F4F: Taint Analysis of Framework-based Web Applications

Manu Sridharan[1] Shay Artzi[1] Marco Pistoia[1] Salvatore Guarnieri[2] Omer Tripp[2,3]
Ryan Berg[2]

[1]IBM T. J. Watson Research Center, Yorktown Heights, NY USA
[2]IBM Software Group, Littleton, MA USA
[3]Tel-Aviv University, Tel Aviv, Israel
{msridhar,artzi,pistoia,sguarni}@us.ibm.com, omert@il.ibm.com, ryan.berg@us.ibm.com

Abstract

This paper presents F4F (Framework For Frameworks), a system for effective taint analysis of framework-based web applications. Most modern web applications utilize one or more web frameworks, which provide useful abstractions for common functionality. Due to extensive use of reflective language constructs in framework implementations, existing static taint analyses are often ineffective when applied to framework-based applications. While previous work has included *ad hoc* support for certain framework constructs, adding support for a large number of frameworks in this manner does not scale from an engineering standpoint.

F4F employs an initial analysis pass in which both application code and configuration files are processed to generate a specification of framework-related behaviors. A taint analysis engine can leverage these specifications to perform a much deeper, more precise analysis of framework-based applications. Our specification language has only a small number of simple but powerful constructs, easing analysis engine integration. With this architecture, new frameworks can be handled with no changes to the core analysis engine, yielding significant engineering benefits.

We implemented specification generators for several web frameworks and added F4F support to a state-of-the-art taint-analysis engine. In an experimental evaluation, the taint analysis enhanced with F4F discovered 525 new issues across nine benchmarks, a harmonic mean of 2.10X more issues per benchmark. Furthermore, manual inspection of a subset of the new issues showed that many were exploitable or reflected bad security practice.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Security

1. Introduction

Taint analysis has emerged as a useful technique for discovering security vulnerabilities in web applications [13–15, 18, 20]. Security taint analysis is an information-flow analysis that automatically detects flows of untrusted data into security-sensitive computations (*integrity violations*) or flows of private data into computations that expose information to public observers (*confidentiality violations*). Information-flow security vulnerabilities account for six of the top ten security vulnerabilities according to the Open Web Application Security Project (OWASP).[1] Previous work has shown that taint analysis can effectively expose such vulnerabilities in real-world web applications [13, 14, 18, 20].

Most modern web applications are built using one or more sophisticated *web application frameworks*. These frameworks are special software libraries that simplify web application development by providing higher-level abstractions for common tasks. For example, many frameworks provide automatic population of user-defined data structures with HTTP request data and mechanisms to ease mixing of static HTML and dynamically generated content (see §2 for further discussion).

Unfortunately, static analysis of web applications is significantly hindered by their use of frameworks. Framework implementations often invoke application code using reflection, based on information provided in configuration files. Extensive use of reflection causes well-known difficulties for static analysis. Real-world bug-finding analyses often ignore reflective code, but for framework-based web applications this leads to many false negatives, since the application code is primarily invoked via reflection. Other techniques model reflection usage based solely on code analysis [4, 12], but

[1] `http://owasp.org`

this is also ineffective for our target applications, as the configuration file information used by the frameworks cannot be precisely recovered via code analysis alone. Handling reflection via code analysis can also cause scalability problems, as excessively over-approximate reflection handling can lead to analysis of a large amount of unreachable code. Besides reflection, complex string manipulation and data structure usage in framework code can also cause static analysis difficulties, to be illustrated in detail in §2.

Some previous analyses have integrated *ad hoc* handling of certain web framework features [6, 20], but none provide a general solution for handling the large number of frameworks in common use today. Wikipedia lists nearly 100 web frameworks, including more than 30 for Java alone [23]. Furthermore, individual frameworks can also vary significantly between versions, necessitating special handling for each version. Handling each individual framework through modification of the core static analysis engine does not scale from an engineering standpoint, since it requires a developer with significant analysis expertise. Also, some previous work on framework handling was based on code analysis alone [20], making framework features heavily dependent on configuration file data impractical to handle.

In this paper, we present Framework for Frameworks (F4F), a novel solution that augments taint analysis engines with precise framework support and allows for handling new frameworks without modifying the core analysis engine.[2] In F4F, a framework analyzer first generates a specification of an application's framework-related behavior in a simple language called WAFL (for Web Application Framework Language). The WAFL specification is generated based on both lightweight code analyses and information found in other relevant artifacts such as configuration files. The taint analysis then uses the WAFL specification to enhance its analysis of the application. This approach has several advantages over previous work:

- By utilizing configuration file data, F4F can yield a far more precise *and* complete handling of framework semantics than previous approaches. Additionally, configuration file data can enhance the usefulness of the issues reported by analysis, e.g., by enabling the association of each issue with the URLs that cause the corresponding code to run.
- With F4F, the analysis engine need only understand WAFL specifications, not the details of handled web frameworks, making the engine design cleaner. Furthermore, WAFL was carefully designed to make adding F4F support to an existing taint analysis straightforward.
- WAFL specification generators can be written by developers unversed in the details of the analysis engine, greatly easing the process of handling new frameworks. The specifications could even be generated by a user of the analysis tool, for example to handle a custom framework not available to the tool developers.
- For analysis of languages such as Java and C#, handling of reflection in the analysis engine can be much less conservative, as much of the relevant behavior is present in the WAFL specification.[3] This less conservative reflection handling can lead to scalability improvements and reduced false positives.

We have implemented WAFL specification generators for several frameworks and added WAFL support to a state-of-the-art taint analysis. In practice, we found that lightweight intraprocedural analyses and configuration file processing sufficed for generating specifications of the framework-related behaviors we encountered. Furthermore, we were able to add WAFL support to the taint analysis with very minimal changes to its code. In an experimental evaluation, we compared the effectiveness of the taint analysis on several applications with and without F4F. F4F made a significant impact: the analysis found 525 more issues with framework support enabled across our nine benchmarks, a harmonic mean of 2.10X more issues per benchmark. Furthermore, manual inspection of a subset of the new issues showed that many were exploitable or reflected bad security practice. We have also added F4F support to a version of Rational AppScan Source Edition,[4] a commercial taint-analysis product.

This paper makes the following contributions:

- We define WAFL, a simple specification language for expressing framework-related behaviors of web applications.
- We describe automatic WAFL generators for several popular Java web frameworks.
- We describe a non-intrusive technique for enhancing an existing taint analysis engine to support WAFL specifications.
- We present an experimental evaluation showing that framework support enabled hundreds of new issues to be reported across a suite of benchmarks.

The remainder of this paper is organized as follows. First, we present a detailed motivating example in §2, showing the difficulty of analyzing framework-based web applications and how F4F aids this analysis. We then describe WAFL, our specification language, in §3. In §4, we describe the WAFL generators we have built thus far and what analyses were required in these generators. §5 explains how we added support for WAFL to a state-of-the-art taint analysis engine.

[2] Although many of our techniques could be generalized to other analyses, in this paper we focus concretely on how to use F4F for taint analysis, due to its importance for web application security.

[3] This could compromise soundness, but we know of no practical security analysis tool that is sound for Java in the presence of reflection, native methods, and dynamic class loading.

[4] `http://www.ibm.com/software/rational/products/appscan/source/`

§6 presents our experimental evaluation of F4F on a range of web applications. Finally, §7 discusses related work, and §8 concludes and discusses future work.

2. Motivating Example

In this section, we give some brief background on taint analysis and Java web frameworks, and then we illustrate how F4F can improve taint analysis of framework-based applications via a detailed example.

2.1 Taint Analysis

Security bugs in web applications can often be discovered by *taint analysis*. Web-application vulnerabilities are often due to either flow of untrusted information into a security-sensitive operation (integrity violations), or flow of confidential information into publicly-observable parts of the application (confidentiality violations). Taint analysis is an information-flow analysis that models both integrity and confidentiality violations in a natural way. The client specifies a set of security rules, where a rule is a triple comprised of *sources*, *sinks* and *sanitizers*. Sources introduce untrusted or confidential data into the application. Sinks represent either security-sensitive operations (for integrity rules) or release points (for confidentiality rules). Finally, sanitizers represent operations that endorse the data, either by declassifying it or by modifying it to make it benign. Given a rule r, taint analysis tries to find data-flow paths in an application from sources of r to sinks of r that do not pass through a sanitizer of r—any such path is an indication of a potential security vulnerability. For a more detailed discussion of taint analysis, see previous work, e.g., [13, 20].

2.2 Java Web Frameworks

Java EE Java Platform, Enterprise Edition (Java EE, formerly J2EE) [10] is the framework upon which most Java web applications and web frameworks are built. At its core, a typical web application accepts a request from a client, performs some computation (possibly interacting with a database), and sends a response back to the client (usually HTML and JavaScript). The goal of a web framework is to provide abstractions that ease programming the more tedious, error-prone parts of this process. Java EE provides several such abstractions, two of which are particularly important from a security perspective: *servlets* and *Java Server Pages (JSPs)*.

In a Java EE application, the logic for each web page resides in a *servlet* class, typically in a method with a signature like the following:

```
void doGet(HttpServletRequest req, HttpServletResponse resp);
```

Given a client request, Java EE invokes `doGet()` on an appropriate servlet to generate the response. The `HttpServletRequest` and `HttpServletResponse` parameters respectively give typed interfaces to the request and response (a friendlier interface than raw network data). A *deployment descriptor* configuration file describes which servlet the framework should invoke for each URL, saving the developer from writing custom dispatch code. Also, servlets may use the *session state* abstraction provided by Java EE to store data across multiple client requests, e.g., to maintain a shopping cart in an e-commerce application.

Java Server Pages (JSPs) [11] ease the process of sending an HTML response to a client. In a servlet, HTML can be sent to the client by passing strings to a `java.io.Writer` object obtained from the `HttpServletRequest`, a rather low-level API. In contrast, in a JSP file, one writes the desired HTML directly, using special syntax to execute Java code and include its output in the HTML. The Java EE framework compiles JSPs to Java code that sends the contents of the JSP to the client when executed. A servlet can "invoke" a JSP by forwarding to the JSP's URL, passing data by storing it in the session state or in the `HttpServletRequest` object. In this manner, a developer can obtain some separation between the logic of handling a request (done in the servlet) and the rendering of the response (done in the JSP).

The Struts Framework Apache Struts[5] is a framework built atop Java EE, with higher-level abstractions to further ease web development. Like many other web frameworks, Struts encourages a model-view-controller (MVC) design [5], yielding a clean separation between the core logic for handling a request (the controller), the rendering of the response (the view), and how state is communicated between the two (the model). At runtime, Struts connects the model, view, and controller components based on settings in a configuration file (to be illustrated in §2.3).

The top graph in Figure 1 shows the flow of server-side code that occurs when a Struts application processes a request. Initially, the request is handled by internal Struts code (the first shaded node), which (1) parses the request URL to determine which controller should be invoked and (2) populates a *form object* of some user-defined type (specified in a configuration file) with data from the request. The term "form object" stems from the common case of request data including values entered into a web form. Struts's automatic form object population saves the developer from writing code to extract each piece of form data from the request, perform type conversions, etc. After form object population, Struts invokes the controller (the first oval, a sub-class of the Struts `Action` class), which reads the form object and performs necessary database interactions and business logic. The controller returns a view name, which Struts uses to forward to the appropriate view, again based on configuration file information (the second shaded node); this indirection enables the view technology to be changed without modifying the controller code. Finally, the view (often a JSP) renders the final response.

[5] `http://struts.apache.org/1.x/`

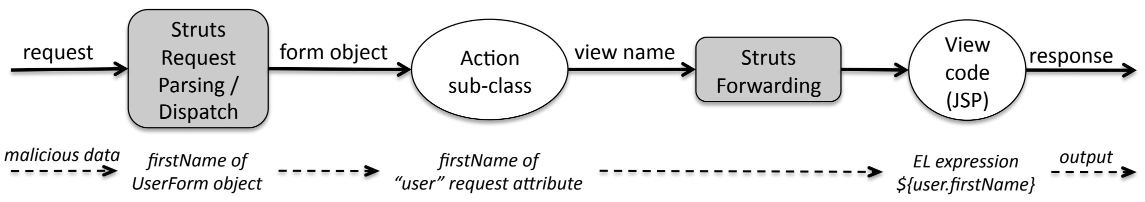
request
Struts Request Parsing / Dispatch
form object
Action sub-class
view name
Struts Forwarding
View code (JSP)
response
malicious data
firstName of UserForm object
firstName of "user" request attribute
EL expression ${user.firstName}
output

Figure 1. Request-to-response flow when using the Struts framework (top) and an illustration of an XSS vulnerability in the example of Figures 2 and 3 (bottom). Shaded rectangles indicate functionality within Struts, while ovals indicate application code. The bottom graph gives program names for the malicious data in the exploit, vertically aligned with their corresponding Struts phase in the top graph.

```
 1  class UserForm extends ActionForm {
 2    private String firstName, lastName;
 3    public String getFirstName() { return firstName; }
 4    public void setFirstName(String firstName) {
 5      this.firstName = firstName;
 6    }
 7    public String getLastName() { return lastName; }
 8    public void setLastName(String lastName) {
 9      this.lastName = lastName;
10    }
11  }
12  class UserAction extends Action {
13    public ActionForward execute(ActionMapping mapping,
14                                 ActionForm form, ...) {
15      UserForm userForm = (UserForm) form;
16      request.setAttribute("user", userForm);
17      ActionForward fwd = mapping.findForward("showuser");

18      return fwd;
19    }
20  }
21  class ShowUser_jsp {
22    public void _jspService(...) {
23      ...; String s = evaluateEL("${user.firstName}");
24      out.write("<p>" + s + "</p>"); ...
25    }
26  }
27  //ShowUser.jsp
28    <p>${user.firstName}</p>
```

Figure 2. Code for our example.

```
 1  <form-beans>
 2      <form-bean name="userForm" type="UserForm"/>
 3  </form-beans>
 4  <action-mappings>
 5      <action name="userForm"
 6              path="/user"
 7              type="UserAction"
 8              validate="false">
 9          <forward name="showuser"
10                   path="/pages/ShowUser.jsp"
11                   redirect="false"/>
12      </action>
13  </action-mappings>
```

Figure 3. Relevant portion of the Struts XML configuration file for our example.

Figure 1 elides the flow of the request and session state, which Struts retains from Java EE. In Struts (and many Java web frameworks), the Java EE `HttpServletRequest` and `HttpServletResponse` objects are essentially global variables that may be read or mutated by any module. In particular, an attribute map associated with these objects is often used to pass the model data from the controller to the view. Tracing data flow through these objects and through the Java EE session state is often critical to finding security vulnerabilities.

2.3 Running Example

Figures 2 and 3 respectively show the application code and configuration information for a Struts-based application to be used as a running example. As we present this example, we shall highlight features that cause difficulties for static analyses, motivating F4F.

Lines 2, 5, and 7 in Figure 3 indicate that the controller class `UserAction` should be invoked with a form object of type `UserForm`. This data flow corresponds to the second edge of the top graph in Figure 1. Struts implements this functionality by using reflection to instantiate the classes named in the configuration file and invoke the appropriate controller method. This reflection is very difficult to model using static code analysis alone, as the relevant type names are only present in the configuration file.

Struts automatically populates `UserForm` objects with (tainted) HTTP request data based on the *Java bean* naming convention [3]. Java beans have *properties* defined by the presence of appropriately-named "getter" and "setter" instance methods. For our example, objects of type `UserForm` have properties `firstName` and `lastName`, based on the methods `getFirstName()`, `setFirstName()`, etc. in Figure 2. When initializing a `UserForm` object, Struts invokes the object's `setFirstName()` and `setLastName()` methods with the HTTP request parameters `firstName` and `lastName`, respectively. This initialization is again accomplished via reflection, making precise modeling of the data flow via pure static code analysis challenging.

The `UserAction.execute()` method—the main logic of the controller—sets the request attribute `"user"` to the provided `UserForm` object (line 16 of Figure 2). Then, it returns the view name "showuser", wrapped in an ActionForward object (lines 17–18). The configuration in Figure 3 shows that the view "showuser" is associated with the URL path /pages/ShowUser.jsp (lines 9 and 10), so Struts forwards control to this JSP by invoking the ShowUser_jsp._jspService() method in Figure 2. (Recall from §2.2 that JSPs are executed via compilation to Java code.) Reflection is once again used by Struts to achieve the control transfer to JSP code, based on strings in the configuration file that are opaque to typical static analyses.

An excerpt of ShowUser.jsp shown on line 28 in Figure 2 includes a string ${user.firstName} in Java EE's *Expression Language (EL)* [7]. EL expressions ease accessing the state of Java bean objects stored in Java EE's request or session state, and they are evaluated at runtime by a custom interpreter. In our case, the EL interpreter evaluates ${user.firstName} to the result of invoking getFirstName() on the UserForm object stored in the "user" request attribute at line 16 in Figure 2. Line 23 shows the call to the EL expression evaluator in the corresponding generated Java code, and line 24 shows the result being written to the response.

Two aspects of the JSP code in our example cause additional difficulty for code analysis. First, the EL expression interpreter invoked on line 23 in Figure 2 makes heavy use of reflection. Second, the use of a request attribute to store and retrieve the UserForm object (lines 16 and 23) is difficult to analyze precisely. Request attributes are typically implemented with one or more map data structures, and the analysis must distinguish distinct entries in these maps (where the key objects are passed across several procedure calls) to precisely track attribute flow. In our experience, the inability to distinguish request attributes during taint analysis can lead to a large number of false positives.

Our example program contains a cross-site scripting (XSS) security vulnerability [17] whose data flow is represented in the bottom graph of Figure 1. An XSS vulnerability exists when a web site includes some unvalidated user data on a web page, allowing, e.g., for an attacker to add malicious JavaScript code to the page. In our example, say a user includes malicious data in the "firstName" parameter of an HTTP request. In populating the form object, Struts copies this data to the firstName field of a UserForm object, which is passed to UserAction.execute(). Then, as described above, the UserForm object is stored in the "user" request attribute, and its firstName field (holding the malicious data) is read via evaluation of the ${user.firstName} EL expression, causing the malicious data to reach the output. The goal of F4F is to enable static analyses to identify vulnerabilities like this one without needing to precisely analyze all the corresponding reflection usage in framework implementations.

```
 1  fun entrypoint UserAction_entry(request) {
 2    UserForm f = new UserForm();
 3    f.setFirstName(request.getParam("firstName"));
 4    f.setLastName(request.getParam("lastName"));
 5    (new UserAction()).execute(_,f,...);
 6    (new ShowUser_jsp())._jspService(...);
 7  }
 8  global request_user;
 9  global session_user;
10  replaceCall setAttribute() 16 {
11    request_user = argToOrigCall(2);
12  }
13  replaceCall evaluateEL() 23 {
14    l = (UserForm)nondet(request_user, session_user);
15    argToOrigCall(-1) = l.getFirstName();
16  }
```

Figure 4. WAFL specification for our example.

2.4 The F4F Solution

Framework for Frameworks (F4F) employs framework-specific handlers to automatically generate a specification of a program's framework-related behaviors. The Web Application Framework Language (WAFL) specification generated for our running example is shown in Figure 4. We use Java-like syntax in the figure for readability, but WAFL specifications also have a language-independent representation (see §3 for details). We shall show how this specification exposes the entire XSS vulnerability from our example for easy discovery by a taint analysis.

The first key element of the specification is a *synthetic method* UserAction_entry() (Figure 4, lines 1–7). This synthetic method models how Struts invokes the code in Figure 2 based on the configuration in Figure 3: given a request, it populates a UserForm object with (tainted) request data (lines 2–4), invokes UserAction.execute() with that object (line 5), and finally invokes the _jspService() method corresponding to the target view name returned by execute() (line 6) . The synthetic method makes explicit the write of tainted request parameter data into a UserForm object and the control transfer to the JSP reading that data. Critically, the synthetic method reflects information obtained from both the configuration file and the code—it would be infeasible to build such a model via code analysis alone. Finally, note that the synthetic method is marked as an entrypoint, useful for call graph construction in the core analysis engine.

The other key elements of the WAFL specification are the *call replacements* (lines 10–16), indicating call sites that should be replaced with more analyzable code. For our example's XSS vulnerability, the call replacements help the taint analysis discover the tainted flow through the "user" request attribute. The first call replacement says that the setAttribute() call at line 16 of Figure 2 should be replaced with an assignment of actual parameter 2 of the call (the userForm local; numbering starts from 0) to a global variable request_user, declared at line 8 of Fig-

location	l	::=	$v \mid e.f$
assignable	a	::=	$l \mid \textbf{argToOrigCall}(\texttt{-1})$
expression	e	::=	$l \mid a := e' \mid \texttt{f}(e_1, e_2, \dots)$
			$\mid \textbf{taint} \mid \textbf{nondet}(e_1, e_2, \dots)$
			$\mid \textbf{argToOrigCall}(i)$
global declaration	g	::=	$\textbf{global}\ (\textbf{request} \mid \textbf{session})\ v$
			$[\textbf{properties}\ v_1, v_2, \dots]$
synthetic method	m	::=	$\textbf{fun}\ [\textbf{entrypoint}]\ \texttt{f}(v_1, v_2, \dots)$
			$\{(\textbf{var}\ v)^*(e)^*\}$
call replacement	r	::=	$\textbf{replaceCall}\ \mathit{callSiteId}\ e$
specification	p	::=	$(g \mid m \mid r)^*$

Figure 5. A grammar for our specification language WAFL.

ure 4.[6] The second call replacement indicates that the `evaluateEL()` call at line 23 should be replaced by an assignment that copies either `request_user.getFirstName()` or `session_user.getFirstName()` (where the receiver is of type `UserForm`) to the variable holding the return value of the original call, in this case `s`.[7] Together, these call replacements expose the request attribute accesses relevant to the XSS vulnerability.

As a whole, the WAFL specification in Figure 4 exposes the entire XSS vulnerability in our example for easy discovery by a taint analysis, exemplifying the utility of F4F. In our implementation, we also provide URL information to the taint analysis via the WAFL specification. For our example, this would enable the analysis to associate the partial URL `/user` (from line 6 of Figure 3) with the XSS issue. The URL corresponding to an issue can be very helpful information, for example to aid in testing the issue's exploitability.

3. WAFL

In this section, we present the details of the Web Application Framework Language (WAFL), used to specify the framework-related behaviors of web applications. Along with describing WAFL's constructs, we discuss why particular constructs were included (or excluded) and sketch how analysis engines can easily incorporate information from WAFL specifications.

Figure 5 presents a grammar for WAFL. In contrast to the example in Figure 4, which presented a WAFL specification using a Java-like syntax, this grammar uses a simplified, more language-independent syntax, both for clarity and to emphasize that WAFL is not specific to Java.[8] At the top level, a WAFL specification consists of a list of global declarations, synthetic methods, and call replacements, which we shall discuss in turn.

[6] Note that this call replacement does not soundly over-approximate the full behavior of `setAttribute()`. In general, our system is designed to expose key framework behaviors to a typical bug-finding (i.e., unsound) taint analysis, not to provide specifications that over-approximate program behavior.

[7] EL expressions may implicitly refer to either request or session attributes, necessitating the two globals; see §4.2.1 for further discussion.

[8] Our tool can output similar WAFL specifications in XML format.

3.1 Globals

Global declarations are useful for representing possible data flows across disparate parts of an application. For example, our WAFL generators use globals to represent flows through request or session attributes (see the `request_user` and `session_user` globals in Figure 4). The use of globals to model data flow across disparate application code can simplify the subsequent taint analysis, since with globals, the flow can be determined syntactically with good precision.[9] Without the globals, the taint analysis may be forced to track many possible aliases to the same data to discover the flow. Request and session attributes in Java web applications are often accessed through many different aliases, and the difficulty of automatically discovering which request / session pointer is in scope at some program point partially motivated our introduction of globals.

Globals can be annotated with either **`request`** or **`session`** to respectively indicate whether they are scoped to a single HTTP request or to a client session. The distinction is useful since taint analyses often analyze the entrypoint method for each request separately, necessitating special handling for session-scoped state that lives across multiple entrypoints.

Finally, it is sometimes useful to associate a set of properties with a global. We have used properties to model constructs like *dynamic beans*, which store internal state in a map instead of in separate fields (e.g., the Struts `DynaActionForm` class, discussed further in §4.2.2). Framework configuration files often specify the exact set of properties used in these dynamic beans, and exposing them directly in the WAFL specification saves the taint analysis from having to re-discover the properties through code analysis (often non-trivial due to complex map implementations). We expect these properties to be especially useful in handling web frameworks for dynamic languages, which often make use of maps for storing object state.

3.2 Synthetic Methods

WAFL specifications typically use *synthetic methods* to model how a framework invokes application code. A synthetic method consists of a list of local variable declarations followed by a sequence of expressions, whose possible forms are shown in Figure 5.[10] Expressions include standard assignments, variables / field accesses, and method calls. A **`taint`** expression represents some form of tainted data, e.g., an HTTP request parameter; the representation of this expression will vary by programming language. A **`nondet`**

[9] This assumes a (control-)flow-insensitive model of data flow through globals, the standard approach for scalable taint analyses.

[10] For simplicity, expressions are statements are not distinguished in WAFL. Also, we elide type information in this presentation for clarity. Our implementation associates any known type information with variables but elides downcasts, as they can easily be inserted while processing the specification.

expression non-deterministically evaluates to the value of one of its arguments. **argToOrigCall** expressions are only relevant for call replacements, to be discussed in §3.3.

It is worth noting what types of expressions are *not* present in synthetic methods. There are no standard control constructs like conditional branches, gotos, loops, etc. Such constructs are typically not interpreted by taint analyses, and excluding them eases processing of synthetic methods by analysis engines. While we elide allocation calls like Java's new expressions from WAFL, such calls can easily be added by the analysis engine if needed (e.g., if call graph reasoning is based on points-to analysis); the example in Figure 4 includes such inserted constructor calls. Note that since allocation is elided, method calls in WAFL specifications cannot use dynamic dispatch and must specify the exact method to be invoked.

3.3 Call Replacements

Many framework behaviors can be modeled via the versatile *call replacement* construct. The semantics of call replacements are straightforward: a method invocation at a certain call site, indicated by the $callSiteId$, should be replaced by an expression e. Within the replacement expression, **argToOrigCall** expressions can be used to refer to either actual parameters at the call site or to the value returned by the call at that site (written **argToOrigCall**(-1)). Figure 4 illustrates a couple of uses of call replacements (for modeling request attribute accesses and one type of EL expression); we shall discuss more uses in §4.

Limiting application code modification to call replacements greatly eases integrating WAFL specifications into analysis engines, while still accommodating the framework modeling needs we have seen thus far. Handling call replacements is particularly easy for engines that already handle dynamic dispatch or higher-order functions; the replacement code is simply placed in a new method, which is marked as the sole possible target of the replaced call site. With this technique, no mutation of the original code is required. Even in an approach where the call is actually replaced in an intermediate representation, the transformation is still simpler than if arbitrary blocks of code could be replaced (which could require complex patching of control-flow graphs, symbol tables, etc.).

4. WAFL Generators

Here we discuss our work thus far on WAFL generators for several Java web application frameworks. First, we present two example WAFL generators in detail (§4.1), illustrating the simplicity and lightweight nature of the generators we have written thus far. Then, we present an overview of our other WAFL generators (§4.2), and finally discuss lessons learned in the process of building them (§4.3). While we have only implemented WAFL generation for a few of the many Java web frameworks available [23], we expect that

```
HANDLEACTIONENTRYPOINT($a, m$)
 1  $n \leftarrow a$.name, $T \leftarrow a$.type
 2  $f \leftarrow$ <form-bean> in config s.t. $f$.name $= n$
 3  $F \leftarrow f$.type
 4  $beanProps \leftarrow \{$ prop | $F$.setProp() exists $\}$
 5  ADDSTMT($m$, '$F$ f = new $F$()')
 6  for each prop in $beanProps$ do
 7          ADDSTMT($m$,
             'f.setProp(req.getParam("prop"))')
 8  ADDSTMT($m$, '$T$.execute($f$, ...)')
 9  for each call ActionMapping.findForward($w$)
      in $T$.execute() do
10          for each string constant $s$ flowing to $w$ do
11                  $r \leftarrow$ <forward> in config s.t. $r$.name $= s$
12                  ADDSTMT($m$, INVOKEJSPENTRY($r$.path))
```

Figure 6. Pseudocode for generating WAFL for Struts Action entrypoints.

many other frameworks will be similarly structured to those discussed below, and hence similar techniques will apply.

4.1 Detailed Examples

Here, we give pseudocode for two WAFL generators, respectively handling a core feature of the Struts framework and accesses to Java EE request and session attributes (both introduced in §2). These examples typify all our WAFL generators in that they only require configuration file parsing and lightweight, local program analyses to successfully model framework behaviors.

Note that our pseudocode elides error handling; our implementation includes thorough checks for malformed inputs. Also, we enclose strings in single quotes, where italicized expressions within the quotes are evaluated before being concatenated.

Struts Figure 6 gives pseudocode for generating WAFL synthetic methods to model the semantics of Struts `Actions`. Given a parsed `<action>` declaration a from a Struts configuration file (e.g., lines 5–12 in Figure 3) and a synthetic method m, the HANDLEACTIONENTRYPOINT procedure populates m with statements reflecting how Struts interprets a. First, statements are added to populate a form bean object with HTTP request data (e.g., lines 2–4 in the WAFL example of Figure 4). Bean properties are determined by inspecting the names of setter methods in the bean's class (line 4),[11] and each setter is invoked with the corresponding request parameter (line 7). Line 8 adds an invocation to the appropriate `execute()` method, passing the populated form bean.

Lines 9–12 generate statements modeling the control flow from controller (the `Action`) to view in the Struts MVC architecture (e.g., line 6 in Figure 4). To discover the possible view names, an analysis is performed to find string constants

[11] We access XML attributes as fields, e.g., f.type on line 3.

```
HANDLEONEACCESS($c, a, scope, isReadAccess$)
1  $stmts \leftarrow \emptyset$
2  $names \leftarrow \{$ $'scope\_\_attr'$ |
                   string constant $attr$ flows to $a$ $\}$
3  for each $name \in names$ do
4        ADDGLOBAL($name, scope$)
5        if $isReadAccess$
6          then $stmt \leftarrow$ '$\texttt{argToOrigCall}(-1) := name$'
7          else $stmt \leftarrow$ '$name := \texttt{argToOrigCall}(2)$'
8        $stmts \leftarrow stmts \cup stmt$
9  ADDCALLREPLACEMENT($c$, 'nondet( $stmts$ )')

HANDLEATTRIBUTEACCESSES()
1  for each call $c$ = ServletRequest.getAttribute($a$) do
2        HANDLEONEACCESS($c, a$, REQUEST, TRUE)
3  for each call $c$ = ServletRequest.setAttribute($a, v$) do
4        HANDLEONEACCESS($c, a$, REQUEST, FALSE)
5  for each call $c$ = HttpSession.getAttribute($a$) do
6        HANDLEONEACCESS($c, a$, SESSION, TRUE)
7  for each call $c$ = HttpSession.setAttribute($a, v$) do
8        HANDLEONEACCESS($c, a$, SESSION, FALSE)
```

Figure 7. Pseudocode for generating WAFL for request / session attribute accesses.

passed as a parameter to `ActionMapping.findForward()` calls within the `execute()` method (e.g., line 17 in Figure 2). Our implementation discovers these constants using intraprocedural def-use chains; we have not seen a need for more sophisticated techniques. The `<forward>` element from the configuration matching each view name is found (line 11), and a call to the appropriate JSP entrypoint method is added (line 12).

Request and Session Attributes Figure 7 gives pseudocode for generating call replacements to handle Java EE request and session attribute accesses. The core logic is in HANDLEONEACCESS, which generates a call replacement given a call site c, the actual parameter a at c holding the attribute name, a scope $scope$ for the access (either REQUEST or SESSION), and a boolean $isReadAccess$ indicating if the access is a read or write. Handling all accesses in the code is done by scanning for calls to access methods and calling HANDLEONEACCESS with the right parameters for each call, shown in HANDLEATTRIBUTEACCESSES.

Recall from §2 that we model request and session attributes as global variables, so attribute accesses become reads and writes to these globals. Line 2 of HANDLEONEACCESS creates names for the globals, based on the access scope and the possible attribute names (discovered using the same intraprocedural flow analysis used for view names in the Struts example). The subsequent loop declares each possible global (line 4) and then creates a statement for the read or write access as appropriate (lines 5–7). (On line 7, `argToOrigCall(2)` references the standard parameter position of the new attribute value for attribute setter methods.) Finally, line 9 adds a call replacement that replaces c with a non-deterministic execution of one of the global access statements.

4.2 Overview of Other WAFL Generators

4.2.1 Java EE, JSP, and EL

Understanding of basic Java EE servlets is essential to any Java taint analysis, and hence this support is built in to previous systems [13, 20]. Our WAFL generator provides deeper support for core Java EE functionality, particularly in exposing data flow from servlets to JSPs. As discussed in §2.2, traditional Java EE applications are often structured in an MVC style: the servlet is the controller, the view is a JSP, and the model is communicated via request or session attributes (very similar to the flow of Struts in Figure 1). Apart from our modeling of attribute accesses (see Figure 7), we also model explicit forwards to JSPs, discovering the target URL using our simple intraprocedural flow analysis (see §4.1), thereby fully capturing the MVC behavior in these applications.

To handle JSP code that uses the Expression Language (EL) (e.g., `${user.firstName}` in line 28 of Figure 2) we built a partial interpreter for the expressions. After discovering the contents of an EL expression (again using our simple local flow analysis), we determine the possible state that could be accessed by scanning the application code for matching getter methods, based on Java bean naming conventions. (So, for an EL access to `firstName`, we look for classes with a method `getFirstName()`.) We then model the EL evaluation as a WAFL **`nondet`** expression that invokes some matching getter method on an object read from either a request or session attribute.[12] For our example from §2, lines 14–15 in Figure 4 give our modeling of the `${user.firstName}` EL expression. While some cases of the non-deterministic read may be infeasible, taint analysis would only report an issue when a read matches a write elsewhere in the code, a case likely relevant to the user.

4.2.2 Struts

Struts was presented in detail in §2, where we showed the WAFL specification for a simple excerpt from a Struts application. Here, we discuss some more advanced Struts features also handled by our generator.

The Struts `DynaActionForm` class motivated our introduction of properties for global variables in WAFL specifications (see §3.1). Figure 6 showed how to generate WAFL to populate a standard Struts form bean, which sub-classes `ActionForm`. `DynaActionForm` objects differ from standard `ActionForms` in that their properties are accessed via methods that take the property name as a parameter, rather than having a getter and setter method per property (e.g.,

[12] EL expressions can also make the scope explicit (e.g., `${request.user}`), in which case we resolve it.

get("name") rather than getName()). As with ActionForms, Struts is able to automatically populate a DynaActionForm with request data, based on a list of property names provided in the XML <form-bean> declaration. Our WAFL generator parses these property names and creates a fresh global with the same properties. The global properties are initialized in the synthetic method for the corresponding Action entrypoint. Call replacements are used to change generic DynaActionForm.get() calls to reads of the corresponding global properties (again employing local String constant analysis to discover the property being accessed), thereby fully exposing DynaActionForm data flow to the taint analysis.

We also handle the Tiles framework,[13] which is very often used in conjunction with either the Struts or Spring frameworks. Tiles is a templating system used to create a common theme for pages in a web site, based on composing JSPs for the page header, footer, body, etc. Our WAFL generator analyzes Tiles configuration files and then treats a forward to a tile as forwarding control to each of the component JSPs.

4.2.3 Spring

Spring[14] is the most complex framework handled by our WAFL generators thus far. Spring provides a very wide variety of features to ease application development, including a sophisticated MVC architecture and a general dependency injection mechanism. Furthermore, Spring features are often highly configurable via XML and method overriding, making the generation of WAFL for certain Spring features somewhat non-trivial. Here, we describe how our WAFL generator handles a few notable Spring features.[15]

Generating synthetic entrypoint methods that model a Spring application's MVC logic (analogous to the methods generated by Figure 6 for Struts) requires handling many different types of controllers. The basic Spring Controller interface provides a single method (parameter types condensed for clarity):

```
ModelAndView handleRequest(Request req, Response resp);
```

A class directly implementing Controller takes the raw HTTP request and response as parameters and returns a ModelAndView object representing the name of the desired view and the model data to be rendered. However, Spring provides several abstract Controller implementations that application code can subclass to obtain more functionality. For example, to get Struts-style automatic population of a bean object with HTTP request data, a developer can subclass AbstractFormController. Each abstract Controller implementation in Spring has its own protocol for which of its methods can or must be overridden in subclasses, what order methods are invoked in, etc. Hence, we have built separate handlers for each of these abstract implementations, and we choose the most suitable one for each application controller based on its supertypes. Each handler is comparable in complexity to those described in §4.1, again relying primarily on configuration file parsing and lightweight code analysis.

Another notable twist in Spring is its *inner beans*, which enable automatic initialization of nested bean data structures. Say that bean type B has accessor methods for a field inner of type Inner, and that Inner has accessor methods for a field name of type String. In automatically populating a B object with request data, Spring will populate the nested Inner object as well, equivalent to the following code (assuming b points to the B object):

```
b.getInner().setName(request.getParameter("inner.name"));
```

Our WAFL generator discovers inner beans by inspecting bean property types, and it generates statements like the above in WAFL synthetic methods to fully model their initialization. Also, our handling of EL expressions (see §4.2.1) is sufficiently general to handle reads from inner beans via EL expressions in JSPs.

4.3 Discussion

Our experience with WAFL specifications thus far has convinced us that our architecture is much more effective than building support for frameworks directly into an analysis engine. The key to the success of the architecture is that only *lightweight* code analyses are required to extract the relevant information for the specifications.[16] If generating WAFL required sophisticated interprocedural analysis in practice, then computing the framework behaviors simultaneously with the taint analysis could yield performance benefits by eliminating redundant work (call graph construction, etc.). However, in our experience, the redundant work between WAFL generation and the subsequent taint analysis (essentially just class hierarchy construction) is quite small. The engineering benefits of separating framework analysis from the core analysis engine, especially as the number of supported frameworks grows, clearly outweigh this small redundancy.

Beyond supporting standard frameworks, our architecture enables (sophisticated) users of a taint analysis engine to help the analysis better handle the difficult constructs in their own applications. In some cases, web applications are built on a private framework or on a customized version of some public framework. We believe that in such cases, a sufficiently-motivated user (e.g., a security auditor tasked with thoroughly checking an application for vulnerabilities) could write her own WAFL generator to at least partially expose the framework behaviors to the security analysis.

We have recently implemented a higher-level API for generating common WAFL constructs to ease the process

[13] http://tiles.apache.org/

[14] http://www.springsource.org/

[15] Our implementation currently handles versions of Spring up to 2.5.

[16] Our code analyses are implemented using the Watson Libraries for Analysis (WALA) [22].

of writing WAFL generators. A sales engineer used the API to create a WAFL generator for the Enterprise Java Beans framework[17] in roughly one week, an encouraging initial success. We are actively studying further techniques for easing creation of new WAFL generators, e.g., a domain-specific language for writing the generators.

We have also recently developed a WAFL generator for the ASP.NET framework[18] and confirmed that the generated specifications increase taint analysis precision for certain applications. The design of ASP.NET is quite different from that of most Java EE frameworks—web page components are abstracted in a manner similar to GUI controls, so, e.g., GUI-style event handlers process user form submissions. Hence, our success in modeling ASP.NET constructs in WAFL shows the flexibility and generality of F4F. A systematic evaluation of the effectiveness of WAFL generation for ASP.NET remains as future work.

5. Taint Analysis Integration

In this section, we present a simple technique based on source code generation for adding WAFL specification support to a taint analysis engine. We then briefly discuss some details of the taint analysis engine used in our experimental evaluation.

5.1 Java Code Generation from WAFL Specifications

As discussed in §3, WAFL was designed to be easy to integrate into analysis engines. The key features that must be added to the analysis engine are the following:

- Processing of WAFL synthetic methods, including generation of an analyzable representation and inclusion of the methods in the application call graph (as entrypoints if marked as such).
- Replacement of call sites in application methods with a representation of the corresponding WAFL call replacement expression.

Here, we present an approach to the above based on generation of Java source code from WAFL specifications and minor changes to the underlying analysis engine. This approach is appealing because the changes to the engine are minimal, the generated Java code can be made understandable for developers, and much of the work can be reused across analysis engines. For support of multiple languages, an approach based on modification of the analysis engine's intermediate representation (IR) may be more maintainable if the IR is shared among the languages.

We created a helper tool called WAFL2Java that generates Java code from WAFL specifications for use with taint analysis. For each synthetic method in the input WAFL specification, WAFL2Java generates a corresponding Java method in a synthetic class whose statements correspond to those in the synthetic method. The most interesting case in this translation is handling method calls. One issue with calls is that in our implementation, WAFL specifications may omit values for some actual parameters if they are not directly relevant to potential tainted flow. The translator can use default values for these parameters in most cases, but handling of the receiver argument is slightly tricky, since the analysis engine may reason that a method call on `null` must be unreachable. A general solution to this problem would involve finding some valid constructor with which to create a receiver object for the call. In our implementation, we exploit the fact that our analysis engine reasons about virtual call targets via local (intraprocedural) type inference, by making the receiver a fresh local assigned the return value of another dummy method; this is sufficient to ensure the invoked method appears in the call graph.

Another tricky situation is when WAFL2Java must generate an invocation of a default- or protected-scope method m, which by default cannot be invoked except from the same package p. WAFL2Java handles this restriction by generating a fresh public class $p.C$ in the desired package that contains a public method $C.m'$ that simply invokes m and returns its value. Since we are not executing the generated code, we need not worry about sealed packages preventing the generation of C.

For the most part, the rest of the translation is straightforward. **`taint`** expressions are translated to an invocation of `HttpServletRequest.getParameter()` (we use the same technique used for method calls if a request pointer is not in scope). We translate **`nondet`** expressions by generating a switch on the value of a public static integer field, which the Java compiler will be unable to simplify. WAFL globals are translated to static fields, with an appropriate fresh class generated for the global if it has properties.

WAFL2Java handles WAFL call replacements by generating a fresh method m_c for each call replacement c. The signature of m_c (i.e., its argument and return types) match the method invoked at the call site to be replaced. The analysis engine can then model c by treating the original call site as if it could only dispatch to m_c. With this translation, handling of the WAFL **`argToOrigCall`** construct is easy: it simply becomes an access of the appropriate formal parameter of m_c, or a **`return`** statement for the case of assigning to **`argToOrigCall`**`(-1)`.

In the end, WAFL2Java provides three outputs to the analysis engine: (1) the synthetic Java code (compiled to bytecode), (2) a list of methods to be treated as entrypoints, and (3) a mapping from each replaced application call site to the generated Java method serving as the replacement target. The analysis engine simply incorporates the synthetic code as part of the application being analyzed, and it incorporates the provided entrypoint methods during call graph construction. Call replacements are also handled during call graph construction: dispatch at the replaced call sites is simply

[17] `http://www.oracle.com/technetwork/java/javaee/ejb/`

[18] `http://www.asp.net/`

redirected to the appropriate generated method. With these simple steps, an analysis engine can fully support WAFL specifications.

5.2 Taint Analysis Engine

For our evaluation, we added WAFL specification support to a version of ACTARUS [8] that analyzes Java bytecodes. ACTARUS is an improved version of TAJ [20], with better scalability and precision. ACTARUS includes a rich set of security rules covering all aspects of the web application's interaction with its environment (including the file system and backend databases); addition of framework support does not modify the rules database used for analysis.

In contrast to TAJ [20], ACTARUS does *not* attempt to model the behavior of reflective calls in framework implementation code. As we analyzed larger programs, we found that TAJ's reflection modeling became a scalability bottleneck. Furthermore, we found that the reflection modeling in TAJ could not discover essential framework behaviors without introducing too much imprecision. The goal with ACTARUS was to improve scalability by ignoring reflection and still capture framework behaviors via F4F.

ACTARUS also does not include the built-in framework support implemented for TAJ [20]. The combination of ACTARUS and F4F discovers many more framework-related issues than TAJ, as TAJ did not have support for important frameworks like Spring and Tiles. We chose to enhance ACTARUS with WAFL support (rather than enhancing TAJ) as it was the best taint analysis available to us (in terms of scalability and precision). We believe that the benefits of F4F should be largely independent of the details of the analysis engine, as WAFL specifications can expose many behaviors that would be difficult for any engine to discover.

6. Evaluation

6.1 Experimental Methodology

Ideally, a taint analysis enhanced with F4F would have the following properties when compared to analysis without F4F:

More discovered vulnerabilities The taint analysis should find significantly more true-positive (i.e., exploitable) security vulnerabilities in framework-based applications.

Few additional false positives The taint analysis should output at most a small number of additional false positives; otherwise, the additional true-positive issues could be lost in the noisy output.

Acceptable running time While the running time of the taint analysis may increase with F4F (since more code / flows are being analyzed), the slowdown should be proportional to the amount of additional work being done by the taint analysis. WAFL specification generation should not be a performance bottleneck.

While experimental measurement of performance overhead was relatively straightforward, we found that performing a complete evaluation of the first two desired properties over a large benchmark suite was impractical. Determining the exploitability of even a small number of taint analysis issues is a significant undertaking, especially when the application code is unfamiliar. Even running some web applications can be non-trivial, due to dependencies on particular databases, servlet containers, etc.; this can make testing exploitability of issues very difficult. On the other hand, ensuring that any particular issue is truly not exploitable can also be challenging, as it may require careful reasoning about possible attack vectors and the behavior of complex validation code. In our experiments, the taint analysis found hundreds of additional issues with F4F enabled, and determining the exploitability of all of these issues was beyond our capability.

In lieu of determining the exploitability of all discovered issues, we performed a more limited classification that still provides useful insights into the effectiveness of F4F. We first manually categorized all new issues discovered with F4F based on whether they were *path-insensitive flows*, i.e., feasible flows of tainted data *ignoring conditional branches*. A path-insensitive flow may not be exploitable, e.g., if a validation check prevents truly dangerous data from reaching the sensitive sink. However, a path-insensitive flow cannot be one of several types of false positives, e.g., those due to infeasible call targets, overwriting of memory locations, or an overly conservative model of some framework feature. Furthermore, we believe that most path-insensitive flows are worthy of manual inspection—even if the flow is a false positive due to validation, security auditors often prefer to manually check the validation for sufficiency. (Analyses to automatically detect validation and sanitization routines continue to advance [19], and F4F can aid such analyses by exposing calls to validation routines performed by frameworks.) We also categorized a subset of issues discovered only with F4F based on their exploitability to gain further insight into the types of vulnerabilities discovered and the causes of false positives.

All our experiments were run on a Red Hat Enterprise Linux 4 machine with four Intel Xeon 3.8GHz CPUs (only one of which was used) and 5GB RAM. We used the Oracle Hotspot JVM version 1.6.0_22 running with a 2.8GB maximum heap.

6.2 Benchmarks

We evaluated F4F on a diverse set of nine subject programs, eight of which are open source. One proprietary application has been anonymized as `AppA`. The programs include an online chat system (`AjaxChat`), systems for managing articles (`StrutsArticle`), photos (`Photov`), documents (`Contineo`), and CVs (`GestCV`), a bulletin board (`JBoard`), a job posting application (`JUGJobs`), and the sample pet store management system provided with Spring (`JPetstore`). Ta-

Benchmark	Benchmark Properties						WAFL specification		
	version	**Java loc**	**classes**	**config. files**	**JSPs**	**frameworks**	**globals**	**methods**	**replacements**
`AppA`	N/A	N/A	250	5	69	Spring, Tiles, EL	9	39	21
`AjaxChat`	0.8.3	4147	29	2	5	Struts, EL	11	9	14
`StrutsArticle`	1.1	7897	45	3	11	Struts, EL	21	8	18
`Photov`	2.1	210304	239	4	48	Struts	41	25	106
`Contineo`	2.2.3	65744	790	5	88	Struts, Tiles	130	134	187
`GestCV`	1.0.0	11524	127	5	17	Struts, Spring, Tiles, EL	13	8	8
`JBoard`	0.3	17500	185	16	47	Struts, Tiles, EL	6	12	14
`JUGJobs`	1.0	4815	30	4	7	Struts, Tiles	11	6	13
`JPetstore`	2.5.6	25820	116	6	45	Struts, Spring, EL	26	38	27

Table 1. Subject program characteristics. In addition to lines of code and number of classes, we also list the number of configuration files and JSPs for each program, along with the frameworks used. For the corresponding generated WAFL specifications, we give the number of globals, synthetic methods, and call replacements.

ble 1 presents the characteristics of our subject programs and of the WAFL specifications we generate for the programs. Links to the above applications and the corresponding WAFL specifications generated by F4F are available online at `http://bit.ly/F4Fbenchmarks`.

Overall, the suite thoroughly exercises our supported frameworks: eight subjects use Struts, three use Spring, five use Tiles, and six use EL. Also note that the size of the WAFL specifications can be substantial (e.g., over 100 synthetic methods and call replacements for `Contineo`), showing the benefit of automatic WAFL generation.

6.3 Results

Table 2 shows the number of issues, call graph size, and running time when each of the benchmarks was analyzed with and without framework support. F4F had a significant impact: The analysis found a total of 525 new issues with F4F, a harmonic mean of 2.10X more issues per benchmark (ranging from 1.1X–14.9X). Note that our WAFL specification generator does not yet handle all features of the supported frameworks, and we have observed several cases where additional support (in particular, better modeling of certain JSP tag libraries) could enable the taint analysis to discover many more flows.

Precision Of the new issues discovered with frameworks, nearly all were path-insensitive flows, i.e., feasible flows if conditional branches were ignored (see §6.1). The few additional issues that were false positives even ignoring conditional branches stemmed from either infeasible dynamic dispatch or over-approximate reasoning about aliasing in the taint analysis.

Note that F4F caused some issues that were reported without framework support to be suppressed. Since WAFL call replacements can replace arbitrary application code, analysis with F4F can lead to suppression of false positives (if imprecisely-analyzed code is replaced) or true positives (if security-relevant code is replaced). There were five suppressed issues in our experiments, of which three were false positives, caused by overly conservative handling of String-based maps like session attributes or Struts `DynaActionForm` state (see §4.2.2) without F4F modeling. The two suppressed path-insensitive flows were due to obscure behaviors like a statement that logged the entire session state; since F4F uses call replacements to more precisely model accesses of session attributes, the session object itself is no longer tainted, causing the analysis to miss the potential log forging vulnerability. These missed flows could easily be addressed by augmenting our WAFL generators; we did not do so as the behaviors were obscure.

Table 3 presents a deeper classification of ten path-insensitive flows discovered only with F4F enabled for each benchmark. Via manual inspection of the code, we classified each issue as exploitable, dangerous, safe, or unknown. A dangerous issue is one in which untrusted data is placed in a store like the database or session state without validation; while not exploitable on its own, such behavior is considered risky from a security perspective.

Only seven of the 90 inspected issues were clearly not exploitable (i.e., "safe"), due to use of standard validation code to prevent vulnerabilities or application-specific semantics that made a particular tainted flow unexploitable. For the cases with standard validation, the taint analysis could easily be enhanced to remove the false positives. The issues categorized "unknown" were cases where we could not be fully confident in complex validation code in the application. Better string analyses [19] could hypothetically verify such code, but for now, security auditors would likely want to inspect these issues by hand. Several types of exploitable issues were discovered only with F4F enabled, including SQL injection, open redirects, and log forging. In summary, our evaluation indicated that F4F meets the goal of exposing more vulnerabilities in framework-based applications without introducing too many false positives.

Performance When running with framework support, running time was no more than 2.43X as long as running without framework support, adding no more than 5.5 minutes of

Benchmark	Without F4F				With F4F			
	Issues	Excl. (PIF)	CG Size	Time (s)	Issues	Excl. (PIF)	CG Size	Time (s)
`AppA`	264	0 (0)	3382	2129	301 (**1.1X**)	37 (28)	3534	2555
`AjaxChat`	3	1 (1)	461	17	14 (**4.7X**)	12 (12)	526	30
`StrutsArticle`	11	0 (0)	771	18	25 (**2.3X**)	14 (13)	898	35
`Photov`	104	1 (0)	5304	101	178 (**1.7X**)	75 (75)	5625	229
`Contineo`	105	0 (0)	4633	391	228 (**2.2X**)	123 (123)	5431	573
`GestCV`	57	2 (1)	2382	622	101 (**1.8X**)	46 (31)	2606	948
`JBoard`	34	1 (0)	1832	239	74 (**2.2X**)	41 (41)	2025	330
`JUGJobs`	24	0 (0)	511	18	39 (**1.6X**)	15 (15)	655	32
`JPetstore`	12	0 (0)	2068	30	179 (**14.9X**)	167 (167)	2243	73

Table 2. The number of issues reported, call graph size, and running time for each of our benchmarks analyzed with and without F4F. The "Excl. (PIF)" column gives the number of issues reported exclusively in each configuration, with the number of path-insensitive flows (defined in §6.1) given in parentheses. The "Issues" column with F4F includes the factor increase over the number of issues found without F4F. Running time with F4F includes WAFL generation time.

Benchmark	Classification
`AppA`	2 exploitable, 8 unknown
`AjaxChat`	7 exploitable, 3 safe
`StrutsArticle`	6 exploitable, 4 dangerous
`Photov`	4 exploitable, 6 dangerous
`Contineo`	2 exploitable, 7 dangerous, 1 unknown
`GestCV`	2 exploitable, 8 unknown
`JBoard`	10 exploitable
`JUGJobs`	2 exploitable, 4 dangerous, 4 safe
`JPetstore`	10 dangerous

Table 3. A deeper classification of a subset of issues discovered only with F4F for each benchmark.

running time (as shown in Table 2). WAFL generation took an average of 31 seconds to run, ranging from 11–74 seconds, showing that most of the additional execution time with F4F enabled was spent in the taint analysis. This result was expected, as more code (indicated by the increased call graph size) and data flows are analyzed with framework support enabled. In all cases, the additional running time was justified by at least a proportionate increase in the number of security issues reported.

6.4 Threats to Validity

One threat to the validity of our results is that the frameworks we have supported thus far may not be representative of web frameworks in general. We have studied the documentation of many other Java and .NET web frameworks [23], and we believe that the techniques employed thus far in our specification generator would extend to these other frameworks in a straightforward manner. We have not studied the structure of frameworks for dynamic languages like PHP in detail, and it is possible that additional specification constructs / generation techniques would be needed in those cases.

Another possible threat is that the chosen benchmarks may not be representative of frameworks-based applications in general. Our benchmarks represent a variety of real-world applications that exercise most features of the supported frameworks. While the degree of benefit may differ for substantially different applications, we still expect significant improvement with framework support.

Finally, it is possible that a disproportionate number of the issues discovered only with framework support are not exploitable compared to those discovered without the support, potentially lessening the benefit of reporting the additional issues. While we were unable to check exploitability of all the new issues discovered with F4F for our benchmarks, the results of classifying a subset of the issues (see Table 3) suggested that in fact most of the additional issues were worthy of inspection.

7. Related Work

Static analyses have long incorporated specifications for code that is difficult to analyze; here, we discuss some of the most closely related work in this space.

Many Java static analysis engines allow for specifications of the behavior of native methods [16, 21, 22], which are substituted at any call to such methods. Additionally, WALA [22] allows for pointer analysis clients to provide code that generates a new version of a method, possibly based on analysis information available at a particular call site. A difference in WAFL is that call replacements enable particular call sites (rather than all call sites to a method) to be replaced in a declarative manner; we are unaware of previous systems with a similar facility.

Jaspan and Aldrich [9] present FUSION, a language for specifying relationships between objects imposed by frameworks, and static analyses to enforce FUSION specifications. Our approaches are complementary: FUSION specifications are written manually, associated with the framework code itself, and focused on typestate-like properties, whereas WAFL specifications are typically automatically generated, associated with each framework-based application, and designed to expose security-relevant behaviors. The alias annotations in the PLAID language of Aldrich et al. [1] could also provide additional support in exposing relevant data flow in framework code.

Ball et al. [2] use training to automatically obtain a model of the Windows kernel when applying model checking to the domain of Windows device drivers. Given kernel procedure k and set $\{d_1, \ldots, d_n\}$ of drivers that utilize k, the verifier is run on each driver linked together with kernel procedure k. This results in n boolean abstractions, which are joined into a refined abstraction of procedure k, b_k. In future runs of the verifier, b_k is used in place of procedure k, effectively replacing the C code by a boolean program specification. The analysis approach taken in F4F is quite different, but the goal of a better environment model (in our case, the web framework) is the same.

Livshits et al. [14] take a probabilistic approach for automatic inference of information-flow specifications. Their tool, MERLIN, models information-flow paths using probabilistic constraints, and then solves the resulting system to classify nodes in the program's information-flow graph as sources, sinks, sanitizers or regular nodes. This approach is complementary to ours: combined with F4F, we suspect that MERLIN could do a significantly better job of inferring information-flow specifications for framework-based applications.

The TAJ system for Java taint analysis included some support for the Struts framework and Enterprise Java Beans (EJBs) [20, §4.2.2]. That system handled tainted flows from Struts `ActionForm` objects, but it did not handle many other key Struts features like `DynaActionForm` objects (see §4.2.2) and forwards to JSPs via `ActionForward` objects (see Figure 2). Furthermore, their framework support was implemented by modifying the analysis engine itself, which becomes an engineering bottleneck as more frameworks need to be supported.

Zheng et al.'s Analysis Preserving Language Transformation (APLT) technique [24] was used to automatically build Java models for C methods suitable for applying a Java security analysis. The models constructed by APLT may deviate from the original semantics in ways that do not matter to the client program analysis, for example ignoring array index values if the static analysis does not track them precisely. In a similar manner, WAFL specifications may only provide a partial model of framework behavior, sufficient for exposing potential security vulnerabilities to the taint analysis. F4F differs from APLT in its use of call replacements, the restricted nature of WAFL (APLT could generate arbitrary Java code as a specification), and its use of both code and configuration files to generate specifications.

8. Conclusion and Future Work

We have presented F4F, an architecture for enabling effective taint analysis of web applications based on frameworks. F4F exposes framework-related behaviors by generating specifications that can easily be processed by analysis engines. This architecture enables taint analyses to find many more framework-related security issues without compromising scalability or precision (shown in our evaluation), and support for analyzing new frameworks can be added without any modification of the core analysis engine.

In future work we plan to expand the applicability of our techniques to more static analyses and frameworks in other domains. In particular, we plan to apply similar techniques to better handle RPC constructs, distributed transaction systems, and other configuration-file-based platforms like Eclipse. We believe that the general approach of leveraging all available artifacts for static analyses with a clean architecture could significantly increase the effectiveness of the analyses on large-scale, real-world applications.

Acknowledgments We thank John Peyton, Babita Sharma, Robert Wiener and the rest of the Rational AppScan development team for their feedback on the design of F4F and their essential help in implementing the system. We also thank the anonymous reviewers for their detailed, helpful feedback.

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA Onward!*, 2009.

[2] T. Ball, V. Levin, and F. Xie. Automatic creation of environment models via training. In *TACAS*, 2004.

[3] Java SE Desktop Technologies – Java Beans. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html`.

[4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[5] S. Burbeck. Applications programming in Smalltalk-80: How to use model-view-controller (MVC). `http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html`, 1992.

[6] P. Centonze, G. Naumovich, S. J. Fink, and M. Pistoia. Role-Based Access Control Consistency Validation. In *ISSTA*, 2006.

[7] The Unified Expression Language. `http://java.sun.com/products/jsp/reference/techart/unifiedEL.html`.

[8] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the World Wide Web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[9] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *ECOOP*, 2009.

[10] Java EE at a Glance. `http://www.oracle.com/technetwork/java/javaee/`.

[11] JavaServer Pages Technology. `http://java.sun.com/products/jsp/`.

[12] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[13] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005.

[14] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[15] A. C. Myers. JFlow: practical mostly-static information flow control. In *POPL*, 1999.

[16] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[17] OWASP. Cross-site scripting. `http://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`. Accessed 16-August-2011.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[19] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, 2011.

[20] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[21] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON*, 1999.

[22] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`.

[23] Wikipedia. Comparison of web application frameworks. `http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks`. Accessed 16-August-2011.

[24] X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The case for analysis preserving language transformation. In *ISSTA*, 2006.