

# Jasmine: A Static Analysis Framework for Spring Core Technologies

Anonymous Author(s)\*

## ABSTRACT

The Spring framework is widely used in developing enterprise web applications. Spring core technologies, such as Dependency Injection and Aspect-Oriented Programming, make development faster and easier. However, the implementation of Spring core technologies uses a lot of dynamic features. Those features impose significant challenges when using static analysis to reason about the behavior of Spring-based applications. In this paper, we propose Jasmine, a static analysis framework with respect to Spring core technologies extends from Soot to enhance the call graph's completeness while not greatly affecting its performance. We evaluate Jasmine's completeness, precision, and performance using Spring micro-benchmarks and a suite of 18 real-world Spring programs. Our experiments show that Jasmine effectively enhances the state-of-the-art tools based on Soot and Doop to better support Spring core technologies. We also add Jasmine support to FlowDroid and discovered twelve sensitive information leakage paths in our benchmarks. Jasmine is expected to provide significant benefits for many program analyses scenes of Spring applications where more completeness of call graphs are required.

## CCS CONCEPTS

• **Software and its engineering** → *Compilers*; • **Theory of computation** → *Program analysis*.

## KEYWORDS

static analysis, points-to analysis, Spring framework

### ACM Reference Format:

Anonymous Author(s). 2022. Jasmine: A Static Analysis Framework for Spring Core Technologies. In *Proceedings of IEEE/ACM Automated Software Engineering (ASE 2022)*, ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Java frameworks are software designed to make programming easier in Java. They provide collections of pre-written code used by Java developers to build Java applications or web applications. The Spring framework [7, 8], in particular, is the most popular Java framework that is used in the vast majority of enterprise applications, including web services, microservices, and data-driven systems. So far, there are more than 540,000 Spring-related projects

on the Github [9]. The Java world is still a Spring-dominated world, with over half of the market using Spring Boot [5].

Static program analysis is the analysis of computer software performed without executing any programs and often used to detect security vulnerabilities and performance issues. In static analysis research, Java has a series of the well-known research frameworks, such as Soot [57], Doop [16], or WALA [24, 51]. Antoniadis et al. [12] claim that these frameworks do not or only partially support core technologies of Spring: Dependency Injection (DI) [3] and Aspect-Oriented Programming (AOP) [1], thus leading to unsound analysis results [38]. As far as we know, the literature rarely contains techniques specifically for Spring applications in static analysis and often ignores the influence of Spring framework. IBM published F4F [50] to make taint analysis engines perform more accurate analysis of web applications. Alibaba's ANTaint [58] simulates framework to enhance the capabilities of FlowDroid in the enterprise applications. JackEE [12] conducts pointer analysis for enterprise applications by identifying entry points and processing variables related to injection objects. The call-graph and the data-flow graph constructed based on the above techniques ignored the Spring core technologies and affected the accuracy of the results generated by software engineering tools where call graphs are required. For example, based on these incomplete graphs, existing taint analysis tools cannot detect information leakage paths in mall [6] and halo [4] projects (with 50k and 19k stars on Github, respectively, Section 5.4). The primary reason is that these frameworks use pointer analysis that does not support Spring core technologies.

Pointer analysis is a fundamental family of static analysis that computes abstractions of the possible values of pointer variables in a program [30, 33, 52]. Such information is essential for reasoning about inter-procedural control flow and alias analysis in object-oriented programs. Therefore, it is widely used in software engineering tools, such as taint analysis [13, 28, 35], program verification [25, 45], program debloating [27, 44], and bug detection [18, 23, 46, 59]. However, Spring core technologies often use configuration files, reflection, and dynamic proxy to implement corresponding functions. These functions produce specific codes or behaviors when the program runs, and these information is invisible during static analysis [11, 31].

In this article, we provide Jasmine, a static analysis framework for Spring applications. Static analysis tools based on Soot and Doop, in concert with Jasmine, can produce a high-completeness, high-precision static analysis of Spring programs. In summary, We make the following contributions.

- We study two core technologies of Spring: DI and AOP. We divided their effects in static analysis into two aspects, "explicit" and "implicit", that help in explaining where to introduce the effects in static analysis and why existing static analysis methods are unsound. (Section 2 and 3).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ASE 2022, 10-14 October, 2022, Michigan, United States

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

- We propose a simulated approach in the level of intermediate representation (IR) and construct the Jasmine framework. Jasmine parses Spring configuration files to obtain the specific functions. According to these information, Jasmine simulates the possible behavior of Spring core technologies (Section 4).
- We present a micro-benchmark and a real-world benchmark that contains eighteen Spring programs, which are used to evaluate the capability of Jasmine. Jasmine can achieve high degrees of completeness through taking advantage of modeling DI and AOP. Jasmine also supports integration into the current state-of-the-art static analysis tools, based on Soot and Doop. With the help of Jasmine, we use FlowDroid found twelve user information leakage paths in three real Spring programs (Section 5).

## 2 BACKGROUND

We offer a gentle introduction to Java pointer analysis in Section 2.1. In Section 2.2, we briefly describe two core technologies of Spring.

### 2.1 Pointer Analysis

Pointer analysis (points-to analysis) is a significant static analysis technology that determines information on the values of pointer variables during runtime[47]. Such information is essential for reasoning about inter-procedural control flow and alias analysis in object-oriented programs. Therefore, it is widely used in static analysis tools. Pointer sets form a relation between variables and abstract objects. We can compute a points-to relation

$$pt \subseteq Var \times Obj$$

with  $Var$  being the set of program variables and  $Obj$  looking up as the set of abstract objects. Abstract objects are typically represented as allocation sites, i.e., instructions that allocate objects (e.g., **new** in Java)[19].

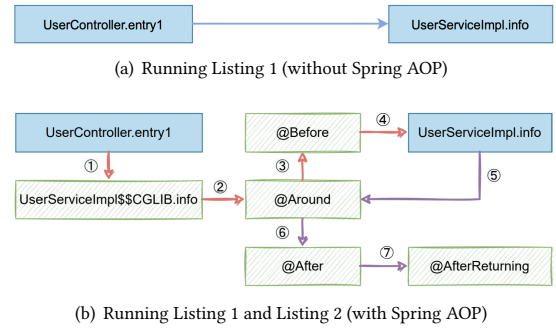
Based on the following rules for statements that involved pointer expressions, pointer analysis can build object flow graph [54] and call graph of a program. For example, an edge  $a \leftarrow b$  in the OFG means that the objects pointed by pointer  $a$  may flow to pointer  $b$ . Although the literature is not entirely consistent on high-level terminology, pointer analysis is a near-synonym of alias analysis.

Type	Stmt	Edge	
assignment	$b = a;$	$b \leftarrow a$	(1)
load	$b = a.f;$ where $o \in pt(a)$	$b \leftarrow o.f$	(2)
store	$a.f = b;$ where $o \in pt(a)$	$o.f \leftarrow b$	(3)
calls/returns	$b = c.m(a);$	$p \leftarrow a$	(4)
	$m(p) \{ \dots \text{return } r; \}$	$m_{this} \leftarrow c$	
		$b \leftarrow r$	

### 2.2 Spring Core Technologies

Spring core technologies[8] contain DI, AOP, Resources, i18n, Data Binding, Type Conversion, SpEL. DI and AOP are the two technologies that significantly affect the completeness of static analysis. Hence, this paper only focuses on DI and AOP.

**2.2.1 Dependency Injection.** "Inversion of Control (IoC) is also known as dependency injection (DI)"[8], is one of the core behaviors of the Spring framework. It is a process whereby objects define



**Fig. 1. Comparison of running with or without Spring AOP.** The execution sequence of (a) omits the complicated calling process inside the Spring framework during the program runs.

their dependencies through arguments to a factory method, constructor arguments, or properties set on the object instance after it is constructed or returned from a factory method. IoC container injects those dependencies when creating the bean (the objects form the backbone of the application and are managed by the Spring IoC container). The container gets instructions on what objects to instantiate, configure, and assemble by reading configuration.

Listing 1 shows that a Controller (which is the core element of a Spring web application), named `UserController`, accepts user's incoming requests (in Line 8) via the URL `"/user/entry1"`. Figure 1(a) shows Listing 1 code execution sequence. In this example, the class `UserController` and `UserServiceImpl` are annotated with `@Controller` and `@Service` annotation (Lines 2 and 21), respectively. Therefore, the IoC container creates the beans of two classes and injects dependency according to the field annotated with `@Autowired`. When the program runs, `userService` points to the object of `UserServiceImpl`, and `entry` invokes `info` in `UserServiceImpl` (in Lines 9 and 23).

**2.2.2 Aspect-Oriented Programming.** Aspect-oriented programming (AOP) is a technology that aims to increase modularity by allowing the separation of cross-cutting concerns. AOP adds additional behavior to existing code (an advice) without modifying the code itself. This allows behaviors that are not central to the business logic (such as logging) to be added to a program. Spring AOP does not require a particular compiler or compilation process. Instead, it is implemented in pure Java and adds additional behavior to existing code by means of JDK dynamic proxy [26] or CGLIB proxy[17].

The Spring AOP example, shown in Listing 2, describes how to link aspects with the target method. Figure 1(b) shows the order in which the codes of Listing 1 and 2 are executed together. Class `LogAspect` is managed by the IoC container as a bean because it is annotated with `@Component`. When the program is running, Spring framework scans the class annotated with `@Aspect` (Line 1) as an aspect, then parses the expression in `@Around` (Line 5) to find the target object (in this case, target class is `UserServiceImpl` and target method is `info`). Spring generates a dynamic proxy method (`UserServiceImpl.CGLIB$1.info` in Figure 1(b)) for target method and `UserServiceImpl.CGLIB$1.info` calls the *advice methods* (the method annotated with `@Around`, `@Before`, `@After`, and `@AfterReturning`) according to ② - ⑦ order in Figure 1(b). Spring framework will generate a call path between `pdj.proceed()` of Listing 2 and `info` of Listing 1 at runtime.

### 3 CAUSES OF INVALIDITY IN POINTER ANALYSIS

The Spring framework has impact on static analysis based on pointer analysis [31], i.e., unsound results. In this section, we dissect how DI and Spring AOP impact for pointer analysis. Explicit impact refers to the case where the function call exists in the application source code, whereas implicit impact means that the calling relationships and data flows are hidden in framework internals and can be only exposed at runtime.

#### 3.1 Explicit Impact

In Section 2.1, we have seen that the goal of pointer analysis is to compute an approximation of the set of program objects that a pointer variable or expression can refer to. If DI processes the field, pointer analysis treats it as uninitialized. For example, in Listing 1, pointer analysis could not compute objects of *userService* refer to and treated *userService* as a null pointer. Because of lacking such information, the call edge between *entry1* and *info* of class *UserServiceImpl* is lost when building the call graph, and *info* is considered unreachable in static analysis. Therefore, taint analysis could not find a leakage path (Line 26) based on this call graph.

Spring framework uses singleton pattern to instantiate objects by default (similar to Line 4 in Listing 1) when executing DI. It means that the bean (object) of class is unique to the whole program. When using the `@Scope("prototype")` in Line 20, Spring framework will use the prototype pattern that each allocation is a different bean (object) during DI (similar to Line 5 in Listing 1). Existing points-to analysis process each DI point with a prototype pattern, which ignores the impact of singleton pattern on pointer analysis (alias analysis). For example, when method *entry1* of *UserController* initializes the field *pwd* of *UserServiceImpl* by the parameter *pwd* of *info* (Line 9 in Listing 1). Then method *entry2* in class *AdminController* obtains the same value by calling *getPwd*, because both *userService* of Lines 3 and 14 point to the same object at runtime.

#### 3.2 Implicit Impact

As described in Section 2.2.2, because there are many reflection and dynamic proxies used in Spring framework, pointer analysis cannot handle the application using spring AOP and lead to the call graph losing the execution sequences in Figure 1(b). An unsound call graph with application's core part will make static analysis tools cause false negative. We could not detect the following example using taint analysis based on the above call graph: *entry1* passes the password information entered by the user to method *info*. Line 7 of Listing 2 obtains and prints the password information.

Spring AOP uses JDK dynamic proxy or CGLIB proxy to add additional behavior to existing code (an advice) without modifying the source code. Since there is no obvious hint on the source code, so static analysis tools don't know how to handle these situations. We dissect the mechanism behind this as follows.

- We change the pointcut expression of *LogAspect* to the comment in Line 3 of Listing 2. In Line 25 of Listing 1, *info* calls method *invoke* belonging to the same class, the dynamic proxy was not triggered, and advice methods did not execute. This is the reason why some interceptors that use AOP as permission checks cannot work (the Permission in Table II).

- CGLIB creates a proxy class that does not contain private or static methods of the superclass and does not initialize any fields inherited by the superclass, including final fields. Method *entry2* in Listing 2, a private entry point that can be accessed via the URL address `"/admin/entry2"`. Changing the pointcut expression to the comment in Line 4 of Listing 2, field *userService* in class *AdminController* is a null pointer and will cause null pointer exception when we accessed private method (the *EntryPoint* in Table II).

```

1 @RequestMapping("/user")
2 @Controller public class UserController {
3     @Autowired UserService userService;
4     // userService=SingletonFactory.getUserServiceImpl();
5     // userService=new UserServiceImpl();
6     // userService=new UserService$$CGLIB();
7     @GetMapping("/entry1")
8     public void entry1(String pwd){
9         userService.info(pwd);
10    }
11 }
12 @RequestMapping("/admin")
13 @Controller public class AdminController {
14     @Autowired UserService userService;
15     @GetMapping("/entry2")
16     private void entry2(){
17         Print(userService.getPwd());
18     }
19 }
20 // @Scope("prototype")
21 @Service class UserServiceImpl implements UserService {
22     private static String pwd;
23     public String info(String pwd){
24         this.pwd = pwd;
25         invoke();
26         Print("pwd:"+pwd);
27         return pwd;
28     }
29     public void invoke() {...}
30     public String getPwd() {return pwd;}
31 }

```

Listing 1: Example of using Dependency Injection

```

1 @Aspect
2 @Component public class LogAspect {
3     // execution(* *.. UserServiceImpl.invoke(..))
4     // execution(* *.. AdminController.entry2(..))
5     @Around("execution(* *.. UserServiceImpl.info(..))")
6     public Object doAround(ProceedingJoinPoint pdj){
7         Print(pdj.getArgs());
8         return pdj.proceed(); //Execute target method
9     }
10    @AfterReturning(returning="res",
11        value="execution(* *.. UserServiceImpl.info(..))")
12    public void returning(JoinPoint jp, Object res){...}
13    // @Before, @After, @AfterReturning method
14 }

```

Listing 2: Example of using Spring AOP

## 4 METHODOLOGY

This section introduces Jasmine: the approach simulates the core technologies of Spring Framework for solving the problems of Section 3 by stubbing on Soot IR [57]. Figure 2 shows the overview of Jasmine, which contains two components: Identifying Spring Framework ( ① in Figure 2) and Modeling Dynamic Features ( ② in Figure 2). The input of Jasmine is the origin IR, the XML file of the Spring application, and the matching rules. After processing by Jasmine, the output is the new IR that can describe DI and AOP.

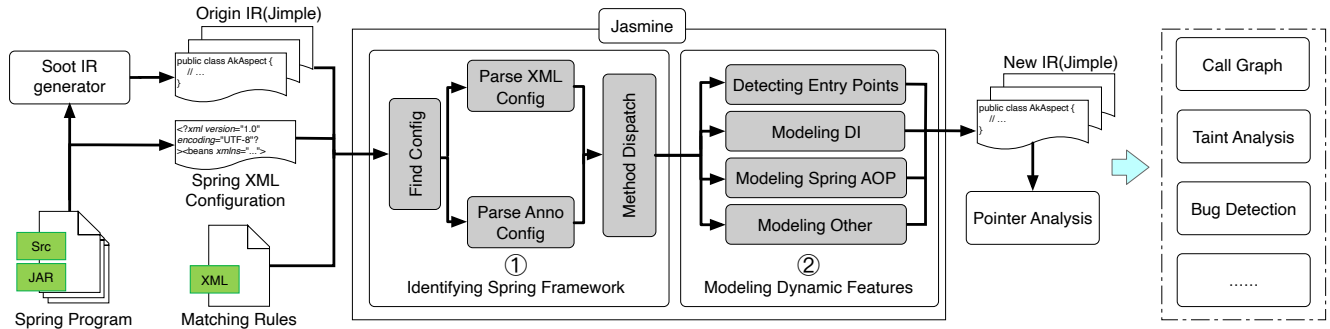


Fig. 2. Overview of Jasmine

## 4.1 Identifying Spring Framework

**4.1.1 Parsing Annotation.** Soot first parses the JAR of a program under test to generate origin IR. Jasmine takes the origin IR, the Spring XML configuration file of the target application, and the Matching Rules file as input to the Identifying Spring Framework module (① in Figure 2). Algorithm 1 shows how to process the annotation and identify the function of a method in a given program. The parsing of the Spring XML configuration is similar.

TABLE I. Matching Rules

Name	Example	# of Rules
Beans	(class,"Lorg/springframework/.../Service;")	12
Prototype	(class,"Lorg/springframework/context/.../Scope;")	1
Entrys	(class,"Lorg/springframework/.../Controller;")	4
EntryMethod	(method,"Lorg/springframework/.../PostMapping;")	6
AOPs	(class,"Lorg/aspectj/lang/annotation/Aspect;")	2
Pointcut	(method,"Lorg/aspectj/lang/annotation/Pointcut;")	1
Advices	(method,"Lorg/aspectj/lang/annotation/Around;")	5
Injects	(field,"Lorg/springframework/.../Autowired;")	4

Three sets are initialized to empty collection in Line 1: the Beans set of Singleton or Prototype pattern and the entry point methods. The algorithm starts by iterating through all the application classes in origin Soot IR, and obtaining the annotation information on the class (Line 2 in Algorithm 1). The annotation of the class will match with Beans, Entrys, and AOPs rules separately in Table I (Lines 3, 8, 12 in Algorithm 1). Specifically, Singleton and Prototype Beans are added to the different sets if a class has prototype annotation in Lines 4-7. For a class identified as entry points, the Algorithm 1 analyses the annotations of each method in this class and places the suitable methods into the EPMethods collection (Lines 9-11). The collections are output to the next module for subsequent processing.

**4.1.2 Processing Spring AOP.** In Algorithm 2, the variable *targetM* and one set are initialized in Line 1: *targetM* points to target method of Spring AOP, a set of advice methods that will enhance the target method's behavior, and *AOPMap*, which denotes a map of target method of Spring AOP and a set of related advice methods[8]. Iterating through all annotations on the method in *class* of the Input. If the annotation in Line 3 is pointcut expression, then the algorithm parses pointcut expression to find the target method and assigns it to *targetM*. Otherwise, advice methods are stored in the

### Algorithm 1: ANNOTATION PROCESS

**Input:** *JimpleClass*: Input the set of class jimple.  
**Output:** *SBeans*: the set of Singleton Bean.  
*PBeans*: the set of Prototype Bean.  
*EPMethods*: the set of entry method.

```

1 PBeans ← {}, SBeans ← {}, EPMethods ← {};
2 foreach class ∈ JimpleClass do
3   if Rule.Beans match class.anno then
4     if Rule.Prototype match class.anno then
5       add class to PBeans;
6     else
7       add class to SBeans;
8   if Rule.Entrys match class.anno then
9     foreach method m of class do
10      if Rules.EntryMethod match m.anno then
11        add m to EPMethods;
12 if Rule.AOPs match class.anno then
13   AOPProcess(class); // calls Algorithm 2

```

### Algorithm 2: AOP PROCESS

**Input:** *class*: Input the set of class jimple.  
**Output:** *AOPMap*: a map of target and advice methods.

```

1 AOPMap ← {}, AdviceList ← {}, targetM ← null;
2 foreach method m of class do
3   if Rules.Pointcut match m.anno then
4     targetM ← parse expression;
5   else if Rules.Advices match m.anno then
6     add m to AdviceList;
7 if targetM ≠ null then
8   add AOPMap.get(targetM) to AdviceList;
9   replace AOPMap(targetM) with Sort(AdviceList);

```

*AdviceList*. In Lines 7-9, if the *targetM* is not empty, get the advice methods in the *AOPMap* with the *targetM* key and add them to the *AdviceList*. Algorithm 2 sorts the *AdviceList* according to the value of *@Order*, class name, and the order of annotations *@Around*, *@Before*, *@AfterReturning*, and *@After*. Finally, Algorithm 2 saves *targetM* and *AdviceList* as key-value in *AOPMap*, and then output *AOPMap* to the next module.



<b>Names:</b>	$\frac{A \in (a...zA...Z\$<>)*}{name(A)}$	<b>Variables:</b>	$\frac{name(A)}{var(A)}$	<b>Types:</b>	$\frac{name(A)}{type(A)}$
<b>Assignment:</b>	$\frac{var(A) \quad var(B)}{instruction(assign(A,B))}$	<b>Allocation:</b>	$\frac{var(A) \quad type(T)}{instruction(alloc(A,T))}$	<b>Return:</b>	$\frac{var(R)}{instruction(return(R))}$
<b>Invocation:</b>	$\frac{var(Base) \quad name(Sign) \quad var(Arg) \quad var(Ret)}{instruction(call(Base,Sign,Arg,Ret))}$	<b>Expression:</b>			$\frac{instruction(E)}{instruction(exp(E))}$
<b>Class:</b>	$\frac{name(Name) \quad type(Super) \quad var(Field) \quad fun(Sign)}{class(Name,Super,Field,Sign)}$	<b>Function:</b>			$\frac{name(Sign) \quad var(Arg) \quad program(Body)}{fun(Sign,Arg,Body)}$
<b>Insert:</b>	$\frac{exp(E_1) \quad fun(Sign) \quad exp(E_2)}{insert(Sign,E_1,[before after]site(E_2))}$	<b>Site:</b>	$\frac{exp(A)}{site(A)}$	<b>Program:</b>	$\frac{instruction(I) \quad program(P)}{program(I,P)}$

Fig. 3. Jasmine Syntax

## 4.2 Modeling Dynamic Features

In this section, we describe how to manipulate code in the level of intermediate code to model the Spring framework's core technologies. Method Dispatch receives output of Preprocessing module and passes each collection to its corresponding modeling within the Module Dynamic Features module (② in Figure 2).

**4.2.1 Jasmine Syntax.** We use the syntax shown in Figure 3 to describe how Jasmine processes the core technologies of Spring. The above and below of rules in Figure 3 are constraints and IR operation commands, respectively. Each parameter in the IR operation command must strictly abide by the constraints. For example, for the **Insert** rule, when we want to obtain **insert** operation command for inserting statement  $E_1$  before (or after) the  $E_2$  statement of the *Sign* method, we should provide the insert statement  $E_1$ , the target method *Sign*, and the target statement  $E_2$  under the constraints.

We explain the contents of Figure 3 in more detail below:

- **Names, Variables, and Types** are Jasmine's three most basic commands. Jasmine uses **Names** to represent unique identifiers such as class names, method names, and variable (field) names. **Variables** and **Types** respectively indicate the unique identifier in **name** instruction as a variable (field) or type.
- **Allocation, Assignment, Invocation, Return and Expression** are Jasmine's core operating commands. These commands cover Jasmine's operations for the method body during the simulation. Most of them are self-explanatory but some deserve explanation. The **instruction** operation indicates that the rule of **Program** and **Expression** prerequisites can only be operations marked with **instruction**. Both **Assignment** and **Allocation** follow the rule of assigning values from right to left ( $A = B$  or  $A = new T$ ). Jasmine uses **Invocation** operation command to express  $Ret = Base.Sign(Arg)$  statement, where *Arg* and *Ret* are optional.
- Jasmine uses **Program, Function and Class** operation commands to construct method bodies, functions, and synthetic classes. Noted that, according to the needs of the Jasmine simulation process, we must provide a parent class or interface (*Super*) for the synthetic class.
- **Insert** and **Site** operation commands are used for Jasmine's insert operation. Jasmine uses Sites to find the line number of expression *A* in the method body of *Sign*.

**4.2.2 Modeling DI.** The DI Modeling module receives the *PBeans* and *SBeans* from Method Dispatch. It replaces the classes in the two collections that are the targets of Spring AOP with the synthetic proxy classes implemented in Section 4.2.3. Algorithm 3 shows

### Algorithm 3: DI PROCESS

```

Input: SBeans: the set of Singleton Bean.
        PBeans: the set of Prototype Bean.
1 Function DIPROCESS (tCla)
2   generateSingleton();
3   fa ← tCla.field.anno, fd ← tCla.field, iMet ← fd.<init>;
4   if Rule.Injects match fa then
5     if fd.type not in PBeans then
6       iMet ← SingletonFactory.get(fd.type);
7     insertComm(tCla.<init>, iMet, return exp,"before");
8 Function generateSingleton ()
9   vars ← JSyn.var(SBeans.value.name);
10  mets ← JSyn.name("get" + SBeans.value);
11  JSyn.class("SingletonFactory", None, [vars], mets);
12  JSyn.fun(<clinit>, None, JSyn.alloc(vars, SBeans.value));
13  foreach sclass ∈ SBeans do
14    getFunComm(sclass);
15 Function getFunComm (class, field)
16   body ← JSyn.program(JSyn.return(field));
17   return JSyn.fun("get" + class.name, None, body);
18 Function insertComm (TM, IM, expr, BorA)
19   insertCall ← JSyn.call(IM.name, IM.Sign, IM.param);
20   insertSite ← JSyn.site(JSyn.exp(expr));
21   // insert IM into TM before/after expr
   JSyn.insert(TM, insertCall, BorA, insertSite);

```

the modeling of dependency injection. The **JSyn.xx** indicates that Soot IR is operated through the rules provided by Jasmine Syntax. Jasmine generates a singleton factory, named *SingletonFactory* (Line 2), whose static fields are all the elements in the *SBeans*. After that, Jasmine uses the **JSyn.alloc** command to initialize all static fields in <clinit> method of *SingletonFactory*, and constructs a *getter* method for each static field, which is used to simulate dependency injection by returning a global singleton object in Lines 11-14.

Finally, Jasmine scans fields, constructors, and setter methods annotated with special annotations (such as *@Autowired*, *@Inject*, etc.) and gets the declared class about variable or parameter. The DI Modeling module finds the implementation of these classes in the *PBeans* or *SBeans*, and initializes them according to prototype or singleton pattern (Lines 4-7 in Algorithm 3).

**4.2.3 Modeling Spring AOP.** Spring AOP weaves enhanced code into the target class by means of JDK dynamic proxy or CGLIB proxy when the program runs. In this section, we describe how to implement spring AOP behavior by simulating CGLIB proxy. The

CGLIB proxy is implemented by generating the corresponding bytecode directly after the program runs so that information, including method calls and data flows, is not available when static analysis for a project that uses Spring AOP.

Algorithm 4 demonstrates Jasmine models Spring AOP that weaves enhanced code into target methods. Jasmine simulates CGLIB proxy behavior by generating a synthetic subclass of the *TClass*, named *TClass\$\$CGLIB* using the approach as Line 3 in Algorithm 4. Class *TClass\$\$CGLIB* declares and initializes two private fields of type *TClass* and *aopClass*, respectively (Lines 19-20). Lines 21-24 illustrate the generation of proxy methods with the same signature for each inheritable method, and Jasmine decides to assign the proxy method to *proxyM* according to whether the method is the target method. The *curMet* and *preMet* variables of Line 4 in Algorithm 4 are assigned using the synthetic proxy method *proxyM* generated above. *AdviceList* corresponding to the target method is obtained from the *AOPMap* output in Section 4.1 and iterated. After that, algorithm handles different enhancement operations.

First, Algorithm 4 clones method *aopMet* and assigns it to *aroundM* when *aopMet* is a method annotated with *@Around*. This step refers to the compiler optimization by means of function cloning[20, 42, 43]. In order to facilitate the parameter flows from the proxy method to the target method, we add all the formal parameters of *targetM* as additional parameters to method *aroundM* (Line 8 in Algorithm 4). Second, Jasmine inserts the call statement of *aroundM* (Lines 9-11 and 31-34 in Algorithm 4) into the body of the proxy method. Finally, *aroundM* is assigned to *curMet*, and Algorithm 4 starts the next loop. Similarly, Lines 13-16 in Algorithm 4, Jasmine inserts the statement that calls method annotated with *@Before*, *@After* or *@AfterReturning* into *curMet* or *preMet*, respectively. When all methods in *AdviceList* are processed, Jasmine inserts the call statement of the target method *targetM* into the body of *curMet* (Line 17 in Algorithm 4).

**4.2.4 Detecting Entry Points.** The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers. The default handler is based on the *@Controller* and *@RequestMapping*. Unlike traditional Java applications that use the main method as the entry point, an application using the Spring MVC framework has more entry points. The entry point (named Controller in Spring MVC) is annotated with *@Controller* and *@RequestMapping*. Our approach to obtaining possible entry points according to annotations and XML configuration. If the target class handled by Spring AOP is a controller, the actual entry point method is the synthetic proxy method generated by the mock CGLIB as described in Section 4.2.3.

It is impossible to call the private entry point methods directly because of Java's encapsulation principle. Jasmine constructs a synthetic method *call\_Entry* for each controller and makes they invoke all entry point methods in their controllers. Meanwhile, Jasmine instantiates parameter objects of each entry point in the method *call\_Entry*. Finally, the method *main* of Application calls *call\_Entry* of all classes. Jasmine will also construct a *dummyMain* class similar to FlowDroid as the starting point for static analysis, because some applications do not have a *main* method.

#### Algorithm 4: WEAVE PROCESS

**Input:** *AOPMap*: a map of target and advice methods.  
*targetM*: target method.

```

1 TClass ← targetM.getClass, aopClass ← AOPMap.getValClass;
2 Function weaveProcess ()
3   proxyClassComm(TClass);
4   curMet ← proxyM, preMet ← proxyM;
5   foreach aopMet ∈ AOPMap.get(targetM) do
6     aa ← aopMet.Anno, am ← aopMet, site ← "before";
7     if @Around match aa then
8       preMet ← curMet, aroundM ← clone and modify am;
9       if aroundM is not proxyM then
10        site ← "after";
11       insertComm(curMet, aroundM, target exp, site);
12       curMet ← aroundM;
13     else if @Before match aa then
14       insertComm(curMet, aroundM, target exp, site);
15     else if @After or @AfterReturning match aa then
16       insertComm(preMet, am, return exp, site);
17   // calls the insertComm function of Algorithm 3
18   insertComm(curMet, targetM, return exp, site);
19 Function proxyClassComm (class)
20   JSyn.class(class.name+"$$CGLIB", class, [target, aspect],
21     class.methods);
22   JSyn.fun(<init>, None, JSyn.alloc(target, TClass);
23   JSyn.alloc(aspect, aopClass););
24   foreach oriMethod ∈ class do
25     proxyFun ← funComm(oriMethod);
26     if oriMethod is targetM then
27       proxyM ← proxyFun;
28 Function funComm (met)
29   if met is targetM then
30     callStat ← JSyn.call(met.name, met.Sign, met.param);
31     returnStat ← JSyn.return(met.return);
32     body ← JSyn.program(callStat; returnStat);
33     return JSyn.fun(met.name, met.param, body);

```

**4.2.5 Modeling Other.** Jasmine synthesizes a mock object to address the situation that the specific implementation class is dynamically generated on the bytecode or obtained in third-party lib packages through the reflection. For example, the implementation of the interface method is generated by dynamic runtime (in particular the mapper method in Mybatis), the data flow affected by the JoinPoint object in the application, and the parameters initialization of the entry point method.

## 5 EVALUATION

In this section, we investigate the following research questions for evaluating our Jasmine framework.

**RQ1.** How does Jasmine support the core technologies of the Spring framework compared to the state-of-the-art techniques?

**RQ2.** How well does pointer analysis tool perform when being combined with Jasmine in Spring application?

**RQ3.** What is the effect Jasmine brings to existing taint-analysis tools on the analysis result?

**TABLE II. Spring micro-benchmark and results. ID 1-4 are Spring features analysis and ID 5-6 are scenes analysis. *Jasmine<sup>d</sup>* and *Jasmine<sup>s</sup>* represent Jasmine's Doop version and Soot version, respectively.**

ID	Benchmark	Exp	Doop						Soot					
			JackEE		Default		<i>Jasmine<sup>d</sup></i>		SPARK		CHA		<i>Jasmine<sup>s</sup></i>	
			Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	EntryPoint	4	50%	100%	50%	100%	100%	100%	0	0	50%	100%	100%	100%
2	DI(Singleton)	11	100%	54.6%	0	0	100%	72.73%	0	0	0*	0*	81.82%	81.82%
3	DI(Prototype)	7	0	0	0	0	100%	100%	0	0	0*	0*	77.78%	100%
4	Spring AOP	8	0	0	0	0	100%	87.50%	0	0	0	0	100%	87.50%
5	InfoLeak	7	0	0	—	—	33.33%	71.43%	0	0	100%	28.57%	100%	100%
6	Permission	1	0*	0*	0*	0*	0*	0*	0	0	0	0	100%	100%
7	Reachable Methods	158	92.09%	81.01%	81.68%	98.73%	100%	87.34%	94.87%	23.42%	92.48%	77.85%	100%	81.01%
8	Application Edges	346	95.86%	67.05%	96.32%	68.21%	97.38%	96.53%	100%	4.62%	66.37%	86.71%	97.01%	93.64%

0\*: These analysis tools produce 0 as the output if they do not support detection. We use "0\*" to indicate such cases and differentiate it from 0 caused by false alarms.

## 5.1 Experimental Setup

**Implementation.** We have implemented Jasmine as a stand-alone open-source tool, available at Github<sup>1</sup>. Benefiting from manipulating code in the level of intermediate code, Jasmine aims to provide support for different static analysis frameworks. Therefore, each static analysis tool can cooperate with Jasmine to support relevant analysis of applications using the Spring framework, such as Soot[57], Doop[16], and many tools[13, 28, 29, 49] based on them. Interacting with existing tools is very easy, while the output of Jasmine is just IR. For example, we only need to modify the Doop's Fact-generator to enable Doop to support applications using the Spring framework. In addition, Jasmine can cooperate with the well-known taint analysis tool, FlowDroid[13], to support taint analysis for Spring applications.

**Benchmarks.** We present a Spring micro-benchmark<sup>2</sup> to measure Jasmine's support for Spring core technologies on eight metrics. We selected first 4 metrics according to the Spring official documents[8] and other metrics were chosen as they are commonly used in the literature and related work[12, 13, 26]. Table III lists the real-world benchmark programs used for evaluation. All the benchmark programs are open-source programs using the Spring framework: the first 4 entries are from the JackEE benchmarks[12] and remain are selected on Github based on the number of forks and stars. We believe that forks and stars are a good selection criterion because they may indicate these projects' popularity.

All the experiments were carried out on a Docker environment which deployed on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (4 x 12 core) and 128GB of RAM.

## 5.2 RQ1: The Core Technologies Support and Completeness Verification

To answer RQ1, we evaluated eight metrics on the Spring micro benchmark: four Spring technologies analysis of ID 1-4 in Table II (the number of the methods that receive user input, the number of dependency injections in singleton or prototype mode, and the number of examples that use Spring AOP), and ID 5-6 are two scenes analyses that influenced by Spring technologies (the number

of paths that leak passwords and the number of times that permission verification fails), a method reachability analysis (the number of reachable application methods), and a call-graph construction analysis (the number edges of application). We use Precision and Recall in Table II to show the support of each tool for the Spring core technologies.

**5.2.1 The Core Technologies Support.** In order to avoid differences in results due to different framework implementations, we compared the Soot (*Jasmine<sup>s</sup>*) and Doop (*Jasmine<sup>d</sup>*) versions of Jasmine with existing tools, respectively. Doop group compares *Jasmine<sup>d</sup>*, Default (Doop framework contains a web-app logic), and JackEE[12]. Soot group compares *Jasmine<sup>s</sup>*, SPARK [33], and CHA [21].

Context-insensitive pointer analysis does not contextually distinguish heap allocations. Therefore, the result of context-insensitive analysis in prototype mode is similar to that in singleton mode. To differentiate the two modes, we evaluated metrics 2-3 in Table II using a 2-object-sensitive pointer (2obj) analysis (with one context element for heap objects) [32, 39, 40]. Because related dependency packages of the context-sensitive framework Paddle[34] in Soot were unavailable, we used TURNER[29], a selective context-sensitive pointer analysis, to evaluate *Jasmine<sup>s</sup>*. It also demonstrated that Jasmine can support selective context-sensitive pointer analysis for Spring applications. The rest of the metrics were evaluated with context-insensitive. We manually added entry points for Soot-SPARK and Soot-CHA to make evaluations smoothly.

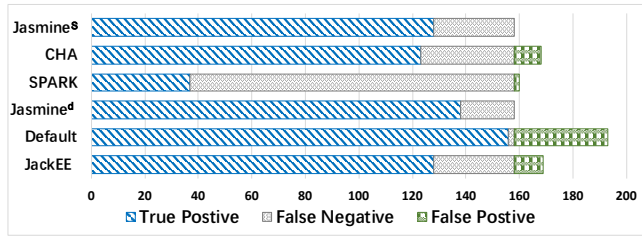
Precision and Recall for ID 1-4 show that Jasmine has better support for Spring core technologies than other tools. Metrics 2 and 4, in both version Jasmine, have false positives and false negatives because we did not consider that bean given different id is treated as different objects in singleton pattern and the AOP approach to interface type declaration. In the Doop group, we used P/Taint[28] to evaluate information leakage. The "-" part indicates time out. The results show that although *Jasmine<sup>d</sup>* has some false positives, and *Jasmine<sup>d</sup>* still detects some correct information leakage paths. We analyzed the datalog[2] rule of P/Taint and found that P/Taint introduced related false positives when processing *Integer.parseInt* and *Integer.valueOf*. We have discussed related issues with the P/Taint maintainer and hope to solve this problem in the future. In the scenes analysis of Soot group, using our taint analysis and authority verification tools, *Jasmine<sup>s</sup>* well supports sensitive information leakage and permission checking of Spring applications.

<sup>1</sup>Publicly available at <https://github.com/SpringJasmine/Jasmine>

<sup>2</sup>Publicly available at <https://github.com/SpringJasmine/Spring-micro-benchmarks>

**TABLE III. Production-benchmark and results. The mall project contains three modules and FEBS-Cloud contains two modules.**

Benchmark	Stars	Fork	Application KLoC	Application Classes	Total Classes	Edge Count (Total)		Reachable Method (Total)	
						JackEE	Jas_JackEE	JackEE	Jas_JackEE
pybbs	1.3k	616	9.3	181	27,717	1,168 (220,711)	2,896 (322,428)	438 (32,207)	715 (42,522)
shopizer	2.1k	2k	90.4	1,339	55,054	13,358 (117,507)	25,678 (129,253)	6,254 (21,574)	6,610 (22,259)
SpringBlog	1.6k	706	1.5	100	18,493	854 (192,412)	1,015 (248,564)	202 (26,691)	226 (33,040)
WebGoat	4.6k	2.6k	15.8	96	12,829	307 (177,557)	372 (227,995)	107 (25,823)	119 (31,620)
mall-admin			60.6	413	18,321	3,140 (161,982)	13,114 (247,618)	1,336 (26,141)	1,947 (36,302)
mall-search	50.1k	21.5k	55.2	271	25,691	806 (149,274)	1,370 (222,414)	305 (22,623)	313 (30,574)
mall-portal			58.6	339	16,148	2,224 (149,390)	5,864 (226,382)	949 (23,485)	1,171 (32,630)
FEBS-Cloud-auth	1.6k	742	4.6	112	18,999	826 (297,283)	1,266 (305,624)	267 (38,477)	327 (39,426)
FEBS-Cloud-system			5.1	134	22,202	1,914 (247,001)	6,165 (276,454)	438 (32,393)	725 (35,966)
jeesite	7.7k	5.9k	25.4	292	20,365	8,685 (324,703)	9,651 (329,086)	1,373 (45,071)	1,688 (45,566)
FEBS-Shiro	5.5k	2.2k	7.2	173	23,923	3,691 (359,274)	8,628 (389,031)	758 (45,054)	1,097 (47,896)
ForestBlog	3.1k	1.4k	3.4	81	10,163	1,106 (128,689)	1,604 (129,284)	369 (16,415)	458 (16,613)
Jeecg-boot	23.9k	9.2k	36.5	549	33,862	26,218 (515,079)	41,559 (560,324)	2,028 (59,300)	3,680 (62,878)
My-Blog	2.1k	672	3.1	59	10,051	548 (99,302)	1,232 (140,676)	130 (16,630)	264 (22,245)
Halo	19.2k	6.6k	28.6	532	30,764	35,653 (519,291)	40,902 (552,997)	2,861 (63,404)	3,599 (66,981)
ruoyi	1.6k	657	21.5	291	23,789	6,858 (438,831)	9,464 (463,446)	1,421 (54,168)	1,938 (56,969)
favorites-web	4.5k	1.7k	6.3	94	11,510	1,233 (95,834)	2,490 (216,665)	499 (16,802)	582 (29,678)
Vblog	6.1k	2.7k	1.1	27	10,474	182 (114,506)	334 (142,507)	98 (18,609)	131 (23,007)
vhr	21.7k	9k	3.7	91	15,648	729 (207,844)	977 (247,123)	267 (30,552)	317 (35,274)
MCMS	1.2k	624	2.3	36	17,524	3,346 (166,721)	3,681 (213,510)	153 (23,955)	199 (30,688)
SpringBlade	5.5k	1.2k	6.2	377	25,219	4,122 (368,985)	6,054 (376,163)	11,74 (46,546)	1477 (47,116)

**Fig. 4. Reachable method.**

**5.2.2 Completeness Verification.** To compare application edges and reachable methods on Spring micro-benchmark call graphs constructed by tools, we used Grays[41] to obtain the runtime call stack information of Spring micro-benchmark. We compared the call graph constructed by the static analysis tool with dynamic, then we filled metrics 7-8 in Table II and drew Figure 4. The results show that JackEE directly uses the Controller and AOP methods as entry points to guide pointer analysis and construct call graph, which ignores some calling relationship between application methods. However, Jasmine fills in the connection and data flow between related methods by synthesizing the methods generated at runtime, so that the static analysis tools can do analysis on Spring applications based on the call graphs.

### 5.3 RQ2: The Enhancements of Jasmine brings to JackEE

To answer RQ2, we extended JackEE with Jasmine, named Jas\_JackEE, and took advantage of Jasmine's support for Spring core technologies. We selected 18 open-source projects on Github as the production-benchmark, shown in Table III, to evaluate the analysis effect of the modified JackEE for Spring applications.

**5.3.1 Completeness.** Columns 4-6 in Table III are the basic properties of the production-benchmark: application KLoC is the number of lines of Java code in the application, and Total Classes contains all the classes about the application and dependent libraries. Edge Count and Reachable Method is the number of edges and reachable methods related to the application generated by using a context-insensitive pointer analysis on Total Classes. In the place marked in red, the programs crashed during the analysis of Total Classes, and the results are produced by running only Application Classes. The columns of Edge Count and Reachable Method in Table III show that Jas\_JackEE uses the facts generated by Jasmine's to have an inevitable increase in the number of reachable methods and calling edges compared to native JackEE.

```

1 public class SmsService { ...
2     public boolean sendSms (...) { ...
3         Map responseMap = JsonUtil
4             .jsonToObject(response.getData(), Map.class);
5         if (responseMap.get("Code").equals("OK")) return true;
6     }
7 }

```

**Listing 3: Example of cause spurious call edges**

Because of space limitation, we have computed the proportion of application edges introduced by Modeling Dynamic Features modules (DI, EntryPoint, AOP and Mocking in Table IV) out of the edges produced by Jas\_JackEE, excluding edges generated by JackEE. Table IV shows that Jas\_JackEE adds many Spring AOP-related edges and dynamic proxy methods for six benchmarks (mall, FEBS-Cloud, FEBS-Shiro, Jeecg-boot, Halo and ruoyi). Both pybbs and My-Blog define dependency injection via specific annotations, which JackEE ignores, and Jasmine considers these cases when simulating in the intermediate code level. The column of Spurious indicates the proportion of spurious edges in added edges. We



TABLE IV. Analysis added edges of Jas\_JackEE

Benchmark	DI	EntryPoint	AOP	Mocking	Spurious
pybbs	74.07%	27.26%	0.85%	7.60%	19.98%
shopizer	30.89%	53.91%	0.98%	6.43%	13.07%
SpringBlog	40.00%	47.37%	0.00%	19.47%	7.89%
WebGoat	22.62%	53.57%	0.00%	15.48%	0.00%
mall-admin	5.51%	6.61%	75.33%	11.99%	4.79%
mall-search	18.65%	3.29%	56.27%	32.13%	3.76%
mall-portal	10.49%	7.20%	78.57%	15.97%	4.97%
FEBS-Cloud-auth	16.11%	19.84%	76.42%	26.13%	20.04%
FEBS-Cloud-system	3.87%	6.85%	49.14%	8.53%	20.77%
jeesite	30.95%	51.31%	0.00%	12.48%	9.36%
FEBS-Shiro	4.89%	9.28%	46.16%	3.74%	23.77%
ForestBlog	40.19%	68.65%	2.12%	24.23%	0.00%
Jeecg-boot	10.07%	14.32%	47.43%	9.50%	12.75%
My-Blog	53.87%	34.81%	0.69%	20.72%	8.70%
Halo	15.21%	8.43%	58.07%	15.30%	22.11%
ruoyi	15.07%	38.10%	41.58%	14.02%	1.67%
favorites-web	43.42%	35.49%	41.01%	25.06%	9.12%
Vblog	41.83%	61.44%	0.00%	30.72%	0.00%
vhr	38.79%	56.20%	0.00%	26.39%	0.00%
MCMS	12.57%	40.11%	18.98%	12.83%	19.25%
SpringBlade	7.44%	27.49%	36.65%	6.04%	1.45%

analyzed and found that these spurious edges are caused by the same kind of problems. We take the spurious edges in pybbs[10] project as an example. In Listing 3, Jasmine handles DI so that method *SmsService.sendSms* becomes reachable in pointer analysis. When analyzing the *sendSms* method, pointer analysis found that the *responseMap* in Line 5 points to an Object object. Therefore, when analyzing call statement of *get("Code")* and *equals("OK")* in Line 6. In order to guarantee the completeness of the results, pointer analysis generated many spurious edges.

Figure 5 shows the completeness of JackEE and Jas\_JackEE's reachable app methods, measured as the proportion of the number of reachable app methods to the number of all app methods. Jas\_JackEE compared with JackEE, the completeness of the reachable method increased by 28.14% (for My-Blog) at most and 0.21% (for mall-search) at least. Because there is no effective dynamic analysis tool that can automatically obtain the reachable methods and calling edges of Spring projects, we could not accurately compute the precision and recall of Jasmine in real-world Spring programs.

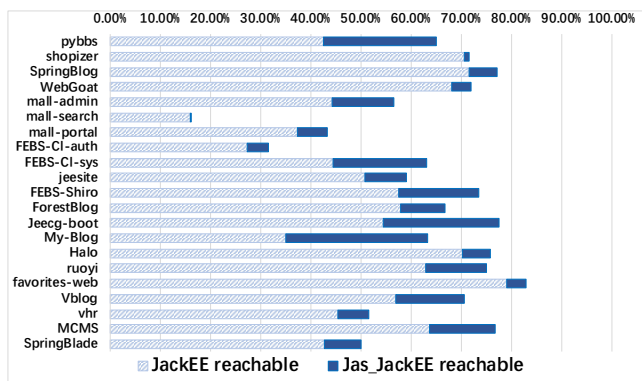
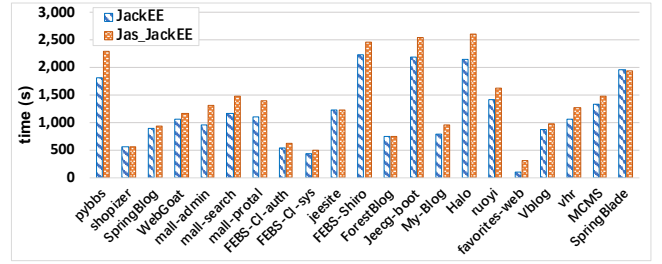
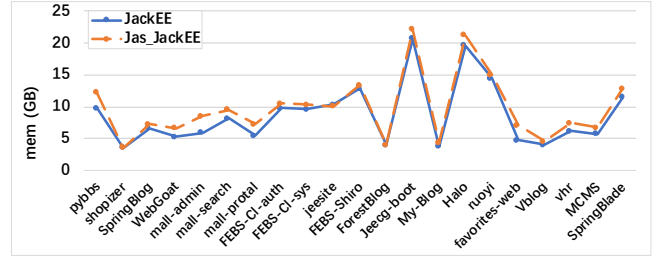


Fig. 5. App methods reachability.



(a) executing time



(b) usage of memory

Fig. 6. Comparison of performance for benchmarks.

5.3.2 *Performance.* While Jasmine increases the number of edges and reachable methods to JackEE, more resources is inevitable. Figure 6 shows the performance of both JackEE and Jas\_JackEE. As we can see, time and memory only increase less than 10% when Jas\_JackEE analyzes most of the projects and improves the coverage for call graph construction and reachable method. For projects that heavily using Spring AOP and dependency injection (pybbs, mall, halo, etc.), although it takes longer time and more memory space than native JackEE, it brings higher coverage. We believe that compared with the increased completeness of the call graph, the overhead of performance from Jasmine can be accepted.

## 5.4 RQ3: The Benefits of Jasmine Brings to FlowDroid

In this section, we intend to answer the benefits Jasmine brings to FlowDroid. First, we used FlowDroid to perform taint analysis on Java projects without changing its origin logic. Then, we combined FlowDroid with Jasmine (denoted as FlowDroid\_Jasmine<sup>3</sup>, using SPARK mode for analysis). At last, we evaluated FlowDroid\_Jasmine by comparing it with FlowDroid\_CHA and FlowDroid\_SPARK on the production-benchmark. We chose the user's input as sources and writing methods as sinks, which can be used to illustrate the advantages Jasmine brings to the static analysis of real-world programs. Table V shows the result on the production-benchmark. As we can see, based on the same pair of sources and sinks, FlowDroid\_Jasmine detects more taint paths than FlowDroid\_CHA and FlowDroid\_SPARK. By analyzing the results, we summarized two reasons for this phenomenon. Firstly, Jasmine supports pointer analysis to process Spring core technologies and patch relevant data flow information. Secondly, in Spring MVC, controllers use class *HttpServletRequest*, entity class, or string as formal parameters to accept

<sup>3</sup>Publicly available at [https://github.com/SpringJasmine/FlowDroid\\_Jasmine](https://github.com/SpringJasmine/FlowDroid_Jasmine)

**TABLE V. Taint analysis results for FlowDroid. "-" on the left is the number of taint paths. On the right is the number of information leakage paths. In parentheses is the number of leakage paths verified after executing programs.**

Benchmark	FlowDroid _CHA	FlowDroid _SPARK	FlowDroid _Jasmine
pybbs	3 - 0	4 - 0	7 - 0
shopizer	61 - 8	20 - 0	97 - 8
SpringBlog	0 - 0	0 - 0	0 - 0
WebGoat	0 - 0	0 - 0	0 - 0
mall-admin	2 - 0	0 - 0	41 - 6(3)
mall-search	0 - 0	0 - 0	5 - 0
mall-portal	1 - 0	0 - 0	20 - 6(3)
FEBS-Cloud-auth	2 - 0	0 - 0	17 - 0
FEBS-Cloud-system	3 - 0	0 - 0	16 - 4
jeesite	10 - 2	5 - 2	19 - 4
FEBS-Shiro	0 - 0	0 - 0	0 - 0
ForestBlog	4 - 0	0 - 0	4 - 0
Jeecg-boot	5 - 0	5 - 0	65 - 4
My-Blog	0 - 0	0 - 0	0 - 0
Halo	0 - 0	0 - 0	31 - 4(4)
ruoyi	0 - 0	0 - 0	65 - 4
favorites-web	0 - 0	0 - 0	20 - 4(2)
Vblog	0 - 0	0 - 0	0 - 0
vhr	0 - 0	0 - 0	0 - 0
MCMS	0 - 0	0 - 0	0 - 0
SpringBlade	4 - 0	4 - 0	4 - 0

incoming requests. Their initialization and process of receiving request information are carried out within the Spring framework. FlowDroid\_CHA and FlowDroid\_SPARK could not detect these as sources, and Jasmine can solve these problems mentioned in Section 4.2.5.

We find that Jasmine can detect forty-four sensitive information leakage paths in nine projects. These leakage paths write the user sensitive information into a log file or print it to the console. After deploying the project and running the functions corresponding to these paths, we manually verified the existence of these twelve sensitive information leakage paths (shown with red background in Table V). These leaks are all related to behaviors generated by core technologies of Spring. We have submitted the vulnerability to the issue of the Github project or emailed it to the author.

We also evaluated P/Taint, a information-flow analysis based on Doop, on benchmark projects. However, whether using JackEE or Jasmine in collaboration with P/Taint, there are many false positives in the detection results and come with heavy efficiency cost. The false positive is that P/Taint generates information-flow between some methods in Java core library is inaccurate. We think that how to improve the accuracy of P/Taint is a topic worthy of study.

## 6 RELATED WORK

In this section, we mainly discuss related work that leverages the idioms of Java web frameworks to address the challenges of analyzing web applications.

### 6.1 Static Reflection Analysis

Several static analyses have resolved calls that use dynamic features. Solar [36, 37] is the first reflection analysis that allows its

soundness to be reasoned about when some assumptions are met and produces significantly improved under-approximations otherwise. Barros et al. [14] present static analysis for Java reflection and Android intents, which frequently appear in Android apps. This work helps to resolve where control flows and what data is passed and improve the precision of downstream analyses. Smaragdakis et al. [48] present an approach for handling reflection in a pointer analysis based on the combination of string-flow and pointer analysis augmented with modeling of partial string flow. Fourtounis et al. [26] observe that the dynamic proxy API is stylized enough to permit static analysis and show how the semantics of dynamic proxies can be modeled straightforwardly as logical rules in the Doop. The above patterns rarely occur in framework implementations. Moreover, frameworks are difficult to analyze for reasons beyond their use of reflection and dynamic proxies.

### 6.2 Framework Analysis

Dietrich et al. [22] generate a driver that supplies such an entry method for Java EE web applications to overcome the challenges of identifying entry points by processes XML configuration files, annotations, and JSPs. Using the information in configuration files, particularly Android [13, 15], has been considerable work on improving analysis precision and sound for framework-based applications. JackEE [12] also introduces techniques and general concepts for identifying and modeling the entry points of enterprise application in a largely framework-independent way. John et al. [53] present Concerto, a system for analyzing framework-based applications by soundly combining concrete interpretation at the framework implementations and abstract interpretation at the application code. TAJ [56] is a taint analysis tool and addresses various attack vectors with techniques to handle reflective calls, flow through containers, nested taint, and issues in generating valuable reports. IBM published F4F [50], a system for effective taint analysis of framework-based web applications. It supports WAFL language to model framework and process configuration files in Java EE that taint analysis engines can use to perform more accurate analysis of web applications. Andromeda [55] is an analysis tool that computes data-flow propagations on demand through constructs call-graph lazily. It resolves virtual calls according to an interprocedural type-inference. ANTaint [58] addresses the problems that applications make heavy use of libraries, native methods, and enterprise-specific frameworks in FlowDroid [13]. It improves scalability by expanding the CG and applying taint propagation on demand for libraries.

## 7 CONCLUSION

With the widespread use of the Spring Framework in enterprise applications, it is impractical to ignore it for static analysis. We have presented Jasmine, a static analysis framework for the Spring programs. Jasmine successfully handles the fundamental problems of static analysis for core technologies of the Spring by manipulating code in the level of Soot IR. Our evaluation on Spring micro-benchmarks and real-world Spring programs demonstrate Jasmine makes static analysis and call graph more completeness. In addition, by combining Jasmine with FlowDroid, we have detected twelve sensitive information leakage paths in three open-source projects. We believe that these results establish Jasmine as a new sweet spot in the well-established static analysis for Spring programs.

## REFERENCES

- [1] 2021. Aspect-oriented programming. [https://en.wikipedia.org/wiki/Aspect-oriented\\_programming](https://en.wikipedia.org/wiki/Aspect-oriented_programming). Accessed July 28, 2021.
- [2] 2021. Datalog. <https://en.wikipedia.org/wiki/Datalog>. Accessed July 28, 2021.
- [3] 2021. Dependency-injection. [https://en.wikipedia.org/wiki/Dependency\\_injection](https://en.wikipedia.org/wiki/Dependency_injection). Accessed July 28, 2021.
- [4] 2021. Halo Project. <https://github.com/halo-dev/halo>. Accessed July 28, 2021.
- [5] 2021. jvm-ecosystem-report-2021. <https://snyk.io/jvm-ecosystem-report-2021/>. Accessed July 31, 2021.
- [6] 2021. Mall Project. <https://github.com/macrozheng/mall>. Accessed July 28, 2021.
- [7] 2021. Spring Boot. <https://spring.io/projects/spring-boot>. Accessed July 28, 2021.
- [8] 2021. Spring Framework. <https://spring.io/projects/spring-framework>. Accessed July 28, 2021.
- [9] 2021. Spring Projects. <https://github.com/search?l=Java&q=Spring&type=Repositories>. Accessed July 28, 2021.
- [10] 2022. pybbs Project. <https://github.com/tomoya92/pybbs>. Accessed Feb 14, 2022.
- [11] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2019. A study of call graph construction for JVM-hosted languages. *IEEE transactions on software engineering* (2019).
- [12] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 794–807.
- [13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oceau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
- [14] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d’Amorim, and Michael D Ernst. 2015. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 669–679.
- [15] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. 2015. Droidel: A general approach to Android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. 19–25.
- [16] 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*. 243–262.
- [17] cglib. 2019. cglib-project. <https://github.com/cglib/cglib>.
- [18] Satish Chandra, Stephen J Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 363–374.
- [19] David R Chase, Mark Wegman, and F Kenneth Zadeck. 1990. Analysis of pointers and structures. *ACM SIGPLAN Notices* 25, 6 (1990), 296–310.
- [20] Dibyendu Das. 2003. Function inlining versus function cloning. *ACM SIGPLAN Notices* 38, 6 (2003), 23–29.
- [21] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.
- [22] Jens Dietrich, François Gauthier, and Padmanabhan Krishnan. 2018. Driver Generation for Java EE Web Applications. In *2018 25th Australasian Software Engineering Conference (ASWEC)*. IEEE, 121–125.
- [23] Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding bugs efficiently with a SAT solver. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 195–204.
- [24] Stephen J. Fink et al. 2019. T.J. Watson Libraries for Analysis (WALA). <http://wala.sourceforge.net>.
- [25] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective tpestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–34.
- [26] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 209–220.
- [27] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1749–1766.
- [28] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
- [29] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- [30] Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–47.
- [31] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 507–518.
- [32] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices* 42, 6 (2007), 278–289.
- [33] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.
- [34] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18, 1 (2008), 1–53.
- [35] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Oceau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.
- [36] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *International Static Analysis Symposium*. Springer, 162–180.
- [37] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.
- [38] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.
- [39] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.
- [40] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 1 (2005), 1–41.
- [41] oldmanpushcart. 2019. greys-anatomy. <https://github.com/oldmanpushcart/greys-anatomy>.
- [42] Dmitry Petrashko, Vlad Ureche, Ondřej Lhoták, and Martin Odersky. 2016. Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 394–409.
- [43] Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic dispatch of context-sensitive optimizations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.
- [44] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. Blankit library debloating: Getting what you want instead of cutting what you don’t. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 164–180.
- [45] Michael Pradel, Ciera Jaspán, Jonathan Aldrich, and Thomas R Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 925–935.
- [46] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706.
- [47] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.
- [48] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.
- [49] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
- [50] 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*. 1053–1068.
- [51] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*. Springer, 196–232.
- [52] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–291.
- [53] John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming*

- Languages* 3, POPL (2019), 1–29.

[54] Paolo Tonella. 2005. Reverse engineering of object oriented code. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. IEEE*, 724–725.

[55] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 210–225.

[56] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[58] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. 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*. 1477–1486.

[59] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: an evolving graph system for flow-and context-sensitive analyses of million lines of C code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 914–929.