AutoWeb: Automatically Inferring Web Framework Semantics via Configuration Mutation

Haining Meng[1,2], Haofeng Li[1], Jie Lu[1], Chenghang Shi[1,2], Liqing Cao[1,2], Lian Li[1,2,3] (✉), and Lin Gao[4]

[1] SKLP, Institute of Computing Technology, CAS, China
{menghaining, lihaofeng, lujie, shichenghang21s, caoliqing19s, lianli}@ict.ac.cn
[2] University of Chinese Academy of Sciences, China
[3] Zhongguancun Laboratory, China
[4] TianqiSoft Inc, China gaolin@tianqisoft.cn

Abstract. Web frameworks play an important role in modern web applications, providing a wide range of configurations to streamline the development process. However, the intricate semantics, facilitated by framework configurations, present substantial challenges when conducting static analyses on web applications. To mitigate this issue, existing approaches resort to manually modeling framework semantics for static analysis tools. Unfortunately, these manual works are both time-consuming and error-prone, especially considering the vast array of web frameworks and their frequent updates.

In this paper, we present the first automated method for inferring web framework semantics. Our innovative approach can automatically deduce framework specifications by mutating configurations. We have developed a prototype called AutoWeb and performed extensive experiments on three popular Java web frameworks. The empirical results show that AutoWeb is comparable to these manual approaches in terms of precision, with a false negative rate of 8.2% and no false positives.

Keywords: static analysis · framework modeling · web framework · Java.

1 Introduction

Modern web applications are commonly built on top of web frameworks. Those frameworks (e.g., Spring [25] and Spring Boot [24]) offer high-level abstractions for common web tasks, thereby greatly simplifying the development process. For instance, many frameworks provide concepts such as *controllers* in the popular model-view-controller (MVC) design pattern: developers can simply declare handler methods for a particular URL without knowing the intricate request dispatching mechanism, and the framework is responsible for processing incoming requests and dispatch them to corresponding handlers.

To effectively analyze web applications, it is crucial to precisely model the semantics of their underlying frameworks, representing the possible frameworks

behaviors at runtime. Otherwise, many existing static analyses become inapplicable. For instance, web applications are driven by frameworks to interact with user requests, and there is no `main` method to start with. Furthermore, existing static analyses often fail to capture the dynamically introduced points-to relations and call relations that arise from dependency injection and dynamic dispatching mechanisms within frameworks, resulting in unsound and imprecise results. Unfortunately, frameworks are notoriously hard to analyze statically. They often employ hard-to-analyze dynamic patterns (e.g. reflection) to interact with application code, and their concrete semantics are customized by the application via configuration files or annotations. It is a daunting task, if not impossible, to automatically analyze frameworks with good precision.

In practice, researchers resort to manually modeling framework semantics to analyze web applications. Framework features were hard-coded in the analysis implementation [29,5,21]. In addition, researchers have proposed several more general solutions to specify framework semantics effectively modeling behaviors under given configurations [26,1]. For instance, IBM's F4F [26] defined the Web Application Framework Language (WAFL) to express framework-related behaviors for specific web applications, where WAFL specifications are generated by hand-crafted generators (one for each framework). JackEE [1] declares framework-related behaviors using Datalog rules, effectively mapping framework configurations to static relations, which can be processed by the Doop [4] analysis engine. Nevertheless, those existing approaches still need manual efforts and can be labor-intensive and error-prone, particularly when frameworks undergo frequent updates.

This paper presents an automatic method for inferring web framework semantics. To the best of our knowledge, this is the first approach of such an attempt. Since it is generally infeasible to directly obtain the concrete semantics by analyzing the complex implementation details of web frameworks, we do not consider various framework-specific concepts such as *filters* and *controllers*. Instead, we focus on the relations that affect the application code, which are framework-introduced but commonly required by static analyses, i.e., entry points, points-to relations, and call relations[5]. Previous work [30,27] has also shown that the above relations are crucial for the static analysis of web applications. Web frameworks provide developers with hundreds of configuration parameters, allowing for the customization of applications that exhibit rich semantics. Our goal is to automatically deduce the semantics under given configurations, i.e., demystifying how entry methods and call/points-to relations are introduced by frameworks using particular configuration parameters. The framework semantics are abstracted as mappings from configuration parameter sets to relation types, referred to as *specification* in this paper. Specifications are framework-related, yet application-independent, and can be applied to specific applications to model framework semantics.

Based on the observation that a framework-introduced relation is declared by the *minimal sufficient and necessary set* (MSNS), which is the set of minimum

[5] Techniques described in this paper are applicable to infer other user-defined relations.

configuration parameters to trigger this relation, we propose a *mutation-based approach* to identifying the MSNS for each relation. Specifically, given an application program construct P with framework-introduced relation R, our approach first identifies the *necessary* condition of relation R by removing configuration parameters from P until R cannot be triggered at runtime. Then, new configuration parameters (mutated from identified necessary conditions) are introduced to further verify that the set of necessary configuration parameters is *sufficient* to trigger relation R.

We develop a prototype tool, AutoWeb, to demonstrate the effectiveness of our approach. AutoWeb observes framework-introduced relations during execution, then mutates configuration parameters to identify the MSNS for a relation. We have experimented with AutoWeb on three popular Java web application frameworks, namely Servlet, Spring, and Apache Struts2. Experimental results demonstrated that AutoWeb can automatically generate specifications as precise as the state-of-the-art manual approaches. To summarize, this paper makes the following contributions:

- We propose the first automated method to infer web framework semantics, by identifying the MSNS for framework-introduced relations.
- We develop AutoWeb, utilizing a novel mutation-based approach to deduce the framework specifications automatically.
- We experimented AutoWeb on three popular Java web frameworks, and experimental results demonstrated that the inferred specifications are comparable with hand-written specifications over precision and soundness.

The rest of the paper is organized as follows. Section 2 motivates our approach with an example, and Section 3 describes AutoWeb in detail. We evaluate the tool AutoWeb in Section 4. Section 5 reviews related work and Section 6 concludes this paper.

2 Motivation

In this section, we aim to introduce the dynamic relations facilitated by web frameworks, illustrating their impact on static analysis through an example.

2.1 Motivating Example

Figure 1 gives an example application built on top of the Spring framework. The execution flows for two URLs are illustrated in Figure 2. The example processes two URLs: `/root1/path1` and `/root2/path2`. URL `/root1/path1` is handled by `Controller1.handle1` and URL `/root2/path2` is firstly processed by `doFilterInternal` before being handled by `Controller2.handle2`. In the example shown in Figure 1, an SQL injection vulnerability exists in line 49. This vulnerability occurs because `Controller1.handle1()`(line 8) invokes `ServiceImpl2.service()` (line 46) at line 11, which directly passes input data

```java
@Controller
public class Controller1 {
    @Autowired
    @Qualifier("service2")
    ServiceInterface srv;

    @GetMapping("/root1/path1")
    public String handle1(HttpServletRequest request){
        ...
        data = request.getParameter("name");
        srv.service(data);
        ...
    }
}

@Controller
@RequestMapping("/root2")
public class Controller2 {
    @RequestMapping("/path2")
    public String handle2(HttpServletRequest request){
        ...
        data = request.getParameter("name");
        sql = "update users set hit=hit+1 where name='"+data+"'";
        statement.executeUpdate(sql);
        ...
    }
}

public class Filter1 extends OncePerRequestFilter{
    protected void doFilterInternal(HttpServletRequest request, ...){
        if(validateSqlCharactor(request)) // Santitizer
            chain.doFilter(request, response);
        ...
    }
}

@Service("service1")
public class ServiceImpl1 implements  ServiceInterface{
    public String service(String name) {
        // safe SQL operation
        ...}
}

@Service("service2")
public class ServiceImpl2 implements ServiceInterface{
    public String service(String name) {
       ...
       String sql = "select * from users where name='"+name+"'";
       stmt.executeQuery(sql); // SQL injection
       ...}
}
```

(a) Application code.

```xml
<!--web.xml configuration file-->
<web-app>
    <filter>
        <filter-name> myFilter </filter-name> <filter-class> Filter1 </filter-class>
    </filter>
    <filter-mapping>
        <filter-name> myFilter </filter-name> <url-pattern> /root2/* </url-pattern>
    </filter-mapping>
</web-app>
```

(b) XML configuration file. "myFilter" aliases the class "Filter1" in Figure 1a.

Fig. 1: Motivating example of a Spring-based application.

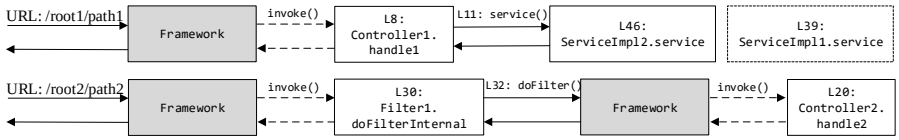


Fig. 2: Execution flow of the example in Figure 1. Solid lines indicate direct call edges, while dotted lines denote indirect calls. The content above the solid line represents the callsite, with the line number appearing to the left of the colon.

to a SQL query (line 49). Note that `Controller2.handle2` (line 20) also passes input data to SQL query statements (line 24). However, it is considered safe since the input request is sanitized by `Filter1` at line 31, before being processed by this handler.

The SQL injection in the above example can be detected via a classical taint analysis which computes whether the parameters of SQL queries are input data without being sanitized or not. However, without awareness of framework-introduced relations, traditional static taint analyses often fail to recognize the execution flow as in Figure 2, resulting in ineffective analysis results.

2.2 Framework-introduced Relations

Entry point Relation. Static analysis including taint analysis typically process on a call graph consisting of all reachable methods from *entry points*. In stand-alone Java applications, the `main` method is considered the entry point, whereas in web applications, entry points are often the request-handling methods directly invoked by frameworks. In Figure 1, `Controller1.handle1()` and `Filter1.doFilterInternal()` are entry points.

Entry point relations are defined by configuring methods and/or their declaring classes with Java annotations or XML configurations. For example, as shown in Figure 1a, the annotation `@Controller` (line 1) defines the entry class, and the annotation `@GetMapping` (line 7) declares the entry method. Note that the parameter values of annotations also specify corresponding request URLs (lines 7, 17, and 19). Additionally, entry points can be declared via XML configurations as shown in Figure 1b.

Points-to Relation. Points-to relations denote the set of heap objects referred to by a pointer variable. Frameworks can dynamically introduce points-to relations: frameworks can create objects outside the application code and inject these objects managed by frameworks into particular field references of application code. In our example, an object with type `ServiceImpl2` is managed by the framework and injected into field reference `Controller1.srv` (line 5). Such framework-introduced points-to relations cannot be computed by existing points-to analyses. Consequently, a call graph algorithm based on points-to analysis will

miss the call relation from line 11 to `ServiceImpl2.service()` (line 46), resulting in false negatives. On the other hand, a CHA-based call graph construction algorithm will introduce a spurious call relation to `ServiceImpl1.service()` (line 39), resulting in false positives.

Points-to relations are specified by annotating fields and the classes of injected objects. In our example, the field `Controller1.srv` is annotated with `@Autowired` (line 3) and `@Qualifier` (line 4), indicating that the field is injected with framework-managed object `service2`. The `@Service` annotation at line 37 and 44 suggest that object `service1` and `service2` are managed by the framework, with type `ServiceImpl1` and `ServiceImpl2`, respectively. Note that the parameter value of `@Qualifier` matches with that of `@Service` (line 44), indicating that object `service2` is injected into field `Controller1.srv`.

Call Relation. Call relations can be statically computed using techniques such as class hierarchy analysis (CHA) [11] or points-to analysis [18], to establish connections between invocation sites and corresponding callee methods. Nevertheless, these analyses are unable to handle indirect call relations introduced by frameworks. In such cases, applications invoke framework APIs, which, in turn, call back into application methods, making it challenging for static analysis techniques to accurately track and resolve these dynamic interactions.

Indirect call relations can be specified in various ways. In the example shown in Figure 1a, the method `Controller2.handle2()` at line 20 is indirectly called by `chain.doFilter()` at line 32 because it handles the URL `/root2/path2` which matches with the URL `/root2/*` (line 7) in the XML configuration for filter-mapping (Figure 1b).

Objective. The framework-introduced relations mentioned above are concrete because they are tied to a particular application. Our objective is to automatically abstract the general framework semantics (e.g., the annotations, `@Controller` and `@GetMapping`, together specify an entry method relation), which referred as *specifications*. Specifications can be leveraged by a static analysis tool to analyze a wide range of applications that are built on similar frameworks with different concrete configuration parameter values. Alternatively, we can manually produce specifications for frameworks, one by one. Nonetheless, such manual work can be error-prone and labor-intensive.

3 Methodology

We present AutoWeb to automatically infer framework specifications, which describe how relations are introduced by frameworks for given configurations. The key idea of AutoWeb is to automatically identify the minimal sufficient and necessary set (MSNS) for a relation by mutating configurations. In practice, AutoWeb takes a set of sample applications as inputs to infer the general framework semantics and the generated specifications can be used when analyzing other web applications.

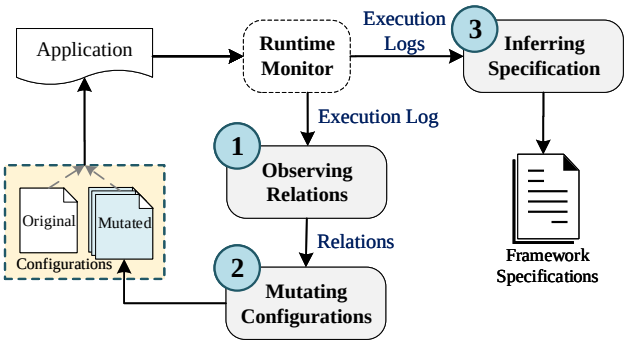


Fig. 3: Overview of AutoWeb. Black solid lines depict the workflow, while gray dotted lines indicate a singular choice.

Figure 3 overviews AutoWeb. The input is a runnable web application, and we partition it into two distinct logical modules: configurations and others in the input application. The steps are shown below:

1. The input application is firstly instrumented and traced to acquire the initial execution log using the original configurations. This step aims to observe the concrete relations including entry points and call/points-to relations.
2. Mutated configurations are generated by removing or adding configuration parameters from the original configurations. The MSNS for each relation can be identified by comparing the dynamic relations of different mutations. The application runs using mutated configurations in this step.
3. Concrete relations and configuration parameters are symbolized to generate specifications that map each relation type to the configuration set.

Next, we will elaborate on each step using the example in Figure 1.

3.1 Observing Relations

This step aims to collect the concrete framework-introduced relations from the initial execution information. In the Runtime Monitor component, we employed Javaassist [6,7,8] to instrument the input application, modifying its Java bytecode before class loading. To capture entry point and call relations, we insert logging statements before and after each method call, as well as at the entry and exit of each method. We log field read statements for points-to relations. Additionally, the entry and exit of the request handling methods of web containers (e.g., *Apache Tomcat* [16]) are also logged to track coming requests. These request sequences will later be used to interact with mutated applications.

As web applications handle multiple user requests concurrently, each log statement includes its thread identifier. Relations including entry methods and call/points-to relations can be easily deduced from execution logs within the same thread, as follows:

Rule$_{entry}$: Method m is an entry method if there is a log instance "[mtd] m" immediately following "[Req]".
Rule$_{call}$: Method m is called directly or indirectly at call site c if there is a log instance "[mtd] m" immediately following "[callsite] c".
Rule$_{ptsto}$: Field f references to an object of type t if there is a log instance "[fieldRead] f t".

Listing 1.1 shows the simplified execution logs of our motivating example. The logs are grouped by thread id ㉘ and ㉗, triggered by the coming request "/root1/path1" (line 1–11) and ends with the log instance "/root2/path2" (line 12–21), respectively.

```
1  ㉘[Req]/root1/path1
2  ㉘[mtd]Controller1.handle1(...)
3  ...
4  ㉘[fieldRead]Controller1.srv:ServiceImpl2
5  ㉘[callsite]line:11
6  ㉘[mtd]ServiceImpl2.service(...)
7  ...
8  ㉘[mtdEnd]ServiceImpl2.service(...)
9  ㉘[returnSite]line:11
10 ㉘[mtdEnd]Controller1.handle1(...)
11 ㉘[ReqEnd]/root1/path1
12 ㉗[Req]/root2/path2
13 ㉗[mtd]Filter1.doFilterInternal(...)
14 ...
15 ㉗[callsite]line:32
16 ㉗[mtd]Controller2.handle2(...)
17 ...
18 ㉗[mtdEnd]Controller2.handle2(...)
19 ㉗[returnSite]line:32
20 ㉗[mtdEnd]Filter1.doFilterInternal(...)
21 ㉗[ReqEnd]/root2/path2
```

Listing 1.1: Simplified runtime logs of the example in Figure 1.

The execution logs precisely capture all triggered relations. To focus solely on the relations introduced by the framework, we exclude the relations that can already be computed by existing static analyses. In Listing 1.1, line 2 and line 13 confirm that `Controller1.handle1` and `Filter1.doFilterInternal` are entry methods (*Rule*$_{entry}$). Lines 15–16 suggest that `Controller2.handle2` is indirectly invoked at line 32 of application code in Figure 1a by the invocation `doFilter` (*Rule*$_{call}$). Line 4 states that field `Controller1.srv` refers to an object of type `ServiceImpl2` (*Rule*$_{ptsto}$). Column 2 in Table 1 shows the set of concrete relations.

Table 1: Observed relations and corresponding configurations.

Relation Type	**Concrete Relation** R	**Minimal Sufficient and Necessary Set** $\mathcal{S}$
Entry point	*entry* : `Controller1.handle1`	{`@Controller`$\prod$`Controller1`, `@GetMapping`$\prod$`handle1`}
	entry : `Filter1.doFilterInternal`	{`@filter`$\prod$`Filter1`,`Filter1`$\trianglelefteq \ldots$, `doFilterInternal`$\trianglelefteq \ldots$}
Points-to	`Controller1.srv:ServiceImpl2`	{`@Autowired`$\prod$`srv`, `@Qualifier`$\prod$`srv`, `@Service`$\prod$`ServiceImpl2` }
Call relation	`32:Controller2.handle2`	{`API:doFilter`, $C_{32}/m_{32} \trianglelefteq \ldots$, `@filter`$\prod C_{32}$, $\mathcal{S}\prod$`{Controller2,handle2}` }

3.2 Mutating Configurations

After step 1, we observed the set of dynamically triggered concrete relations. Given a framework-introduced relation R, we try to identify the MSNS $\mathcal{S}$ for R, i.e., the minimum set of configuration parameters triggering R. To this end, we mutate configuration parameters based on the following guidelines.

> **Necessity:** Configuration parameter C is a necessary condition of R, i.e., $C \in \mathcal{S}$, if the resulting effect of removing C is that R is not triggered, while other relations remain unaffected.
> **Sufficiency:** The configuration set $\mathcal{S}$ is sufficient to trigger R, if a mutated configuration set $\mathcal{S}'$ can trigger a correspondingly mutated relation R'.

Hence, our approach firstly identifies necessary conditions for relation R by removing each configuration parameter one by one until R cannot be triggered. Next, the set of necessary conditions is mutated to further verify whether it is sufficient to trigger a correspondingly mutated relation R'.

In processing relation R, we only need to consider configurations related to R, which is the set of Java annotations or XML attributes attached to related program constructs of R. Hereafter, we use the notation $\mathcal{S}\prod p$ to denote configurations $\mathcal{S}$ on program construct p. Entry point relation $entry : C.m$ involves configurations $\mathcal{S}\prod\{C, m\}$, where $C.m$ represents the entry method m inside class C; points-to relation $f : C$ relates to configurations $\mathcal{S}\prod\{f, C\}$, where f and C are the field and its type respectively; and Call relation $c : C.m$, denoting that method m inside class C is invoked at callsite c, involves configurations $\mathcal{S}\prod\{c, C_c, m_c, C, m\}$, where C_c and m_c refer to the containing class and containing method of callsite c, respectively. The types of configuration parameters include Java annotations and XML configuration files. In addition, we also consider extensions of framework APIs including sub-typing and method overriding as special configurations, denoted by the notation $\trianglelefteq$.

Table 2: Mutation strategy.

original relation	mutated relation
$\mathcal{S}\prod\{C, m\} \implies entry : C.m$	$\mathcal{S}\prod\{C', m'\} \implies entry : C'.m'$
$\mathcal{S}\prod\{f, C\} \implies f : C$	$\mathcal{S}\prod\{f', C'\} \implies f' : C'$
$\mathcal{S}\prod\{c, C_c, m_c, C, m\}$ $\implies c : C.m$	$\mathcal{S}\prod\{c', C_{c'}, m_{c'}, C', m'\}$ $\implies c' : C'.m'$

Table 2 presents the mutation strategy to verify sufficient conditions. In summary, original program constructs p related to relation R are duplicated and

renamed to p'. Configurations $\mathcal{S}$ on p are moved to the mutated construct p' instead. Each mutation action generates a *mutated configuration set*, which is used to replace original configurations and then executed to verify whether a correspondingly mutated relation R' on p' can be triggered or not.

Next, we elaborate on how the MSNS for distinct relations are identified for our motivating example. The result for each observed relation is shown in Column 3 of Table 1.

Entry point Relation. In Figure 1, `Controller1.handle1` and `Filter1.doFilterInternal` are entry points. Let us consider the concrete relation *entry* : `Controller1.handle1`. There are two related configuration parameters of this method: `@Controller`$\prod$`Controller1` and `@GetMapping`$\prod$`handle1`, and both parameters are necessary since the relation cannot be triggered if either of them is removed. Furthermore, a new relation *entry* : `Controller1'.handle1'` can be triggered by applying our mutation strategy in Table 2. As a result, we have identified the MSNS {`@Controller`$\prod$`Controller1`, `@GetMapping`$\prod$`handle1`} for this concrete relation. The entry method `Filter1.doFilterInternal` involves the XML configuration on class `Filter1` (`filter`$\prod$Filter1) and two additional configurations derived from API extension, sub-typing from `OncePerRequestFilter` (denoted as `Filter1`$\trianglelefteq \dots$) and overriding of method `doFilterInternal` (denoted as `doFilterInternal`$\trianglelefteq \dots$), which form the MSNS for this relation.

Points-to Relation. We observe a points-to relation `srv:ServiceImpl2` in our motivating example. There are three related configuration parameters, which are `@Autowired`$\prod$`srv`, `@Qualifier`$\prod$`srv`, and `@Service`$\prod$`ServiceImpl2`. Removing either annotation will disable the points-to relation. To mutate the set of configurations, we introduce a new class `ServiceImpl2'` and a new field `srv'`, duplicated from `ServiceImpl` and `srv`, respectively. Next, the set of configurations is removed from the original program constructs and applied to the newly duplicated field and class instead. In another word, the configuration set $\mathcal{S} \prod$ {`srv, ServiceImpl2`} is mutated to another set $\mathcal{S} \prod$ {`srv', ServiceImpl2'`}. The mutated application will trigger the points-to relation `srv':ServiceImpl2'`. Hence, the three configuration parameters consist of the minimal sufficient and necessary set for the points-to relation `srv:ServiceImpl2`.

Call Relation. Call relation preserves the semantics of indirectly invoking application methods via framework API. That is, application invokes framework APIs, which in turn, call back into application methods. For this, we consider configuration parameters on the callsite (including its containing class and method), as well as configuration parameters on the invoked method (including its containing class). For concrete call relation `32:Controller2.handle2` in our example, the callsite (line 32 in Figure 1a) is constrained with the following configurations: line 32 invokes API `doFilter` (`API:doFilter`), m_{32} derived from `doFilterInternal` ($m_{32} \trianglelefteq \dots$), C_{32} (class `Filter1`) derived from `OncePerRequestFilter` ($C_{32} \trianglelefteq \dots$), and C_{32} configured as `filter` in XML

(`filter` $\prod C_{32}$). These configurations, together with configurations on the invoking method ($\mathcal{S} \prod$ {`Controller2, handle2`}) are the corresponding minimal sufficient and necessary set.

Table 3: Inferred Specifications of the motivating example shown in Figure 1.

Relation type	**Relation** R	**Specification** (content for relation type)
Entry point	$entry : C.m$	{`@Controller`$\prod C$, `@GetMapping`$\prod m$ }
		{`@filter` $\prod C$, $C \trianglelefteq$ `OncePerRequestFilter`, $m \trianglelefteq$ `doFilterInternal` }
Points-to	$f : C$	{`@Autowired`$\prod f$, `@Qualifier`$(S_1) \prod f$, `@Service`$(S_2) \prod C$, $S_1 \sim S_2$}
Call relation	$c : C.m$	{`API:doFilter`, `@filter` $\prod C_c$, $C/m \trianglelefteq \ldots$, `@Controller` $\prod C$, `@RequestMapping` $\prod m$ }

3.3 Inferring Specifications

Until this point, we have obtained a set of relations with their corresponding minimal sufficient and necessary sets. However, the relations and configurations are concrete: they are tied to concrete program constructs and the value (if any) of a configuration parameter is a constant string. In this step, we generalize the relation R and its corresponding configurations to abstract program constructs, according to the following rule.

> **Generalization:** Concrete application program constructs (classes, methods, and fields) are generalized to abstract program constructs. The constant string parameters of a configuration are generalized to a symbolic string with constraints matching the string to the name of a program construct, or to the parameter of another configuration. Symbolic strings with no matching constraints can be discarded.

Table 3 summarizes the specifications inferred from our example in Figure 1. The specifications look similar to the minimal sufficient and necessary set for a concrete relation, with concrete program constructs and constant configuration parameters symbolized.

As shown in Table 3, we have inferred two rules for entry point relation. The first rule states that the two configurations which are `@Controller`$\prod C$ and `@GetMapping`$\prod m$, collectively declare that $C.m$ is an entry method, regardless of their parameters. The second rule indicates that method $C.m$ is an entry method if $C.m$ extends from `OncePerRequestFilter.doFilterInternal` and C is configured as a filter by `@filter`$\prod C$. Note that in this rule, we only generalize application classes and methods while preserving concrete framework constructs. In the third rule, points-to relation $f : C$ holds under the following conditions: f is annotated with `@Autowired` and `@Qualifier`(S_1), C is annotated with `@Service`(S_2), and the two symbolic string parameter S_1 and S_2 match with each other ($S_1 \sim S_2$). Here S_1 and S_2 are parameters of configuration `@Qualifier` and `@Service`, respectively. The last rule indicates that if c invokes API `doFilter` in a method derived from `doFilterInternal` ($C/m \trianglelefteq \ldots$), it may indirectly invoke $C.m$ if $C.m$ is a request handler (`@Controller` $\prod C$, `@RequestMapping` $\prod m$) and C_c is a filter (`@filter` $\prod C_c$) with parameter matching the name of C ($S \sim C.name$).

Limitations. Points-to/Call relation connects a field/callsite to corresponding class/methods. The connection between distinct program constructs are often indicated by their configuration parameters, where the parameter may match with another parameter or with the name of a program construct. For the points-to relation `srv:ServiceImpl2` in our example, the parameter of configuration `@Qualifer("service2")` on field `srv` matches with the parameter of configuration `@Service("service2")` on class `ServiceImpl2`.

However, it is often challenging to statically determine whether a parameter matches another due to the flexibility provided by frameworks. Parameters can be configured using options such as regular expressions or string manipulation operations. For instance, the call relation `32:Controller2.handle2` happens because the URL processed by the containing class of line 32 (class `MyFilter`) matches with the URL handled by `Controller2.handle2`. However, the URL processed by `MyFilter` is configured as a regular expression `/root/*`. Moreover, we need to join parameters of the two configurations `@RequestMapping(/root2)` and `@RequestMapping(/path2)` together to construct the full URL handled by method `Controller.handle2`. It is rather challenging to recognize the above intricate connection automatically, and our approach will discard the matching constraints between the two URL parameters.

4 Evaluation

4.1 Experimental Setup

Implementation. We implemented a prototype that includes all the components as depicted in Figure 3, and took the benchmarks as inputs to generate framework specifications. Additionally, we applied the specifications inferred by AutoWeb to JackEE, a web application analysis engine built on top of Doop [4] for static program analysis.

Platform. The experiments for inferring framework specifications were conducted on an Intel Core(TM) i5-4590 (3.3GHz) laptop equipped with 32 GB of RAM, operating on the Windows 10 Professional version.

Benchmarks. We experimented AutoWeb on three popular web application frameworks: *Servlet* [13], *Spring*(including *Spring* [25] and *Spring-boot* [24]), and *Apache Struts2* [15]. Our experimental benchmarks comprise two parts.

- A collection of 16 open-source web applications[6], as listed in Table 4, was curated from various open platforms and filtered based on the underlying web frameworks, application categories (such as blogging systems and e-shops), and their respective star ratings. This benchmark encompasses a diverse array of applications, encompassing both popular and lesser-known examples, as well as those with complex and straightforward architectures.
- A collection of 8 web applications from JackEE, which are suggested by experts or top-popularity representatives of major classes of enterprise applications. One is free-binary-only, and the others are open-source.

[6] https://gist.github.com/menghaining/38286f83c8b674fab771be66d5bf371f

Table 4: Details of collected open-source benchmarks.

ID	Benchmark	Properties				
		Application Classes	Total Classes	Stars	Forks	Frameworks
1	community	94	19544	2.3k	739	Servlet, Spring
2	halo	425	45359	22.1k	7.5k	Servlet, Spring
3	iCloud	22	9498	183	115	Servlet, Spring, Struts2
4	jpetstore	24	6847	521	745	Servlet, Spring
5	logicaldoc	2013	51494	61	31	Servlet, Spring
6	LMS	33	14329	420	187	Servlet, Spring, Struts2
7	B2CWeb	41	17434	481	343	Servlet, Spring, Struts2
8	newbee-mall	89	13463	9.4k	2.5k	Servlet, Spring
9	NewsSystem	66	20065	19	8	Servlet, Spring, Struts2
10	openkm	2968	88843	527	255	Servlet, Spring
11	RuoYi	290	38320	3k	1k	Servlet, Spring
12	showcase	42	8937	5k	3.8k	Servlet, Spring
13	petclinic	24	29092	395	1.8k	Servlet, Spring
14	WebApp	75	28722	1.3k	610	Servlet, Spring
15	struts-examples	170	13507	405	543	Servlet, Struts2
16	Struts2-Vuln	20	4569	170	38	Servlet, Struts2

The rest of the figures and tables of this paper would use "ID" to represent each benchmark instead of benchmark names.
Benchmarks **struts-examples** and **Struts2-Vuln** are two collections that contain 41 and 16 micro-benchmarks, respectively.

Our experiments aims to answer the following research questions:

RQ1 Can our approach automatically infer framework-introduced semantics of web applications?
RQ2 How precise are the inferred specifications?
RQ3 How is the quality of generated specifications compared to manually written ones?

4.2 RQ 1: Feasibility

AutoWeb. AutoWeb offers an automated and user-friendly solution that minimizes the need for manual setup throughout the workflow. Human involvement is solely required during the initial phase, primarily to interact with the deployed applications, which can be streamlined through using tools like [31]. Then, AutoWeb automates the following steps leveraging information directly derived from the initial phase. Compared to human learning, which involves static analysis and framework knowledge, the cost is notably lower. Even though resources like StackOverflow [10] and official documentation facilitate rapid initiation, they often lack insights into the correlation between framework usage and static analysis semantics. Establishing these connections requires intricate, labor-intensive manual intervention, making it a challenging and time-consuming endeavor.

The applications in Table 4 are used as the input of AutoWeb to generate specifications. In Table 5, columns 2–5 detailed the runtime information of AutoWeb. The size of the execution log produced by each application using the original configurations is outlined in column 2. Column 3 displays the number

of mutated configuration sets generated by AutoWeb. Each mutation concerns one configuration for one relation. Column 4 denotes the success rate of application execution using these mutated configurations. Apart from benchmarks 2 and 5, all other benchmarks achieved success rates exceeding 75% during execution. Even in benchmarks 2 and 5, a significant number of successful runs amounted to 277 and 488, respectively. Column 5 presents the recall ($TP/(TP + FN)$), demonstrating that all benchmarks, except benchmark 9 (to be discussed in 4.3), achieved recall rates surpassing 90%. The extensibility of AutoWeb lies in two aspects, namely new frameworks and new relation types. New frameworks are already supported by AutoWeb (discussed shortly), while certain components need to be enhanced to support new relation types.

Table 5: Detailed results of inferred specifications and runtime information.

ID	Instrument	Mutation		Inferring	Configurations			Entry		Inject		Call	
	Log Size(M)	Testcase Number	Trigger Success%	Recall%	All	Reach.	Spec.	FN	FP	FN	FP	FN	FP
1	47.6	167	86.75%	90.00%	19	15	9	**1**	0	0	0	0	0
2	12.5	398	69.41%	94.44%	80	60	17	0	0	0	0	**1**	0
3	0.5	242	96.61%	100.00%	6	6	4	0	0	0	0	0	0
4	36.6	104	98.08%	100.00%	6	6	2	0	0	0	0	0	0
5	122	780	62.50%	100.00%	18	9	9	0	0	0	0	0	0
6	0.5	213	84.13%	100.00%	8	7	6	0	0	0	0	0	0
7	0.8	155	82.96%	100.00%	7	4	4	0	0	0	0	0	0
8	172	117	76.92%	100.00%	19	14	8	0	0	0	0	0	0
9	158	93	76.92%	60.00%	9	9	3	0	0	0	0	**2**	0
10	207	421	98.05%	100.00%	67	30	5	0	0	0	0	0	0
11	2.79	310	75.28%	92.31%	59	27	12	0	0	**1**	0	0	0
12	1.4	695	78.71%	100.00%	29	28	12	0	0	0	0	0	0
13	1.75	337	75.00%	100.00%	33	31	10	0	0	0	0	0	0
14	1.56	290	80.07%	100.00%	34	19	7	0	0	0	0	0	0
15	11.2	1022	98.33%	100.00%	29	16	8	0	0	0	0	0	0
16	0.3	176	99.43%	100.00%	3	3	2	0	0	0	0	0	0

Columns 2–5 show the details of runtime information. Columns 6–8 show the configuration numbers. Columns 9–14 show the accuracy of inferred specifications.
Reach. denotes reachable configurations at runtime.
Spec. denotes specifications inferred by AutoWeb (Section 3.3)

Specification. The benchmarks encompass a total of 152 configuration parameters: 60 for classes, 39 for methods, and 53 for fields. Among these configuration parameters, 121 are specific to the three frameworks under manual investigation, while the remaining parameters are associated with other frameworks utilized within the applications, such as Stripes [17]. Additional details regarding the count of distinct configuration parameters in each benchmark can be found in columns 6–8 of Table 5.

The specifications derived from the 16 benchmarks encompass 96 entry point types, 9 points-to types, and 17 indirect call types. These specifications involve configuration parameters for 17 classes, 20 methods, 10 fields, as well as 46 additional sub-types within the framework API. To enhance understanding and application of these specifications, we provide an excerpt of inferred entry point relations. Both points-to and call relations exhibit similarities.

Table 6 displays a snip of the results regarding entry point types in the inferred specifications. Each row indicates when a method and its associated class satisfy the specified configuration parameters, the method can be considered as an entry point, and the class becomes the entry class. EP 1, 3, and 6 correspond to annotation configurations, while EP 2, 4, and 5 are associated with XML configurations. In the case of EP6, the method also needs to override the designated method. The results shown are divided by frameworks: EP 1–2, EP 3–4, EP 5, and EP 6 belong to frameworks Spring, Struts2, Servlet, and Strips [17], respectively. It is worth noting that initially, we did not consider Strips but AutoWeb still inferred the corresponding specification (EP 6), thereby validating our approach's potential for generalization to other novel frameworks.

Table 6: Part of inferred entry-point (EP) specifications.

Line	Class	Method
EP1	@RestController	@PostMapping
EP2	beans->bean[class]	beans->bean[destroy-method]
EP3	-	@Action
EP4	struts->package->action[class]	struts->package->action[method]
EP5	web-app->filter->filter-class	Filter.doFilter(...)
EP6	-	@DefaultHandler

The "-" symbol represents any configuration.

Since specifications are generalized and not specific to any application, they can be utilized by existing static analyzers to analyze any web application built on the frameworks outlined in the specifications. After that, static analyzers can understand framework semantics to facilitate various analyses, such as call graph construction [18], and information analysis [3].

Conclusion. To sum up, our proposed technique is feasible in practice: AutoWeb can automatically generate precise framework specifications, saving heavy human effort. Moreover, AutoWeb is readily applicable to other frameworks (e.g., Stripes) and relation types, confirming the generality of our approach.

4.3 RQ 2: Specification Accuracy

The specifications generated by AutoWeb should exhibit minimal or zero false positives to prevent any adverse impacts for later direct use. To this end, we manually verify the accuracy of all inferred specifications by examining the source code, with a summary of the results provided in columns 9–14 of Table 5. During the verification process, we focus on the following two issues:

- **False Positives.** Are there any incorrectly inferred relations that contradict framework semantics?
- **False Negatives.** Are there any correct relations that were not inferred but were observed during runtime?

False Positives Table 5 (columns 10, 12, 14) reveals no false positives in any of the benchmarks. This is due to the specifications being inferred from runtime

information, which is subsequently verified by the actual execution of the application. As a result, AutoWeb effectively avoids false positives, ensuring the correctness of the framework semantics introduced in the static analysis.

False Negatives There are only five false negatives for all benchmarks. These false negatives are hidden behind certain factors during runtime, preventing them from being apparent. One factor is embedding the configuration content directly in the code, rendering the configuration on the code(annotations or XML files) ineffective. For example, in the `community` project, removing the parameter `@RequestMapping` does not take effect at runtime because the application has a default response function that produces the same result as the configuration parameter. Another factor is the language feature. For example, in `RuoYi`, the false negative for the field-inject relation is related to the `@Value` annotation. This occurs because all fields annotated with this configuration are Java primitive types, which always have default values. Moreover, complex string configurations lead to false negatives. In the `NewsSystem`, a false negative is caused by intricate string manipulation. Specifically, the configuration `<action name="AdminAction_*" method="1">` utilizes the implicit configuration value "{1}" to represent the method name, relying on the incoming URLs. Another false negative arises from multi-layer references in XML attributes, which are not yet supported.

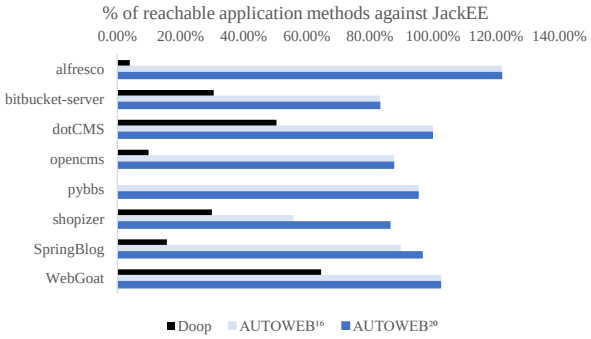
Note that false negatives in one application may appear as true positives in other applications. For instance, the Entry-point relation `@RequestMapping` is FN in `community` appears as true positive in `newbee-mall`. More input projects would bring more complete specifications, which will be discussed in 4.4. The average false negative rate summarized from all inferred specifications is 8.2%.

Conclusion. Upon analysis, we identified that the false negatives in our inferred specifications were a consequence of configurations outside the scope of our analysis. Importantly, the inferred specifications exhibit no false positives, indicating their direct applicability as framework knowledge for existing static analysis tools.

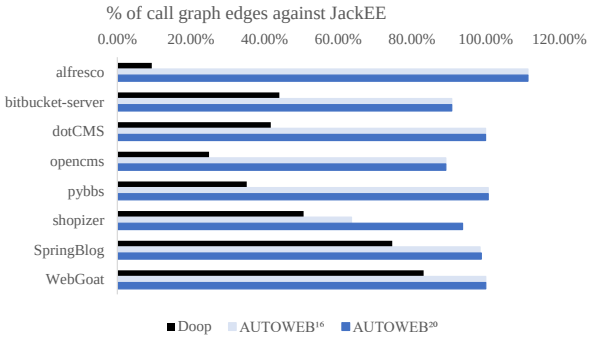
4.4 RQ 3: Comparison with Existing Work

The state-of-the-art tool, JackEE [1], offers open-source specifications across various web frameworks. JackEE is built on top of Doop [4], which is a collection of various analyses expressed in the form of Datalog [23] rules, and all the framework specifications are written in Datalog rules. To assess the efficacy of the specifications generated by AutoWeb, we transformed them into Datalog rules, replacing all the configuration specifications in JackEE. With JackEE's specifications serving as the baseline, we evaluated the specifications produced by AutoWeb alongside the default specifications from Doop.

Comparison Dimension. Framework knowledge helps static analyses to better understand the application behaviors. To evaluate the quality of specifications, we focus on *reachable application methods* and *call graph edges* (same metrics used in JackEE).



(a) Application methods reachability against JackEE.



(b) Call graph edges against JackEE.

Fig. 4: Reachability for different metrics.

Comparison Method. We adopt JackEE (with its manual specifications) as the baseline for comparison, evaluating the efficacy of JackEE, Doop, and AutoWeb. We considered the 8 benchmarks from JackEE (mentioned in 4.1) for evaluation. Among these benchmarks, one benchmark entails a prolonged setup duration, two encounter deployment problems, and one is not open-sourced. As a result, we use 4 benchmarks to enrich the specifications generated by AutoWeb. Hence, we have two sets of specifications: (1) AutoWeb^{16} contains the specifications generated using 16 benchmarks in Table 4. (2) AutoWeb^{20} denotes the enriched specifications using 20 (16 + 4) benchmarks.

Result. Figure 4a and Figure 4b compare Doop, AutoWeb^{16}, and AutoWeb^{20} against JackEE on reachable (application) methods and call graph edges, respectively. When comparing AutoWeb^{16} with JackEE, the average ratio on reachable application methods is 91.85%, while for call graph edges, the average is 94.27%. In some cases, AutoWeb^{16} even outperformed JackEE like `alfresco` and `WebGoat`. When comparing AutoWeb^{20} with JackEE, the ratios range from 83.14% to 121.73% on reachable methods, averaging 96.59%; For call graph edges, the numbers vary from 89.18% to 111.48%, averaging 98.09%. On the other hand, AutoWeb^{20} exhibited improvements over AutoWeb^{16}, particularly in benchmarks such as `shopizer` and `SpringBlog`, where previously

missing configurations, absent in the initial 16 benchmarks, were later inferred. This underscores that using a more extensive set of inputs can lead to richer specifications, as discussed in section 4.3. Furthermore, our specifications include correct configuration parameters that may have been overlooked manually. For example, we identified annotations like `@PostMapping` and `@ExceptionHandler` as entry-point relations, while JackEE did not recognize them. This highlights the presence of unsystematic and incomplete issues of manually configured specifications.

Conclusion. Concerning the quality of constructed call graphs, the specifications inferred by AutoWeb are comparable with those manual ones in JackEE. Furthermore, our approach excels at identifying overlooked configuration parameters during manual writing.

5 Related Work

The related work encompasses two main areas: modeling framework behaviors, and automatic summarizing of program semantics.

Modeling Framework Behaviors. As mentioned in section 1, previous works modified each analysis engine with human knowledge on-demand, which suffers from limited reusability. Some studies attempted to develop reusable models for specific frameworks to address this limitation, which still rely on human knowledge. ANTaint [30] needs manually modeled core features of Spring like bean injection and AOP. The Oracle team [12] manually wrote rules to identify entry points only for Java EE Servlet applications. GenCG [19,20], F4F [26,28], and JackEE [1] all need human effort to obtain knowledge of the framework and static analysis. Static Analysis Refining Language (SARL) [14] can also obtain framework-introduced relations via iterative software analysis. However, the analyzer also needs to point out and add the missing framework knowledge. Unlike AutoWeb, all these approaches rely on the knowledge of frameworks and static analysis, and require extra manual effort for new frameworks.

Automatic Summarizing Program Semantics. Research on exploiting automatic approaches to summarizing framework library specifications used in static analysis [22,9,2] became more popular. These approaches mined information flow specifications with additional running information over libraries, rather than writing them by hand. However, the purpose of these approaches is to summarize the framework library APIs' semantics, especially for Android, not to deal with complex but frequently used configurations (e.g., XML files). Therefore, these approaches do not apply to web applications that mostly use non-code configurations.

6 Conclusion

Web applications heavily rely on web frameworks, making it imperative to precisely model framework semantics for static analysis. In this paper, we pro-

posed the first automated method to produce specifications that represent general framework semantics. To that end, we identify the minimal necessary and sufficient set for a framework-related relation by mutating configurations. Experimental results on three mainstream Java frameworks demonstrate that our technique is comparable to existing state-of-the-art manual approaches, obtaining a marginal 8.2% false negatives with no false positives.

Acknowledgments. We thank all anonymous reviewers for their valuable feedback which has significantly improved the quality of this manuscript. This work is supported by the National Key R&D Program of China (2022YFB3103900), and the National Natural Science Foundation of China (NSFC) under grant numbers 62132020 and 62202452.

References

1. Antoniadis, A., Filippakis, N., Krishnan, P., Ramesh, R., Allen, N., Smaragdakis, Y.: Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 794–807 (2020)
2. Arzt, S., Bodden, E.: Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 725–735. IEEE (2016)
3. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 259–269. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594299, https://doi.org/10.1145/2594291.2594299
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 243–262 (2009)
5. Centonze, P., Naumovich, G., Fink, S.J., Pistoia, M.: Role-based access control consistency validation. In: Proceedings of the 2006 international symposium on Software testing and analysis. pp. 121–132 (2006)
6. Chiba, S.: Javassist, https://www.javassist.org/
7. Chiba, S.: Load-time structural reflection in java. In: European Conference on Object-Oriented Programming. pp. 313–336. Springer (2000)
8. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient java bytecode translators. In: International Conference on Generative Programming and Component Engineering. pp. 364–376. Springer (2003)
9. Clapp, L., Anand, S., Aiken, A.: Modelgen: mining explicit information flow specifications from concrete executions. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 129–140 (2015)
10. Community, S.O.: stackoverflow, https://stackoverflow.com/
11. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: ECOOP'95—Object-Oriented Programming,

9th European Conference, Åarhus, Denmark, August 7–11, 1995 9. pp. 77–101. Springer (1995)
12. Dietrich, J., Gauthier, F., Krishnan, P.: Driver generation for java ee web applications. In: 2018 25th Australasian Software Engineering Conference (ASWEC). pp. 121–125. IEEE (2018)
13. Edition, J.E.: Jakarta ee, https://jakarta.ee/
14. Ferrara, P., Negrini, L.: Sarl: Oo framework specification for static analysis. In: Software Verification, pp. 3–20. Springer (2020)
15. Foundation, A.: Apache struts 2, https://struts.apache.org/
16. Foundation, A.S.: Apache tomcat, https://tomcat.apache.org/
17. Framework, S.: Stripes framework, https://stripesframework.atlassian.net/
18. Lhoták, O., Hendren, L.: Scaling java points-to analysis using s park. In: Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12. pp. 153–169. Springer (2003)
19. Luo, L.: A general approach to modeling java framework behaviors. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1680–1682 (2021)
20. Luo, L.: Improving Real-World Applicability of Static Taint Analysis. Ph.D. thesis, Universität Paderborn (Oct 2021), https://www.bodden.de/pubs/phdLuo.pdf
21. Martínez, S., Cosentino, V., Cabot, J.: Model-based analysis of java ee web security configurations. In: 2016 IEEE/ACM 8th International Workshop on Modeling in Software Engineering (MiSE). pp. 55–61. IEEE (2016)
22. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. ACM SIGSOFT Software Engineering Notes **27**(4), 229–239 (2002)
23. Smaragdakis, Y., Bravenboer, M.: Using datalog for fast and easy program analysis. In: International Datalog 2.0 Workshop. pp. 245–251. Springer (2010)
24. Spring: Spring boot, https://spring.io/projects/spring-boot
25. Spring: Spring framework, https://spring.io/projects/spring-framework
26. Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4f: taint analysis of framework-based web applications. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. pp. 1053–1068 (2011)
27. Toman, J., Grossman, D.: Taming the static analysis beast. In: 2nd Summit on Advances in Programming Languages (SNAPL 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
28. Tripp, O., Pistoia, M., Cousot, P., Cousot, R., Guarnieri, S.: Andromeda: Accurate and scalable security analysis of web applications. In: International Conference on Fundamental Approaches to Software Engineering. pp. 210–225. Springer (2013)
29. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. ACM Sigplan Notices **44**(6), 87–97 (2009)
30. Wang, J., Wu, Y., Zhou, G., Yu, Y., Guo, Z., Xiong, Y.: Scaling static taint analysis to industrial soa applications: a case study at alibaba. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1477–1486 (2020)
31. Yu, X., Jin, G.: Dataflow tunneling: mining inter-request data dependencies for request-based applications. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 586–597. IEEE (2018)