

CHENNAI MATHEMATICAL INSTITUTE



Call Graph Construction for Spring Framework

Author:
Mugdha Khedkar

Supervisor:
Prof. Dr. Eric Bodden
Dr. Johannes Späth

*A thesis submitted in fulfillment of the requirements
for the degree*

*Master of Science
in*

Computer Science

May 8, 2020

Declaration of Authorship

I, Mugdha Khedkar, declare that this thesis titled, "Call Graph Construction for Spring Framework" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Name: Mugdha Khedkar

Date: 8th May 2020

CHENNAI MATHEMATICAL INSTITUTE

Abstract

Master of Science

Call Graph Construction for Spring Framework

by Mugdha Khedkar

Call graphs[28] maintain the record of the calls between methods in a given program. Such information is necessary for compilers to determine whether specific optimizations can be applied. Additionally, numerous software engineering tools use call graph information to help software engineers increase their understanding of a program[8].

Constructing precise call graphs is crucial for any practical static analysis that needs to handle a program consisting of multiple procedures and procedure calls. Call graph construction in the presence of direct calls is straightforward. However, indirect calls (supported by features such as function pointers, virtual functions, and reflection etc.) hide the caller callee relationships statically. This complicates the construction of call graphs resulting in either imprecision (spurious caller-callee relationships) or unsoundness (missed caller-callee relationships). This, in turn, affects the precision or soundness of a client analysis that uses such a call graph. The objective of this thesis is to explore the issues in call graph construction for web frameworks like Spring and to study how the existing algorithms fare on the task.

Programs written in Spring framework have two unique features: they have an abundant use of annotations (in some cases, annotations decide the entry points of the program) and a separate framework for Inversion of Control[29]. These two features lead to major unsoundness in existing call graph construction algorithms. The client analyses that use these call graphs end up missing critical findings.

One contribution of this thesis is the observation that the unsoundness is because of reflective calls in the Spring API. Static analysis of the Reflection API has attracted significant research effort since the past few decades, some such methods have been discussed in this thesis.

Another contribution of this thesis is that it presents a concept for a new hybrid analysis algorithm. This algorithm generates a simple, non-reflective version of the input Spring program written in Java. Such an algorithm would work on programs written in web frameworks designed on top of the Spring framework and the call graph constructed for these programs would be sound and precise, as required.

Acknowledgements

It gives me great pleasure in presenting the project report on

'CALL GRAPH CONSTRUCTION FOR SPRING FRAMEWORK'

I take this opportunity to express my profound gratitude to my supervisors *Prof. Dr. Eric Bodden* and *Dr. Johannes Späth* for their exemplary guidance, monitoring and constant encouragement throughout the course of this thesis work. Everything that I learnt from them will help me a long way in the journey of life and the next adventure I embark upon.

This research project would not have been possible without the support of the *Software Engineering group of Universität Paderborn*. I am very grateful for their valuable suggestions. They became my first friends in Germany and this help is immeasurable.

My special thanks to all the staff members of Chennai Mathematical Institute for their continuous support. A special mention to my faculty advisor, *Prof. B Srivathsan* who has been a constant help throughout my time at CMI.

No words are enough to express the gratitude I have towards my parents and my sister. Staying thousands of miles away from them was not easy but their constant support made this journey worthwhile. A special thanks to my friends as well.

You all made this journey a great experience, and for this I cannot thank you all enough.

Mugdha Khedkar
Chennai Mathematical Institute

To Aai, Baba and Sneha who lived in two timezones for three months...

Contents

Declaration of Authorship	i
Abstract	ii
Acknowledgements	iii
1 Introduction	1
2 Background	4
2.1 Call Graph	4
2.2 Call Graph Construction Algorithms	4
2.2.1 Class Hierarchy Analysis	4
2.2.2 Rapid Type Analysis	5
2.2.3 Variable Type Analysis	6
2.2.4 SPARK	7
2.3 Spring Framework	10
2.3.1 Spring Architecture	10
2.3.2 Inversion of Control	12
2.3.3 Spring Annotations	13
2.3.4 An Example	15
2.4 Reflection	17
2.4.1 Object Creation	17
2.4.2 Method Invocation	17
2.4.3 Accessing Fields	18
2.4.4 Applications of Reflection	18
2.4.5 Reflection in Static Analysis	19
2.5 Summary	19
3 Illustrating Unsoundness in Call Graph Construction	20
3.1 A Simple Spring Application	20
3.2 Spring Application with Core Annotations	23
3.3 Summary	25
4 Our Solution	26
4.1 The Key Idea	26
4.2 Bean Dependence Graph	29
4.3 Handling Different Containers and Scopes	29
4.3.1 Spring IoC containers	29
4.3.2 Bean Scopes	30
4.4 Algorithm	32
4.5 An Example	34
4.6 Limitations	35
4.7 Summary	35

5	Related Work	36
5.1	General Call Graph Construction	36
5.1.1	Precise Analysis of String Expressions	36
5.1.2	Call Graph Discovery using Points-to Analysis	36
5.1.3	Averroes	37
5.1.4	Call Graph Construction for Java Libraries	38
5.2	Reflection and Framework Handling for Call Graph Construction . . .	39
5.2.1	DOOP Framework	39
5.2.2	Tamiflex	39
5.2.3	Framework for Frameworks	40
5.2.4	Self-Inferencing Reflection Resolution for Java	40
5.2.5	Sound Static Handling of Java Reflection	41
5.3	Summary	42
6	Conclusion	43
	Bibliography	48

1 Introduction

Call graph is a static abstraction of all the methods that can be called by a program at any program point. Call graphs[28] maintain the record of the calls between methods in a given program. Such information is necessary for compilers to determine whether specific optimizations can be applied. For example, a compiler can use a call graph to eliminate unreachable code (e.g. debugging code and unused parts of libraries) or to propagate constants across a program. Additionally, numerous software engineering tools use call graph information to help software engineers increase their understanding of a program[8]. For example, an integrated development environment (IDE) like Eclipse uses call graph information to provide features like code navigation and completion, automated code refactoring etc. Call graphs can also be used to detect anomalies or code injection attacks[9].

Constructing precise call graphs is crucial for any static analysis that needs to handle a program consisting of multiple procedures and procedure calls. Dealing with direct calls is straightforward. However, the presence of indirect calls (supported by features such as function pointers, virtual functions, and reflection etc.) complicates the construction of call graphs because they hide the caller-callee relationships statically. This results in either imprecision (spurious caller-callee relationships) or unsoundness (missed caller-callee relationships)[16]. This, in turn, affects the precision or soundness of a client analysis that uses such a call graph.

Indirect calls are also the main source of imprecision in a call graph of a Spring application. The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java Enterprise Edition platform. Spring's layered architecture organizes the middle tier objects. Separating all the layers eliminates the need to use a variety of custom-property file formats. The Spring layered architecture is discussed in detail in Section 2.7. Spring Framework is non-invasive and uses POJO and POJI models:

- POJO (Plain Old Java Objects): A Java class not coupled with any technology or any framework.
- POJI (Plain Old Java Interfaces): A Java interface not coupled with any technology or any framework.

For running the Spring application, a server is not mandatory. Spring has its own container for running any application. It does not need a server and thus prevents the hassles of saving the program again and again when restarting a server. Spring Framework is loosely coupled because it has concepts like Dependency Injection (Section 2.3.2), Inversion of Control etc. These features help in reducing dependency and increasing the modularity within the code. All these features make Spring extremely popular.

A Spring application consists of an XML file which defines beans (Spring objects) and Java files that define the classes and the main method. Listings 1.1, 1.2, 1.3, 1.4 illustrate a sample Spring application. We discuss more about Spring Framework in Chapter 2.


```
1 package Education;
2
3 public class Student
4 {
5     private Integer age;
6     private String name;
7
8     public void setAge(Integer age) {
9         this.age = age;
10    }
11    public Integer getAge() {
12        return age;
13    }
14    public void setName(String name) {
15        this.name = name;
16    }
17    public String getName() {
18        return name;
19    }
20 }
```

LISTING 1.1: A Spring Application (1): Student.java

```
1 package Education;
2
3 public class Profile
4 {
5     private Student student;
6
7     public void printAge() {
8         System.out.println("Age : " + student.getAge() );
9     }
10    public void printName() {
11        System.out.println("Name : " + student.getName() );
12    }
13 }
```

LISTING 1.2: A Spring Application (2): Profile.java

```
1 public class MainApp
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("Bean.xml");
6         Profile profile = (Profile) context.getBean("profile");
7         profile.printName();
8         profile.printAge();
9     }
10 }
```

LISTING 1.3: A Spring Application (3): MainApp.java

```
1 <?xml version = "1.0" encoding = "UTF-8" ?>
2 <beans xmlns = "http://www.springframework.org/schema/beans"
3     xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context = "http://www.springframework.org/schema/context"
5     xsi:schemaLocation = "http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9 <context:component-scan base-package="Education"/>
10 <context:annotation-config/>
11 <!-- Definition for profile bean -->
12 <bean id = "profile" class = "Education.Profile"/>
13 <!-- Definition for student1 bean -->
14 <bean id = "student1" class = "Education.Student">
15     <property name = "name" value = "Zara" />
16     <property name = "age" value = "11"/>
17 </bean>
18 <!-- Definition for student2 bean -->
19 <bean id = "student2" class = "Education.Student">
20     <property name = "name" value = "Neha" />
21     <property name = "age" value = "2"/>
22 </bean>
23 </beans>
```

LISTING 1.4: A Spring Application (4): Beans.xml

Spring applications are difficult to analyze statically due to programming features like reflection (Section 2.4) and annotations (Section 2.3.3). The motivation behind this Master's thesis is finding out which features in Spring hinder sound and precise call graph construction and what can be done to ensure soundness in call graph construction algorithms and precision in the resultant call graphs.

There are various web frameworks built on top of the Spring framework. By studying the soundness in call graph construction algorithms and precision in the resultant call graphs, we can extend this study further to other web frameworks.

The main contributions of this thesis are summarized below:

1. A detailed study of call graph construction algorithms and concrete reasons why these algorithms are unsound and unable to construct precise call graphs for web frameworks like Spring.
2. A concept for a hybrid analysis (a combination of static and dynamic analysis) algorithm to ensure soundness in call graph construction algorithms and construction of precise call graphs for Spring.

This thesis is structured as follows. Chapter 2 discusses the technical concepts needed to understand the process of call graph construction for Spring framework. These include call graphs (Section 2.1), call graph construction algorithms (Section 2.2), Spring framework (Section 2.3) and Reflection (Section 2.4). Chapter 3 uses two examples to illustrate unsoundness in call graphs. These examples are illustrated in Section 3.1 and Section 3.2. Chapter 4 discusses a proposed hybrid analysis concept to deal with unsoundness in call graph construction algorithms. Chapter 5 shows some work done in the past to deal with reflection in static analyses. Finally, Chapter 6 concludes this work and discusses the possible directions this work could take in the future.

2 Background

This chapter provides the technical concepts needed to better understand call graph construction for Spring framework. We discuss call graphs (Section 2.1), call graph construction algorithms (Section 2.2), Spring framework (Section 2.3) and Reflection (Section 2.4).

2.1 Call Graph

A call graph is a static abstraction of the caller-callee relation between the methods in a program. A call graph is a graph that consists of nodes (representing procedures) linked by directed edges (representing calls from one procedure to another). An interprocedural analysis requires an approximation of the call graph. A call graph construction algorithm is said to be sound if the resulting call graph does not miss any caller-callee relationship. A call graph is precise if it does not include caller-callee relationships which are spurious with respect to any given potential execution.

Call graphs can be constructed in two ways: either by examining the source code or by observing the caller-callee relationships during execution (using profiling). In C, examining the source code to construct a call graph is difficult because of function pointers. In C++, we additionally have virtual functions. In Java and web frameworks like Spring, there is use of reflection and all calls are virtual by default. Handling programming features like reflection, annotations have become crucial to getting a sound and precise call graph[28].

2.2 Call Graph Construction Algorithms

We will use the example in Listing 2.1 throughout this Section to illustrate the differences between some well known call graph construction algorithms: CHA (Section 2.2.1), RTA (Section 2.2.2), VTA (Section 2.2.3) and SPARK (Section 2.2.4).

The example consists of class *Shape* and its subclasses *Quadrilateral* and *Circle*. *Square* is further inherited by *Quadrilateral*. All these classes have a method *print()*. We want to see which print method is called when we call *b2.print(c2)* (Line 9 in Listing 2.1).

2.2.1 Class Hierarchy Analysis

Class Hierarchy Analysis[6] is a standard method for conservatively approximating the run-time types of receivers. To implement this analysis, the compiler gains static information from the class hierarchy, which is a tree constructed based on a class subclass relation. The class hierarchy is studied to approximate the types that could have called a method of that class.

The class hierarchy of our running example is shown in Figure 2.1. The function *print()* is a part of classes *Shape*, *Quadrilateral*, *Square* and *Circle*. The algorithm looks at the class hierarchy and calls *print()* for all classes belonging to the subtree with

```
1 public class Example
2 {
3     public static void main(String[] args)
4     {
5         Shape b1 = new Quadrilateral();
6         Shape c1 = new Square();
7         Shape b2 = b1;
8         Shape c2 = c1;
9         b2.print(c2);
10    }
11    public static class Shape extends Object
12    {
13        public void print(Shape object) { ....}
14    }
15    public static class Quadrilateral extends Shape
16    {
17        public void print(Shape object) { ....}
18    }
19    public static class Square extends Quadrilateral
20    {
21        public void print(Shape object) { ....}
22    }
23    public static class Circle extends Shape
24    {
25        public void print(Shape object) { ....}
26    }
27 }
```

LISTING 2.1: Call Graph Construction Example

Shape as the root (these include *Shape*, *Quadrilateral*, *Square* and *Circle*) ignoring the classes are instantiated or which types are allocated in the program. The call graph generated by CHA is shown in Figure 2.2. As evident from the figure, this graph is an overapproximation. The print function of class *Circle* would never be a part of any execution of the program because there is no object belonging to class *Circle* anywhere in the program.

CHA is a very simple flow-insensitive algorithm. A flow-insensitive algorithm does not take into consideration the order of program statements. It is very efficient because subclass relations are easy to compute. The resulting call graph contains edges for all calls that the program could execute. There is very little scope for unsoundness. It is most effective in situations where the compiler has access to the source code of the entire program, since the whole inheritance hierarchy can be examined and the locations of all method definitions can be determined.

The disadvantage of the resulting graph is that it is very imprecise. The calls by most edges will never be made.

2.2.2 Rapid Type Analysis

Rapid Type Analysis[6] starts with a call graph generated by performing Class Hierarchy Analysis. It uses information about instantiated classes to further reduce these to exclude those that have not been instantiated, thereby reducing the size of the call graph. RTA is a refining algorithm. This version of RTA is called pessimistic RTA since it starts with the complete conservative call graph built by CHA and looks for all instantiations in method in that call graph. This may find an instantiation which is in a method that should really be removed from the call graph[25].

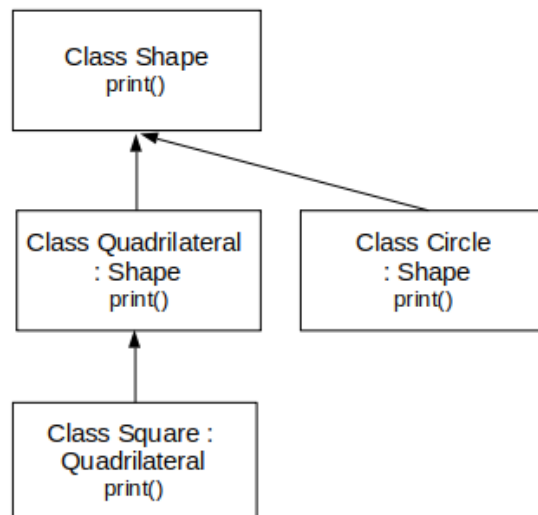


FIGURE 2.1: Class Hierarchy

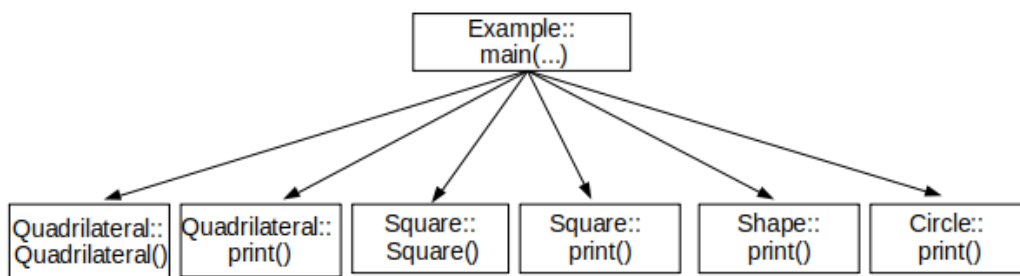


FIGURE 2.2: CHA Call Graph

For the given example, RTA refines the CHA call graph (Figure 2.2) by removing the classes not instantiated (for this example, class *Circle*). The refined call graph is illustrated in Figure 2.3.

RTA inherits the limitations and benefits of CHA: it must analyze the complete program. Like CHA, RTA is flow-insensitive and does not keep per-statement information, making it very fast.

2.2.3 Variable Type Analysis

The idea of Variable Type Analysis[6] is to start at every allocation site and record its type. Then the algorithm looks at every assignment and propagates its effects by building a graph.

For the given example, VTA looks at the allocation sites and only includes types associated with them in the final call graph. The allocation sites are of type *Quadrilateral* and *Square*. Hence they are the only classes recorded in the final call graph. The final call graph is illustrated in Figure 2.4.

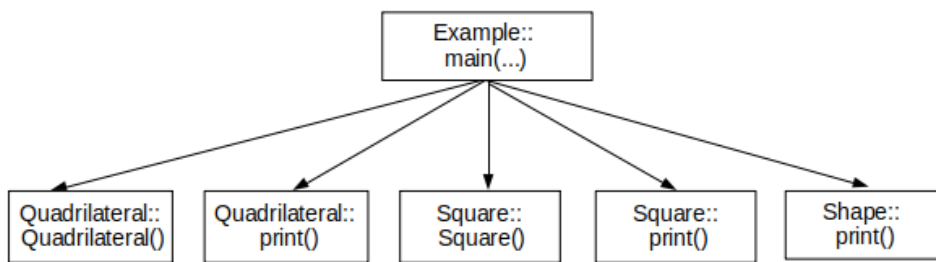


FIGURE 2.3: RTA Call Graph

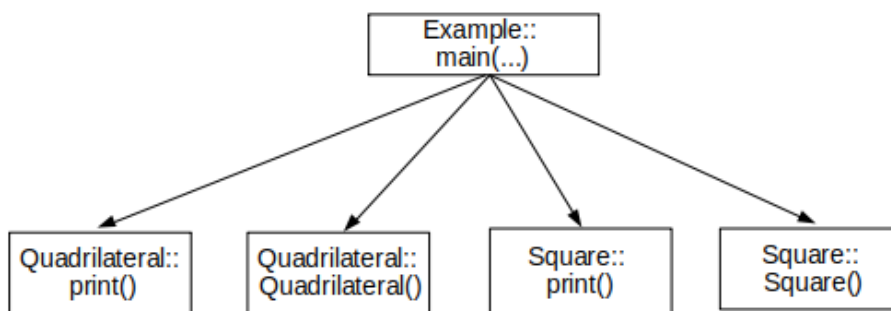


FIGURE 2.4: VTA Call Graph

The algorithm looks remarkably similar to RTA in that it is a fixed point algorithm using a worklist. The difference between VTA and RTA is that the latter ignores type allocation sites. Since we are considering much more information than RTA (RTA ignores all assignments and VTA iterates over them), the cost of running VTA is much higher. In most cases the precision of VTA is more than RTA.

2.2.4 SPARK

The Soot Pointer Analysis Research Kit (SPARK)[11] is a flexible framework for experimenting with points-to analyses for Java built on top of Soot.

Soot[26] is a Java optimization framework. It provides some intermediate representations for analyzing and transforming Java bytecode. One such intermediate representation is Jimple, a typed three-address intermediate representation suitable for optimization. Soot accepts as input the Java byte code and transforms it into Jimple. It then performs the suitable analysis and optionally converts the code back to Java byte code. SPARK is built on top of Soot. It supports both subset-based (Anderesen[3]) and equality-based (Stensgaard[24]) flow-insensitive pointer analyses. The results of SPARK can be used by other analyses and transformations in Soot.

The execution of SPARK can be divided into three stages:

1. Pointer assignment graph construction.
2. Pointer assignment graph simplification.
3. Points-to set propagation.

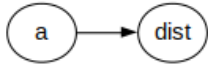
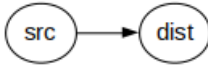
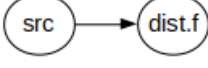
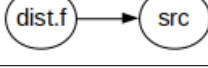
Statement	Edge	Example
Allocation Edge: $a: \text{dist} = \text{new } C()$		$X \ x = \text{new } X()$
Assignment Edge: $\text{dist} = \text{src}$		$x = y$
Store Edge: $\text{dist}.f = \text{src}$		$x.f = y$
Load Edge: $\text{dist} = \text{src}.f$		$x = y.f$

TABLE 2.1: Pointer Assignment Graph Construction

SPARK uses a pointer assignment graph as its internal representation of the program being analyzed. The first stage of SPARK is the pointer assignment graph builder which determines how features of the program, such as field references, array element references and parameters passed to methods are represented.

The nodes in the pointer assignment graph are connected with four types of edges reflecting the pointer flow, corresponding to the four types of constraints imposed by the pointer-related instructions in the source program (illustrated in Table 2.1):

- Allocation edge: It is an edge from an allocation node to a variable node. It represents an assignment of pointers to the objects represented by the allocation node to the location represented by the variable node. In the Table 2.1, a is the allocation node and dist is the variable node.
- Assignment edge: It is an edge from a variable node to another variable node. It represents an assignment from the location represented by the first variable node to the location represented by the second variable node. In the Table 2.1, src and dist are the variable nodes.
- Store edge: It is an edge from a variable node to a field reference node. It represents a store from the location represented by the variable node to the appropriate field of some object pointed to by the base of the field reference node. In the Table 2.1, src is the variable node and $\text{dist}.f$ is the field reference node.
- Load edge: It is an edge from a field reference node to a variable node. It represents a load from the appropriate field of some object pointed to by the base of the field reference node to the location represented by the variable node. In the Table 2.1, dist is the variable node and $\text{src}.f$ is the field reference node.

```

1 public class Example
2 {
3   public static void main(String[]
4     args)
5   {
6     Shape b1 = new Quadrilateral();
7     Shape c1 = new Square();
8     Shape b2 = b1;
9     Shape c2 = c1;
10    b2.print(c2);
11  }
12  ...

```

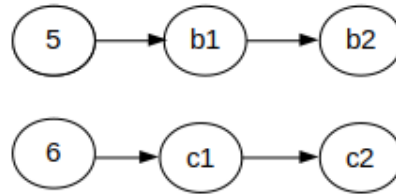
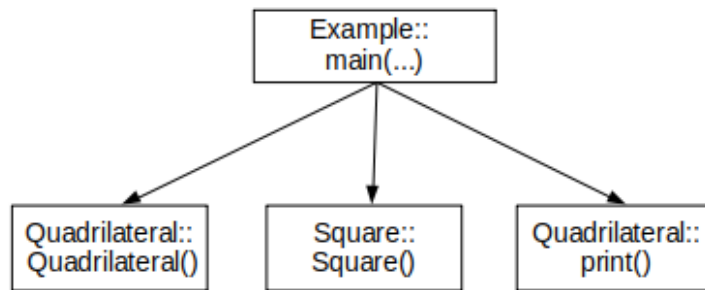
LISTING 2.2: SPARK:
Input ProgramFIGURE 2.5: Pointer
Assignment Graph

FIGURE 2.6: SPARK Call Graph

The pointer assignment graph may then be simplified by merging nodes that are known to have the same points-to sets. This simplification reduces the amount of processing required to compute the points-to sets. Finally, the points-to set propagator computes the points-to set for each variable by propagating sets along assignments in the program (which are represented by edges in the pointer assignment graph).

For a call `v.foo()`, SPARK can overestimate the set of receivers through the set `points-to(v)` that it computes: if `v` can point to an object of type `Foo`, then `Foo.foo()` is a possible receiver of the call `v.foo()`. To every call target determined that way, SPARK inserts an edge into the call graph [4].

For our running example illustrated in Listing 2.2, SPARK constructs a pointer assignment graph depicted in Figure 2.5. Allocation nodes (5 and 6) are the line numbers where the variables `b1` and `c1` have been instantiated. From this pointer assignment graph, SPARK finds the type of `b2`. Since `b1` points to `b2`, SPARK concludes that the type of `b2` is same as that of `b1` (`Quadrilateral`). Thus SPARK only calls the print method of class `Quadrilateral`. The final call graph is illustrated in Figure 2.6.

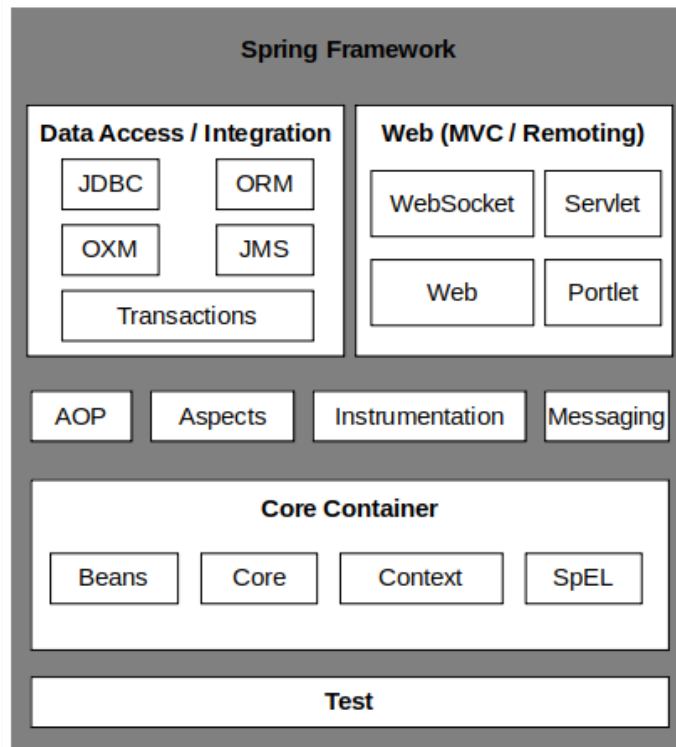


FIGURE 2.7: Spring Architecture

2.3 Spring Framework

The Spring Framework[21][30] provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform. It is commonly used for web applications. It is an application framework and inversion of control container for the Java platform. Section 2.3.4 provides a complete example of a Spring application.

2.3.1 Spring Architecture

Spring has a modular architecture as shown in Figure 2.7[20]. Some of the major components of the architecture are:

- **Spring Core Container:** The Spring container is at the core of the Spring Framework. It uses dependency injection to manage the components that make up an application. The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations or Java code. The container will then create the objects, wire them together, configure them and manage their complete life cycle from creation till destruction[22].
- **Spring Beans:** The objects that form the backbone of any Spring application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled and managed by a Spring IoC container. These beans are created with the configuration metadata that is supplied to the container. The metadata can be a bean configuration file or it can

```

1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <beans xmlns = "http://www.springframework.org/schema/beans"
3     xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context = "http://www.springframework.org/schema/context"
5     xsi:schemaLocation = "http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9 <context:component-scan base-package="Education"/>
10 <context:annotation-config/>
11 <!-- Definition for profile bean -->
12 <bean id = "profile" class = "Education.Profile"/>
13 <!-- Definition for student1 bean -->
14 <bean id = "student1" class = "Education.Student">
15     <property name = "name" value = "Zara" />
16     <property name = "age" value = "11"/>
17 </bean>
18 <!-- Definition for student2 bean -->
19 <bean id = "student2" class = "Education.Student">
20     <property name = "name" value = "Neha" />
21     <property name = "age" value = "2"/>
22 </bean>
23 </beans>

```

LISTING 2.3: Bean Configuration File: Beans.xml

be using Spring annotations (Section 2.3.3)[22]. *Beans.xml* (Listing 2.3) defines three beans: *profile*, *student1* and *student2*. All three of them have classes associated with them and (may) have properties associated with them.

- **Context:** Context module builds on Core and Bean modules and is a medium to access the objects that have been instantiated by the core.
- **SpEL:** SpEL module provides a powerful expression language to query and manipulate object graph at runtime. It was created so Spring could have a single well supported expression language that can be used across all the products in the Spring portfolio.
- **Data Access/Integration :** This layer consists of the JDBC, ORM, OXM, JMS and Transaction modules which are essentially all modules for JDBC related coding or XML mapping or transaction management.
- **Web:** The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules which are used to handle Spring's Model-View-Controller implementation.
- **Other modules:** A few other important modules are as follows:
 - AOP: This module provides an aspect-oriented programming implementation that helps increase modularity by allowing separation into distinct parts called concerns.
 - Aspects: It provides integration with AspectJ, a powerful and mature AOP framework.
 - Instrumentation: It provides class instrumentation support and class loader implementations to be used in certain application servers.

- Messaging: It provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- Test: It supports the testing of Spring components with JUnit or TestNG frameworks.

2.3.2 Inversion of Control

Inversion of control (IoC)[29] is a programming principle that inverts the flow of control as compared to traditional control flow. In IoC, custom-written portions of a computer program receive the flow of control from a generic framework[29].

A software architecture with this design inverts control as compared to traditional procedural programming. In IoC, the framework takes care of custom code. Inversion of control is used to increase modularity of the program and make it extensible.

Dependency Injection is a way of inverting control in Spring. Every Java-based application references object that interact with each other to present what the end-user sees as a working application. When writing a complex Java application, the application classes should be as independent as possible of other Java classes, to test them independently of other classes while unit testing and also to increase the possibility to reuse them. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent[22].

Consider an application which has a text editor component and we want to provide a word counter. A typical code would look something like this:

```
1 public class TextEditor
2 {
3     private WordCount wordCount;
4
5     public TextEditor()
6     {
7         wordCount = new WordCount();
8     }
9 }
```

In the code snippet above, we have created a dependency between *TextEditor* and *WordCount*. In an inversion of control scenario, we would instead do something like this:

```
1 public class TextEditor
2 {
3     private WordCount wordCount;
4
5     public TextEditor(WordCount wordCount)
6     {
7         this.wordCount = wordCount;
8     }
9 }
```

Here, the *TextEditor* should not worry about *WordCount* implementation. *WordCount* will be implemented independently and will be provided to *TextEditor* at the time of *TextEditor* instantiation. This entire procedure is controlled by the Spring

Framework. Here, we have removed object creation from the *TextEditor* and the dependency (i.e. class *WordCount*) is being injected into the class *TextEditor* through a class constructor[22].

The second method of injecting dependency is through Setter Methods of the *TextEditor* class where we will create a *WordCount* instance. This instance will be used to call setter methods to initialize *TextEditor's* properties.

Thus, dependency injection exists in two major variants:

1. Constructor-based DI: It is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.
2. Setter-based DI: It is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

2.3.3 Spring Annotations

An XML file is used to describe the bean configuration in a Spring application. Instead of using XML to describe a bean wiring, the bean configuration can be moved into the component class itself by using annotations on the relevant class, method, or field declaration. Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches. Once `<context:annotation-config/>` is configured, the code can be annotated to indicate that Spring should automatically wire values into properties, methods, and constructors[18]. Some important annotations are as follows:

- **@Configuration**: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- **@Bean**: Marks a factory method which instantiates a Spring bean. The return value of this method is instantiated as a Spring bean. Consider the Java file below:

```
1 @Configuration
2 public class A
3 {
4     @Bean
5     public B b()
6     {
7         return new B();
8     }
9 }
```

The same information can be represented as an XML file:

```
1 <beans>
2   <bean id = "b" class = "package.B" />
3 </beans>
```

- **@Autowired**: Marks a dependency which Spring is going to resolve and inject. It can be used with a constructor, setter or field injection.

```
1 @Configuration
2 public class Profile
3 {
4     @Autowired
5     @Qualifier("student1")
6     private Student student;
7
8     ...
9 }
```

LISTING 2.4: Example of @Autowired: Profile.java

- **@Qualifier**: Used along with **@Autowired** to provide the bean id or bean name to be used in ambiguous situations. It can be used with a constructor, setter, or field injection. Consider the file *Profile.java* (Listing 2.4). We have used the **@Autowired** annotation to wire the *student* bean into the *Profile* class. In *Beans.xml* (Listing 2.3), since there are two beans *student1* and *student2* of the type *Student*, we need to use the **@Qualifier** annotation. The absence of **@Qualifier** throws an ambiguous bean exception.
- **@ComponentScan**: Specifies which packages contain classes that are annotated.
- **@Lazy**: By default, beans and components get initialized eagerly. This behavior can be changed using this annotation. Can be used with **@Bean** or **@Component**. If annotated, the component/bean will not be initialized until another bean explicitly references it and it is needed for the application to run smoothly.
- **@Value**: Can be used for assigning default values to fields, reading environment variables and setting default values for parameters if used within a method or constructor.
- **@DependsOn**: If a bean depends on some other beans for correct instantiation, Spring can guarantee that all the beans it depends on will be created before it. **@DependsOn** annotation specifies the dependence relation.
- **@Primary**: The **@Primary** annotation is often used alongside the **@Qualifier** annotation. Used to define the "default" bean for autowiring when no further information is available.
- **@Scope**: When defining a `<bean>`, there is an option to declare a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, the bean's scope attribute should be defined as `prototype`. The Spring Framework supports the following scopes:
 1. *singleton*: Scopes the bean definition to a single instance per Spring IoC container (default). The default scope is always `singleton`. If one and only one instance of a bean is needed, the scope property can be set to `singleton` in the bean configuration file.
 2. *prototype*: Scopes a single bean definition to have any number of object instances. The Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made.
 3. *request*: Warrants the instantiation of a single bean for each HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

```
1 package Education;
2
3 public class Student
4 {
5     private Integer age;
6     private String name;
7
8     public void setAge(Integer age) {
9         this.age = age;
10    }
11    public Integer getAge() {
12        return age;
13    }
14    public void setName(String name) {
15        this.name = name;
16    }
17    public String getName() {
18        return name;
19    }
20 }
```

LISTING 2.5: A Simple Spring Example (1): Student.java

```
1 package Education;
2
3 public class Profile
4 {
5     private Student student;
6
7     public void printAge() {
8         System.out.println("Age : " + student.getAge() );
9     }
10    public void printName() {
11        System.out.println("Name : " + student.getName() );
12    }
13 }
```

LISTING 2.6: A Simple Spring Example (2): Profile.java

4. *session*: Instantiates the annotated bean with a lifecycle-dependent of the HTTP session.
5. *application*: Works similarly to the singleton scope. An application scoped bean's life cycle depends on the application, or rather, the ServletContext.
6. *websocket*: Ties the bean's life cycle to the life cycle of the WebSocket's session[19].

2.3.4 An Example

Let us consider an example application written in Spring. It consists of the following files:

- **Java files:** The file *Student.java* (Listing 2.5) defines the class *Student* with its getter and setter methods. *Profile.java* (Listing 2.6) defines the class *Profile* and injects the dependency to a *Student* object. The main class *MainApp* (Listing 2.7) creates an IoC container for the bean configuration file *Beans.xml*

```
1 public class MainApp
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("Bean.xml");
6         Profile profile = (Profile) context.getBean("profile");
7         profile.printName();
8         profile.printAge();
9     }
10 }
```

LISTING 2.7: A Simple Spring Example (3): MainApp.java

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <beans xmlns = "http://www.springframework.org/schema/beans"
3     xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context = "http://www.springframework.org/schema/context"
5     xsi:schemaLocation = "http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9 <context:component-scan base-package="Education" />
10 <context:annotation-config/>
11 <!-- Definition for profile bean -->
12 <bean id = "profile" class = "Education.Profile" />
13 <!-- Definition for student1 bean -->
14 <bean id = "student1" class = "Education.Student">
15     <property name = "name" value = "Zara" />
16     <property name = "age" value = "11" />
17 </bean>
18 <!-- Definition for student2 bean -->
19 <bean id = "student2" class = "Education.Student">
20     <property name = "name" value = "Neha" />
21     <property name = "age" value = "2" />
22 </bean>
23 </beans>
```

LISTING 2.8: A Simple Spring Example (4): Beans.xml

(Listing 2.8), retrieves beans from the XML file and calls functions on the bean objects.

- **Bean configuration file:** An XML file which acts as a cement that glues the beans, i.e. the classes together. *Beans.xml* (Listing 2.8) defines the beans *profile*, *student1* and *student2*.

2.4 Reflection

Reflection in Java allows the developer to perform runtime actions given the descriptions of the objects involved. Reflection is useful to change the behaviour of an object at runtime. Reflective APIs can be used for object creation, method invocation and field access as discussed in Sections 2.4.1, 2.4.2 and 2.4.3. There are various other reflective functions but the ones relevant to this thesis have been covered in this Section.

2.4.1 Object Creation

Reflection APIs in Java provide a way to programmatically create objects of a class, whose name is provided at runtime. Obtaining a class given its name is most typically done using a call to one of the static functions *Class.forName(String, ...)* and passing the class name as the first parameter. Creating an object with an empty constructor is achieved through a call to *newInstance(args)* on the appropriate *java.lang.Class* object, which provides a runtime representation of a class. Both these methods are described below:

```
1 class Example
2 {
3     void message() {System.out.println("Hello Java");}
4 }
5 class Test
6 {
7     public static void main(String args[])
8     {
9         try {
10             Class c = Class.forName("Example");
11             Example new_example = (Example) new_example.newInstance();
12             e.message();
13         } catch (Exception e) {System.out.println(e);}
14     }
15 }
16 }
```

2.4.2 Method Invocation

Methods are obtained from a Class object by supplying the method signature or by iterating through the array of methods returned by one of Class functions. Methods are subsequently invoked by calling *method.invoke(...)* as follows:

```
1 public class Demo
2 {
3     public static void main(String[] args)
4     throws IllegalAccessException, IllegalArgumentException,
5         InvocationTargetException
```



```
5 {
6     Method[] methods = Student.class.getMethods();
7     Student sampleObject = new Student();
8     methods[1].invoke(sampleObject, "data");
9     System.out.println(methods[0].invoke(sampleObject));
10 }
11 }
12
13 class Student
14 {
15     private String name;
16
17     public String getName() {return name;}
18
19     public void setName(String name) {this.name = name;}
20 }
```

2.4.3 Accessing Fields

Fields of Java runtime objects can be read and written at runtime. Calls to *Field.get(...)* and *Field.set(...)* can be used to get and set fields containing objects[14] as follows:

```
1 public class Test
2 {
3     public static void main(String[] args)
4         throws Exception
5     {
6         Employee emp = new Employee();
7         Field field = Employee.class.getField("uniqueNo");
8         field.set(emp, (short)1213);
9     }
10 }
11
12 // Sample class
13 class Employee
14 {
15     // Static values
16     public static short uniqueNo = 239;
17     public static double salary = 121324.13333;
18 }
```

2.4.4 Applications of Reflection

Reflection is widely used at the backend of many softwares. Some real applications of Reflection are discussed below:

- Code analyzer tools: Code analyzers tools perform a static analysis of syntax, show optimization tips and even report error conditions. They are written in a way such that they can analyze any class file passed to them to analyze. They use reflection to perform this analysis.
- Eclipse (Other IDEs): One example of usage of reflection is Eclipse or any other IDE. Eclipse is able to provide us method suggestions whenever we hit CTRL+SPACE, even before we can finish writing that class. This is achieved through Reflection.

- **Marshalling and unmarshalling:** JAXB/Jattison and other marshalling/unmarshaling libraries heavily use reflection for XML (or JSON) to/from java beans code. They look up all annotated attributes in java bean, analyze their overall attributes and generate XML tags for them. The same is valid for unmarshaling as well.
- **Junit Testcases:** In previous versions of Junit, to run a testcase we had to name a method starting with test e.g. *testMethod1()*, *testCode2()* etc. Junit processor used reflection to iterate over all methods in class in order to find out methods starting with test and run this as testcase.

2.4.5 Reflection in Static Analysis

Obtaining a "whole program" yields many challenges when analyzing Java programs that use reflection, or load classes using custom class loaders. Industrial Java applications frequently use custom class loaders or generate classes on the fly. A static analysis may have no access to these class loaders. The same programs also frequently use reflection to invoke methods or instantiate objects of types that programmers cannot fully determine at compile time. To construct a complete call graph, a static analysis needs to be aware of these calls. Even if a static analysis is aware of reflective calls and has access to all classes that are loaded at runtime, researchers need to modify the analysis to handle the reflective calls and all the program's classes correctly[4].

When a Java program accesses a class by supplying a class name as a parameter in *Class.forName(...)* library call, the static analysis needs to either conservatively over-approximate (e.g., assume that any class can be accessed), or to perform a string analysis that will allow it to infer the contents of the *forName* string argument. The conservative over-approximation may never become constrained enough by further instructions to be feasible in practice. On the other hand, precise string analysis is impractical when it comes to programs of realistic size, thereby reducing scalability to a large extent[17].

Due to this reason, many projects that use static analysis for optimization, error detection and other purposes ignore the use of reflection. This makes the information computed by static analysis tools incomplete[15] because some parts of the program may not be included in the call graph and potentially unsound, because some operations, such as reflectively invoking a method or setting an object field, are ignored[14]. One such example is SPARK. Tools like TamiFlex were designed with the purpose of dealing with reflection in applications.

2.5 Summary

Call graphs are abstractions extremely crucial to any interprocedural analysis. In this chapter, we discussed some call graph construction algorithms and then introduced Spring framework. Towards the end of this chapter, we elaborated on the concept of reflection which would help us in understanding the forthcoming chapters.

3 Illustrating Unsoundness in Call Graph Construction

This chapter illustrates unsoundness in call graph construction algorithms for Spring programs using two examples. A call graph construction algorithm is said to be sound if the resultant call graph does not miss any caller-callee relationship for the application code.

3.1 A Simple Spring Application

We consider a simple Spring application illustrated in sub section 2.3.4 (Listings 2.8, 2.5 and 2.6). It is simple because it takes input beans *profile*, *student1* and *student2* from the Bean configuration file and does not make use of any new features like annotations. We construct call graphs for this application using CHA (Section 2.2.1), RTA (Section 2.2.2), VTA (Section 2.2.3) and SPARK (Section 2.2.4). The total number of edges in the call graph is 84,522 in algorithms like CHA, RTA and VTA. Some of the edges from the constructed call graph relevant to the input (edges corresponding to the source code and not library) are shown in Figure 3.1.

The call graph constructed by CHA, RTA and VTA is not an underapproximation in that it doesn't miss any caller-callee relationship for the application code. On the other hand, it is found to be an overapproximation¹ since it includes spurious caller-callee relationships.

¹Overapproximation not depicted in Figure 3.1.

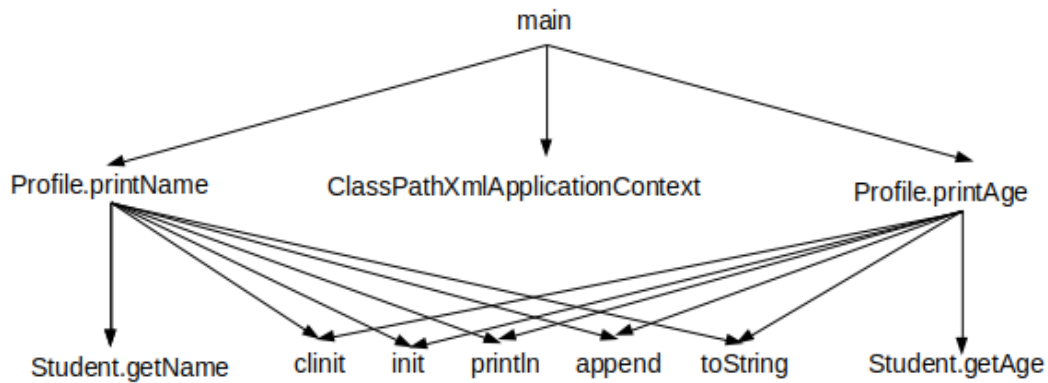


FIGURE 3.1: CHA/RTA/VTA Call Graph

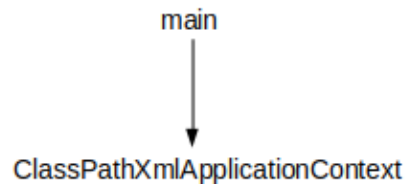


FIGURE 3.2: SPARK Call Graph

Usage of SPARK for constructing call graph for the example leads to a significantly smaller call graph (Total number of edges in the call graph: 40,646). Some of the edges from the constructed call graph relevant to the input (edges corresponding to the source code and not library) are shown in Figure 3.2.

The call graph constructed by SPARK is an underapproximation for the application code because it misses most edges related to the class *Profile*, *Student*. SPARK does not recognise *profile* and *student1* as beans and hence no methods are called. On debugging the Spring implementation, we found many reflective calls. Figure 3.3 shows the return statement that calls a reflective method *newInstance(args)* when instantiating a bean. So, every call to the method *getBean(...)* calls some reflective calls within the Spring framework which SPARK cannot resolve. SPARK is unable to establish the link between the class (defined in the java file) and the bean (defined in the XML file) due to its inability to handle this reflective method call. Section 2.4 provides more information about *newInstance(args)* and other reflective calls.

```
public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationException {
    Assert.notNull(ctor, "message: " + "Constructor must not be null");
    try {
        ReflectionUtils.makeAccessible(ctor);
        return ctor.newInstance(args);
    } catch (InstantiationException var3) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Is it an abstract class?", var3);
    } catch (IllegalAccessException var4) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Is the constructor accessible?", var4);
    } catch (IllegalArgumentException var5) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Illegal arguments for constructor", var5);
    } catch (InvocationTargetException var6) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Constructor threw exception", var6.getTargetException());
    }
}

public static Method findMethod(Class<?> clazz, String methodName, Class<?>... paramTypes) {
```

FIGURE 3.3: Reflective Calls in Spring

3.2 Spring Application with Core Annotations

Listings 3.1, 3.2 and 3.3 show an example application written in Spring using core annotations.

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2
3 <beans xmlns = "http://www.springframework.org/schema/beans"
4     xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:context = "http://www.springframework.org/schema/context"
6     xsi:schemaLocation = "http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8     http://www.springframework.org/schema/context
9     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10
11     <context:component-scan base-package="Vehicle"/>
12
13     <context:annotation-config/>
14 </beans>
```

LISTING 3.1: Spring Application with Core Annotations (1): Bean.xml

```
1 @Component("engine")
2 public class Engine
3 {
4     @Autowired
5     Engine Engine()
6     {
7         return new Engine();
8     }
9     void print()
10    {
11        System.out.println("In print()");
12    }
13 }
```

LISTING 3.2: Spring Application with Core Annotations (2): Engine.java

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("
6         file:/home/mugdha/eclipse-workspace/SpringProject/src/Bean.xml");
7         Engine e = (Engine) context.getBean(Engine.class);
8         e.print();
9     }
}
```

LISTING 3.3: Spring Application with Core Annotations (3): App.java

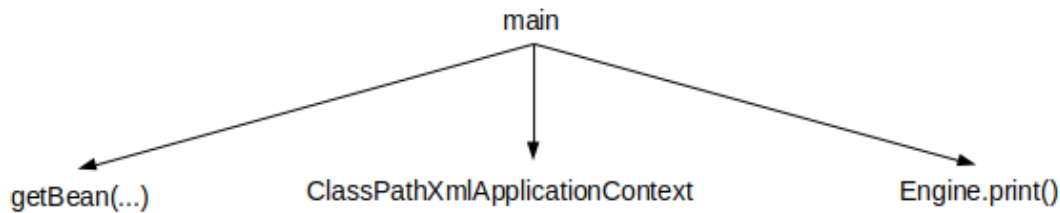


FIGURE 3.4: CHA/RTA/VTA Call Graph

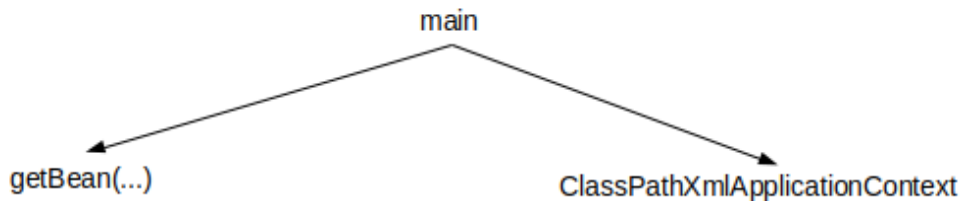


FIGURE 3.5: SPARK Call Graph

This example uses the `@Autowired` annotation on a constructor to initiate the bean we need (Listing 3.2). The constructor `Engine()` is annotated with `@Autowired` and it instantiates the bean `engine` (Line 9 in Listing 3.2). An important observation is that there is no separate definition of the bean "engine" in the XML file (Listing 3.1). More information and examples of annotations can be found in Section 2.3.3.

For the given example, in algorithms like CHA, RTA and VTA, the total number of edges in the call graph is 52,553. Some of the edges from the constructed call graph relevant to the input (edges corresponding to the source code and not library) are shown in Figure 3.4.

Call graph constructed by the algorithms CHA, RTA and VTA is an underapproximation. The call graph misses edges to the method annotated with core annotations (`Engine()` in this example).

Usage of SPARK for constructing call graph for the example leads to a significantly smaller call graph (Total number of edges in the call graph: 20,537). Some of the edges from the constructed call graph relevant to the input (edges corresponding to the source code and not library) are shown in Figure 3.5.

Call graph constructed by SPARK is an underapproximation because it misses edges to the method annotated with core annotations (`Engine()` in this example). It also misses the call to `print()` due to its inability to detect the `Engine` bean. This is found to be true for Spring core annotations apart from `@Autowired` (namely `@Bean`, `@Qualifier`, `@Required`, `@Value`, `@Primary`).

On debugging the Spring implementation, we observed that the classes handling annotations (`AutowiredBeanPostProcessor`, `QualifierBeanPostProcessor` etc.) are responsible for invoking the annotated methods. These classes contain reflective method invocation calls which are completely ignored by these call graph construction algorithms. Figure 3.6 shows a reflective call to `method.invoke(...)`. More information about `method.invoke(...)` and other reflective calls can be found in Section 2.4.

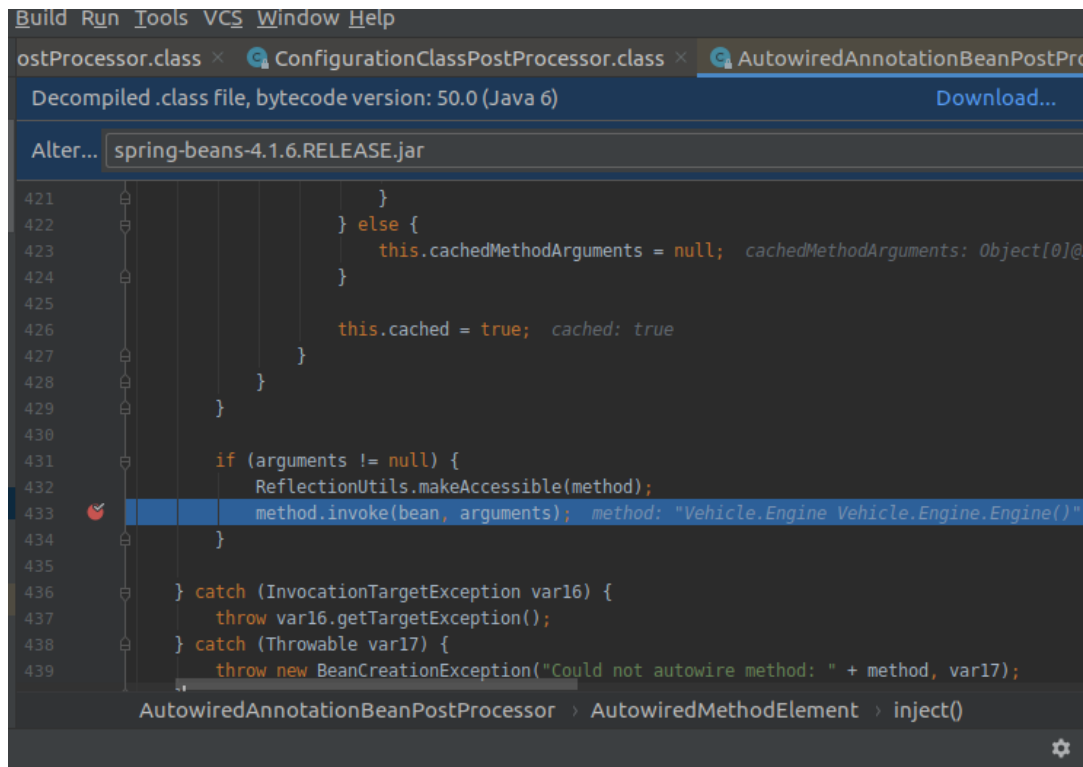


FIGURE 3.6: Reflective Calls in Annotation Handler

3.3 Summary

The examples show that the existing algorithms CHA, RTA, VTA and SPARK are not sound for the application call graph of Spring applications due to their inability to handle reflective calls. Reflective calls are used inside the Spring framework for the Spring bean implementation as well as Spring annotation implementation. Spring beans are the backbone of any Spring application and SPARK is unable to construct a call graph that takes into account beans instantiated in the program (both using XML file or using annotations).

4 Our Solution

This chapter discusses some concepts to handle reflective calls and proposes our final solution.

Partial Call Graph construction involves constructing a call graph that explicitly represents the call relations between the application and summarizes the library by a single node in the graph. It is described in detail in Section 5.1.3. We can use this concept and the *-no-bodies-for-excluded* option can be used to ensure that Soot never loads the bodies of the excluded classes. In that case, SPARK ignores it. Here, the excluded classes could be all the classes from the package *org.springframework*. The Spring application uses calls like *getBean(...)* which return information from Spring to our application. These are the call back edges from the library i.e. call graph edges anchored at a variable from within the library that call a method within the application. Since Soot has no information about call sites in the library, these edges may just be depicted as edges from library to the application. These may not give us enough information about the call *getBean(...)*. Also, the output call graph would be small in size (because it will not have edges within library) but the approach would be unsound. Due to these reasons, we decided that this would not be the best approach.

Some tools handle reflection in an input program. One such tool is TamiFlex, which is used for taming reflection. It is described in detail in Section 5.2.2. In this thesis, we explore handling reflection by converting reflective calls in the input code into non-reflective calls by making suitable changes in the code.

We now discuss our solution that can be used to deal with reflective calls in a Spring application without making any changes to the underlying call graph construction algorithm used. We will use the example in Listings 4.1 and 4.2 to illustrate our solution.

4.1 The Key Idea

We observed that when Spring IoC container loads, it initializes all beans by default. It does not wait for an explicit *getBean(...)* call in the main function of any application (eg. *getBean(...)* call in Listing 4.2). The bean data can be extracted from the initialized IoC container. The idea is to stop the execution of the program once the container loads. Information from this container can be used to replace calls and generate a Java program that does not use reflection. This generated Java program will serve as an input to a call graph construction algorithm. This combines dynamic and static analysis and hence can be categorised as hybrid analysis.

This idea is illustrated in Figure 4.1. The input is a program P (Listing 4.2) and a specification (a list of functions whose presence is to be checked in the call graph). We execute P until the Spring IoC container loads and initializes all beans. The output of this dynamic analysis is a Bean Dependence Graph (Section 4.2) which we use along with program P as an input to our transformer.

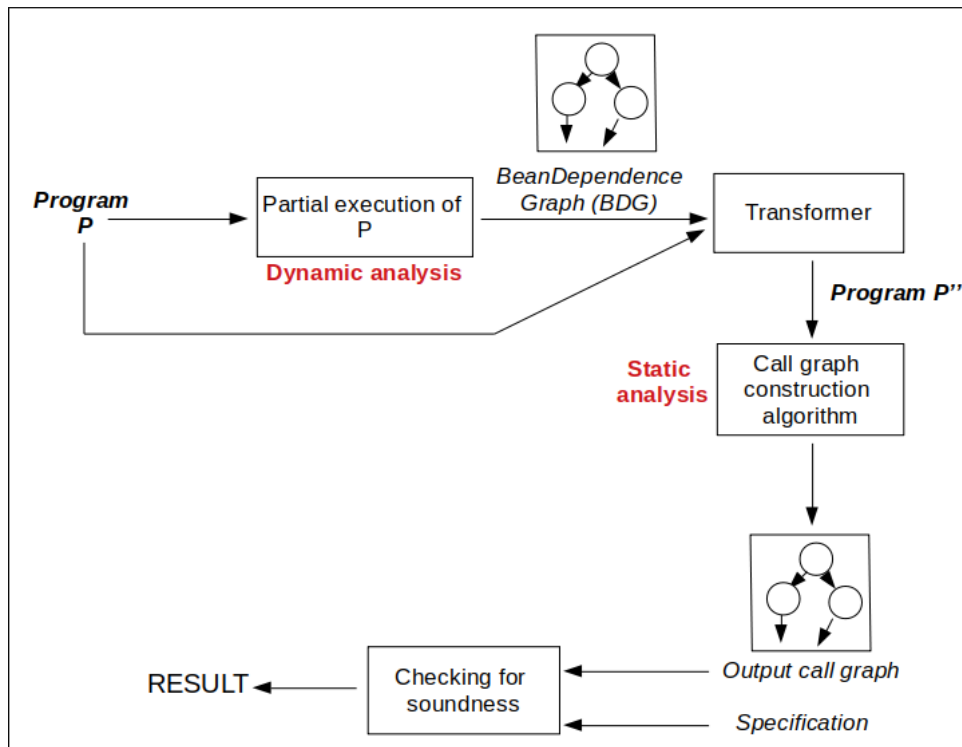


FIGURE 4.1: Our Solution: Extracting Bean Data from Spring Container

Our transformer converts program P into a program P'' . P'' is a simple program statically equivalent to P but which does not use reflection. This program is fed to any call graph construction algorithm. The output call graph is compared with the specification provided. If the functions in the specification are present in the call graph, the call graph construction is sound. Otherwise we conclude it is unsound. We are further exploring if model checking techniques can be used for this soundness check.

For example, Listing 4.2 shows an example program. Our proposed approach would follow the following steps on the original program:

1. Execute the program till the IoC container loads and initializes all beans (Line 10 in Listing 4.2).
2. Extract the bean *Car*.
3. Replace the `getBean(...)` call¹ (Line 11 in Listing 4.2) by a constructor of class *Car* (Line 10 in Listing 4.3).
4. Call the annotated methods `engine()` and `setEngine(...)` (Lines 12 and 13 in Listing 4.3).
5. Give the resulting Java program (shown in Listing 4.3) as input to CG constructor.

¹We focus on `getBean(...)` calls in this algorithm. A call to `getBean(...)` instantiates beans and this is the first step in fetching beans in any Spring application. There may be other functions which need replacing, we have not considered them in this algorithm

```
1 @Component
2 public class Car
3 {
4     private Engine engine;
5     @Autowired
6     public void setEngine(Engine engine)
7     {
8         this.engine = engine;
9     }
10    @Bean
11    Engine engine()
12    {
13        return new Engine();
14    }
15    void print() {...}
16 }
17 public class Engine
18 { .. }
```

LISTING 4.1: Our Solution(1): Classes of the input program

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("
6         file:Bean.xml");
7         Car c = (Car) context.getBean(Car.class);
8         c.print();
9     }
}
```

LISTING 4.2: Our Solution(2): Original Program P

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Car c = new Car();
6         Engine e = new Engine();
7         e = c.engine();
8         c.setEngine(e);
9         c.print();
10    }
11 }
```

LISTING 4.3: Our Solution(3): Updated Program P''

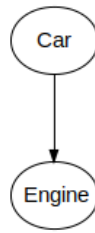


FIGURE 4.2: Bean Dependence Graph

4.2 Bean Dependence Graph

We introduce a new data structure, `BeanDependenceGraph` (BDG) which is used to extract bean data.

Definition 4.2.1. *Bean Dependence Graph is a directed acyclic graph $G = (V, E)$ where every vertex is a pair of a bean and a list of abstract objects (objects of the bean class and its subclasses). We use $node@b$ for the vertex $(b, listOfAbstractObjects) \in V$.*

$\forall u, v \in V$ where $u = (u_b, u_listOfObjects)$ and $v = (v_b, v_listOfObjects)$,

$(u, v) \in E$ iff bean v_b is autowired in bean u_b .

Every bean b in BDG points to a list of beans which are autowired as fields of class b .

For the example in Listing 4.1, the BDG created is illustrated in the Figure 4.2. Since the class `Car` has an `@Autowired` method `setEngine(...)` (Line 6 in Listing 4.1), it autowires the bean `Engine`. Hence `Car` would have an outgoing edge to `Engine`. *For our proposed algorithm, we assume that the Bean Dependence Graph extracted is acyclic.*

In some cases (discussed in Section 4.3.1), Spring constructs a dependence graph which can be refined to be used as a BDG. The dependence graph created by Spring consists of beans as nodes. The autowired relationship between beans is captured in the form of the edges. An obvious refinement to this dependence graph is to add a list of objects per node. In some other cases, we may have to construct the whole graph explicitly.

4.3 Handling Different Containers and Scopes

Spring framework consists of different IoC containers and beans have different scopes. Our algorithm has to handle all such possible containers and scopes.

4.3.1 Spring IoC containers

Spring framework has four IoC containers. They all have different ways of initializing beans from the inputs (XML file and/or annotations). They are categorized as follows:

- `XmlBeanFactory`: `XmlBeanFactory` considers input only from the XML file. Annotations are ignored and Spring does not store any specific information which will help construct a BDG.

```
1 public class App
2 {
3     public static void main(String[] args) {
4         ApplicationContext context = new ClassPathXmlApplicationContext
5         (...);
6         Car c = (Car) context.getBean("car");
7         c.print();
8         Car cc = (Car) context.getBean("car");
9         cc.print();
10 }
```

LISTING 4.4: Original Program

- `AnnotationConfigApplicationContext`: This container looks at Java classes as input but it looks at bean definitions only in those classes. It ignores all annotations except `@Component` and `@Bean`. Spring ignores `@Bean` annotations but creates a dependency graph. We can update the dependency graph (if required) to construct a BDG for this container.
- `ClassPathXmlApplicationContext` and `FileSystemXmlApplicationContext`: These containers consider both XML file and Java classes as input. Spring constructs a dependency graph. For building this graph, these containers consider functions with `@Autowired` annotation but not `@Bean` annotation. We can update the dependency graph (if required) to construct a BDG for this container.

All these containers need to be handled separately due to the different inputs they consider. In this algorithm, we will focus on `ClassPathXmlApplicationContext` and `FileSystemXmlApplicationContext` since they are the most generic containers ie. consider both XML file and Java classes as inputs.

4.3.2 Bean Scopes

As mentioned in Section 2.3.3, Spring beans have different scopes. These scopes can be specified either in the XML file or in annotations. The basic scopes are singleton and prototype. The default scope is singleton and it scopes the bean definition to a single instance per Spring container. If the scope is prototype, the Spring container creates a new bean instance of the object every time a request for that specific bean is made.

Listing 4.4 shows the input program where we have multiple `getBean(...)` calls (Lines 5 and 7) for the same class `Car`.

If the scope of the bean `Car` is singleton, $cc = c$. If the scope is prototype, Spring creates a new instance for every `getBean(...)` call. Hence $cc \neq c$. This shows the importance of considering scopes when processing `getBean(...)` calls.

Since we execute the program till the container is initialized, all the bean information (including scoping of beans) will be captured in the container. For every `getBean(...)` call, the algorithm will check its bean scope and transform the input program accordingly. Listing 4.5 shows the output program our algorithm generates if the scope of the bean `Car` is singleton. Listing 4.6 shows the output generated if the scope is prototype.

```
1 public class App
2 {
3     public static void main(String[] args) {
4         Car c = new Car();
5         Engine e = new Engine();
6         e = c.engine();
7         c.setEngine(e);
8         c.print();
9         Car cc = c;
10        cc.print();
11    }
12 }
```

LISTING 4.5: Output Program: Singleton Scope

```
1 public class App
2 {
3     public static void main(String[] args) {
4         Car c = new Car();
5         Engine e = new Engine();
6         e = c.engine();
7         c.setEngine(e);
8         c.print();
9         Car cc = new Car();
10        Engine e2 = new Engine();
11        e2 = cc.engine();
12        cc.setEngine(e2);
13        cc.print();
14    }
15 }
```

LISTING 4.6: Output Program: Prototype Scope

4.4 Algorithm

For every `getBean(...)` call in the input program P , we create a BDG rooted at the bean created. This BDG is passed as an input to the algorithm 1. Algorithm 1 traverses the BDG in postorder and calls annotated functions on the objects created. These objects are stored in every node of a BDG. It does not construct any objects or call any functions on the root. It just returns the root of the BDG to the main algorithm (Algorithm 2). The main algorithm transforms the input program P to a non-reflective output program P' using the bean information retrieved from the BDG.

Algorithm 1 Traverse the BDG in postorder and call annotated functions

```

1: procedure POPULATEBEANS( $BDG$ ):
2:   for (node  $n \in BDG$ ) do
3:     Instantiate object  $obj$  for all children( $n$ )
4:     Call all methods annotated by @Autowired on  $obj$ 
5:     if ( $n$  is not the root) then
6:       Instantiate object  $obj$  for  $n$ 
7:       Call all methods annotated by @Autowired on  $obj$ 
   return root( $BDG$ )

```

Algorithm 2 Transform Code Using Bean Information

```

Input: Program  $P$ ,  $BDG$ 
Output: Program  $P'$ 
1: procedure MAIN2:
2:   Execute input program  $P$  till initialization of container
3:   for every getBean call  $C$  in input program  $P$  do
4:      $b$  = bean returned by  $C$ 
5:     root = populateBeans( $BDG$ )
6:     Update the output program:
7:       Instantiate object  $obj$  for root
8:       Call all methods annotated by @Autowired on  $obj$ 
9:       Call all direct methods of class( $root$ ) on  $obj$ 

```

This approach considers both XML file and annotations as input. It extracts information from Spring container and reduces redundant programming. This approach is sound and the resulting call graph is precise. It is a combination of dynamic as well as static analysis, and can be categorised as hybrid analysis.

```
1 @Component
2 public class Engine
3 {
4     @Autowired
5     Wheel w;
6     @Autowired
7     Fuel f;
8     @Autowired
9     public void setWheel(Wheel w1)
10    {
11        this.w = w1;
12    }
13    @Autowired
14    public void setFuel(Fuel f1)
15    {
16        this.f = f1;
17    }
18    public void func() {...}
19 }
20 @Component
21 public class Wheel
22 { .. }
23 @Component
24 public class Fuel
25 { .. }
```

LISTING 4.7: Example(1): Classes of the input program

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("
6         file:Bean.xml");
7         Engine e = (Engine) context.getBean("engine");
8         e.func();
9     }
}
```

LISTING 4.8: Example(2): Original Program P


```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6         Wheel w = new Wheel();
7     }
8 }

```

LISTING 4.9: Intermediate Program P'

4.5 An Example

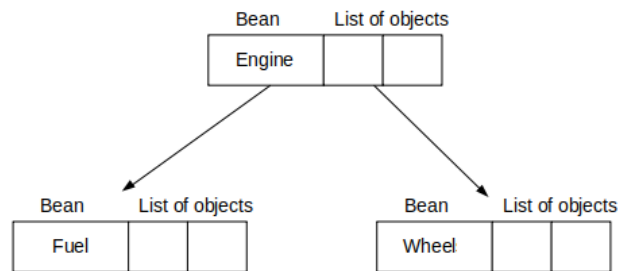


FIGURE 4.3: Bean Dependence Graph (BDG)

Listings 4.7 and 4.8 illustrate the input program and Figure 4.3 shows the BDG created for it. The leaf nodes in the BDG would be *Fuel* and *Wheel* since they do not have *@Autowired* fields. The bean *Engine* is the parent of *Fuel* and *Wheel*.

Algorithm 1 populates the BDG with the suitable objects of classes *Fuel* and *Wheel*. Furthermore, the algorithm updates the output program with the suitable *@Autowired* functions belonging to bean classes of all nodes except the root (in this case, none). The root is handled in algorithm 2. This is because we need to call more functions on the root (in this case, *Engine*). The output program and BDG are shown in Listing 4.9 and Figure 4.4 respectively.

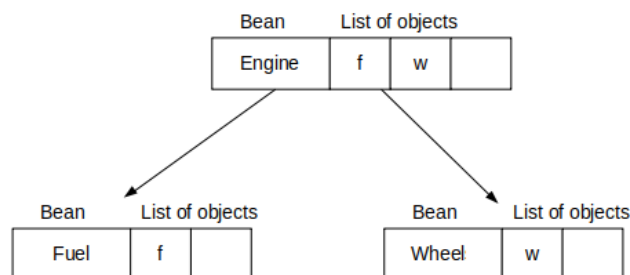


FIGURE 4.4: Intermediate BDG

Algorithm 2 transforms code using the bean information extracted from the BDG. For every *getBean(...)* call in the input program P, this algorithm populates the BDG using algorithm 1. It further updates the input program with object created and all

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6         Wheel w = new Wheel();
7         Engine e = new Engine();
8         e.setWheel(w);
9         e.setFuel(f);
10        e.func();
11    }
12 }
```

LISTING 4.10: Output Program P''

functions (annotated and non annotated) called on the root of the BDG (in this case, *setWheel(...)*, *setFuel(...)* and *func()*). The output program is shown in Listing 4.10.

4.6 Limitations

The algorithm depicted in Section 4.4 deals with the Spring API method *getBean(...)* and some core annotations (*@Bean*, *@Autowired*, *@Qualifier*, *@Component*, *@Configuration*, *@Scope*). At the moment, our prototype proposal does not handle the following:

- Other annotations and reflective calls.
- MVC framework in Spring.
- Applications with a cyclic bean dependency.

As of now, our prototype proposal has no thorough practical implementation and hence we have been unable to evaluate it on real-world examples.

4.7 Summary

This chapter discussed some approaches that we could use to deal with the unsoundness in call graph construction algorithms. The unsoundness is majorly due to reflective calls. All approaches in this chapter gave ideas to handle reflection.

5 Related Work

Static analysis of the Reflection API has attracted significant research effort. Until 2005, the analysis of code which uses reflection was considered to be out of bounds for static analysis. In 2005, Livshits *et al.*[14] published an analysis of how reflection was used in six Java projects, proposing three unsound assumptions and using these to (partially) statically resolve the targets of dynamic method calls (Section 5.1.2). Since then more tools were based on similar assumptions. Soot[26] and WALA[27] are well known frameworks for performing static analysis. In 2017, Landman *et al.*[10] surveyed what static analysis approaches existed for Java and what their limitations were. They also analyzed how real-world Java code uses the Reflection API, and how many Java projects contain code challenging state-of-the-art static analysis. In their corpus, reflection can not be ignored for 78% of the projects. This shows how important it has become to handle reflection soundly in static analyses.

In this chapter, we discuss some more approaches the Program Analysis community has proposed from 2005.

5.1 General Call Graph Construction

This Section discusses some explored call graph construction methods and reflects on how they differ from our approach.

5.1.1 Precise Analysis of String Expressions

In 2003, Christensen *et al.*[7] presented a static analysis technique for extracting a context-free grammar from a program and applying a variant of the Mohri-Nederhof approximation algorithm to approximate the possible values of string expressions in Java programs. They used the Soot framework[26] to parse class files and compute interprocedural control flow graphs. Their algorithm for string analysis can be split into a front-end that translates the given Java program into a flow graph, and a back-end that analyzes the flow graph and generates finite-state automata.

This technique resolves reflective calls with an approximation of string expressions, as opposed to our proposed algorithm, which uses dynamic information to convert reflective calls into semantically equivalent non reflective calls.

5.1.2 Call Graph Discovery using Points-to Analysis

This technique, proposed in 2005 by Livshits *et al.*[14] uses sound points-to analysis to determine all the possible sources of strings that are used as class names. It is one of the first techniques that attempted to resolve a reflective call for the purpose of static analysis. The pointer analysis-based approach fully resolves the targets of a reflective call if constant strings account for all the possible sources. The program points where input strings are defined are called specification points.

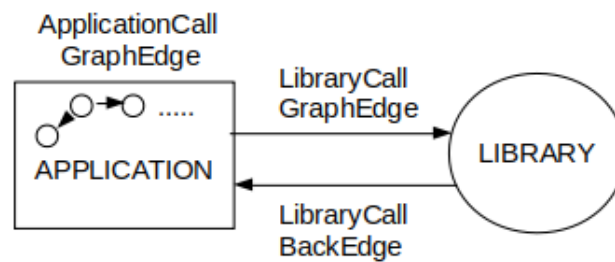


FIGURE 5.1: Partial Call Graph

Unfortunately the number of specification points in a program can be large. Instead of asking users to specify the values of every possible input string, this approach takes advantage of casts, whenever available, to determine a conservative approximation of targets of reflective calls that are not fully resolved. The input program can be represented as a set of relations in *bddb*, a BDD-based program database. The program database and the constraint resolution tool allows program analyses to be expressed in a succinct way as a set of rules in Datalog, a logic programming language. Points-to information is compactly represented in *bddb* with binary decision diagrams (BDDs) and can be accessed and manipulated efficiently with Datalog queries. Finally, user-provided specification is used for the remaining set of calls (calls whose source strings are not all constants) in order to obtain a conservative approximation of the call graph.

Points-to analysis determines all possible sources of strings, which can further resolve the targets of reflective calls. Our proposed algorithm does not determine sources of strings at all, we just use dynamic information to transform the given program into a simpler program with non reflective calls.

5.1.3 Averroes

Whole program call graph construction is challenging for a program. Even if the call graph construction algorithm itself is efficient, just reading all of the library dependencies of a program takes a long time. Additionally, in many cases, the whole program may not even be available for analysis. This generated interest in the development of algorithms that analyze only parts of a program.

A partial call graph is a call graph that explicitly represents the call relations between the analyzed parts of the program, usually the application, and summarizes the unanalyzed parts of the program, usually the library, by a single node in the graph. For partial call graph construction, library is modelled as a single summary node. A call edge is created for each possible call between application methods, but no edges are created to represent calls within the library. All methods in the library are assumed to be reachable from any other method in the library[1].

Figure 5.1 illustrates the types of call graph edges. The call graph edges can be categorized as:

- Application Call Graph Edges: Model the calls within application methods

- Library Call Graph Edges: Represent the calls into the library
- Library Call Back Edges: Define the calls out of the library

A realistic yet very useful assumption is that the code of the library has been compiled without access to the code of the application. This is referred to as the *Separate Compilation Assumption*[1]. Ignoring the library call back edges may render the generated call graph unsound. Thus, it is crucial to precisely define, based on the separate compilation assumption, how the library summary node interacts with the application methods in the call graph.

In 2012, Karim *et al.*[2] proposed *Averroes*, a tool that intends to provide the same input environment to the whole-program analysis, but without analyzing any actual code of the original library classes. For each application class, *Averroes* examines only the constant pool to find all references to library classes, fields, functions. *Averroes* uses this information to build a model of the class hierarchy and the overriding relationships between methods in the program. Since *Averroes* examines only a small fraction of the library, its execution can be much faster than a typical whole-program analysis.

The output of *Averroes* is a placeholder library. *Averroes* uses Soot to generate this library. The application classes together with the generated placeholder library make up a self-contained whole program that can be given as input to any whole-program analysis[2]. *Averroes* models reflective behaviour in the library in two ways. First, whenever a call site in the application calls a library method, *Averroes* assumes that any argument of the call that is a string constant could be the name of an application class that the library instantiates by reflection. For every such string constant that is the name of an application class, *Averroes* generates a new instruction and a call to the default constructor of the class. Second, *Averroes* reads information about uses of reflection in the format of TamiFlex (Section 5.2.2).

Averroes transforms a given input environment into a simpler environment i.e. a placeholder library. This largely differs from our proposed algorithm, where we transform an input program into a simpler program, keeping the environment (library, Spring API) the same.

5.1.4 Call Graph Construction for Java Libraries

Constructing call graphs for Java libraries is challenging. Libraries can be extended by their users via inheritance. Libraries also consist of classes and interfaces that define the public API and those which belong to the library private implementation. Ignoring the first property leads to the construction of call graphs that miss important edges (unsoundness), while ignoring the second property leads to call graphs with many spurious edges (imprecision). In the first scenario, the library is assumed to be open, i.e., all non-private classes, fields, and methods can be accessed; non-final classes can be extended and non-final methods can be overridden. We use the term open-package assumption (OPA) to refer to this assumption. Call graphs built based on OPA represent the unrestricted usage scenarios of the library. In the second scenario, only the code that belongs to a library's public API is used or gets extended by users of it. In Java, e.g., a library's classes, fields and methods with package visibility do not belong to the public API. We refer to this case as the closed-package assumption (CPA).

It is not possible to adequately address both scenarios by using the same call-graph algorithm. If we did, the algorithm would be either unsound or imprecise depending on the scenario in which it is used. Thus this led to two different call

graph algorithms for libraries, both modelled on CHA. The first algorithm is sound but makes very conservative assumptions. The second algorithm gives soundness at the cost of precision[16]. This was studied in 2016.

Our proposed algorithm does not differ between library scenarios (OPA, CPA). Instead our algorithm uses dynamic information to rewrite any input program and passes the rewritten program as an input to any call graph construction algorithm.

5.2 Reflection and Framework Handling for Call Graph Construction

This Section talks about reflection and framework handling for call graph construction.

5.2.1 DOOP Framework

DOOP Framework for points-to analysis of Java programs was proposed by Smaragdakis *et al.*[5] in 2009. DOOP makes use of Datalog language for specifying the program analyses. Datalog, being a declarative language separates the specification of an analysis from its implementation. This allows multiple techniques for efficient execution, all expressed at the level of Datalog evaluation.

In order to execute Datalog programs efficiently, the low-level representation of relations should be compact and an indexing scheme should be in place so that all rules are executed efficiently. A commercial Datalog engine developed by LogicBlox Inc. is used which allows the user to specify maximum cardinalities for the domains of variables. The relations are indexed by very routine data structure, B-trees.

DOOP supports both context-sensitive as well as context-insensitive pointer analysis. Call graph construction is also specified in Datalog. The interdependency between call graph construction (i.e., which methods are reachable in a given context) and points-to analysis is expressed as plain Datalog mutual recursion. This allows call graph discovery to be on-the-fly. DOOP performs sophisticated reflection analysis. For example, DOOP uses distinct representations of instances of `java.lang.Class` for every class in the analyzed program. This resolves some reflection automatically.

DOOP Framework uses Datalog to specify program analyses. Call graph construction is also specified in Datalog. Our proposed algorithm does not use any declarative language to specify analyses. We do not modify any analyses, but we modify the input program into a simpler program that may generate a sounder call graph.

5.2.2 Tamiflex

In 2011, Bodden *et al.*[4] proposed TamiFlex, a dynamic tool for taming reflection. Virtually call graph construction is a whole-program analysis; the analysis must consider the entire program to deliver sound results. Presence of reflective calls in programs can lead to unsound call graphs. TamiFlex aims at logging the reflective calls into a file and estimating the targets of reflective calls. This information can then be fed to any static analysis to get sound call graphs.

It uses two agents: Play-out agent and Play-in agent. Play-out Agent logs information about reflective calls and dumps all classes to disk that the running program loads or generates. In many cases, however, users may want to use static-analysis results to transform classes, e.g., to optimize or instrument them. In these cases,

one faces the problem of re-packaging the transformed classes in such a way that the original program finds the classes where it expects them. Without special tool support, this can be either hard, for instance if the program loads the classes from a remote location, or even impossible, if the program generates the classes on the fly. The Play-in Agent solves this problem by re-inserting offline-transformed classes into a running program. The agent even replaces classes that an application generates at runtime.

If the aim is only to analyze a program, users can use only the Play-out agent. The resulting log file will give the users complete information of the reflective calls. But simply logging reflective calls is not enough as most static-analysis tools would be unable to interpret these logs. In many cases, users may want to use static-analysis results to transform classes, e.g. to optimize or instrument them. Play-in agent can be used for the same.

Tamiflex logs reflective calls into a file which is then used to transform classes and generate a sound call graph. Tamiflex executes the complete input program once and logs reflective calls. As opposed to this, our algorithm executes a part of the input program to gain some dynamic information. This dynamic information is then used to transform the input program into a simpler program with non reflective calls.

5.2.3 Framework for Frameworks

Framework for Frameworks (F4F) is a system for effective taint analysis of framework-based web applications developed by Sridharan *et al.*[23] in 2011. Static analysis of web applications is significantly hindered by their use of frameworks. Framework implementations often invoke application code using reflection, based on information provided in configuration files. The idea is to define Web Application Framework Language (WAFL), a simple specification language for expressing framework-related behaviors of web applications. Automatic WAFL generators can be built for Java web frameworks. Then these WAFL specifications are used to generate a technique to enhance taint analysis.

F4F handles Spring framework. F4F has separate handlers for all abstract implementations (all abstract Controllers), and it chooses the most suitable one for each application controller based on its supertypes. It then uses WAFL to enhance taint analysis. If the aim is only to do a taint analysis, F4F is a very useful tool[23]. F4F was developed primarily for use by IBM customer engineers working with customer code. It is relatively integrated into AppScan products and hence has not been made open source.

F4F can be extended if we use WAFL generation with call graph construction algorithms. WAFL indirectly uses changes in call graphs to generate a correct summary and perform taint analysis. These call graphs may be restricted to specific customer domains. If we talk about only these domains, it may be interesting to observe how WAFL fares in terms of soundness and precision of call graphs.

F4F indirectly uses changes in call graphs to generate a correct summary and perform taint analysis. F4F focuses on performing taint analysis, as opposed to our algorithm which focuses on generating sound call graphs.

5.2.4 Self-Inferencing Reflection Resolution for Java

Despite the large literature on pointer analysis for Java, almost all the analyses reported are unsound in the presence of reflection since it is either ignored or handled

partially. In 2014, Li *et al.* [13] presented a static reflection analysis, Elf which is integrated into Doop (Section 5.2.1) for analyzing Java programs. Some reflection analyses have suggested resolving reflective calls by tracking the flow of class/method/-field names in the program. Elf goes beyond those analyses by taking advantage of a self-inferencing property inherent in reflective code. Elf makes use of a key observation: many reflective calls are self-inferenceable. We can approximate their targets reasonably accurately based on the dynamic types of the arguments of their target calls and the downcasts (if any) on their returned values. Consider the following code snippet:

```
1 Object[] parameters = new Object[] {this};
2 ...
3 for (int i = 0; i < size; i++)
4 {
5     method = target.getClass().getMethod
6     ("_" + cmd, parameterTypes);
7     retval = method.invoke(target, parameters);
8     ...
9 }
```

The method name (the first argument of `getMethod()`) is statically unknown as part of it is read from command line `cmd`. However, the target method (represented by `method`) can be deduced from the second argument (`parameters`) of the corresponding reflective-action call `invoke()` [12].

With respect to pointer analysis, we can divide the pointer-affecting methods in the Java reflection API into three categories: (1) *entry methods*, e.g. `forName()` for creating Class objects, (2) *member-introspecting methods*, e.g. `getDeclaredMethod()` for retrieving Method (Constructor), and (3) *side-effect methods*, e.g. `newInstance()`, `invoke()` that affect the pointer information in the program reflectively. Elf is the first to handle all such accessor methods in reflection analysis. Elf has been evaluated against Doop on 11 DeCapo benchmarks and two Java applications, Eclipse4 and Javac. The results show that Elf can make a decent tradeoff among soundness, precision and scalability while resolving usually more reflective call targets than Doop.

Elf approximates the targets of reflective calls reasonably accurately based on the dynamic types of the arguments of their target calls and the downcasts (if any) on their returned values. On the other hand, our algorithm does not approximate the targets of reflective calls. We transform the input program to rid it of reflective calls which, in turn, simplifies the call graph construction process to a large extent.

5.2.5 Sound Static Handling of Java Reflection

Full soundness can not be practically achieved. Soundness can be looked at something you improve upon. Empirical soundness is a quantification of how much of the dynamic behavior the static analysis covers. Here we look at some techniques proposed in 2015 to enhance empirical soundness of static analysis and statically handling reflective calls:

Richer string flow: The parameter of a reflective call like `forName()` could be any string expression. It could be a class name as well. In order to estimate what classes, fields or methods a string expression may represent, we can use substring matching: all string constants in the program text are tested for prefix and suffix matching against known class, method and field names. The strings that may refer to such entities are handled with more precision than others during analysis.

Use-Based Reflection Analysis: Reflection is one of the few parts of static analysis that is under-approximated rather than over-approximated. The first use-based reflection analysis technique back-propagates information from the use-site of a reflective result to the original reflection call that got under-approximated. The idea is to create a marker object o to stand for unknown objects that a method invocation M may return. This object o flows through points-to analysis. If the receiver of any call C is this marker object o , it remembers its origin (M). Call C also marks a second marker object o' which flows through points-to analysis. When o' reaches the site of a cast, it returns the invocation M . This is referred to as inter-procedural back-propagation. The second analysis consists of inventing objects of the appropriate type at the point of a cast operation that has received the result of a reflection call. This works as a forward propagation technique. The backward propagation technique affects precision. Whenever a special, unknown reflective object flows to the point of a cast, instead of informing the result of invocation M , the technique invents a new, regular object of the right type that starts its existence at the cast site. The “invented” object does not necessarily abstract actual run-time objects. Instead, it exploits the fact that a points-to analysis is fundamentally a may-analysis: it is designed to possibly yield over-approximate results, in addition to those arising in real executions[17].

These techniques approximate the targets of reflective calls. Our algorithm does not approximate the targets of reflective calls, but transforms the input program to rid it of reflective calls. This transformed program is then used as input to any call graph construction algorithm.

5.3 Summary

This chapter briefly discussed the methods tried in the past for sounder static analyses in the presence of reflective calls.

6 Conclusion

Constructing precise call graphs is an important prerequisite to a sound and precise static analysis. While a lot of research effort has gone into development of sound call graph construction algorithms, the area of call graph construction for web frameworks remains almost unexplored.

The first contribution of this thesis is the observation that unsoundness of call graph construction algorithms for Spring framework is only due to the presence of reflective calls. This is described in Chapter 3 and it narrows down our problem to handling reflection in static analysis.

Another contribution of this thesis is that we present a hybrid analysis algorithm, as discussed in Chapter 4. This algorithm generates a simple, non-reflective version of the input Spring program written in Java. This algorithm executes the input program till the initialization of the Spring container. We retrieve some bean information from this execution and use it to transform the input program to a simple non-reflective program. This algorithm would work on programs written in web frameworks designed on top of the Spring framework and the call graph constructed for these programs would be sound and precise, as required.

We hope that the idea of this algorithm helps program analysers and web developers in the future to study and (hopefully) improve soundness of call graph construction algorithms and precision of the resulting call graph.

List of Abbreviations

IDE	I ntegrated D evelopment E nvironment
POJO	P lain O ld J ava O bjects
POJI	P lain O ld J ava I nterfaces
API	a pplication- p rogram i nterface
CHA	C lass H ierarchy A nalysis
RTA	R apid T ype A nalysis
VTA	V ariable T ype A nalysis
SPARK	S oot P ointer A nalysis R esearch K it
IoC	I nversion of C ontrol
DI	D ependency I njection
IoC	I nversion of C ontrol
XML	e Xtensible M arkup L anguage
MVC	m odel- v iew- c ontroller
F4F	F ramework for F rameworks
WAFL	W eb A pplication F ramework L anguage
BDD	B inary D ecision D iagram
OPA	o pen- p ackage a ssumption
CPA	c losed- p ackage a ssumption
WALA	W atson L ibraries for A nalysis

List of Figures

2.1	Class Hierarchy	6
2.2	CHA Call Graph	6
2.3	RTA Call Graph	7
2.4	VTA Call Graph	7
2.5	Pointer Assignment Graph	9
2.6	SPARK Call Graph	9
2.7	Spring Architecture	10
3.1	CHA/RTA/VTA Call Graph	21
3.2	SPARK Call Graph	21
3.3	Reflective Calls in Spring	22
3.4	CHA/RTA/VTA Call Graph	24
3.5	SPARK Call Graph	24
3.6	Reflective Calls in Annotation Handler	25
4.1	Our Solution: Extracting Bean Data from Spring Container	27
4.2	Bean Dependence Graph	29
4.3	Bean Dependence Graph (BDG)	34
4.4	Intermediate BDG	34
5.1	Partial Call Graph	37

Listings

1.1	A Spring Application (1): Student.java	2
1.2	A Spring Application (2): Profile.java	2
1.3	A Spring Application (3): MainApp.java	2
1.4	A Spring Application (4): Beans.xml	3
2.1	Call Graph Construction Example	5
2.2	SPARK: Input Program	9
2.3	Bean Configuration File: Beans.xml	11
2.4	Example of @Autowired: Profile.java	14
2.5	A Simple Spring Example (1): Student.java	15
2.6	A Simple Spring Example (2): Profile.java	15
2.7	A Simple Spring Example (3): MainApp.java	16
2.8	A Simple Spring Example (4): Beans.xml	16
3.1	Spring Application with Core Annotations (1): Bean.xml	23
3.2	Spring Application with Core Annotations (2): Engine.java	23
3.3	Spring Application with Core Annotations (3): App.java	23
4.1	Our Solution(1): Classes of the input program	28
4.2	Our Solution(2): Original Program P	28
4.3	Our Solution(3): Updated Program P''	28
4.4	Original Program	30
4.5	Output Program: Singleton Scope	31
4.6	Output Program: Prototype Scope	31
4.7	Example(1): Classes of the input program	33
4.8	Example(2): Original Program P	33
4.9	Intermediate Program P'	34
4.10	Output Program P''	35

List of Tables

2.1	Pointer Assignment Graph Construction	8
-----	---	---

Bibliography

- [1] Karim Ali and Ondřej Lhoták. “Application-Only Call Graph Construction”. In: *Proceedings of the 26th European Conference on Object-Oriented Programming. ECOOP’12*. Beijing, China: Springer-Verlag, 2012, pp. 688–712. ISBN: 978-3-642-31056-0. DOI: [10.1007/978-3-642-31057-7_30](https://doi.org/10.1007/978-3-642-31057-7_30). URL: http://dx.doi.org/10.1007/978-3-642-31057-7_30.
- [2] Karim Ali and Ondřej Lhoták. “Averroes: Whole-program Analysis Without the Whole Program”. In: *Proceedings of the 27th European Conference on Object-Oriented Programming. ECOOP’13*. Montpellier, France: Springer-Verlag, 2013, pp. 378–400. ISBN: 978-3-642-39037-1. DOI: [10.1007/978-3-642-39038-8_16](https://doi.org/10.1007/978-3-642-39038-8_16). URL: http://dx.doi.org/10.1007/978-3-642-39038-8_16.
- [3] Lars Ole Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. University of Copenhagen, 1994.
- [4] Eric Bodden et al. “Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders”. In: *Proceedings of the 33rd International Conference on Software Engineering. ICSE ’11*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 241–250. ISBN: 978-1-4503-0445-0. DOI: [10.1145/1985793.1985827](https://doi.org/10.1145/1985793.1985827). URL: <http://doi.acm.org/10.1145/1985793.1985827>.
- [5] Martin Bravenboer and Yannis Smaragdakis. “Strictly Declarative Specification of Sophisticated Points-to Analyses”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 243–262. ISSN: 0362-1340. DOI: [10.1145/1639949.1640108](https://doi.org/10.1145/1639949.1640108). URL: <https://doi.org/10.1145/1639949.1640108>.
- [6] Call Graph Algorithms. *Call Graph Algorithms*. URL: <https://ben-holland.com/call-graph-construction-algorithms-explained>.
- [7] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. “Precise Analysis of String Expressions”. In: *Proceedings of the 10th International Conference on Static Analysis. SAS’03*. San Diego, CA, USA: Springer-Verlag, 2003, pp. 1–18. ISBN: 3540403256.
- [8] T. Eisenbarth, R. Koschke, and D. Simon. “Aiding program comprehension by static and dynamic feature analysis”. In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. 2001, pp. 602–611.
- [9] Debin Gao, Michael K. Reiter, and Dawn Song. “Gray-Box Extraction of Execution Graphs for Anomaly Detection”. In: *Proceedings of the 11th ACM Conference on Computer and Communications Security. CCS ’04*. Washington DC, USA: Association for Computing Machinery, 2004, pp. 318–329. ISBN: 1581139616. DOI: [10.1145/1030083.1030126](https://doi.org/10.1145/1030083.1030126). URL: <https://doi.org/10.1145/1030083.1030126>.
- [10] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. “Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study”. In: *Proceedings of the 39th International Conference on Software Engineering. ICSE ’17*. Buenos Aires, Argentina: IEEE Press, 2017, pp. 507–518. ISBN: 9781538638682. DOI: [10.1109/ICSE.2017.53](https://doi.org/10.1109/ICSE.2017.53). URL: <https://doi.org/10.1109/ICSE.2017.53>.

- [11] Ondřej Lhoták and Laurie Hendren. “Scaling Java Points-to Analysis Using SPARK”. In: *Proceedings of the 12th International Conference on Compiler Construction*. CC’03. Warsaw, Poland: Springer-Verlag, 2003, pp. 153–169. ISBN: 3-540-00904-3. URL: <http://dl.acm.org/citation.cfm?id=1765931.1765948>.
- [12] Yue Li, Tian Tan, and Jingling Xue. “Understanding and Analyzing Java Reflection”. In: *ACM Trans. Softw. Eng. Methodol.* 28.2 (Feb. 2019). ISSN: 1049-331X. DOI: 10.1145/3295739. URL: <https://doi.org/10.1145/3295739>.
- [13] Yue Li et al. “Self-Inferencing Reflection Resolution for Java”. In: *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 27–53. ISBN: 9783662442012. DOI: 10.1007/978-3-662-44202-9_2. URL: https://doi.org/10.1007/978-3-662-44202-9_2.
- [14] Benjamin Livshits, John Whaley, and Monica S. Lam. “Reflection Analysis for Java”. In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS’05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 139–160. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_11. URL: http://dx.doi.org/10.1007/11575467_11.
- [15] Benjamin Livshits et al. “In Defense of Soundness: A Manifesto”. In: *Commun. ACM* 58.2 (Jan. 2015), pp. 44–46. ISSN: 0001-0782. DOI: 10.1145/2644805. URL: <https://doi.org/10.1145/2644805>.
- [16] Michael Reif et al. “Call Graph Construction for Java Libraries”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 474–486. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950312. URL: <http://doi.acm.org/10.1145/2950290.2950312>.
- [17] Yannis Smaragdakis et al. “More Sound Static Handling of Java Reflection”. In: *APLAS*. 2015.
- [18] Spring Annotations. *Spring Annotations*. URL: <https://springframework.guru/spring-framework-annotations>.
- [19] Spring Annotations. *Spring Annotations*. URL: <https://stackabuse.com/spring-annotations-core-framework-annotations/>.
- [20] Spring Architecture. *Spring Architecture*. URL: https://www.tutorialspoint.com/spring/spring_architecture.htm.
- [21] Spring Project. *Spring Framework*. URL: <https://spring.io/projects/spring-framework>.
- [22] Spring Tutorial. *Spring Tutorial*. URL: <https://www.tutorialspoint.com/spring>.
- [23] Manu Sridharan et al. “F4F: Taint Analysis of Framework-based Web Applications”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 1053–1068. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048145. URL: <http://doi.acm.org/10.1145/2048066.2048145>.

- [24] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41. ISBN: 0897917693. DOI: [10.1145/237721.237727](https://doi.org/10.1145/237721.237727). URL: <https://doi.org/10.1145/237721.237727>.
- [25] Vijay Sundaresan et al. "Practical Virtual Method Call Resolution for Java". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 264–280. ISBN: 158113200X. DOI: [10.1145/353171.353189](https://doi.org/10.1145/353171.353189). URL: <https://doi.org/10.1145/353171.353189>.
- [26] Raja Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.
- [27] WALA Docs. *WALA Docs: Jan 2020*. URL: <http://wala.sourceforge.net/javadocs/trunk/>.
- [28] Wikipedia contributors. *Call Graph* — *Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Call_graph.
- [29] Wikipedia contributors. *Inversion of Control* — *Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Inversion_of_control.
- [30] Wikipedia contributors. *Spring Framework* — *Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Spring_Framework.