Jasmine: A Static Analysis Framework for Spring Core Technologies

Miao Chen
chenmiao_nsrc@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Tengfei Tu*
tutengfei.kevin@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Hua Zhang
zhanghua_288@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Qiaoyan Wen*
wqy@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Weihang Wang
weihangw@usc.edu
University of Southern California
Los Angeles, USA

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 for 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 complete call graphs are required.

CCS CONCEPTS

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

KEYWORDS

static analysis, points-to analysis, Spring framework

*Corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556910

ACM Reference Format:
Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. 2022. Jasmine: A Static Analysis Framework for Spring Core Technologies. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556910

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 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 is often used to detect security vulnerabilities and performance issues. In static analysis research, Java has many the well-known research frameworks, such as Soot[58], Doop[17], and WALA[25, 52]. Antoniadis et al.[13] 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[39]. As far as we know, the literature rarely contains techniques specifically for the Spring applications in static analysis and often ignores the influence of Spring framework. IBM published F4F[51] to make taint analysis engines perform more accurate analysis of web applications. Alibaba's ANTaint[59] simulates framework to enhance the capabilities of FlowDroid in enterprise applications. JackEE[13] 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 ignore the Spring core technologies and affect the accuracy of the results generated by software engineering tools where call graphs are required. For example, existing taint analysis tools based on these incomplete graphs 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 significant static analysis technology that computes the possible values of pointer variables in a program[31, 34, 53]. Such information is essential for reasoning about call graph and alias in object-oriented programs. Therefore, it is widely used in software engineering tools, such as taint analysis[14, 29, 36], program verification[26, 46], program debloating[28, 45], and bug detection[19, 24, 47, 60]. 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 such information is invisible during static analysis[11, 32].

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).
- We propose an approach to simulate Spring core technologies in the level of intermediate representation (IR) and construct the Jasmine framework. Jasmine parses Spring configuration files to obtain the specific functions. According to this information, Jasmine simulates the possible behaviors of Spring core technologies (Section 4).
- We present a micro-benchmark and a real-world benchmark containing 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 implemented FlowDroid and found twelve user information leakage paths in three real-world 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[48]. Such information is essential for reasoning about call graph and alias in object-oriented programs. Therefore, it is widely used in software engineering tools. Pointer sets describe the relations between variables and abstract objects. We can compute a points-to relation

$$pt \subseteq Var \times Obj$$

with *Var* being program variables set and *Obj* being as abstract objects set. Abstract objects are represented as allocation sites, i.e., instructions that allocate objects (e.g., **new** in Java)[20].

Based on the rules that involved pointer expressions[12, 52], pointer analysis can build the object flow graph (OFG)[55] and call graph of a program. Although the literature is not entirely consistent on high-level terminology, pointer analysis is a near-synonym of alias analysis.

2.2 Spring Core Technologies

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

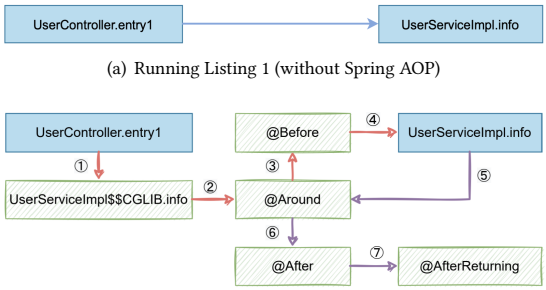
```
@RequestMapping("/user")
@Controller public class UserController{
  @Autowired UserService userService;
  // userService=SingletonFactory.getUserServiceImpl();
  // userService=new UserServiceImpl();
  // userService=new UserService$$CGLIB();
  @GetMapping("/entry1")
  public void entry1(String pword){
    userService.info(pword);
  }
}
@RequestMapping("/admin")
@Controller public class AdminController{
  @Autowired UserService userService;
  @GetMapping("/entry2")
  private void entry2(){
    Print(userService.getPwd());
  }
}
// @Scope("prototype")
@Service class UserServiceImpl implements UserService{
  private static String pwd;
  public String info(String pword){
    this.pwd = pword;
    invoke();
    Print("pword:"+pword);
    return pword;
  }
  public void invoke(){...}
  public String getPwd(){return pwd;}
}
```

Listing 1: Example of using Dependency Injection

```
@Aspect
@Component public class LogAspect {
  // execution(* *..UserServiceImpl.invoke(..))
  // execution(* *..AdminController.entry2(..))
  @Around("execution(* *..UserServiceImpl.info(..))")
  public Object doAround(ProceedingJoinPoint pdj){
    Print(pdj.getArgs());
    return pdj.proceed();  //Execute target method
  }
  @AfterReturning(returning="res",
      value="execution(* *..UserServiceImpl.info(..))")
  public void returning(JoinPoint jp,Object res){...}
  // @Before, @After, @AfterReturning method
}
```

Listing 2: Example of using Spring AOP

2.2.1 Dependency Injection. "Inversion of Control (IoC), also known as dependency injection (DI)"[8], is one of the core technologies of the Spring framework. It is a process whereby objects define their dependencies through factory method arguments, constructor arguments, or properties that are set on the object instance after factory



(a) Running Listing 1 (without Spring AOP)

(b) Running Listing 1 and Listing 2 (with Spring AOP)

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.

method constructs or returns it. 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* (Lines 2 and 21), respectively. Therefore, the IoC container creates the beans of two classes and injects dependencies according to the field annotated with *@Autowired*. When the program runs, *userService* points to the object of *UserServiceImpl*, and *entry1* 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. Without modifying the code, AOP adds additional behavior to existing code (an advice). AOP allows adding behaviors that are not business logic (such as logging) 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[27] or CGLIB proxy[18].

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, the 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, the target class is *UserServiceImpl* and the target method is *info*). Spring generates a dynamic proxy method (*UserServiceImpl$$CGLIB.info* in Figure 1(b)) for the target method and *UserServiceImpl$$CGLIB.info* calls the *advice methods* (the method annotated with *@Around*, *@Before*, *@After*, and *@AfterReturning*) in the order of ②-⑦ illustrated 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[32], which may lead to unsound results, etc. In this section, we dissect how DI and Spring AOP impact on 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 the field is processed by DI, pointer analysis treats it as uninitialized. For example, in Listing 1, pointer analysis could not compute objects that *userService* refers 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. Therefore, taint analysis could not find a leakage path (Line 26) based on this incomplete call graph.

Spring framework uses the singleton pattern to instantiate objects by default (similar to Line 4 in Listing 1) when executing DI, which means that the bean (object) of the class is unique to the whole program. When using the *@Scope("prototype")* in Line 20, the Spring framework will use the prototype pattern and each allocation is considered as 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 *pword* of *info* (Line 9 in Listing 1), method *entry2* in class *AdminController* will obtain the same value by calling *getPwd* since 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 a lot of reflection and dynamic proxies used in Spring framework, pointer analysis cannot handle the application using spring AOP and the generated call graph may lose the execution sequences in Figure 1(b). An unsound call graph of the application's core part may cause false negative during static analysis. 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*, and then Line 7 in Listing 2 obtains and prints the password.

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 in the source code, static analysis tools don't know how to handle these situations. We dissect the mechanism behind this as follows.

- Due to the proxy-based nature of Spring's AOP framework, calls within the target object are not intercepted[8]. As described in Section 2.2.2, we change the pointcut expression of *LogAspect* to the comment in Line 3 of Listing 2. Spring AOP generates a dynamic proxy method for *invoke*. In Line 25

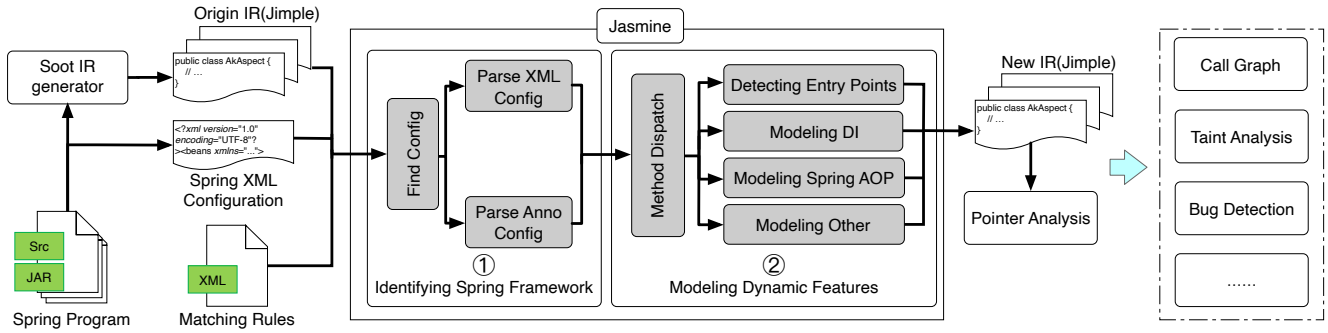


Fig. 2. Overview of Jasmine

of Listing 1, *info* calls method *invoke* belonging to the same class, so the dynamic proxy method was not called, 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 1 is 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$$CGLIB* is a null pointer and will cause a null pointer exception when we accessed private method (the EntryPoint in Table II).

4 METHODOLOGY

This section introduces Jasmine: an approach to simulate the core technologies of Spring Framework for solving the problems mentioned in Section 3 by instrumenting on Soot IR[58]. 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 configuration 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.

TABLE I. Matching Rules

Name	Example	# of Rules
Beans	(class,"Lorg/springframework/.../Service;")	12
Protype	(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

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 (①in Figure 2). We collected annotations related to DI, Spring AOP and entry points from the Spring framework documentation[8], and sorted them into Matching Rules as shown in Table I. Each rule provides the scope and the Jimple IR of annotation. Algorithm 1 shows how to process the annotation and identify the function of a method in a given program. The parsing process of the Spring XML configuration is similar.

Algorithm 1: ANNOTATION PROCESS

```
Input: JimpleClass: 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.Protype 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
```

Three sets are initialized to empty 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 (Line 2 in Algorithm 1), and obtaining the annotation information on the class. 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, Algorithm 1 analyses the annotations of each method in this class and places the suitable methods into the EPMethods set (Lines 9-11). Then, the set are output to the next module for subsequent processing.

4.1.2 Processing Spring AOP. In Algorithm 2, the variable *targetM* and two sets are initialized in Line 1: *targetM* points to target

Names: $\frac{A \in (a...zA...Z\$<>)*}{\boldsymbol{name}(A)}$ **Variables**: $\frac{\boldsymbol{name}(A)}{\boldsymbol{var}(A)}$ **Types**: $\frac{\boldsymbol{name}(A)}{\boldsymbol{type}(A)}$

Assignment: $\frac{\boldsymbol{var}(A) \quad \boldsymbol{var}(B)}{\boldsymbol{instruction}(\boldsymbol{assign}(A,B))}$ **Allocation**: $\frac{\boldsymbol{var}(A) \quad \boldsymbol{type}(T)}{\boldsymbol{instruction}(\boldsymbol{alloc}(A,T))}$ **Return**: $\frac{\boldsymbol{var}(R)}{\boldsymbol{instruction}(\boldsymbol{return}(R))}$

Invocation: $\frac{\boldsymbol{var}(Base) \quad \boldsymbol{name}(Sign) \quad \boldsymbol{var}(Arg) \quad \boldsymbol{var}(Ret)}{\boldsymbol{instruction}(\boldsymbol{call}(Base,Sign,Arg,Ret))}$ **Expression**: $\frac{\boldsymbol{instruction}(E)}{\boldsymbol{instruction}(\boldsymbol{exp}(E))}$

Class: $\frac{\boldsymbol{name}(Name) \quad \boldsymbol{type}(Super) \quad \boldsymbol{var}(Field) \quad \boldsymbol{fun}(Sign)}{\boldsymbol{class}(Name,Super,Field,Sign)}$ **Function**: $\frac{\boldsymbol{name}(Sign) \quad \boldsymbol{var}(Arg) \quad \boldsymbol{program}(Body)}{\boldsymbol{fun}(Sign,Arg,Body)}$

Insert: $\frac{\boldsymbol{exp}(E_1) \quad \boldsymbol{fun}(Sign) \quad \boldsymbol{exp}(E_2)}{\boldsymbol{insert}(Sign,E_1,[before|after]\boldsymbol{site}(E_2))}$ **Site**: $\frac{\boldsymbol{exp}(A)}{\boldsymbol{site}(A)}$ **Program**: $\frac{\boldsymbol{instruction}(I) \quad \boldsymbol{program}(P)}{\boldsymbol{program}(I;P)}$

Fig. 3. Jasmine Syntax

Algorithm 2: AOP PROCESS

Input: *class*: 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 AdivceList;
9     replace AOPMap(targetM) with Sort(AdviceList);
```

method of Spring AOP, AdviceList represents a set of advice methods that will enhance the target method's behavior, and *AOPMap*, which denotes the mapping relations between target method of Spring AOP and a set of related advice methods[8]. The algorithm iterates through all annotations on the method in *class* of the Input. If the annotation in Line 3 is a pointcut expression, then the algorithm parses the pointcut expression to find the target method and assigns it to *targetM*. Otherwise, advice methods are stored in the *AdviceList*. In Lines 7-9, if the *targetM* is not empty, get the advice methods in *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 outputs *AOPMap* to the next module.

4.2 Modeling Dynamic Features

In this section, we describe how to manipulate code in the level of intermediate representation to model the Spring framework's core technologies. The Method Dispatch receives the output from the Identifying Spring Framework module and passes each set to its corresponding sub-module within the Modeling Dynamic Features module (②in 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 each rule 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 the **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 basic rules. 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 rules. These rules cover Jasmine's operations for the method body during the simulation. Most of them are self-explanatory but some deserve an 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 the **Invocation** operation command to express *Ret = Base.Sign(Arg)* statement, where *Arg* and *Ret* are optional.
- Jasmine uses **Program**, **Function**, and **Class** operation rules 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 rules are used for Jasmine's insert operation. Jasmine uses **Site** 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 sets that are the targets of Spring AOP with the synthetic proxy classes implemented in Section 4.2.3. Algorithm 3 shows the modeling of dependency injection. The algorithm starts by iterating through all the application classes in origin Soot IR, then tClas, SBeans and PBeans are passed as arguments to the function DIProcess (Lines 1-2). The **JSyn.xx** indicates that Soot IR is operated according to the rules provided by Jasmine Syntax in Figure 3. Function generateSingleton (Line 10) generates a singleton factory, named *SingletonFactory* (Line 13), whose static fields are all the elements in the SBeans. After that, Jasmine uses the **Jsyn.alloc** command to initialize all static fields in the *<clinit>* method of *SingletonFactory*, and function getFunComm constructs a *getter* method for each static field in Lines 15-16 and 17-19, which is used to simulate DI by returning a global singleton object.

Finally, Jasmine scans fields, constructors, and setter methods annotated with special annotations (such as *@Autowired*, *@Inject*,

Algorithm 3: DI PROCESS

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

etc.) and gets the declared class about variable or parameter (Line 6 in Algorithm 3). The DI Modeling module finds the implementation of these classes in the PBeans or SBeans, and initializes them according to the prototype or singleton pattern (Lines 7-9). Function insertComm uses **JSyn.call** to generate a call statement for the insertion method *IM* and locates the statement *expr* at the location of the target method *TM*. After that, use **JSyn.insert** to insert the statement calling *IM* before the statement returns in the *TM* body.

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 performing static analysis on a project that uses Spring AOP.

Algorithm 4 demonstrates Jasmine modeling Spring AOP that weaves enhanced code into target methods. Jasmine simulates CGLIB proxy behavior by generating a synthetic subclass of the *TClass*, named *TClass$$CGLIB* using the approach in Line 3. Class *TClass$$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 variables *curMet* and *preMet* of Line 4 in Algorithm 4

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 @AfterReturing match aa then
16             insertComm(preMet, am, return exp, site);
       // calls the insertComm function of Algorithm 3
17     insertComm(curMet, targetM, return exp, site);
18 Function proxyClassComm(class)
19     JSyn.class(class.name+"$$CGLIB", class, [target, aspect],
         class.methods);
20     JSyn.fun(<init>, None, JSyn.alloc(target, TClass);
         JSyn.alloc(aspect, aopClass););
21     foreach oriMethod ∈ class do
22         proxyFun ← funComm(oriMethod);
23         if oriMethod is targetM then
24             proxyM ← proxyFun;
25 Function funComm(met)
26     if met is targetM then
27         callStat ← JSyn.call(met.name, met.Sign, met.param);
28     returnStat ← JSyn.return(met.return);
29     body ← JSyn.program(callStat; returnStat);
30     return JSyn.fun(met.name, met.param, body);
```

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 begin to handle different enhancement operations.

First, Algorithm 4 clones *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[21, 43, 44]. In order to facilitate the parameter flowing from the proxy method to the target method, we add all the formal parameters of *targetM* as additional parameters to the method *aroundM* (Line 8 in Algorithm 4). Second, Jasmine inserts the call statement of *aroundM* (Lines 9-11 in Algorithm 4) into the body of the proxy method. Finally, *aroundM* is assigned to *curMet*, and the algorithm starts the next iteration. Similarly, Lines 13-16, 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).

TABLE II. Spring micro-benchmark and results. ID 1-4 are Spring features analysis and ID 5-6 are scenes analysis. $Jasmine^d$ and $Jasmine^s$ represent Jasmine's Doop version and Soot version, respectively. P. indicates precision and R. indicates recall.

ID	Benchmark	Exp	Doop									Soot								
			JackEE			Default			$Jasmine^d$			SPARK			CHA			$Jasmine^s$		
			P.(%)	R.(%)	F1(%)	P.(%)	R.(%)	F1(%)	P.(%)	R.(%)	F1(%)	P.(%)	R.(%)	F1(%)	P.(%)	R.(%)	F1(%)	P.(%)	R.(%)	F1(%)
1	EntryPoint	4	50	100	66.67	50	100	66.67	100	100	100	0	0	0	50	100	66.67	100	100	100
2	DI(Singleton)	11	100	54.60	70.63	0	0	0	100	72.73	84.21	0	0	0	0*	0*	0*	81.82	81.82	81.82
3	DI(Prototype)	7	0	0	0	0	0	0	100	100	100	0	0	0	0*	0*	0*	77.78	100	87.50
4	Spring AOP	8	0	0	0	0	0	0	100	87.50	93.33	0	0	0	0	0	0	100	87.50	93.33
5	InfoLeak	7	0	0	0	—	—	—	33.33	71.43	45.45	0	0	0	100	28.57	44.44	100	100	100
6	Permission	1	0*	0*	0*	0*	0*	0*	0*	0*	0*	0	0	0	0	0	0	100	100	100
7	Reachable Methods	158	92.09	81.01	86.20	81.68	98.73	89.40	100	87.34	93.24	94.87	23.42	37.57	92.48	77.85	84.54	100	81.01	89.51
8	Application Edges	346	95.86	67.05	78.91	96.32	68.21	79.86	97.38	96.53	96.95	100	4.62	8.83	66.37	86.71	75.19	97.01	93.64	95.30

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

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 based on the Spring MVC framework has multiple entry points. The entry point (named Controller in Spring MVC) is annotated with *@Controller* and *@RequestMapping*. Our approach identifies 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 them invoke all entry point methods in controller. 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.

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 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.

4.3 Threats to Validity

Threats to the validity of our study could come from some aspects. In addition to DI and AOP, the remaining Spring core features also affect the completeness of the call graph (yet not as significant as DI and AOP). Jasmine will gradually support them in the future. Jasmine only focuses on Spring programs. Some Java applications are developed without using Spring framework, and Jasmine cannot improve the performance of static analysis for these applications. Despite these limitations, Jasmine has made great efforts to supplement the construction of application call graph. We believe Jasmine can provide significant benefits for many program analysis scenes of Spring application where more complete call graphs are required.

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 the pointer analysis tool perform when combined with Jasmine on the Spring application?

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

5.1 Experimental Setup

Implementation. We have implemented Jasmine as a stand-alone open-source tool, available on Github[1]. Benefiting from manipulating code in the level of intermediate representation, Jasmine provides support for different static analysis frameworks and each static analysis tool, including Soot[58], Doop[17], and so on[14, 29, 30, 50], can cooperate with Jasmine to perform relevant analysis on the applications based on the Spring framework. Interacting with existing tools is very easy, since the output of Jasmine is just IR. For example, we only need to modify the Doop's Fact-generator to enable it to support applications using the Spring framework. In addition, Jasmine can cooperate with the well-known taint analysis tool, FlowDroid[14], to support taint analysis for Spring applications.

Benchmarks. We presented a Spring micro-benchmark[2] to measure Jasmine's support for Spring core technologies with eight metrics. We selected the first 4 metrics (ID 1-4 in Table II) according to the Spring official documents[8] and other metrics (ID 5-8 in Table II) are chosen as that are commonly used in the literature and related work[13, 14, 27]. 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[13] and the remains are selected on Github based on the number of forks and stars. We believe that the number of forks and stars is a good selection criterion because they may indicate these projects' popularity. Most of our selected applications have at least two years of history and

[1]Publicly available at https://github.com/SpringJasmine/Jasmine
[2]Publicly available at https://github.com/SpringJasmine/Spring-micro-benchmarks

still under maintenance by the developers currently. As another selection criterion, applications that use different versions of the Spring framework can prove the generality of Jasmine's analysis results. The earliest release time of the Spring framework version used in benchmark applications is December 2015 (jeesite), and the latest release time is October 2021 (ruoyi).

All the experiments were carried out on a Docker environment which deployed on a machine with 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 evaluate eight metrics on the Spring micro-benchmark as illustrated in Table II. ID 1-4 are four Spring technologies analysis metrics (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 metrics that analyse the scenes that influenced by Spring core technologies (the number of paths that leak passwords and the number of times that permission verification fails), ID 7 is a method reachability analysis (the number of reachable application methods), and ID 8 is a call graph construction analysis metric (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 ($Jasmine^s$) and Doop ($Jasmine^d$) versions of Jasmine with existing tools, respectively. Doop group compares $Jasmine^d$, Default (Doop framework contains a web-app logic), and JackEE[13]. Soot group compares $Jasmine^s$, SPARK[34], and CHA[22].

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)[33, 40, 41]. Because related dependency packages of the context-sensitive framework Paddle[35] in Soot were unavailable, we used TURNER[30], a selective context-sensitive pointer analysis tool, to evaluate $Jasmine^s$. 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, Recall and F1 for ID 1-4 show that Jasmine has better support for Spring core technologies than other tools. Metrics 2 and 4, in both versions of Jasmine, have false positives and false negatives because we did not consider the situation that bean given different id is treated as different objects in singleton pattern and the AOP can optionally be declared using an interface. In the Doop group, we used P/Taint[29] to evaluate information leakage. The "—" indicates time out. The results show that although there exist some false positives, $Jasmine^d$ can 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, $Jasmine^s$ well supports sensitive information leakage and permission checking of Spring applications. CHA is a call graph construction algorithm that uses the class hierarchy to compute which method can be invoked by objects of each class type. CHA generates a call graph with low accuracy (may introduce false edges to the call graph)[59]. When the information leakage detection tool uses CHA as the call graph construction algorithm and satisfies the conditions that the entry points are manually added and the process of receiving request information being not carried out within the Spring framework, it can detect the leakage path described in Section 3.1.

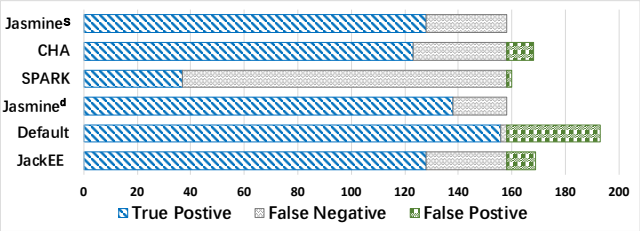


Fig. 4. Reachable method.

5.2.2 Completeness Verification. To compare application edges and reachable methods on Spring micro-benchmark call graphs constructed by different tools, we used Grays[42] to obtain the runtime call stack information of Spring micro-benchmark. We compared the call graph constructed by the static analysis tool with that constructed dynamic, then we filled metrics 7-8 in Table II and drew Figure 4.

In Figure 4, though we manually added entry points for CHA and SPARK to make evaluations smoothly, they still produced high false negatives. Default configuration yields the best true positive result but has the highest false positives compared to others. We analyzed its Datalog[2] rules and found that Default extensively uses the mock approach, where all classes generate a mock object and then call themselves, resulting in most methods being reachable except the method generated by dynamic runtime.

Table II and Figure 4 illustrate the results which ignore some calling relationships between application methods, since JackEE directly uses the Controller and AOP methods as the entry points to guide pointer analysis and construct call graph. Fortunately, 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 analyse Spring applications based on the call graphs. Indeed, Jasmine has false negatives because we did not consider AOP declared using an interface in the framework, we will support this in the future.

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 (mall project contains three modules and

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

Benchmark	Stars	Fork	Spring Version	Application KLoC	Application Classes	Total Classes	Edge Count (Total)		Reachable Method (Total)	
							JackEE	Jas_JackEE	JackEE	Jas_JackEE
pybbs	1.3k	616	5.2.2	9.3	181	27,717	1,168 (220,711)	2,896 (322,428)	438 (32,207)	715 (42,522)
shopizer	2.1k	2k	5.2.11	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	4.3.14	1.5	100	18,493	854 (192,412)	1,015 (248,564)	202 (26,691)	226 (33,040)
WebGoat	4.6k	2.6k	5.3.9	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	5.2.6	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	5.2.1	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	4.1.9	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	5.3.3	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	4.3.19	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	5.2.10	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	5.1.2	3.1	59	10,051	548 (99,302)	1,232 (140,676)	130 (16,630)	264 (22,245)
Halo	19.2k	6.6k	5.3.8	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	5.3.12	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	5.0.4	6.3	94	11,510	1,233 (95,834)	2,490 (216,665)	499 (16,802)	582 (29,678)
Vblog	6.1k	2.7k	5.2.6	1.1	27	10,474	182 (114,506)	334 (142,507)	98 (18,609)	131 (23,007)
vhr	21.7k	9k	5.3.1	3.7	91	15,648	729 (207,844)	977 (247,123)	267 (30,552)	317 (35,274)
MCMS	1.2k	624	5.2.12	2.3	36	17,524	3,346 (166,721)	3,681 (213,510)	153(23,955)	199 (30,688)
SpringBlade	5.5k	1.2k	5.3.8	6.2	377	25,219	4,122 (368,985)	6,054 (376,163)	11,74(46,546)	1477 (47,116)

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%

FEBS-Cloud contains two modules), shown in Table III, to evaluate the analysis effect of the modified JackEE for Spring applications.

5.3.1 Completeness. Columns 5-7 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 (Application Classes) and dependent libraries. Edge Count and Reachable Method is the number of edges and reachable methods related to the application generated by conducting 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 only running the 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 achieve a remarkable 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 on six benchmarks(mall, FEBS-Cloud, FEBS-Shiro, Jeecg-boot, Halo and ruoyi). Both pybbs and My-Blog define dependency injection via specific annotations, which is ignored by JackEE, and Jasmine considers these cases when simulating in the intermediate representation level. The column of Spurious indicates the proportion of spurious edges in the added edges. We 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 the 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.

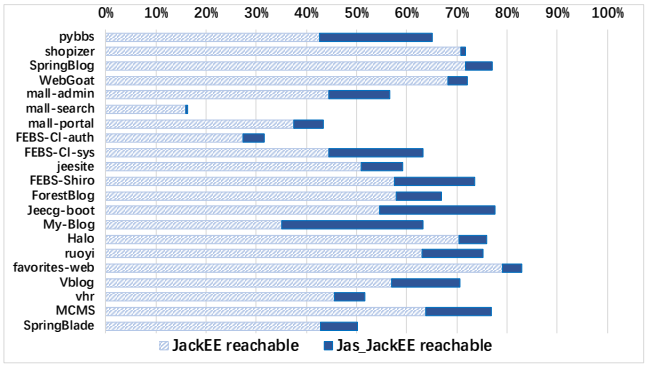


Fig. 5. App methods reachability.

Therefore, when analyzing the call statement of *get("Code")* and *equals("OK")* in Line 6, pointer analysis generated many spurious edges to guarantee the completeness of the results. We mark the edges in added edges of Jas_JackEE as spurious with the following two features. First, the edges that have the same calling method, and the called methods have the same method signature but do not belong to the same class. Second, the edges that the number of them is far more than the number of corresponding call statements in the source code.

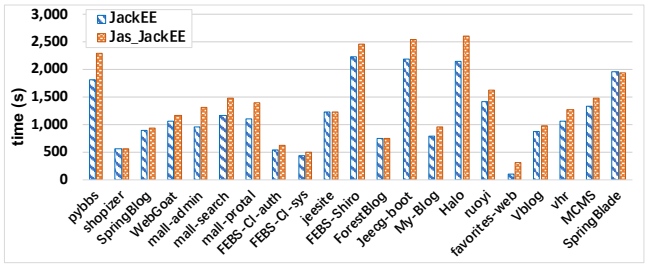
Figure 5 shows the completeness of JackEE and Jas_JackEE's reachable app methods, measured by the proportion of the number of reachable app methods in the number of all app methods. Compared Jas_JackEE with JackEE, the completeness of the reachable method increased by 28.14% (for My-Blog) at most and 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.

5.3.2 Performance. Since Jasmine finds more edges and reachable methods to JackEE, more resource consumption 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% while Jas_JackEE analyzes most of the projects and improves the coverage for call graph and reachable method identification. For projects that heavily using Spring AOP and dependency injection (pybbs, mall, halo, etc.), although it takes longer time and more memory space to analyse than native JackEE, it brings higher coverage. We believe that compared with the increased completeness of the constructed call graph, the overhead introduced by Jasmine is acceptable.

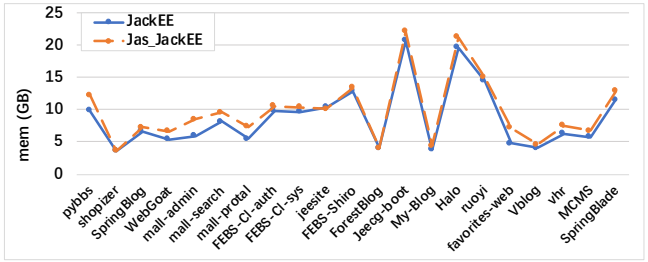
5.4 RQ3: The Benefits of Jasmine Brings to FlowDroid

In this section, we intend to evaluate 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 FlowDriod with Jasmine (denoted as FlowDroid_Jasmine[3], 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, then conducted the above three

[3]Publicly available at https://github.com/SpringJasmine/FlowDroid_Jasmine



(a) executing time



(b) usage of memory

Fig. 6. Comparison of performance for benchmarks.

modes on the production-benchmark and obtained the results as shown in Table V, which can be used to illustrate the advantages that Jasmine brings to the static analysis of real-world programs. 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. We manually compared the taint paths in Table V with the source code to verify that these taint paths are valid. 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 *HttpServlet*, entity class, or string as formal parameters to accept 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 as mentioned in Section 4.2.5.

We found that FlowDroid_Jasmine can detect forty-four sensitive information leakage paths in nine projects. These leakage paths write the user's 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 excluded thirty-two paths that may be blocked-up by control flow statements and manually verified the existence of the other twelve paths (shown with red background in Table V). These leaks are all related to behaviors generated by the core technologies of Spring. We have submitted these security defects to the issue of the Github project or emailed it to the author.

We also evaluated P/Taint, an information-flow analysis tool 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 that come with heavy efficiency cost. The false positive is the inaccurate information-flow

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	0 - 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

between some methods in Java core library that generated by P/-Taint. 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[37, 38] 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.[15] present static analysis for Java reflection. This work helps to resolve where control flows and what data is passed and improve the precision of downstream analyses. Smaragdakis et al.[49] 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.[27] 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. We analyze the source code of the Spring framework and find that the above patterns rarely occur in framework implementations. Moreover, frameworks are difficult to analyze for reasons beyond their use of reflection and dynamic proxies. Jasmine models Spring core technologies instead of analyzing them to avoid scalability or precision loss due to their complexity.

6.2 Framework Analysis

Whole-program Analysis. Dietrich et al.[23] 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[14, 16], has been considerable work on improving analysis precision and sound for framework-based applications. JackEE[13] 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.[54] 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. Jasmine not only focuses on identifying entry points, but also simulates the Spring core technologies on the Soot IR to achieve a complete analysis of Spring applications. Jasmine also supports integration into the current state-of-the-art static analysis tools based on Soot and Doop.

Demand-driven Analysis. TAJ[57] 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[51], 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[56] 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[59] addresses the problems that applications make heavy use of libraries, native methods, and enterprise-specific frameworks in FlowDroid[14]. It improves scalability by expanding the call graph and applying taint propagation on demand for libraries. However, they are specifically geared towards taint analysis and their modeling is incomplete. Jasmine can provide the ability to model all value-flow in the program, which is completely different from the above approaches to model information flow.

7 CONCLUSION

With the widespread use of the Spring Framework in enterprise applications, it is impractical to ignore it for static analysis. We present Jasmine, a static analysis framework for the Spring programs. Jasmine successfully handles the fundamental problems of static analysis for the Spring core technologies by manipulating code in the level of Soot IR. Our evaluation on Spring micro-benchmarks and real-world Spring programs demonstrate that Jasmine makes static analysis and call graph more complete. In addition, by combining Jasmine with FlowDroid, we discover 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.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their insightful comments. This work is supported by Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing.

REFERENCES

[1] 2021. 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. 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, Ondrej Lhoták, Julian Dolby, and Frank Tip. 2019. A study of call graph construction for JVM-hosted languages. *IEEE transactions on software engineering* (2019).

[12] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. Citeseer.

[13] 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.

[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[15] 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.

[16] 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.

[17] 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.

[18] cglib. 2019. cglib-project. https://github.com/cglib/cglib.

[19] 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.

[20] David R Chase, Mark Wegman, and F Kenneth Zadeck. 1990. Analysis of pointers and structures. *ACM SIGPLAN Notices* 25, 6 (1990), 296–310.

[21] Dibyendu Das. 2003. Function inlining versus function cloning. *ACM SIGPLAN Notices* 38, 6 (2003), 23–29.

[22] 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.

[23] 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.

[24] Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding bugs efficiently with a SAT solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 195–204.

[25] Stephen J. Fink et al. 2019. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[26] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–34.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–47.

[32] 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.

[33] 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.

[34] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.

[35] 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.

[36] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, 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.

[37] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *International Static Analysis Symposium*. Springer, 162–180.

[38] 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.

[39] 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 soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[40] 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.

[41] 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.

[42] oldmanpushcart. 2019. greys-anatomy. https://github.com/oldmanpushcart/greys-anatomy.

[43] 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.

[44] 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.

[45] 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.

[46] Michael Pradel, Ciera Jaspan, 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.

[47] 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.

[48] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.