multiple, distinguished mock objects).

`ConcreteApplicationClass`(*c : ClassType*)	*c* is a non-abstract class in application code
`Method_DeclaringType`(*m : Method, c : ClassType*)	maps method to its declaring class
`Field_DeclaringType`(*f : Field, c : ClassType*)	maps field to its declaring class
`Field_Name`(*f : Field, s : symbol*)	maps field to its name
`Var_Type`(*v : Variable, t : Type*)	static type of variable
`Value_Type`(*o : Value, t : ClassType*)	dynamic type of (abstract) object
`Bean_Id`(*o : ClassType, t : string*)	maps a bean class to its id
`Subtype`(*t1 : Type, t2 : Type*)	*t1* is a subtype of *t2*
`FormalParam`(*i : number, m : Method, v : Variable*)	*i*-th formal argument of method
`FormalParam`(*i : number, mi : MethodInvocation, v : Variable*)	*i*-th actual argument of invocation
`AssignReturnValue`(*mi : MethodInvocation, v : Variable*)	return value of invocation assigned to local variable
`GetBeanInvocation`(*mi : MethodInvocation*)	invocation of Spring method `getBean()`

Figure 2. Program-related information, for use in framework specifications (and further analysis).

3.4 Modeling Examples

We next present the modeling of different aspects of enterprise frameworks via examples, using the vocabulary of Figure 1, as well as program-related concepts. We give a brief description of the latter in Figure 2.

3.4.1 Java Servlet API. For a web application implementing the Java Servlet API to be analyzed properly, we need to identify all the concrete subtypes of `javax.servlet.GenericServlet` in the application. `GenericServlet` is at the top of the Servlet class hierarchy so any application request-handling class needs to extend it.

```
Servlet(class) :-
  ConcreteApplicationClass(class),
  SubtypeOf(class, "javax.servlet.GenericServlet").
```

Every servlet class is automatically an entry point class (per framework-independent rules). However, another rule extends the modeling of the Java servlet API specifically: a method of *any* class that accepts a `ServletRequest` or `ServletResponse` parameter is considered an entry point (and is appropriately mocked).

```
ExercisedEntryPointMethod(method) :-
  ConcreteApplicationClass(class),
  Method_DeclaringType(method, class),
  FormalParam(_, method, param),
  Var_Type(param, paramType),
  (SubtypeOf(paramType, "javax.servlet.ServletRequest");
   SubtypeOf(paramType, "javax.servlet.ServletResponse")).
```

Filters are objects that perform filtering tasks to a request to a resource and/or to a response from a resource. Custom filters are implemented by extending `javax.servlet.Filter` and overriding the `doFilterMethod`. Filter classes are also generic entry points.

```
EntryPointClass(class) :-
  ConcreteApplicationClass(class),
  SubtypeOf(class, "javax.Servlet.Filter").
```

3.4.2 Java RESTful Web Services API Entry. The following framework-modeling rule marks application methods with http-request annotations from `javax.ws.rs.*` as entry points that need to be exercised, and the declaring class of every such method as a `RESTResource` class which is an `EntryPointClass`. This logic can be easily extended to support other REST API specifications and implementations.

```
EntryPointClass(class),
RESTResource(class),
ExercisedEntryPointMethod(method) :-
  ConcreteApplicationClass(class),
  Method_DeclaringType(method, class),
  (Method_Annotation(method, "javax.ws.rs.POST");
   Method_Annotation(method, "javax.ws.rs.PUT");
   Method_Annotation(method, "javax.ws.rs.GET");
   Method_Annotation(method, "javax.ws.rs.HEAD");
   Method_Annotation(method, "javax.ws.rs.DELETE")).
```

3.4.3 Spring MVC Entry Points. Each Spring application defines its own application-level controllers that handle requests, annotated with the `@Controller` annotation. The `@RequestMapping` annotation is used to bind the annotated controller class or method to a specific URL. The following framework-modeling rules mark all application non-abstract classes annotated with `@Controller` as entry points and any individual methods annotated with `@RequestMapping` as entry points to be mocked. We abridge package names in the following examples. The symbol @ denotes annotation interfaces.

```
Controller(class),
EntryPointClass(class) :-
  ConcreteApplicationClass(class),
  Class_Annotation(class, "org.spring...@Controller").

Controller(class),
ExercisedEntryPointMethod(method) :-
  ConcreteApplicationClass(class),
  Method_DeclaringType(method, class),
  Method_Annotation(method, "org.spring...@RequestMapping").
```

Application classes implementing the `HandleInterceptor` interface or extending the `HandleInterceptorAdapter` class are custom Spring interceptors. The framework model marks these classes as entry points and mocks the receivers and arguments of their methods.

```
EntryPointClass(class),
Interceptor(class) :-
  ConcreteApplicationClass(class),
  (Subtype(class, "org.spring...HandleInterceptorAdapter");
   Subtype(class, "org.spring...HandlerInterceptor")).
```

Spring security offers a large array of options for performing authentication in a web application. These options conform to the pattern that an authentication request is processed by an authentication provider. Custom authentication providers may use external or developer-defined authentication services. Such authentication providers with custom authentication services are registered using an XML configuration—for instance:

```
<authentication-manager>
    <authentication-provider
      ref="customAuthenticationProvider" />
</authentication-manager>
```

The `<authentication-manager>` tag instructs Spring to create a new `authentication.ProviderManager` object to register authentication providers to it. Spring then uses the bean id found in the `<authentication-provider>` tag to create an object of the specified custom authentication provider and registers it to the `ProviderManager` object. The rule below identifies custom authentication providers defined in the XML configuration of web application.

```
Interceptor(authProvider) :-
  XMLNode(XMLfil, parntId, _, _, "authentication-manager"),
  XMLNode(XMLfil, nodeId, parntId, _, "authentication-provider"),
  XMLNodeAttr(XMLFile, nodeId, _, _, providerId),
  Bean_Id(authProvider, providerId).
```

3.5 Wiring Together Beans

The framework specification rules we have seen so far capture configuration definitions that conceptually *initialize* a static analysis. However, a framework's logic may need to interact deeply with an analysis: to establish connections between objects and also to query the analysis for its inferences. The best examples of such recursive framework specifications can be found in rules for setting up beans. These appeal directly to the main two relations defined in the static analysis: `ObjectFieldPointsTo`(*o1 : Value, f : Field, o2 : Value*) and `VarPointsTo`(*v : Variable, o : Value*)—capturing what abstract values an object's field or a program variable may have, respectively.

The modeling rules in the rest of this section merge some elements that are framework-specific with implicit, framework-independent logic, for a unified presentation.

Declaring and Setting Up Beans. Web applications rely heavily on beans to take advantage of reusable code. Therefore the bean configuration of a web application is also critical to the completeness of a static analysis. In fact, most of the mechanisms already described as entry points in the previous section rely on beans. For instance, when declaring a custom `AuthenticationProvider` in Spring, that custom authentication provider implementation needs to be registered as a bean so that the Spring framework will know it is responsible for creating it and injecting its dependencies. Either at entry-point level or deeper in the application, web applications rely on the creation and lifecycle management of beans by the web framework to have the bean objects created and provided to them in a complete state for processing.

A static analysis oblivious to the side-effects of the execution of the web framework would encounter references pointing to `null` where a bean object needs to be injected by the framework. As a result, generating the full bean objects along with their transitive dependencies is important so that the analysis can fully analyze bean code.

The most common dependency injection patterns are constructor and field injection, followed by the less common method injection. In even rarer cases it is possible for a field to depend on a collection of beans, such as a `List` or `Map`.

The first task of JackEE is the identification of classes in the application that are declared as beans. There are framework-independent ways to declare a bean—e.g., XML configuration—or framework-dependent ways, such as annotations. JackEE provides rules to identify both.

```
Bean(type) :-
  (Class_Annotation(type, "javax.ejb.@Stateful");
   Class_Annotation(type, "javax.ejb.@Stateless");
   Class_Annotation(type, "javax.ejb.@Singleton");
   Class_Annotation(type, "org.spring...@Component");
   Class_Annotation(type, "org.spring...@Service");
   Class_Annotation(type, "org.spring...@Repository");
   Class_Annotation(type, "org.spring...@Controller")).
```

The above rule registers classes annotated with EJB bean annotations, i.e., `@Stateless`, `@Stateful`, `@Singleton`, or Spring bean annotations. (`@Component` defines that a class is a bean and `@Controller`, `@Service` and `@Repository` are its subtypes.) We omit the bean id creation by the framework which fills the `Bean_Id` relation. A bean can both receive a dependency injection and be injected into another bean.

There are several XML patterns that declare a field injection. We demonstrate one such pattern below.

```
<bean class="targetClass">
      <property name="targetField" ref="beanId"
        />
</bean>

<bean id="beanId" class="beanClass"><bean>
```

The next rule identifies the above pattern and infers a `BeanFieldInjection`, the dependency injection of a bean object *beanObject* into the field *targetField* of another bean *targetClass*.

```
BeanFieldInjection(targetClass, targetField, beanObject) :-
  XMLNode(XMLFile, parentNodeId, _, _, _),
  XMLNodeAttr(XMLFile, parentNodeId, _, "id", _),
  XMLNodeAttr(XMLFile, parentNodeId, _, "class", targetClass),
  XMLNode(XMLFile, nodeId, parentNodeId, _, "property"),
  XMLNodeAttr(XMLFile, nodeId, _, "name", fieldName),
  XMLNodeAttr(XMLFile, nodeId, _, "bean", beanId),
  Field_DeclaringType(targetField, targetClass),
  Field_Name(targetField, fieldName),
  Bean_Id(beanClass, beanId),
  GeneratedObject(beanObject, beanClass).
```

The BeanFieldInjection inference leads to a core static analysis inference that models the dependency injection: an ObjectFieldPointsTo relationship, making the *targetField* of the bean object of type *targetClass* point to *beanObject*.

```
ObjectFieldPointsTo(targetObject, targetField, beanObject) :-
  Value_Type(targetObject, type),
  Field_DeclaringType(targetField, type),
  BeanFieldInjection(targetClass, targetField, beanObject).
```

Similar modeling can be performed with framework-specific techniques for declaring field injection. For instance, injected Spring Beans are identified with the @Autowired annotation on a field, whereas the @Inject annotation is used for the Java extension package version described in the JSR-330[21] specification. The latter modeling is shown as an example:

```
BeanFieldInjection(targetClass, targetField, beanObject) :-
  Field_DeclaringType(targetField, targetClass),
  Field_Annotation(targetField, "@Inject"),
  Bean_Id(beanClass, targetField),
  GeneratedObject(beanObject, beanClass).
```

Getting Beans Programmatically. Finally, beans can be retrieved programmatically instead of based on static elements, such as annotations or XML specifications. This kind of modeling consumes information from the static analysis and produces information for the analysis. The logic can be framework-specific—e.g., when the `T getBean(String name)` method of BeanFactory is used in Spring. An invocation of getBean returns the bean with same id as the value provide in the *name* argument of the invocation. Below we present the rule that identifies such an invocation and returns the requested bean.

```
VarPointsTo(local, beanObject) :-
  GetBeanInvocation(invocation),
  ActualParam(0, invocation, actualParam),
  VarPointsTo(actualParam, beanName),
  Bean_Id(beanClass, beanName),
  GeneratedObject(beanObject, beanClass),
  AssignReturnValue(invocation, local).
```

The above rule infers that if the first (and only) parameter of a getBean invocation points to a beanName string and if that beanName is the id of bean beanClass, then the local variable localVar, to which the invocation return value is assigned, points to beanObject: the generated object for beanClass.

4 Analysis of Web Applications: Precision and Scalability

Completeness of a static analysis, given the complex dynamic features of enterprise applications, is one side of the analysis quality story. The other side is the tradeoff of analysis scalability and precision. The issue is particularly pertinent to realistic enterprise applications due to their large size, as well as due to the very same factors (i.e., high degrees of abstraction and configurability) that hinder completeness.

Background. *Context sensitivity* in static analysis [19, 23–25] is a powerful way to achieve precision without sacrificing scalability. Under context sensitivity, a method is analyzed separately for every distinct context, where contexts are abstractions of the possible settings in dynamic executions. (A second aspect of the approach is a *context-sensitive heap*, which distinguishes allocated objects also based on abstractions of the possible dynamic environment of the allocation.) In the Java setting, *object-sensitivity* [19] has been the context sensitivity approach that constitutes the state of the art for precise analysis of realistic programs. In particular, a 2-object-sensitive analysis with a context-sensitive heap (2objH) is often considered the golden standard[4] of a "precise analysis" in the literature [17, 18, 26]—we briefly postpone discussing what 2objH entails exactly. Notably, a 2objH analysis is excellent for precisely analyzing the java.util package, mainly consisting of the standard Java collections. This is one of the most central pieces of reusable functionality in the Java world: nearly every application will use pre-defined data structures from java.util. A precise treatment of it in static analysis is essential: different use-sites of distinct data structures should not be conflated for good precision. Tellingly, recent research that attempts to develop sophisticated algorithms for adaptive precision and scalability hardwires [18] the context-sensitivity flavor to 2objH for all of java.util or employs heuristics that result in all of java.util being analyzed under 2objH [17].

Problem. A striking finding of our work with enterprise applications has been that a 2objH analysis is extraordinarily heavy, due to its treatment of java.util—the very same functionality that 2objH is key for handling well. For illustration, a 2objH analysis of WebGoat (the smallest application in our experimental set, and much analyzed in past work) incurs over two-thirds (69.8%) of its cost in analyzing java.util alone. (The cost can be attributed to specific packages as a proportion of the context-sensitive variable-may-hold-value inferences made by the analysis.) This is an extraordinarily high number compared to typical Java desktop applications—e.g., for the DaCapo benchmarks, often used to evaluate static analyses, the corresponding number is typically under 20%.

[4]Even more precise analyses are, of course, desirable, if they can be made to scale, however they rarely do.

The finding has been a constant throughout our experimentation with enterprise applications of diverse backgrounds.

The reason for this analysis behavior somewhat varies per-case but is generally attributable to highly-generic, heterogeneous data structures, central to the enterprise application. An example of such a data structure is a cache, applied to a large number of objects, of many different types, and from distant parts of the application's code. Caches feature prominently in enterprise application frameworks, especially because of the distributed nature of applications and need for horizontal scaling. The *identity map* pattern [13] is well-known, regarding database-related objects. At the same time, view objects or business logic objects may also be aggressively cached. Spring, for instance, offers annotations such as `@EnableCaching` and `@Cacheable`, to trigger the caching of any bean object.

Approach. To address the issues with inefficient analysis of `java.util` data structures, we introduce a replacement implementation of key data structures, namely variants of `HashMap` (currently merely two classes: `HashMap`, `LinkedHashMap`) and `ConcurrentHashMap`. The replacement implementations follow a principle that we term *sound-modulo-analysis.*[5]

A *sound-modulo-analysis* model of a code entity C relative to a static analysis S produces a replacement code entity C' for C such that the analysis $S(C')$ is guaranteed to model all dynamic program behaviors of the original program C.

This definition allows the replacement code, C', to result in both more precise and less precise analysis (under fixed S) than analyzing the original program C, as long as all realizable behaviors of C are modeled.

In our case, the analysis S is a context-sensitive but *path-*, *flow-*, and *array-insensitive* static analysis: neither the conditions of branches, nor the order of analyzed statements, nor the exact index of an array's contents affect the static analysis result. We exploit these properties to offer scalable and precise simplified versions of data structures. Figure 3 illustrates a small part of the rewrite of `java.util.HashMap`.

The original code stores all data in a `table` field: an array of `Node` objects (line 3, left hand side). For an array-insensitive analysis, we can replace the table by a single `Node`—`contents`, on the right. The `contents` field is initialized to a node allocated at construction time. Every assignment of a `Node` object to a table position in the original code (not shown) maps to a corresponding assignment of the `contents.key` and `contents.value` fields, without allocating a new `Node`. A traversal of the table, such as that in method `foreach` (lines 6-20) results in behavior that is at least as general on the right-hand-side version, as far as the analysis is concerned. Since all data that enter the `HashMap` get collapsed into a single `Node`, iteration is unnecessary: the code can merely retrieve the node (line 13, r.h.s.) and simulate the field accesses and method calls on it (lines 14-15). A flow-insensitive static analysis will treat equivalently (and possibly only less precisely) the access to a single `contents` field that points to multiple abstract objects and the loop on the left.

As seen in the example, the sound-modulo-analysis modeling goes to great lengths to ensure capturing all of the original behaviors of the Java library. For instance, the simplified model preserves all exceptions thrown (possibly under simplified branching conditions, as seen on line 18, since the analysis is path-insensitive and does not model them).

Scalability. The sound-modulo-analysis simplification of `java.util.HashMap/LinkedHashMap`, and `java.util.concurrent.ConcurrentHashMap` results in a much lighter analysis. For instance, as can be seen in Figure 3, there is no need for the `tab` local alias of the `table` field on line 7 (which would result in an array variable pointing to a large number of abstract objects inside the analysis), nor for local variables for `tab[i]` (line 13). Removing internal complexity from the most-reused part of standard libraries reduces the burden on the static analysis.

The largest complexity-removal factor, however, is the elimination of the `TreeNode` class (nested in `HashMap`). `TreeNode`s are an alternative to regular (linked list) `Node`s in a `HashMap`. The alternative exists for performance and for generality relative to the `LinkedHashMap` specialization of `HashMap` (which we also rewrote in a sound-modulo-analysis simplification). The existence of `TreeNode`s is an optimization that is semantically irrelevant to library clients, and no other part of the Java library refers to the `TreeNode` class. However, a static analysis of the original `java.util.HashMap` code has to capture all possible behaviors and, thus, model all data inserted in any `HashMap` as possibly also stored in a `TreeNode`-based version, incurring significant overhead.

Precision. The sound-modulo-analysis modeling of `HashMap` structures in practice results in a *more* precise analysis, as far as clients of the library are concerned. The reason is that the simplification of the code allows the context sensitivity policy to keep greater precision.

One specific code pattern helps illustrate much of the precision loss in the original `HashMap`. To understand it, we need to consider the definition of 2objH—a 2-object-sensitive analysis. 2objH analyzes every call, "`base.method()`", separately for every possible context, consisting of two elements:

- the abstract receiver object, i.e., the allocation instruction of the object that `base` refers to
- the abstract receiver object of the method that performed that allocation, i.e., that contains the allocation instruction of the previous step.

This kind of context combination is typically information-rich enough to distinguish dynamic invocation conditions of "`base.method()`", keeping the data flow from different calls separate, resulting in precision. For instance, two uses of a

[5]Our modified OpenJDK 8 is publicly available at https://github.com/plast-lab/sound-modulo-analysis-openjdk-8 .

```java
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable { ...
 transient Node<K,V>[] table;

 final class KeySet extends AbstractSet<K> { ...
  public final void forEach(Consumer<..> action) {
    Node<K,V>[] tab;
    if (action == null)
      throw new NullPointerException();
    if (size > 0 && (tab = table) != null) {
      int mc = modCount;
      for (int i = 0; i < tab.length; ++i) {
        for (Node<K,V> e = tab[i];
             e != null;
             e = e.next)
          action.accept(e.key);
      }
    if (modCount != mc)
      throw new ConcurrentModificationException();
  } ...
 } ...
}
```

```java
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable { ...
 transient Node<K,V> contents;

 final class KeySet extends AbstractSet<K> { ...
  public final void forEach(Consumer<..> action) {

    if (action == null)
      throw new NullPointerException();



    Node<K,V> e = contents;
     // forall i, table[i] abstracts to "contents"
    e = e.next;
    action.accept(e.key);

    if (modCount != 42) // mere non-determinism
      throw new ConcurrentModificationException();
  } ...
 } ...
}
```

Figure 3. Example of sound-modulo-analysis modeling of `java.util.HashMap`. Original code on the left, simplified on right.

`HashMap` by a certain program class operating as an intermediary on behalf of two other classes will still be distinguished.

The original `HashMap` code in `java.util` contains several instances of a double-dispatch-like pattern (e.g., in methods `putVal`, `compute`, `merge`, `computeIfAbsent`):

`treeNode.putTreeVal(this, tab, hash, key, value).`

This code pattern significantly weakens the precision of the 2objH analysis for different application-level clients. The receiver object of the call (i.e., the abstract value of `treeNode`) is a `TreeNode` object allocated inside `HashMap`. Therefore, the second element of the `HashMap`'s context is not used in this call—one of the two context elements is dropped, in favor of a locally-allocated `TreeMap` that cannot serve to distinguish the dynamic execution conditions of the library's caller. This effectively reduces a 2objH analysis to a less precise 1objH.

The above pattern serves to remind how otherwise-powerful static analysis techniques are highly sensitive to code idioms. Careful sound-modulo-analysis rewrite of key functionality ensures that precision is kept—in this case, by the simple elimination of all `TreeNode`-based code.

5 Evaluation

We evaluate JackEE on a dataset of well-known open-source applications, selected due to recommendation or popularity. ´ We use a machine with two Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz (each with 12 cores x 2 hardware threads) and 640GB of RAM. We evaluate on three dimensions: completeness, speed and precision. JackEE extends the Doop framework, executed with the Souffle Datalog engine running on 16 threads.

Our benchmarks include:

- *alfresco*: The community edition of one of the most popular content management systems (CMS). It consists of several different libraries packaged and deployed as as single web app. Alfresco community edition is open-source, available on GitHub. It was suggested to us as a good example of a web application for analysis by Oracle researchers. Application classes: 9164. Total classes: 37163.
- *bitbucket-server*: The on-premise version of the popular web-based version-control repository hosting service. Application classes: 581. Total classes: 29984.
- *dotCMS*: Popular CMS. (GitHub, 525 stars, 354 forks.) Application classes: 5473. Total classes: 46027.
- *opencms*: Popular CMS. (GitHub, 421 stars, 342 forks.) Application classes: 2143. Total classes: 16183.
- *pybbs*: A platform for building websites. (GitHub, 910 stars, 487 forks.) Application classes: 172. Total classes: 24692.
- *shopizer*: A platform for building commercial websites. (GitHub, 1.5k stars, 1.4k forks.) Application classes: 1151. Total classes: 33841.
- *SpringBlog*: A blogging system. (GitHub, 1.5k stars, 719 forks.) Application classes: 100. Total classes: 18493.
- *WebGoat*: A web application by OWASP, designed to teach web application security lessons. (GitHub, 2.9k stars, 1.2k forks.) A very small application for the standards of our investigation, included mainly because of its use in many past publications [9, 10, 32]. Application classes: 96. Total classes: 7317.

5.1 Completeness

None of the major frameworks for Java program analysis currently offers realistic support for enterprise applications.

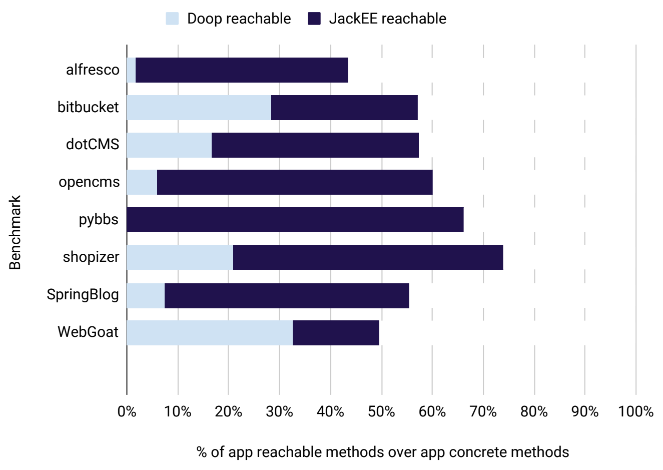

Figure 4. App methods reachability for Doop and JackEE.

WALA [11] has virtually zero analysis completeness after the removal of the J2EE part of the framework. Soot [33] does not provide any automated or semi-automated analysis for web applications, instead requiring manual definition of custom entry points. Consequently, an out-of-the-box Soot analysis provides zero completeness for web applications. The Doop framework [8], on which JackEE is built, contains a basic servlet and web-app "open programs" logic. We enable and compare to Doop, as a point of reference.

Due to the modular underlying architecture of Doop, JackEE can run with every variant of context sensitivity supported. We evaluate the completeness of JackEE's enterprise analysis with a highly-precise analysis: the mod-2objH analysis, which is a 2-object-sensitive analysis used with JackEE's sound-modulo-analysis simplified `HashMaps` and `ConcurrentHashMap`. The only exception is DotCMS, which runs out of memory at around the 15-hour mark for 2objH, so we ran JackEE's context-insensitive and mod-2objH analyses.

We compare this most precise configuration of JackEE to the least precise (context-insensitive) configuration for Doop, thus ensuring that any false completeness (due to imprecision) will count *against* JackEE. Figure 4 plots completeness, measured as the proportion of reachable (concrete) application methods. (Table 1 later shows that this metric is also backed by a quite thorough call-graph exploration.)

JackEE's analysis averages in-app reachability of 58.04%, while dropping to no less than 43.48% (for alfresco). In comparison, Doop averages 14.48% in-app coverage while dropping to approximately 1.8% and 0.0% coverage for two benchmarks (alfresco and pybbs respectively). Both alfresco and pybbs define entry points (and then further functionality) via framework-specific mechanisms—to which Doop is oblivious. Alfresco has both XML-configured entry points and a custom Spring-based REST API. The entry points of pybbs are given using Spring annotations. Both applications then use annotations for dependency injection.

To appreciate the practical meaning of exercising 58% of methods in a large application archive, we computed the same metric for Doop's analysis of standard Java desktop applications. The literature shows a large number of publications evaluating Doop analyses over the DaCapo2006[6] benchmark suite. Doop achieves an average 42.89% in-app reachability for the DaCapo2006 benchmarks, also with the aid of reports from Tamiflex [7]—a dynamic analysis tool for the resolution of reflective call graph edges.

5.2 Performance

JackEE applies the sound-modulo-analysis simplification of `java.util.HashMap`, `java.util.LinkedHashMap`, and `java.util.concurrent.ConcurrentHashMap` to the Java 8 library. Figure 5 shows both the result and the need for the optimization. Observe the (proportional, as well as absolute) cost of `java.util` skyrocket between a context-insensitive analysis and a 2objH. The JackEE replacement succeeds in decreasing 6-fold this cost for the majority of benchmarks. The average speedup is 5.9x, peaking at 15.1x for bitbucket-server. Considering the minimality of the rewrite, its impact is substantial, and shows the importance of map functionality for web applications.

5.3 Precision

To evaluate the precision of the mod-2objH analysis we compare it to the context-insensitive analysis, and the 2objH analysis with the original Java 8 library.

Table 1 presents the results over multiple precision metrics. Although no single metric is fully reliable, by taking all five metrics into account we have a reliable indicator of the overall precision of the analysis. As can be seen, the sound-modulo-analysis replacement of `HashMap` and `ConcurrentHashMap` exhibits non-negligible gains in precision (for reasons analyzed earlier). The size of the call-graph is noteworthy in most cases, demonstrating the thoroughness of the analysis, as well as the size of these applications.

6 Related Work

Past work has attempted to address the challenges of analyzing web applications by taking advantages of the idioms of Java web frameworks.

Concerto [30] addresses the problem of analyzing framework-based applications by combining concrete interpretation at the framework level and abstract interpretation at the application level and taking advantage of the framework configuration information.

To overcome the challenges of identifying entry points, Dietrich et al. [10] generate a driver for exercising a Java EE application. This processes XML configuration files, annotations and JSPs. While it handles servlets, filters, and listeners, the current level of support for frameworks is limited.

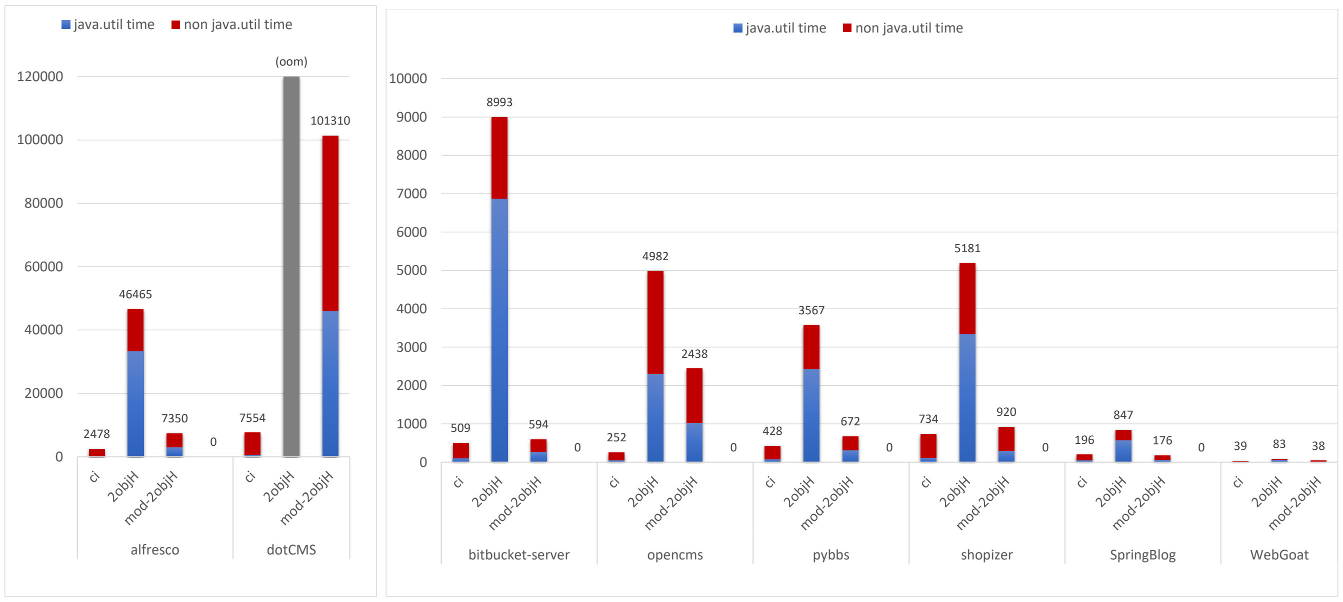

Figure 5. Analysis time for all benchmarks. Alfresco shown separately on the left, due to different scale. Time is broken up heuristically into `java.util` vs. non-`java.util` based on final cumulative sizes of points-to sets. `mod-2objH` is a `2objH` analysis using the JackEE sound-modulo-analysis simplification of `HashMaps` and `ConcurrentHashMap`.

The idea of modeling libraries instead of analyzing them, to avoid precision or scalability loss due to their complexity, has also appeared in the past, in different forms.

TAJ [32], apart from its modeling of key Java EE concepts and framework, is related to our work by also leveraging a sound-modulo-analysis simplification of data structures. However, modeling information flow through a data structure class is quite different from modeling all flows of values, as we do. This is another instance of TAJ being a targeted taint analysis, as opposed to general flow analysis of all values through a large application.

In recent work, ATLAS [5] performs this modeling by running tests on library code and producing points-to specifications which are then consumed by a client points-to analysis. ATLAS's approach is similar to JackEE's as it attempts to collapse the array fields of Collections to a single field in a manner similar to what our sound-modulo-analysis approach does for `HashMap`, `LinkedHashMap` and `ConcurrentHashMap` to achieve higher precision. However, while ATLAS uses this approach for the whole library, JackEE does it only for a critical fraction of the Collections API. This enables a focused manual rewrite, ensuring full soundness, as opposed to automated inference with empirical measurements of soundness. Furthermore, ATLAS summarizes collection behavior leading to a partial analysis of the Collections API, while JackEE leverages a full-featured points-to analysis on modified Collections. While ATLAS needs to only capture taint flows in and out of the libary, JackEE keeps up with all the value flows to and from the Java library, allowing it to be used for a larger set of client analyses. ATLAS targets Android applications—an ecosystem that exhibits some similarities to enterprise applications, but also several differences.

In work concurrent to ours, Fegade and Wimmer [12] propose an idea highly related to "sound-modulo-analysis" rewrites: "semantic models [for collections] obtained by manually simplifying data structure implementations by relying on analysis abstractions". The rewrites they propose are very similar to ours, although it is not clear whether they preserve the full semantics of the library relative to the analysis—e.g., our models maintain the full data structure behavior, including all exceptions thrown, at any level. The Fegade and Wimmer approach can afford to be unsound with respect to the call-graph because, in their setting, two analyses interact: one sound but imprecise models the call-graph; another, over semantic models but more precise, models value flow. Still, this difference is necessitated by the specifics of their domain, which includes closed-world compilation. In principle, the Fegade and Wimmer approach appears no different from our "sound-modulo-analysis" rewrite. Fegade and Wimmer evaluate on relatively small benchmarks, use call-site sensitivity, and aggressively merge abstract objects per type. We believe that the latter two techniques are a suboptimal choice for a precise and scalable analysis, at least in our setting of enterprise applications.

Special treatment of collections is common in static analysis frameworks. For instance, WALA [11] supports a special context-sensitivity policy, ZeroOneContainerCFA, which enables more precise handling of collections. (This enables

Table 1. Precision+speed metrics. In all cases *lower is better.* Dash (-) entries are for analyses that did not terminate in 28h. The 5 precision metrics shown are the average size of points-to sets—for all variables and app-only—(how many heap objects are computed to be pointed-to per-var), the number of edges in the computed call-graph, the number of virtual calls in the application whose target cannot be disambiguated by the analysis, and the number of casts in the application that cannot be statically shown safe.

Benchmark	*Metrics* / *Analyses*	*ci*	*2obj+H*	*mod-2obj+H*
alfresco	avg. objs per var	400.0	101.9	77.2
	avg. objs per app var	327.8	91.8	74.0
	edges (over ~100K meths)	1,145,133	726,751	673,132
	app poly v-calls (of ~117K)	8,193	6,008	5,962
	app may-fail casts (of ~7K)	6,445	4,307	4,176
	elapsed time (s)	2,478	46,465	7,350
bitbucket-server	avg. objects per var	178.2	49.6	28.6
	avg. objects per app var	53.0	16.1	12.0
	edges (over ~48K meths)	402,697	285,905	227,426
	app poly v-calls (of ~4.1K)	210	199	201
	app may-fail casts (of ~0.44K)	169	127	124
	elapsed time (s)	509	8,993	594
dotCMS	avg. objs per var	647.6	-	135.3
	avg. objs per app var	314.8	-	139.3
	edges (over ~122K meths)	2,250,607	-	1,146,463
	app poly v-calls (of ~93K)	12,417	-	10,647
	app may-fail casts (of ~9.8K)	9,459	-	7,859
	elapsed time (s)	7,554	-	101,310
opencms	avg. objs per var	118.3	25.9	22.4
	avg. objs per app var	99.1	24.5	22.5
	edges (over ~33K meths)	277,253	200,115	196,532
	app poly v-calls (of ~67.3K)	5,460	4,837	4,829
	app may-fail casts (of ~2.5)	2,433	1,751	1,713
	elapsed time (s)	252	4,982	2,438
pybbs	avg. objs per var	196.3	61.5	40.8
	avg. objs per app var	91.6	33.4	25.3
	edges (over ~44K meths)	397,771	304,832	277,638
	app poly v-calls (of ~2.4K)	78	66	66
	app may-fail casts (of ~8K)	70	54	55
	elapsed time (s)	428	3,567	672
shopizer	avg. objs per var	219.1	58.1	40.5
	avg. objs per app var	166.4	17.6	12.8
	edges (over ~53K meths)	465,935	333,475	313,376
	app poly v-calls (of ~25.6K)	1,390	990	983
	app may-fail casts (of ~0.68K)	703	280	267
	elapsed time (s)	734	5,181	920
SpringBlog	avg. objs per var	113.1	30.5	16.9
	avg. objs per app var	30.4	9.1	6.5
	edges (over ~25K meths)	186,990	142,957	130,627
	app poly v-calls (of ~0.73K)	56	48	48
	app may-fail casts (of ~0.04K)	28	14	13
	elapsed time (s)	59	172	76
WebGoat	avg. objs per var	34.2	7.9	5.5
	avg. objs per app var	14.1	2.9	2.7
	edges (over ~12K meths)	71,666	58,588	55,989
	app poly v-calls (of ~1.6K)	103	87	87
	app may-fail casts (of ~0.04K)	41	20	19
	elapsed time (s)	39	83	38

object sensitivity of unlimited depth.) The aim of this policy is precision, rather than scalability—in fact, the WALA documentation explicitly warns that the analysis can become "relatively expensive".

Schwarz proposes an approach for the implementation and analysis of web applications [22]. The analysis is, however, designed for a new web application framework, JWIG, without intending to capture the variety of web application technologies in the wild.

7 Conclusions

Enterprise applications have been the elephant in the Java static analysis room. Dominant in practice, they have resisted attempts for analysis, and researchers have largely ignored them. We presented techniques for flexible modeling of enterprise application frameworks and demonstrated how this modeling can lead to high-coverage analysis. Additionally, we have identified scalability problems for high-precision analyses and pinpointed the standard Java `HashMap` and `ConcurrentHashMap` functionality as a central piece in these problems.

Our JackEE framework succesfully tackles the fundamental problems of static analysis in the domain of Java Enterprise applications. Its modular and highly expandable technique of handling web framework functionality guarantees state-of-the-art analysis coverage. At the same time the concept of *sound-modulo-analysis* modeling provides a non-intrusive way of handling core Java data structures, leading to improved precision while achieving very high scalability, one of the greatest challenges of static analysis that aims to model the whole heap of the program. Object-sensitive analyses that otherwise would be infeasible can be made viable using *sound-modulo-analysis* models. All this is achieved in the setting of realistic web applicatiions, while making no compromises in analysis soundness and almost fully preserving the intended functionality of the implementation. Finally, the *sound-modulo-analysis* approach can easily be extended to provide an even higher level of precision and scalability.

We hope that JackEE can be the beginning of research interest in analyzing enterprise applications. Covering a domain of such complexity and size cannot be achieved via a single step. However, collecting a set of sizable, realistic benchmarks, showing that their analysis is feasible, and making progress in its precision are good ways to inspire further research in this high-value area.

Acknowledgments

We gratefully acknowledge funding by the European Research Council, grant 790340 (PARSE) and an Oracle Labs collaborative research grant.

References

[1] 2019. Dependency injection. https://en.wikipedia.org/wiki/Dependency_injection Accessed: 2019-11-22.

[2] 2019. Java Platform, Enterprise Edition. https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition Accessed: 2019-11-19.

[3] 2019. TIOBE Index. https://www.tiobe.com/tiobe-index/ Accessed: 2019-11-19.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[5] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 678–692. https://doi.org/10.1145/3192366.3192383

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

[9] Matthias Buchler, Johan Oudinet, and Alexander Pretschner. 2012. SPaCiTE – Web Application Testing Engine. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, Washington, DC, USA, 858–859. https://doi.org/10.1109/ICST.2012.187

[10] Jens Dietrich, François Gauthier, and Padmanabhan Krishnan. 2018. Driver Generation for Java EE Web Applications. In *Australasian Software Engineering Conference (ASWEC)*. IEEE, 121–25.

[11] Stephen J. Fink et al. [n. d.]. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[12] Pratik Fegade and Christian Wimmer. 2020. Scalable Pointer Analysis of Data Structures Using Semantic Models. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/3377555.3377885

[13] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. https://martinfowler.com/articles/injection.html Accessed: 2019-11-22.

[15] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[16] Rod Johnson. 2002. *Expert One-on-One J2EE Design and Development.* Wiley Publishing, Inc.

[17] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

[18] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[19] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41.

[20] Oracle. 2019. Java EE at a glance. https://www.oracle.com/java/technologies/java-ee-glance.html Accessed: 2019-11-19.

[21] Java Communtity Process. 2019. JSR 330: Dependency Injection for Java. https://jcp.org/en/jsr/detail?id=330 Accessed: 2019-11-22.

[22] Mathias Romme Schwarz. 2013. *Design and Analysis of Web Application Frameworks.* Ph.D. Dissertation. Superviser: Anders Møller.

[23] Micha Sharir and Amir Pnueli. 1981. *Two Approaches to Interprocedural Data Flow Analysis.* Chapter 7, 189–233.

[24] Olin G. Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. Dissertation. Carnegie Mellon University.

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30.

[26] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

[27] Manu Sridharan. 2019. Commit: remove com.ibm.wala.j2ee. https://github.com/wala/WALA/commit/7045a06e51acfe954b950bab3480bc8b436f4481 Accessed: 2019-11-19.

[28] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint Analysis of Framework-based Web Applications. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 1053–1068. https://doi.org/10.1145/2048066.2048145

[29] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer Berlin Heidelberg, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

[30] John Toman and Dan Grossman. 2019. Concerto: A Framework for Combined Concrete and Abstract Interpretation. *Proc. ACM Program. Lang.* 3, POPL, Article 43 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290356

[31] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Fundamental Approaches to Software Engineering*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 210–225.

[32] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008