Dataflow Tunneling

Mining Inter-request Data Dependencies for Request-based Applications

Xiao Yu
Department of Computer Science
North Carolina State University
xyu10@ncsu.edu

Guoliang Jin
Department of Computer Science
North Carolina State University
guoliang_jin@ncsu.edu

ABSTRACT

Request-based applications, e.g., most server-side applications, expose services to users in a request-based paradigm, in which requests are served by request-handler methods. An important task for request-based applications is *inter-request analysis*, which analyzes request-handler methods that are related by *inter-request data dependencies* together. However, in the request-based paradigm, data dependencies between related request-handler methods are implicitly established by the underlying frameworks that execute these methods. As a result, existing analysis tools are usually limited to the scope of each single method without the knowledge of dependencies between different methods.

In this paper, we design an approach called *dataflow tunneling* to capture inter-request data dependencies from concrete application executions and produce data-dependency specifications. Our approach answers two key questions: (1) what request-handler methods have data dependencies and (2) what these data dependencies are. Our evaluation using applications developed with two representative and popular frameworks shows that our approach is general and accurate. We also present a characteristic study and a use case of cache tuning based on the mined specifications. We envision that our approach can provide key information to enable future inter-request analysis techniques.

KEYWORDS

web applications, request-based applications, web frameworks, inter-request analysis, tracing

1 INTRODUCTION

Server-side applications commonly expose services to users in a request-based paradigm, in which user requests are served by application-defined request-handler methods. We refer to such applications as request-based applications. Request-based applications are playing an increasingly important role in the current software ecosystem, providing classic web pages and emerging cloud-based application services. The increasing prevalence of these applications brings new demands and challenges regarding software quality, thereby calling for advanced techniques to ensure application correctness, performance, and security.

Modern request-based applications are usually built on top of some supporting frameworks, e.g., Spring [37] and Struts [5] for web applications. In the request-based execution model, upon receiving a request, the framework invokes one or more request-handler methods, and the invoked method(s) return data objects containing the necessary information to serve the request. These data objects are further processed by the framework to generate a concrete response to be sent to the user side (e.g., in the form of HTML). While each request is served modularly, users can issue a series of related requests, which may be based on the received responses (e.g., through a hyperlink generated dynamically from some data objects). Therefore, the output of one serving request-handler method could become the input of another method, creating data dependencies between these methods. We say two request-handler methods are *related* if there are data dependencies between them.

With the execution model, it is important for program analysis techniques targeting request-based applications to track data dependencies across related methods and perform *inter-request analysis*. While there is a natural analogy between inter-request analysis and interprocedural analysis [40, 43], one cannot directly apply techniques developed for interprocedural analysis for inter-request analysis. The challenges are mainly rooted in the way how data dependencies between request-handler methods are established: (1) because of the separation of server and user sides, data dependencies are not directly established by propagating data objects across methods through return values or parameters, and (2) because of the framework, there is no explicit caller-callee relationship between methods serving different requests. These complications make inter-request data dependencies not perceivable by conventional inter-procedural or summary-based analysis techniques [41, 42, 57, 58].

Existing techniques in different fields are affected by the limited capability of conducting inter-request analysis. In the performance field, a previous study [21] shows that many performance bugs are caused by skippable function calls or inefficient function-call combinations. Request-based applications are more prone to such inefficiencies, as the request abstraction is highly modular, and developers can easily write inefficient code spread across individual request-handler methods. Although various techniques [13, 14, 30, 38] have been proposed to alleviate such inefficiencies, they may not reach their full potential without inter-request capability. In the security field, investigating malicious-data propagation is crucial. Request-to-response propagation within each request is a common target for existing work [47, 53], but propagation that spans multiple

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180171

requests can also create vulnerabilities. Some work [1, 31] has discussed and tried identifying such vulnerabilities by exploring navigation sequences of web pages.

To overcome challenges and build a solid foundation for inter-request analysis on modern request-based applications, we propose an approach called *dataflow tunneling*. Our approach is based on the key observation that framework and user behaviors are of less interest in application-code analysis. For example, knowing how a framework accesses some data objects is less important than knowing what data these objects carry. Therefore, our approach effectively abstracts away the involvement of frameworks and users in propagating data across requests, and directly exposes data dependencies across request-handler methods. We define such dependencies in the form of *data-dependency specifications*, which describe how the output of a method *tunnels* (i.e., being propagated and transformed) through frameworks and users to become the input of another method. The resulting specifications provide necessary information to allow conventional analysis techniques to go across the boundaries of request-handler methods without analyzing supporting frameworks or user behaviors.

Our approach analyzes concrete application executions to capture data dependencies. We design our approach with three key techniques: (1) a static analysis on request-handler methods to identify input and output sites where these methods and their supporting framework pass data objects to each other; (2) on data objects passing through the identified sites, an object-centric tracing technique to capture object accesses and their accessed values throughout the whole processing of every single request; and (3) a mining technique to identify potential data dependencies between request-handler methods in traces from different requests. With this design, our approach adheres to the general request-based execution model instead of specific framework implementations, thereby avoiding ad hoc analyses on framework code.

We have implemented a prototype for Java-based web applications, and evaluated the specification accuracy and tracing overhead on two real-world open-source applications developed with different supporting frameworks. We have also manually inspected the generated specifications to investigate how useful they are for program understanding and future inter-request analysis techniques. The results show good accuracies (above 80%) in typical configurations, and suitable overheads for an in-house testing environment. Our manual inspection helps us gain a better understanding of the two applications. Based on the understanding, we present a use case of the specifications in tuning object caching, which shows promising results in database-query reduction and execution-time reduction. We believe these specifications can help enable future sophisticated inter-request analysis.

Overall, we make four main contributions. First, we elaborate the importance and challenges of inter-request analysis, especially on challenges posed by modern frameworks. Second, we propose an approach to infer inter-request data dependencies. One novelty of our approach is to abstract away the involvement of frameworks without human-assisted modeling. Third, we prototype and evaluate our approach on two open-source applications regarding accuracy and overhead. Fourth, we study the data-dependency specifications generated by our prototype and present a usage scenario on how they can help program understanding and future analysis techniques.

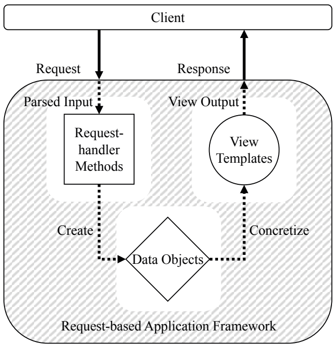

Figure 1: The abstract request-based execution model

2 BACKGROUND AND MOTIVATION

We present an extended discussion on the background of request-based applications and the challenges of inter-request data-dependency analysis. The discussion is based on a real-world example, which involves a popular Java request-based framework.

2.1 Background

Figure 1 illustrates the general request-based application execution model, which captures the commonalities among applications developed with different frameworks. This abstract model shows a single request-to-response flow. Along the flow, three major components of a request-based application are involved: (1) request-handler methods, (2) data objects, and (3) view templates.

Request-handler methods are the entry points of a request-based application to process incoming requests. For each request, the framework determines and invokes the appropriate handler method(s) through a predefined *request-handler mapping*. The request-handler mapping defines requests that each handler method can process. It is specified by developers in different forms depending on the underlying frameworks, e.g., Spring [37] uses Java annotations and Struts [5] uses configuration files. Depending on specific framework implementations, one or more request-handler methods may get invoked to process one request.

During the execution of a request-handler method, it creates *data objects* and passes them to the framework. These data objects encapsulate data that are usually retrieved from a backend database. The framework then references and uses these objects to generate concrete responses. Based on our investigation of popular frameworks for Java listed in an online source [49], the majority of them allow request-handler methods to pass data objects through either method returns or collection-based parameters, while some require request-handler methods to use framework-specific APIs. In this paper, we follow the object-passing model of the majority, but our approach can be extended to include framework-specific APIs.

View templates define how data objects should be further processed to display to the client, and such templates are usually written in different markup languages, such as FreeMarker [4], JSTL [35], and Velocity [3]. With the data objects from request-handler methods, the framework chooses a view template, either programmatically or based on a predefined *view-template mapping*, to populate a *concrete view*. Finally, the concretized view will be sent to the user as the *response*, which can be as complex as a web page or as simple as plain text that is serialized from data objects.

2.2 Motivating Example

Figure 2 shows an example from OpenMRS [33], which is an open-source web-based medical-record system. It uses the Spring Framework [37] as the application framework. The listed code includes three request-handler methods implemented in Java, with excerpts of JSP [34] files that are their corresponding view templates. The request-handler method and view template for handling requests to the path `/patientDashboard.form` are listed from lines 1 to 19. They produce an overview page for the patient specified in the request. The method `renderDashboard()` takes two parameters: (1) an integer `patientId` passed in by the Spring Framework as a request parameter, and (2) a `ModelMap` object `map` carrying data objects to the corresponding view template. The method uses `patientId` to create a `Patient` data object (line 7), and then puts it into `map` (line 8). With the method returning the string “`patientDashboardForm`” (line 10), the framework chooses the view template `patientDashboardForm.jsp` (lines 14 to 19) to produce a concrete patient-overview page. The page contains a button (line 15) that can fire another request to show some visit information of the specified patient. This new request requires the patient identifier to be embedded as a request parameter. At runtime, the Spring Framework evaluates `model.patient.patientId` (line 18) to fill in the request path (lines 17 - 18) with a concrete value. The value evaluation relies on the Java reflection mechanism to find the data object bound to `model.patient`, and invoke the accessor method bound to the field name `patientId`. In this example, the object is the `Patient` object created by `renderDashboard()`, and the accessor method is `Patient.getPatientId()`.

When a user clicks the button on the overview page, a request for the patient’s visit is fired to the request path `/admin/visits/visit`. This request is handled by the code from lines 21 to 45. Lines 21 to 27 comprise the request-handler method `showForm()`, and this method requires a `Visit` object populated by another method `getVisit()` listed from lines 28 to 38. At runtime, the Spring Framework invokes `getVisit()` before `showForm()` to create the `Visit` object with its corresponding `Patient` object from request parameters `visitId` and `patientId`. The view template `visitForm.jsp` accesses the `Visit` and `Patient` objects bound to `visit.patient` multiple times (lines 43 and 45) for different pieces of information.

An inter-request caching opportunity. We show that inter-request analysis is useful to identify an optimization opportunity. For the two requests in the example, their corresponding methods invoke `PatientService.getPatient()` (lines 7 and 35), which queries a database for patient data to create a `Patient` object. With inter-request data dependencies, one can determine that the flow (lines connected by dashed arrows) 4⟶7⟶8⟶18⟶29⟶35 propagates the value of `patientId` from the first method invocation to the second. This propagation hints that one may add a cache mechanism to cache the `Patient` object from the first invocation, so the cached object is available on the second invocation without additional queries to the database. In Section 4.6, we show a use case using Hibernate [39], a data-access framework, to achieve such caching.

```
1   @Controller
2   public class PatientDashboardController {
3     @RequestMapping("/patientDashboard.form")
4     protected String renderDashboard(@RequestParam("patientId")
            Integer patientId, ModelMap map, ...) {
5       // Get the patient
6       PatientService ps = Context.getPatientService();
7       Patient patient = ps.getPatient(patientId);
8       map.put("patient", patient);
9       ...
10      return "patientDashboardForm";
11    }
12  }

14  // patientDashboardForm.jsp
15  <input type="button"
16    onclick="window.location=
17      '/admin/visits/visit.form?patientId=
18        <c:out value=${model.patient.patientId} />'"
19    .../>

21  @Controller
22  public class VisitFormController {
23    @RequestMapping("/admin/visits/visit")
24    public String showForm(@ModelAttribute("visit") Visit visit,
            ModelMap model, ...) {
25      ...
26      return "/admin/visits/visitForm";
27    }
28    @ModelAttribute("visit")
29    public Visit getVisit(@RequestParam("visitId") Integer visitId,
            @RequestParam("patientId") Integer patientId) {
30      Visit visit = null;
31      PatientService ps = Context.getPatientService();
32      if (visitId == null && patientId != null) {
33        visit = new Visit();
34        // Get the patient and set it to the visit
35        visit.setPatient(ps.getPatient(patientId));
36      } else { ... }
37      return visit;
38    }
39  }

41  // visitForm.jsp
42  <input type="hidden" value=
43    <c:out value=${visit.patient.patientId} />
44    ... />
45  <td><c:out value=${visit.patient.personName} /></td>
```

Figure 2: An excerpt of three related request-handler methods and their view templates from OpenMRS. Only method parameters related to our discussion are shown.

The challenges of inter-request data-dependency analysis. Following the discussion in Section 1 that framework behaviors and user interactions complicate the analysis of inter-request data dependencies, we concretely discuss the challenges based on our example with the assumption that we were to extend conventional interprocedural analyses to compute inter-request data dependencies.

First, handling application components in different languages would be necessary but ad hoc. Request-handler methods and view templates are commonly developed in different languages, and concrete responses can also contain scripts that perform client-side operations to propagate data across requests. In our example, three languages are involved to propagate the `patientId` value: Java for handler methods, Expression Language [36] for evaluating concrete values on data objects in view templates, and a piece of JavaScript code (lines 16 to 18) to propagate the `patientId` value by firing the second request from the user’s browser. As data propagation goes through all the three components, ad hoc supports for different languages would be necessary to capture data dependencies between the involved request-handler methods.

Second, analyzing framework code would be necessary but challenging to capture data propagation outside request-handler methods. The framework is the caller of request-handler methods and view templates, and it provides the supports of parsing requests, binding and accessing data objects, and populating concrete views. As our example shows, the framework heavily relies on the reflection mechanism, which performs dynamic object instantiation and method invocations at runtime. For instance, the framework uses the annotation `@ModelAttribute` (line 28) to bind the `Visit` object in Java code to the view template with a textual name `visit` (line 43). It is not until the time of execution that the framework parses these annotations and textual names, and uses the reflection mechanism to invoke the parsed methods on the bound objects. It is known that this runtime mechanism brings difficulties to static analysis. Researchers have proposed different approaches to address this problem with their own pros and cons [12, 25, 28, 45, 47, 50].

3 DATAFLOW TUNNELING

Our approach requires two artifacts: the application code of request-handler methods specified by developers for analysis and test inputs for generating traces. With the two artifacts, our approach performs three key steps: (1) static analysis on request-handler methods to identify input and output sites, (2) instrumenting the methods and tracing application executions under test inputs, and (3) mining the generated traces for data-dependency specifications. The results of our approach are data-dependency specifications across request-handler methods. Such specifications describe how the output of a request-handler method is propagated and transformed into the input of another request-handler method. We use the following notations to represent data-dependency specifications:

(1) a^{method}_{type} denotes that an entity with a name a is of *type* and used in *method*. We use a special notation *obj* to denote an entity without a name, and we use *VIEW* as a special *method* to denote that the entity is used outside a request-handler method.

(2) $a \Longrightarrow b$ denotes that a is propagated to b without transformation (we omit the *method* and *type* on a and b for simplicity). There are two propagation channels: *PARAM* for propagation through a collection-based parameter, or *RETURN* for propagation through a method return. We put the propagation channel over the arrow. For channel *PARAM*, we put the parameter name under the arrow.

(3) $a \longrightarrow b$ denotes that a is transformed and then propagated to b. We put the method name related to the transformation over the arrow. Sometimes a may represent a collection of objects (e.g., a set of `Patient` objects), while b may represent only a single value (e.g., the `patientId` of one of the objects). Such cases indicate a many-to-one relation between a and b. We put the notation *CHOICE* under the arrow to denote these cases.

With these notations, the data-dependency specifications for the example in Figure 2 are:

$$obj^{renderDashboard()}_{Patient} \xRightarrow[map]{PARAM} obj^{VIEW}_{Patient} \xrightarrow{Patient.getPatientId()} patientId^{getVisit()}_{Integer} \tag{1}$$

$$obj^{getVisit()}_{Visit} \xRightarrow{RETURN} visit^{showForm()}_{Visit} \tag{2}$$

Specification (1) describes that a `Patient` object from the method `renderDashboard()` is passed through a parameter channel `map` (denoted by ⇒) to the view. Then the object is transformed by `Patient.getPatientId()` into an integer, which is propagated to the input parameter `patientId` of the method `getVisit()` (denoted by ⟶). Specification (2) describes that a `Visit` object is returned from the method `getVisit()`, and then the object is propagated to the parameter `visit` of the method `showForm()`.

3.1 Identifying Input and Output Sites

To capture data dependencies across request-handler methods, we first need to know what input each method receives and what output each method produces. The goal of this step is identifying the input and output sites in each request-handler method. An input site indicates where a handler method receives input data from the framework, and an output site indicates where the method passes output data to the framework. The identified sites allow the later instrumentation and tracing step to collect necessary information.

We apply a static taint analysis to identify input and output sites, as essentially they are sources and sinks between which data is propagated within the scope of a request-handler method. For each request-handler method, we start with a conservative set of sources and then identify sinks based on a propagation graph computed by the static taint analysis. We further filter sources and sinks using a set of rules that incorporate the request-based execution model. We consider variables in the resulting sources as input sites and variables in the resulting sinks as output sites.

We define the propagation graph as follows, followed by the rules to determine sources and sinks.

Definition 1. *A Propagation Graph is a directed graph* $PG = (V, E)$ *where* V *is a set of nodes for variables labeled by program locations, and* E *contains edges over* V*. Each edge* $v_1^{l_1} \rightarrow v_2^{l_2}$ *indicates a flow of data from variable* v_1 *at program location* l_1 *to* v_2 *at* l_2*.*

In producing such propagation graphs, we consider two types of variables as sources: (1) parameters of a request-handler method and (2) variables that receive data from static method invocations. These two sets of sources reflect two common ways that a request-handler method receives request inputs from the framework: either from parameters or through certain APIs. Starting from these source variables, we apply conventional propagation rules to compute a propagation graph for each method. In particular, we propagate tainted values starting from sources through normal assignments, and arguments and return values in method invocations. To propagate through virtual method invocations, whose method bodies cannot be determined statically for direct analysis, we introduce two heuristics. First, if b is tainted in the case of $a.m(b)$, a should also be tainted (i.e., introducing an edge $b \rightarrow a$). This rule reflects a common pattern that the object a receives data from the object b. For example, line 35 in Figure 2 is a typical case in which the `Visit` object stores the `Patient` object in a field. Second, for the case $a = b.m(c)$, a should be tainted if b or c is tainted. This is an over-approximation that aggressively creates possible flow edges when m is not available for direct analysis.

On a propagation graph constructed based on the propagation rules, the nodes without any incoming edges are likely sources, and the nodes without any outgoing edges are likely sinks. We then apply the following three rules to filter sources and sinks for input and output sites. First, the variable in a sink is an output site if it is a parameter (e.g., a collection object), in a return statement (e.g., an object returned by the method), or an argument of a static method invocation. This rule indicates that a sink variable is an output site only if the data carried by the sink variable is visible outside a request-handler method. Second, an object variable marked as both a source variable and a sink variable is likely to be an output site, as the object referenced by the variable is likely to hold new data derived from other inputs. Later such data may be accessed outside the request-handler method. Third, source and sink variables not affected by the first two rules are input sites and output sites, respectively. For example, we initially consider the parameter `map` of `renderDashboard` in Figure 2 as a source, and then we find that it acts like a sink at a later program location. With the second rule, we classify the parameter `map` as an output site.

3.2 Instrumentation and Tracing

In this step, we instrument the identified input and output sites, and then execute the application with test inputs to trace and collect necessary information for the later mining step. For each method execution, two sets of events are generated: input events and output events. Input events record concrete input values passed into request-handler methods by the framework, and output events record (1) the concrete values in request responses and (2) the method-invocation sequences representing how the framework retrieves these values from output data objects.

We specifically design this step not to ad hoc analyze or trace complex framework executions. Instead, this step focuses on data-object accesses (e.g., getting an object field). This idea is based on the observation that all framework-dependent code, regardless of languages and the use of reflection, finally has to invoke methods on data objects to retrieve concrete data. Therefore, tracing object accesses is a reasonable way to observe data propagation beyond request-handler methods and into framework and view templates.

In particular, we use a dynamic proxy-based technique on data objects. A proxy object is a type-safe delegate of its original object. We can replace a data object with its proxy object, and the proxy object can intercept method invocations made on the original object without affecting application behaviors. This technique involves two key ideas: (1) dynamically creating proxy objects to intercept and then dispatch method invocations to the original objects, and (2) propagating proxy objects based on the context of method executions to capture framework behaviors in different execution points. By dynamically creating and propagating proxy objects, we can record a sequence of methods that the framework invokes on each data object, together with concrete values returned by this sequence. This technique allows us to capture framework behaviors but effectively avoid the scalability issue of ad hoc analysis on different languages and analysis limitations caused by reflection.

Table 1 summarizes the instrumentation and tracing rules. These rules fire tracing events and create proxy objects on demand. We describe these rules in detail as follows.

At the entry and exit points of each request-handler method, we instrument `enterMethod(mName)` and `exitMethod(mName)` to record the start and end of each method execution. These points generate events *EnterMethodEvent* and *ExitMethodEvent* (*RE1* and *RE2*). We introduce these events to distinguish different method executions. They only record the name of the method being executed.

For each variable `varIn` identified as an input site, we instrument `varIn = recordInput(varIn)` at the beginning of the method. For each variable `varOut` identified as an output site, we instrument `varOut = recordOutput(varOut)` to a point that depends on the output type. If `varOut` appears in a return statement, the instrumentation is placed before the statement. In other cases, the instrumentation is placed where the method first initializes *varOut*: either at the beginning of the method if *varOut* is an out-parameter, or right after an API invocation if *varOut* receives a return object. Such instrumentation serves two purposes: recording necessary information on input and output sites, and creating proxy objects to further trace objects passed through these sites.

The instrumented input and output sites generate events of *InputEvent* (*RI1*, *RI2*) and *OutputEvent* (*RO1*, *RO2*). The information recorded in these events depends on the type of `varIn` or `varOut`: a simple value for a simple type, or an object identifier for a complex class. The instrumented sites also create proxy objects to replace the original objects in order to trace method invocations. In particular, the rules *RI2* and *RO2* describe two situations where proxy objects are created. First, if an output-site variable `varOut` references a complex object, the instrumentation creates a proxy object to replace the referenced object (*RO2*). As a result, the tracing step can trace how the request-handler method and the framework use the referenced object. Second, the instrumentation creates a proxy object on the input site that references an `HttpRequest` object (*RI2*). This is for backward compatibility with request-handler methods that directly access the `HttpRequest` object to retrieve request inputs.

With the initial proxy objects created at instrumentation sites, we discuss rules *PM1*, *PM2*, *PF1*, and *PF2*, which specify the behaviors of proxy objects when they intercept method invocations.

Rule *PM1* or *PM2* applies when a proxy object intercepts a method invocation within a request-handler method. If the proxy object is an `HttpRequest` object, we consider that it presents an input site, therefore an event of *InputAccessEvent* is fired (*PM2*). The *InputAccessEvent* represents the behavior of retrieving request input from the `HttpRequest` object, and records the return value and argument values of the method invocation.

The cases other than invocations on an `HttpRequest` object imply that the target proxy object serves as an output site of the method, and the invocation arguments are *likely* to be visible and used by the framework. Therefore, we apply *PM1* to fire an event of *OutputEvent* on each argument involved in the invocation, and replace each argument that is of a complex type with a proxy object. This rule captures output data that is passed out by the handler method, as well as propagates proxy objects to the framework to further trace runtime behaviors outside the method.

For invocations intercepted outside the scope of a request-handler method, we introduce rules *PF1* to trace an intermediate invocation that returns another complex object, and *PF2* to trace the final invocation that returns a simple value. These two rules are based on the observation that the framework may invoke a series

Table 1: Instrumentation and tracing rules. Trac Point: whether a trace event is fired on an instrumentation point (Inst) or on a proxy object (Proxy); Exec Point: the current scope of execution, in a request-handler method or the framework; Involved Statement: the instrumented code for an instrumentation point, the intercepted method invocation for a proxy point(*obj* always references a proxy object); Tracing Action and Conditions: the tracing step performs the action under certain conditions.

Rule	Trac Point	Exec Point	Involved Statement	Conditions	Tracing Action
RE1	Inst	Method	`enterMethod(mName)`		Fire an event of *EnterMethodEvent*, recording the method name $mName$.
RE2	Inst	Method	`exitMethod(mName)`		Fire an event of *ExitMethodEvent*, recording the method name $mName$.
RI1	Inst	Method	`recordInput(var)`	var is not of the type `HttpRequest`.	Fire an event of *InputEvent*, recording the value or object identifier of var.
RI2	Inst	Method	`recordInput(var)`	var is of the type `HttpRequest`.	Fire an event of *InputEvent* that records the object identifier of var, and create a proxy object for the object referenced by var.
RO1	Inst	Method	`recordOutput(var)`	var is a simple value.	Fire an event of *OutputEvent*, recording the value of var.
RO2	Inst	Method	`recordOutput(var)`	var references a complex object.	Fire an event of *OutputEvent* that records the object identifier of var, and create a proxy object for the object referenced by var.
PM1	Proxy	Method	`obj.m(arg1, ...)`		For each arg_i, fire an event of *OutputEvent*, and create a proxy object for arg_i if it is of a complex type.
PM2	Proxy	Method	`obj.m(arg)`	obj is of the type `HttpRequest`, and m has a return value.	Fire an event of *InputAccessEvent* to record arg and the returned value.
PF1	Proxy	Framework	`obj.m(...)`	m returns a complex object.	Create a proxy object to replace the returned object, and append m to the invocation sequence that leads to the current invocation.
PF2	Proxy	Framework	`obj.m(...)`	m returns a simple value.	Fire an event of *OutputAccessEvent* to record the returned value and the previously recorded invocation sequence plus m.

of methods on a complex object to retrieve a simple value (e.g., `visit.patient.patientId`). If an intercepted method invocation returns a complex object, we consider it as an intermediate step towards the final output value, and we apply *PF1* to create a new proxy object to replace the return object to further trace subsequent invocations and record the current invocation in the invocation sequence. Otherwise, if the invocation returns a simple value, we apply *PF2* to fire an event of *OutputAccessEvent*, which records the returned value, as well as the sequence of invocations recorded along with all the intermediate invocations plus the final one.

Overall, this tracing step produces a trace for each request. Each trace is in the form of *(EnterMethodEvent (InputEvent | InputAccessEvent | OutputEvent)* ExitMethodEvent OutputAccessEvent*)**, indicating that zero or more request-handler methods are executed for one request. During the execution of each method, there can be zero or more *InputEvent*, *InputAccessEvent*, and *OutputEvent*. After each method execution, there can be zero or more *OutputAccessEvent*. For every access event (*InputAccessEvent* or *OutputAccessEvent*) in a trace, the same trace must have one corresponding source event (*InputEvent* or *OutputEvent*) before the access event, and the source event records information about the accessed root object.

An example with Figure 2. We use `renderDashboard()` in Figure 2 as an example to show how the analysis and tracing steps work. The static-analysis step first identifies the parameter `patientId` as a source variable and the parameter `map` as a sink variable. Then we instrument two lines of code `patientId = recordInput(patientId)` and `map = recordOutput(map)` at the beginning of the method. We also instrument events `enterMethod` and `exitMethod` at the beginning and end of the method, respectively.

In the tracing step, the execution of `renderDashboard()` yields a trace consisting of *EnterMethodEvent*, *InputEvent* on `patientId`, *OutputEvent* on `map`, *OutputEvent* on the `Patient` object, *ExitMethodEvent*, *OutputAccessEvent* representing a method invocation of `Patient.getPatientId`. The *InputEvent* is triggered by *RI1*, and the first *OutputEvent* by *RO2*. A proxy object for `map` is created to replace the original object during this process. Then this proxy object fires the second *OutputEvent* by *PM1* when it intercepts `map.put()`. It also creates and passes a proxy object for the original `Patient` object into `map`. When the framework executes `patientDashboardForm.jsp`, the proxy object for the `Patient` object intercepts an invocation `Patient.getPatientId()`, and fires *OutputAccessEvent* by *PF2* as the method returns a simple value.

3.3 Specification Mining

Based on the trace structure defined in Section 3.2, each trace represents the processing of a single request. We first leverage the user-session information collected along tracing to distinguish and group traces of requests from different user sessions. We then use a k-length sliding window algorithm to infer data-dependency specifications from traces from the same user session. Our algorithm maintains a k-length sliding window because data dependencies may exist across non-adjacent requests. For example, the two requests mentioned in Section 2.2 are actually not adjacent, which is separated by a background request triggered by the first patient-overview page. In practice, data dependencies are unlikely to exist between methods of a long request distance, and we expect the sliding window size to be small. One can determine a proper window size in different ways, depending on the environment where the application under analysis is running. In an in-house testing setting, one may repeat test runs and gradually increase the window size until no true specification is found. In a production environment where requests and workloads are not repeatable, one may use the windows size determined during in-house testing or heuristically adjust the size online, and we leave the latter for future work.

Algorithm 1 presents an outline of the k-length sliding window algorithm. For each incoming trace t, the algorithm pairs t with $k - 1$ previous traces from the same user session as well as t itself, and applies the subroutine `MineSpec` on these pairs. Mining each of these trace pairs yields specifications that describe data dependencies between request-handler methods in the paired traces. Note that mining the reflective pair (t, t) yields specifications when multiple request-handler methods are involved in processing the

Algorithm 1: Mining Traces with a K-Length Sliding Window

```
Input: A trace $t$ from a trace stream of the application
Global: A configurable integer $k$, a map $M$ associating every session
        identifier with a list storing up to $k - 1$ previous traces, and a
        database $DB$ storing existing data-dependency specifications
$list \leftarrow M$[GetSessionId($t$)];
AppendLast($list$, $t$);
foreach $t'$ in $list$ do
    $methodPairs$, $specs \leftarrow$ MineSpec($t'$, $t$);
    foreach $p$ in $methodPairs$ do
        UpdateMethodPairStats($p$, $DB$);
    foreach $s$ in $specs$ do
        UpdateSpecDB($s$, $DB$);
if Size($list$) = $k$ then
    RemoveFirst($list$);
```

same request. From the subroutine MineSpec, the algorithm gets a set of method pairs and a set of specifications. These pairs cover request-handler methods between which the algorithm tries to identify data dependencies. The algorithm stores these pairs (via UpdateMethodPairStats()) for the statistical purpose of calculating and updating the *confidence* of each mine specification via UpdateSpecDB(). For a specification s, we define its *confidence* as the frequency of having the specification s identified between two request-handler methods when these two methods appear together in different trace pairs. We use confidence as an indicator of the likelihood that a specification is true and worth further analysis.

Algorithm 2 shows the subroutine MineSpec, which does the actual work to infer data dependencies across two methods. The basic idea of this algorithm is to check whether some output values from the trace of one request and some input values from the trace of another request are equal. A pair of equal values indicates a possible inter-request data dependency. The adoption of this equivalence relation is backed by the observations that (1) unique values likely exist to approximate the actual dependencies, and (2) these values are usually carried from the output of one request to the input of another without complex computations in between. For example, the value of patient.patientId would be unique for every patient, and the value is not involved in any complex computation.

In particular, Algorithm 2 takes two traces t_1 and t_2 to perform the following three major steps. First, it extracts output and input related events from t_1 and t_2. The extracted events include *OutputEvent* and *OutputAccessEvent*, indicating the output of request-handler method(s) in t_1, and *InputEvent* and *InputAccessEvent*, indicating the input of method(s) in t_2. Since *OutputAccessEvent* and *InputAccessEvent* record method-invocation sequences, the same invocation sequence in different events essentially reflects either repetitive or iterative data-access actions, which may indicate a many-to-one relation between the output and input of the request-handler methods (i.e., the *CHOICE* case in specifications). Therefore, the algorithm clusters events that record an identical invocation sequence (via ClusterOutputEvents() and ClusterInputEvents()) to form an event set, which is used in later steps as a whole to represent identical object-access actions.

Given the clustered input and output events, the second step constructs a matching table to compute potential data dependencies using concrete values recorded in these events. As shown below, the rows of the table represent clustered output events in trace t_1,

Algorithm 2: MineSpec on a Trace Pair

```
Input: Two traces $t_1$ and $t_2$
Output: A set $methodPairs$ storing pairs of request-handler methods, and
        a set $specs$ storing derived data-dependency specifications
1  $methodPairs \leftarrow$ [];
2  $specs \leftarrow$ [];
3  $oEventList \leftarrow$ ClusterOutputEvents($t_1$);
4  $iEventList \leftarrow$ ClusterInputEvents($t_2$);
5  $matchTable \leftarrow$ ConstructMatchTable($oEventList$, $iEventList$);
6  for $i \leftarrow 0$; $i <$ Size($oEventList$); $i++$ do
7      for $j \leftarrow 0$; $j <$ Size($iEventList$); $j++$ do
8          $matchType \leftarrow matchTable[i][j]$;
9          if $matchType$ is not no-match then
10             $s \leftarrow$ CreateSpec($matchType$, $oEventList[i]$,
                 $iEventList[j]$);
11             Add($specs$, $s$);
12         $method1 \leftarrow$ GetMethodName($oEventList[i]$);
13         $method2 \leftarrow$ GetMethodName($iEventList[j]$);
14         Add($methodPairs$, ($method1$, $method2$));
15 return $methodPairs$, $specs$;
```

and the columns represent clustered input events in trace t_2.

	t_2Event_{Input1}	t_2Event_{Input2}
$t_1Event_{Output1}$	*no-match*	*one-to-one*
$t_1Event_{Output2}$	*no-match*	*no-match*
$t_1Event_{Output3}$	*many-to-one*	*no-match*
$\cdots$	$\cdots$	$\cdots$

For each cell of the table, the algorithm extracts concrete values from the events, and checks for matched values, resulting in one of three types of outcomes: (1) no value is matched, then the algorithm marks the cell as *no-match*; (2) a single value is matched, and the output and input events each contains only one concrete value, then the algorithm marks the cell as *one-to-one*; (3) a single value is matched, and the output event contains multiple concrete values, then the algorithm marks the cell as *many-to-one*. We currently do not consider the case of *many-to-many*, as it is not commonly observed, but our approach can be extended to support such cases.

With a constructed matching table, the algorithm goes over each cell in the table and creates a specification for the cell that indicates a potential dependency (lines 6 - 14 in Algorithm 2). In particular, CreateSpec() (line 10) identifies (1) output and input entities by tracking back from *OutputAccessEvent* or *InputAccessEvent* to its corresponding *OutputEvent* or *InputEvent*, (2) the channel (⇒) that propagates an output object to the framework by determining whether *OutputEvent* is from a return statement or an out-parameter, and (3) the transformation actions (⟶) by extracting the invocation sequence from *OutputAccessEvent* or *InputAccessEvent*. With the matching type from each cell, the algorithm creates a specification describing how an output object of a method is propagated and transformed into the input of another method.

4 EVALUATION

We have built a prototype of our approach for Java-based web applications. We use Soot [11] for static analysis and instrumentation, and Byte Buddy [56] for dynamic proxy creation. With the prototype, we evaluated the following aspects of our approach.

Specification Accuracy. We measure how accurate the generated specifications are in describing data dependencies between request-handler methods. We consider a specification to be true if it reveals a real data dependency between the involved methods but not due to coincidentally equal values, and we manually examine and label each specification based on our understanding of the subjects' code and runtime behaviors.

Tracing Overhead. We measure the runtime overhead imposed by the tracing step. Measuring the tracing overhead helps us determine the applicability of our approach in various scenarios. A manageable overhead would allow us to run more tests to achieve a better coverage. If the measured overhead is low enough and suitable in a production environment, we can apply our approach in the field to extract specifications based on real-world workload.

Characterization and Application. We manually study all the generated specifications to understand their characteristics and how they can facilitate program understanding and future inter-request analyses. We discuss the lessons learned and the experience of using the specifications for tuning object cache policies.

4.1 Subjects

We evaluated our approach on two open-source Java-based web applications: ITracker [20] (version 3.3.2), an issue-tracking system, and OpenMRS [33] (version 1.11.5), a medical-record system.

ITracker has about 37,000 lines of Java code and about 8,000 lines of view-template code. It uses Struts [5] as the supporting framework. The request-handler methods defined under Struts follow the convention that method inputs and outputs are passed through special objects, such as `HttpRequest` objects. For the evaluation, we randomly generated a data set containing 10 developers, 20 projects, and 200 bugs randomly distributed among the 20 projects.

OpenMRS has about 204,000 lines of Java code and about 25,000 lines of view-template code. It uses Spring [37] as the supporting framework, and it contains request-handler methods defined in three different forms: (1) methods using special objects to pass inputs and outputs, (2) a workflow-based model requiring multiple methods to process a single request (e.g., a method to create a view and another method to create data objects), and (3) a modern model passing method inputs and outputs directly through method parameters by simple values and collection-based objects. We use an anonymized data set with 5,000 patients and 500,000 observations.

These two applications use two widely adopted request-based frameworks in a non-trivial way. The two frameworks have different programming interfaces and paradigms, which lead to differences in the implementation of request-handler methods. We use the two applications to show the generality and usefulness of our approach on applications using different frameworks.

4.2 Experiment Design

To generate concrete application executions, we have implemented a random workload generator using Selenium [46], a web-browser based testing framework. The generator drives a web browser to simulate user actions on web pages, such as clicking hyperlinks. These actions can trigger requests to the applications. In particular, the generator identifies actionable elements on each page, and then performs an action on a randomly selected element. The identified elements include hyperlinks, web forms, input elements, and so on. For each element, the generator applies a corresponding action: clicking for a hyperlink or a button, and generating random but domain-specific data for a web form. With these capabilities, the generator can keep running to cover a wide range of request sequences on the subject applications. Given the random nature of the generator, the workload it generates may not cover all possible real-world request sequences. Therefore, our results need to be interpreted with the random workload in mind.

Table 2: Specification accuracy of ITracker and OpenMRS

Subject (Window Size)	# of Specs	# of True Specs	Accuracy
ITracker (reflective)	0	0	N/A
ITracker (2)	64	54	84.38%
ITracker (3)	16	14	87.50%
ITracker (4)	2	0	0.00%
ITracker (5)	0	0	N/A
ITracker (Total)	82	68	82.92%
OpenMRS (reflective)	15	15	100.00%
OpenMRS (2)	20	17	85.00%
OpenMRS (3)	32	20	62.50%
OpenMRS (4)	11	1	9.09%
OpenMRS (5)	12	0	0.00%
OpenMRS (Total)	90	53	58.89%

We configure the generator with 10 distinct random seeds to exercise different request sequences in the subjects. For each seed configuration, we run the generator 10 times. Each run starts with the same database state and consists of 500 simulated user actions. Our results show that this setup allows our approach to discover most of the possible and valid specifications. Although we aim at producing repeatable results, the generator may encounter nondeterministic and unexpected behaviors from Selenium and the network. So we choose a relatively small number of actions for each run to reduce the potential divergences on the request sequences, which are supposedly fixed by each random seed, and repeatedly run each configuration to get stable results. Nevertheless, our reported results may still be affected the likely nondeterminism.

To evaluate specification accuracy, we use a sliding window with an increasing size to process the generated traces. Starting from size two, we keep increasing the window size until our manual inspection determines that the increased window does not yield new true specifications not seen in smaller windows. In the results, we only consider specifications observed at least 100 times with a confidence (defined in Section 3.3) of at least 0.2. These thresholds are chosen to ensure that most true specifications would likely be reported, while most false specifications would likely be filtered out. We consider the reported results worth further inspection.

For tracing overhead, we record the processing time for each request on the server side during experiment runs. Then we calculate the overhead by comparing the recorded time of corresponding requests on the instrumented and uninstrumented subjects. We include requests from repeated runs to amortize the likely nondeterminism and performance anomaly.

4.3 Specification Accuracy

Table 2 presents the accuracy results for ITracker and OpenMRS in different sliding-window sizes. Each row shows the statistics of unique specifications that are not found in smaller windows. The column "# of Specs" shows the total specifications in the window of size shown by the first column, and "# of True Specs" shows the total true positives that we manually identified. Note that a specification

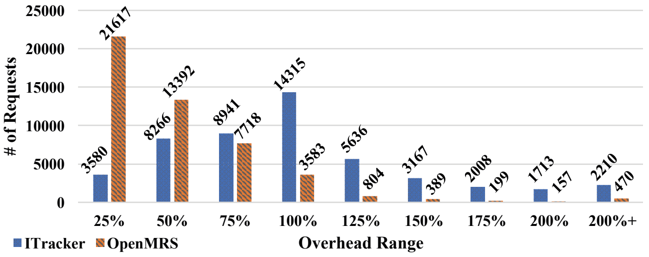

Figure 3: Tracing overhead by requests

may be found in windows of different sizes, so we attribute it to the smallest window. For example, "ITracker (3)" shows statistics of the specifications that can only be found in window size three or above, and "OpenMRS (reflective)" indicates specifications found between request-handler methods that process the same request. As a result, we observe decreased numbers of specifications in larger windows.

The results show that our approach is effective and can reach a balance between testing efforts and result accuracy with the reporting threshold. We find that increasing the window size does produce more specifications. The additional specifications are either infrequent or not discovered under smaller windows. On the other hand, a larger window introduces more false positives, which lower the overall accuracy. As we can observe from the results, a break point exists from which we may not have more true unique specifications. For instance, the accuracy of ITracker is steady under windows of sizes two and three, but no new true specifications are found in larger windows. This observation suggests that most specifications can be found in small windows, and we can determine the proper window size by gradually increasing the mining window.

We also look into the false positives to investigate the causes. In our mining step, any coincidentally equal output and input values would lead to false positives. Confidence values can mitigate such false positives to some extent. However, a specification with a confidence of 100% does not necessarily mean a true positive, e.g., some of the false positives in OpenMRS have very high confidence values. This is usually due to a combination of overlapped value ranges and insufficient data points of different kinds of data objects. For instance, multiple distinct data objects in OpenMRS, such as hospital locations and question forms, have the same range of identifier values, and sometimes appear in the same output. When only a small number of such data objects exist in the database, the chance of overlapped identifiers increases. Using more diverse data sets (e.g., real-world data sets) can mitigate such problems.

4.4 Tracing Overhead

The tracing overhead on the total execution time from all experiment runs is 74.96% or ITracker, and 33.43% for OpenMRS. Figure 3 presents the overhead results by requests. Each bar represents the number of requests whose overhead falls into the range indicated by the x-axis, e.g., 100% means the range from 75% (exclusive) to 100% (inclusive). Due to the space limit, we aggregate all the requests whose overhead is above 200% into the last bar. Also note that the total number of requests for each subject is not exactly 50,000, because some user actions may not trigger requests, and some web pages may trigger background requests automatically.

Overall, the results show that the overhead of nearly 83% of all requests falls within 100%. Given the current overhead measurements, we conclude that our approach is suitable for in-house testing and feasible to be selectively applied in a production environment with less strict performance requirements.

We also observe some requests with high overhead that is related to the large number of object accesses during request processing. In ITracker, a frequent request path of high overhead is `/list_issues` with a median overhead of 166.67%. To show a list of bugs, the view template accesses every bug object multiple times to retrieve the bug identifier, name, and other information, with an average of 695 method invocations per request on those objects. The situation is similar in OpenMRS, e.g., `/concept.htm` with a median overhead of 75.57% and 488 invocations per request on average.

4.5 Characterization

The mined specifications allow us to investigate common characteristics of data dependencies across requests in the level of request-handler methods. We next discuss our findings regarding the propagation and use of inter-request data.

Among 121 true specifications, 86 of them show the propagation of entity identifiers, such as the identifier of a patient in OpenMRS or a project in ITracker. OpenMRS and ITracker store domain-specific entities with identifiers in databases, and create data objects representing these entities when accesses are needed. To achieve certain features, these applications may need the same entity in processing multiple related requests. For instance, ITracker needs the same piece of project information in processing requests `/list_issues`, which lists all issues in the project, and `/view_issue`, which shows a particular issue. To share information across requests, these applications choose to propagate entity identifiers instead of data objects. On incoming requests, request-handler methods receive these identifiers and then use them to recreate the corresponding data objects from the database.

For the rest of the specifications, 15 of them represent data dependencies between request-handler methods that process the same request (e.g., `getVisit()` and `showForm()` in Figure 2). As such methods are executed in the same context without going across the client side, they directly propagate data objects to the supporting framework, which further passes the objects to other methods.

The remaining 20 specifications involve values for pagination and flags, and the propagated objects are not instances of application-defined classes. Examples include an integer value indicating the number of total data entries, which allows a method to calculate how many remaining entries need to be displayed, and a string instructing the next action of a receiving method.

4.6 Application: Tuning Cache Policies

As revealed by our characterization study, entity identifiers derived from data objects are common in inter-request data propagation. The request-handler methods tend to use the propagated identifiers to recreate the corresponding data objects. Such behaviors lead to repetitive object creation, which can be expensive, because it usually involves database queries.

The mined specifications can help pinpoint repetitive object creation between related request-handler methods. For the example in Figure 2, a hypothetical analyzer could work in the following way: (1) applying dataflow analysis starting from renderDashboard() to identify the creation of the Patient object (line 7) and its output site (line 8), (2) using the specification on renderDashboard() and getVisit() as a summary to confirm that the parameter patientId is an alias of patientId retrieved from the Patient object, (3) continuing the analysis into getVisit() to identify the creation of another Patient object using the same patientId (line 35). The analyzer could then infer that the two Patient objects represent the same entity, and suggest an optimization strategy, e.g., caching the first Patient object.

We follow the methodology of this hypothetical analyzer to manually optimize repetitive object creation in our evaluation subjects by leveraging the object cache built in Hibernate [39]. Hibernate is used by both subjects as their data-access layer, and its cache is controlled by configuration files specifying classes of objects to be cached and their caching lifetimes.

We first rank the mined specifications by their observed times to identify data objects whose identifiers are frequently propagated across requests. The top identified objects are: Issue and Project for ITracker, and Concept, Obs (Observation), Patient, and Visit for OpenMRS. In ITracker, Issue and Project objects are not cached, and we directly create a new long-term cache policy for them. In OpenMRS, there is a default cache policy applying a long-term cache policy with a 12,000-second lifetime for Concept objects and a short-term policy with a 30-second lifetime for Patient objects. However, the cache of Concept objects is insufficient, as the objects referenced by Concept objects (e.g., ConceptName and ConceptDescription) are not included in the default policy. In addition, the lifetime for Patient objects is too short for multiple requests of related patient features. Therefore, we amend the default cache policy to include Obs and Visit objects, objects referenced by Concept objects, and change the policy for Patient objects to be long-term.

We run the same experiment described in Section 4.2 with and without the updated cache policies, and we measure both database queries and execution time. A reduction in database queries indicates that fewer data objects are created by querying data from a database, as the required objects are cached in the application. The results show an average query reduction of 8.50% for ITracker and 5.05% for OpenMRS under our random workload, and the reduction in the total execution time is 3.32% for ITracker and 2.80% for OpenMRS. The presented execution time is measured in an experimental setup where the database server and application server are deployed on the same machine. We expect a greater time reduction in a realistic setting where the network latency in accessing the database introduces more overhead in the overall execution time. On the other hand, the query count will remain stable, as it is less sensitive to possible instabilities in the execution environment.

5 RELATED WORK

Analysis of web applications. There is a wide range of work on web applications, such as testing [2, 7, 24, 26, 27, 44, 54, 59], program slicing [32, 51, 53], and security analysis [1, 19, 31, 47, 52, 55]. The proposed approaches perform their analysis based on some forms of data- or control-flows.

Some of the approaches [1, 6, 7, 26, 32, 51] extract page navigation and/or data-flows across web pages. These approaches commonly rely on a language-specific model to analyze program code (e.g., PHP) that directly generates dynamic page content. With a modernized point of view on framework-based web applications, our approach focuses on request-handler methods and their data dependencies, eliminating the involvement of web pages. We make this choice because framework-based web applications are popular nowadays and they expose request-handler methods as application entry points instead of web pages.

Some other approaches, such as TAJ [53], F4F [47], and ANDROMEDA [52], perform analysis on framework-based web applications. These approaches require framework-specific modeling, and treat each request-handler method as a single entry point of analysis. On the contrary, our approach is not tied to specific framework implementations, and can enhance these approaches by providing inter-request data dependencies for end-to-end analysis.

Program specification mining and its applications. Existing work has shown program specifications inferred from traces to be useful in program analysis, such as data-flow specifications [15, 17] for complex libraries and dynamic language features, and specifications in finite state machines [9, 10, 18, 22, 23, 29] for component interactions, API usages, and temporal invariants.

With the purpose of inter-request analysis, our inferred specifications describe data dependencies between request-handler methods that are modularized by underlying frameworks. Our evaluation has shown a use case of tuning cache policies to reduce database operations. Future work can use our specifications to automate such optimization and perform more fine-grained analysis in combination of many existing approaches [30, 38, 48]. For instance, Tamayo *et al.* [48] propose an approach to construct dependency graphs for database operations. Ramachandra *et al.* [38] propose an approach to identify database operations in the interprocedural scope to enable prefetching. Our specifications can enable these approaches to perform inter-request analysis for more optimization opportunities. As another example, we also envision that our specifications can allow information-flow analysis to detect security vulnerabilities and malicious flows across multiple requests.

Dynamic taint tracking. Our tracing approach is similar to dynamic taint tracking, which incurs a non-negligible runtime overhead. Dytan, a generic taint tracking system [16], reports an overhead from 3000% to 5000%. Phosphor, a recent system for Java [8], incurs an average overhead of 53.31% and 220% at worst. We consider the overhead of our approach to be on a par with these systems.

6 CONCLUSION

We present *dataflow tunneling*, which derives data-dependency specifications between request-handler methods for request-based applications. The outcome specifications can enable *inter-request analysis*, which allows program analysis to work in the scope across request-handler methods. As future work, we plan to develop inter-request analysis techniques that can leverage the outcome specifications to improve the quality of request-based applications.

REFERENCES

[1] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. 2016. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 641–652. https://doi.org/10.1145/2976749.2978380

[2] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. 2010. Scalability Issues with Using FSMWeb to Test Web Applications. *Information and Software Technology* 52, 1 (Jan. 2010), 52–66. https://doi.org/10.1016/j.infsof.2009.06.002

[3] Apache. 2010. The Apache Velocity Project. (2010). Retrieved March 8, 2017 from http://velocity.apache.org/

[4] Apache. 2016. Apache FreeMarker. (2016). Retrieved March 8, 2017 from http://freemarker.org/

[5] Apache. 2017. Apache Struts. (2017). Retrieved March 8, 2017 from https://struts.apache.org

[6] Snigdha Athaiya. 2017. Inferring Page Models for Web Application Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 412–415. https://doi.org/10.1145/3092703.3098240

[7] Snigdha Athaiya and Raghavan Komondoor. 2017. Testing and Analysis of Web Applications Using Page Models. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 181–191. https://doi.org/10.1145/3092703.3092734

[8] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[9] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-inference Algorithms Through Declarative Specification. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 252–261. http://dl.acm.org/citation.cfm?id=2486788.2486822

[10] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 267–277. https://doi.org/10.1145/2025113.2025151

[11] Eric Bodden. 2017. Soot - A framework for analyzing and transforming Java and Android Applications. (2017). Retrieved April 15, 2017 from https://sable.github.io/soot/

[12] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[13] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *Proceedings of the VLDB Endowment* 5, 11 (July 2012), 1471–1482. https://doi.org/10.14778/2350229.2350262

[14] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. Sloth: Being Lazy is a Virtue (when Issuing Database Queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 931–942. https://doi.org/10.1145/2588555.2593672

[15] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: Mining Explicit Information Flow Specifications from Concrete Executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2771783.2771810

[16] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490

[17] Tom Deering, Ganesh Ram Santhanam, and Suresh Kothari. 2015. FlowMiner: Automatic Summarization of Library Data-Flow for Malware Analysis. In *Proceedings of the 11th International Conference on Information Systems Security - Volume 9478 (ICISS 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 171–191. https://doi.org/10.1007/978-3-319-26961-0_11

[18] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 339–349. https://doi.org/10.1145/1453101.1453150

[19] Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-Based Taint Analysis for Java Web Applications. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*. Springer-Verlag New York, Inc., New York, NY, USA, 140–154. https://doi.org/10.1007/978-3-642-54804-8_10

[20] itracker. 2016. itracker. (2016). Retrieved March 8, 2017 from http://itracker.sourceforge.net/

[21] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[22] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 178–189. https://doi.org/10.1145/2635868.2635890

[23] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. 2010. Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 179–182. https://doi.org/10.1145/1810295.1810324

[24] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 449–459. https://doi.org/10.1145/2635868.2635913

[25] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 27–53. https://doi.org/10.1007/978-3-662-44202-9_2

[26] Chien-Hung Liu. 2006. Data Flow Analysis and Testing of JSP-based Web Applications. *Information and Software Technology* 48, 12 (Dec. 2006), 1137–1147. https://doi.org/10.1016/j.infsof.2006.06.003

[27] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. 2000. Object-Based Data Flow Testing of Web Applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00) (APAQS '00)*. IEEE Computer Society, Washington, DC, USA, 7–16. http://dl.acm.org/citation.cfm?id=786446.786478

[28] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 139–160. https://doi.org/10.1007/11575467_11

[29] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic Generation of Software Behavioral Models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 501–510. https://doi.org/10.1145/1368088.1368157

[30] Amit Manjhi, Charles Garrod, Bruce M. Maggs, Todd C. Mowry, and Anthony Tomasic. 2009. Holistic Query Transformations for Dynamic Web Applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE Computer Society, Washington, DC, USA, 1175–1178. https://doi.org/10.1109/ICDE.2009.194

[31] Michael Martin and Monica S. Lam. 2008. Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. USENIX Association, Berkeley, CA, USA, 31–43. http://dl.acm.org/citation.cfm?id=1496711.1496714

[32] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2015. Cross-language Program Slicing for Dynamic Web Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 369–380. https://doi.org/10.1145/2786805.2786872

[33] OpenMRS. 2016. OpenMRS. (2016). Retrieved March 8, 2017 from http://openmrs.org

[34] Oracle. 2013. JavaServer Pages Technology. (2013). Retrieved March 8, 2017 from http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[35] Oracle. 2013. JSP Standard Tag Library. (2013). Retrieved March 8, 2017 from https://jstl.java.net/

[36] Oracle. 2013. Unified Expression Language. (2013). Retrieved March 8, 2017 from https://uel.java.net/

[37] Pivotal. 2017. Spring Framework. (2017). Retrieved March 8, 2017 from https://projects.spring.io/spring-framework/

[38] Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2213836.2213852

[39] RedHat. 2017. Hibernate ORM. (2017). Retrieved March 8, 2017 from http://hibernate.org/orm/

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[41] Atanas Rountev, Scott Kagan, and Thomas Marlowe. 2006. Interprocedural Dataflow Analysis in the Presence of Large Libraries. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 2–16. https://doi.org/10.1007/11688839_2

[42] Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE Dataflow Analysis in the Presence of Large Object-oriented Libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 53–68. http://dl.acm.org/citation.cfm?id=1788374.1788380

[43] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT '95)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 131–170. http://dl.acm.org/citation.cfm?id=243753.243762

[44] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. 2004. A Scalable Approach to User-Session Based Testing of Web Applications Through Concept Analysis. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. IEEE Computer Society, Washington, DC, USA, 132–141. https://doi.org/10.1109/ASE.2004.6

[45] Jason Sawin and Atanas Rountev. 2009. Improving Static Resolution of Dynamic Class Loading in Java Using Dynamically Gathered Environment Information. *Automated Software Engineering* 16, 2 (June 2009), 357–381. https://doi.org/10.1007/s10515-009-0049-9

[46] Selenium. 2016. Selenium WebDriver. (2016). Retrieved March 8, 2017 from http://www.seleniumhq.org/projects/webdriver/

[47] 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

[48] Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. 2012. Understanding the Behavior of Database Operations Under Program Control. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 983–996. https://doi.org/10.1145/2384616.2384688

[49] TechEmpower. 2016. Framework Benchmarks for Java Web Applications. (2016). Retrieved March 8, 2017 from https://github.com/TechEmpower/FrameworkBenchmarks/tree/master/frameworks/Java

[50] Andreas Thies and Eric Bodden. 2012. RefaFlex: Safer Refactorings for Reflective Java Programs. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2338965.2336754

[51] Paolo Tonella and Filippo Ricca. 2005. Web Application Slicing in Presence of Dynamic Code Generation. *Automated Software Engineering* 12, 2 (April 2005), 259–288. https://doi.org/10.1007/s10515-005-6208-8

[52] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

[53] 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

[54] D. A. Turner, M. Park, J. Kim, and J. Chae. 2008. An Automated Test Code Generation Method for Web Applications Using Activity Oriented Approach. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 411–414. https://doi.org/10.1109/ASE.2008.61

[55] Gary Wassermann and Zhendong Su. 2007. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/1250734.1250739

[56] Rafael Winterhalter. 2017. Byte Buddy. (2017). Retrieved April 15, 2017 from http://bytebuddy.net/

[57] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Rethinking Soot for Summary-based Whole-program Analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP '12)*. ACM, New York, NY, USA, 9–14. https://doi.org/10.1145/2259051.2259053

[58] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 221–234. https://doi.org/10.1145/1328438.1328467

[59] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. 2014. Virtual DOM Coverage for Effective Testing of Dynamic Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 60–70. https://doi.org/10.1145/2610384.2610399