MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

AN INVESTIGATION INTO THE UNSOUNDNESS OF STATIC PROGRAM ANALYSIS

A thesis presented in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science
at Massey University, Palmerston North,
New Zealand.

Li Sui

2021

Contents

List of Tables

List of Figures

Abstract

Static program analysis is widely used in many software applications such as in security analysis, compiler optimisation, program verification and code refactoring. In contrast to dynamic analysis, static analysis can perform a full program analysis without the need of running the program under analysis. While it provides full program coverage, one of the main issues with static analysis is imprecision – i.e., the potential of reporting false positives due to overestimating actual program behaviours. For many years, research in static program analysis has focused on reducing such imprecision while improving scalability. However, static program analysis may also miss some critical parts of the program, resulting in program behaviours not being reported. A typical example of this is the case of dynamic language features, where certain behaviours are hard to model due to their dynamic nature. The term "unsoundness" has been used to describe those missed program behaviours. Compared to static analysis, dynamic analysis has the advantage of obtaining precise results, as it only captures what has been executed during run-time. However, dynamic analysis is also limited to the defined program executions.

This thesis investigates the unsoundness issue in static program analysis. We first investigate causes of unsoundness in terms of Java dynamic language features and identify potential usage patterns of such features. We then report the results of a number of empirical experiments we conducted in order to identify and categorise the sources of unsoundness in state-of-the-art static analysis frameworks. Finally, we quantify and measure the level of unsoundness in static analysis in the presence of dynamic language features. The models developed in this thesis can be used by static analysis frameworks and tools to boost the soundness in those frameworks and tools.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisors: Associate Professor Jens Dietrich and Dr. Amjed Tahir for the continuous support of my study and related research, for their patience, motivation, and immense knowledge. Their guidance has pointed me in the right direction throughout the work.

I also would like to express my appreciation to Dr. Catherine McCartin for her patient guidance and advice on writing this thesis. My sincere thanks also go to Dr. George Fourtounis and Mr. Shawn Rasheed for their active collaborations on writing related papers.

This dissertation would not have been possible without funding from the Science for Technological Innovation National Science Challenge of New Zealand under the project: Closing the Gaps in Static Program Analysis.

Last but not the least, I must express my very profound gratitude to my parents and to my partner Shuwen, with their unfailing support and encouragement throughout the development of this thesis and my life in general.

Chapter 1

Introduction

Software systems are ubiquitous in society, penetrating all aspects of modern life. However, poor software quality and vulnerabilities that can be exploited for malicious activities cause significant problems. Novopay [10] is a good example of why software quality matters. Novopay is a payroll system responsible for the pay of 110,000 teachers at 2,457 schools in New Zealand. By the year 2015, Novopay had cost over $45 million in order to be fixed [9]. Bugs can also be fatal. An Airbus that crashed in Spain in 2015 [1] caused 4 casualties due to software bugs. The Panda Burning Incense virus spread widely between 2006 and 2007 in China, with more than 10 million infected devices [115]. During this time, attackers were able to bypass security processes in the Windows operating system and replaced infected files with an hilarious image of a panda burning incense. Furthermore, a staggering number from the Common Vulnerabilities and Exposures (CVE) dataset shows a total of 134,671 vulnerability entries in 2018, of those, 2,088 are Java-related vulnerabilities [3].

Program analysis plays a very important role in modern software development. The focus of this thesis is on static program analysis. There are many applications for static analysis, including bug detection and security analysis, that use program analysis techniques to correctly and fully model how programs behave, so that those bugs or vulnerabilities can be detected.

Static and dynamic analysis are the two main categories of program analysis. Static analysis performs an analysis without the need to execute the program. Static analysis techniques, in particular, have been widely used to detect bugs and other issues **early** in the development cycle, when it is much cheaper to find and fix bugs. Companies like Google and Facebook are increasingly using static analysis techniques and integrating them into their work flow, as part of their development pipeline, to detect different types of bugs and issues [62, 165]. On the other hand, analysis which requires the execution of actual programs is referred to as dynamic program analysis. Commonly used techniques include program testing, debugging and profiling. Program analysis

works for all program languages but this thesis will focus on Java, as it remains one of the most popular programming languages since 2012 [14, 15, 37], and is used across different platforms.

Program bugs can be simple. Consider the code snippet demonstrated in Listing 1.1, which shows a simple infinite recursive loop. If the code is executed and it calls itself recursively, the method stack will exceed its capacity and throw an exception eventually. Another common bug pattern is dereferencing a null pointer as shown at line 2 in Listing 1.2. The code will compile as normal, but a runtime exception will be thrown due to the null pointer error.

Listing 1.1: Infinite recursive loop

```
1 public void foo(){  //infinite recursive loop
2   foo();
3 }
```

Listing 1.2: Dereferencing the null pointer

```
1  public void bar(){
2    //dereference the null pointer
3    mayBeNull(0).toString();
4  }
5  public Object mayBeNull(int i){
6    Object a=null;
7    if(i>0){
8      a=new Object();
9      return a;
10   }
11   return a;
12 }
```

Generally, static analysis reasons about the relationship between different models, based on a program's semantics, to predict program behaviours. Those behaviours are represented by calling relationships among methods, which can be modelled using a directed graph. We refer to such graphs as **call graphs**. Considering the code in Listing 1.1 for example, method `foo()` is calling itself, which forms a circular pattern (`foo()` → `foo()`). This indicates a potential infinite recursive loop. Static analysis also considers all possible relationships among program models: in the example shown in Listing 1.2, by looking at the method `mayBeNull()`, we can predict the null value by inspecting the state of variable `a` under two circumstances: (1) assume `i>0`, then `a` has been assigned a new object and (2) if `i<=0`, then `a` is never assigned. In general, static analysis assumes both can happen, but this is imprecise, as only one of the conditions can actually be satisfied. To model this precisely, we could use a dynamic technique, such as testing, to invoke the method `mayBeNull()` with a parameter `i=-1`. As a result,

the second circumstance (`i<=0`) is achieved.

1.1 Research Motivation

There is a recent increase in awareness that more research on the soundness of static analysis is needed. In 2015, a manifesto on the soundness of static analysis [129] was published, which outlines the unsoundness issue in static analysis. Many researchers have been working on pushing the boundaries of what static program analysis can detect. However, there is no single static analysis tool that can handle complex language features soundly in reality [129]. A sound static analysis is expected to model all possible program behaviours. To achieve this, an over-approximation strategy is used to estimate how programs behave. A precise analysis requires to fully model actual program behaviours. A typical strategy is to reduce "noise" in obtaining more precise results. Dynamic analysis, on the other hand, guarantees that only actual program behaviours are captured. We refer to models produced from dynamic analysis as soundness **oracles**, which can be used to assess statically modelled program behaviours. Moreover, there are parts of the program behaviour that static analysis may under-approximate (shown as **the gaps in static analysis** in the Figure 1.1), but that dynamic analysis is able to capture. The goal of the thesis is to identify the gaps in static analysis by using **oracles** which are generated by means of dynamic analysis to assess static analysis. The term **recall** that is used later in this thesis represents quantitative measurement of the level of unsoundness with respect to a given oracle. Figure 1.1 also visually demonstrates that it is possible to have a part of the program behaviour that neither static nor dynamic analysis can model, due to the low quality of the oracle (e.g., low program coverage).

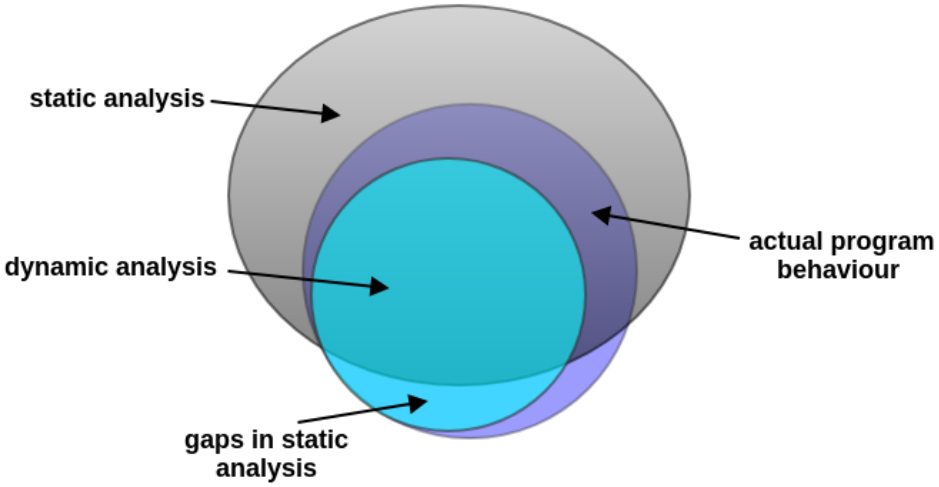


Figure 1.1: Precision and Soundness

Security analysis is one application of static program analysis. It motivates this research, since Java security issues have increased over the years as the language and its usage have evolved. Unsound analysis could miss a security breach, which may lead to serious problems. The soundness of static analysis therefore is desirable, in order to find as many security holes as possible. However, the presence of dynamic language features in Java makes it hard for static analysis to provide such sound analysis. For instance, features like *reflection*, *serialisation*, *dynamic class loading* and the use of *native libraries* can all be potential sources of unsoundness.

1.2 Research Objectives and Questions

The main objectives of this thesis are:

1. Investigate the impact of dynamic language features in Java on the soundness of static analysis and identify potential usage patterns of such features.
2. Explore an alternative way of oracle generation through mining software repositories.
3. Assess state-of-the-art Java static analysis tools and their ability in handling dynamic language features.
4. Quantify and measure the level of unsoundness in static analysis.

Specifically, we investigate the following five research questions:

RQ1 What are the sources of unsoundness in static analysis? (Results are reported in Chapter 4.)

RQ2 Are state-of-the-art static analysis tools able to successfully model dynamic language features in Java? (Results are reported in Chapter 4.)

RQ3 Can information obtained from stack traces improve the soundness of static analysis? (Results are reported in Chapter 5.)

RQ4 What is the level of recall achieved by state-of-the-art static analysis tools? (Results are reported in Chapter 6.)

RQ5 Which particular language features cause unsoundness in real-world programs? (Results are reported in Chapter 6.)

1.3 Overview of the Thesis

This thesis is divided into five main chapters, as follows:

- **Chapter 2** presents a detailed literature review as well as background of program analysis. This chapter covers related work in program analysis techniques and frameworks, including call graph construction, and static, dynamic and hybrid analysis techniques.
- **Chapter 3** describes the research methodology we followed in this thesis.
- **Chapter 4** categorises Java dynamic language features and presents a benchmark for static program analysis. Such a benchmark provides an intuitive way of investigating soundness in static analysis tools. The results are reported in the following publication:
  - **Sui, L.**, Dietrich, J., Emery, M., Rasheed, S., & Tahir, A. (2018). On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation. In Asian Symposium on Programming Languages and Systems (APLAS). Springer, Cham.
- **Chapter 5** presents the result of the using mining techniques to extract stack traces from on-line resources, to construct oracles which are used to assess statically built ones. The results are reported in the following publications:
  - Dietrich, J., **Sui, L.**, Rasheed, S., & Tahir, A. (2017). On the construction of soundness oracles. In the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP), ACM.
  - **Sui, L.**, Dietrich, J., & Tahir, A. (2017). On the use of mined stack traces to improve the soundness of statically constructed call graphs. In 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE.

- Chapter 4 and Chapter 5 discuss the notion of the soundness of static analysis in a binary form: either the analysis is able to capture the calling relationship (sound) or not (unsound). In **Chapter 6**, we report on a large empirical study that we conducted to measure the level of unsoundness (recall) in real-world programs. The results are published in the following publications:
  - Dietrich, J., Schole, H., **Sui, L.**, & Tempero, E. (2017). XCorpus—An executable Corpus of Java Programs. Journal of Object Technology, 16(4).
  - **Sui, L.**, Dietrich, J., Tahir, A., & Fourtounis, G. (2020). On the Recall of Static Call Graph Construction in Practice. In the 42nd International Conference on Software Engineering (ICSE). ACM.
- **Chapter 7** presents the main conclusions from this work and outlines a number of future work directions.

Chapter 2

Literature Review

2.1 Introduction

Understanding how programs behave is an essential part of software engineering. With the increase in programs' size and complexity, there is a growing need for automated approaches to analyse such large programs. Program analysis offers an automated mechanism to predict a program's behaviour at either compile-time or run-time [147, sect 1.1]. In general form, program analysis falls into two main categories: **static** and **dynamic** analysis. Static program analysis is usually performed on programs without running them. There are various static analysis techniques that are widely used including: points-to, control flow and data flow analysis. On the other hand, dynamic program analysis samples program behaviour at run-time by means of execution. Typical dynamic program analysis techniques include: profiling, monitoring, testing and program slicing. Hybrid analysis capitalises on the advantages of both static and dynamic analyses by focusing on the combination of both techniques. It is typically used in malware detection.

Program analysis techniques are widely used to validate the quality of software systems. Applications are extended to security analysis. Particularly, static analysis can identify security issues early in the development cycle [50, 51, 136]. Program analysis has also been used for compiler optimisation. For instance, Java Just-In-Time compiler (JIT) [145] compiles byte code to native code at run-time to improve the Java virtual machine (JVM)'s performance. During execution, the JIT compiler optimises methods that are often invoked by inlining them. This can save a great amount of computing resources. Another usage is dead code elimination [110, 89]. The compiler can perform data flow analysis to remove the parts of the source code that are either unreachable or do not impact the output. Other examples of program analysis usage are complexity analysis, anti-pattern/code smell detection and performance evaluation.

This chapter provides a detailed overview of program analysis techniques, with a

focus on static analysis, and discusses advantages and limitations (in terms of precision and recall) of static analysis. The remainder of the chapter is structured as follows: static analysis algorithms and applications are presented in Section 2.2.1, followed by a discussion of dynamic analysis techniques in Section 2.2.2, and finally, Section 2.3 discusses the gaps in static analysis, alongside a number of related works that deal with filling these gaps.

2.2 Background

2.2.1 Static Analysis

In many applications, static analysis is used for client analysis for the purpose of quality control at an early stage of the development. It usually takes source code or compiled code (i.e., byte code in Java) as input, and then applies various analysis techniques to model the program's behaviours. The program behaviours can be represented by a points-to set or a call graph. A points-to set is a collection of relationships between pointers and a call graph indicates calling relationships among methods. More details of points-to and call graph analyses are given in following sections.

2.2.1.1 Points-to Analysis

Points-to analysis is a foundation of static analysis as it provides information about the relationships between pointers. The points-to set contains heap references of created instances (i.e., objects). A heap reference is a memory location which indicates where an object is located. The analysis of the heap shows which variables may point to which objects. There are three ways to describe a pointer operation: referencing, dereferencing and aliasing.

Listing 2.1: Pointer Operation in Java

```
1  class Foo {
2
3   String bar;
4
5   public static void main(String[] args){
6     Foo foo=new Foo();
7     foo.bar;
8     Foo foo2=foo;
9   }
10 }
```

Referencing refers to a variable being created and assigned to a specific address (refers to allocation site in Java). The statement at line 6 in Listing 2.1 indicates that

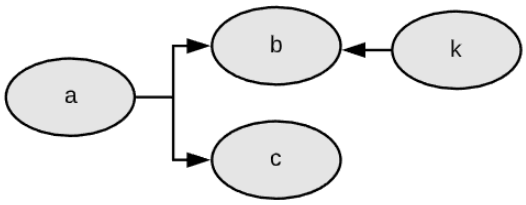
variable `foo` is assigned to (references) a new object `Foo`. The process for a variable accessing an allocation site is called dereferencing. Line 7 shows the variable `foo` being used to access the field `bar`. At line 8, the variable `foo` is copied to another variable `foo2`. We refer to this process as aliasing.

A points-to set can form a directed graph where the vertices are variables/objects and edges indicate points-to relationships. In the most simple form, we can assume all objects "point to" each other and the result of having such a points-to set is useless. There are two foundational algorithms that can help to find a more precise points-to set: Steensgaard's and Andersen's algorithms.

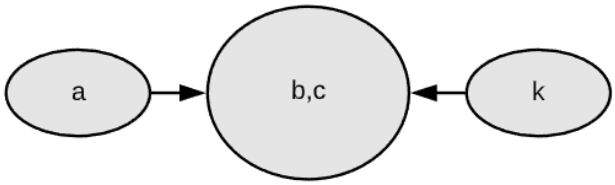
- Steensgaard's points-to analysis [174] performs in almost linear time $O(n\alpha(n,n))$ where α is the inverse Ackermann's function [20] and n is the size of the input program. The algorithm can be summarised as finding a union of two points-to sets that contain possible allocation sites of a variable. Considering the aliasing example which is shown in Listing 2.1, the points-to set would be:
  $p(foo2) \cup p(foo)$. The downside for using such equality constraints is that the merged allocation sites are represented in either direction, therefore the result is imprecise.

- Andersen's points-to analysis [31] delivers a relatively more precise analysis than Steensgaard's analysis. The difference is Andersen's analysis computes a transitive closure to propagate points-to relations. However this entails a compromise in performance, as any transitive closure computation requires cubic running time $O(n^3)$.

To demonstrate the differences between the two approaches, consider the set of points-to relations: $p(a) = \{b, c\}$ and $p(k) = \{b\}$. Figure 2.1(a) demonstrates the models produced by Andersen's analysis which is the correct representation. Steensgaard's analysis yields an incorrect relation where `k` also points to `c` (shown in Figure 2.1(b)). Andersen's analysis provides a greater level of precision with a reduction in performance. Collapsing strongly connected components is an important optimization that can be applied in the case of Andersen's analysis [92]. All strongly connected components share the same points-to relation, so therefore can be reduced to one representation (see Figure 2.2).



(a) Andersen's static analysis



(b) Steensgaard's static analysis

Figure 2.1: Andersen vs Steensgaard analysis

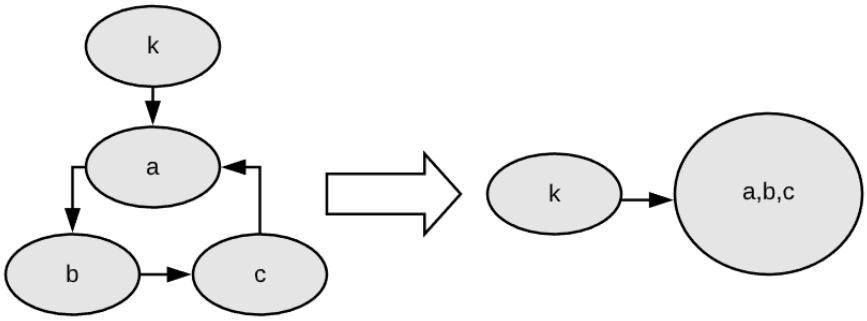


Figure 2.2: Cycle elimination

To further improve the precision of points-to analysis, we have to consider sensitivity in static analysis. There are two major factors that contribute to a precise analysis: program flow and execution context. In the following sections, we explore sensitivity analysis techniques in more detail.

2.2.1.2 Sensitivity in Static Analysis

Flow-sensitivity [184, 49] emphasises the program execution order. Considering the example provided in Figure 2.3, the code on the left side is evaluated top-down. A flow insensitive analysis computes a conservative prediction and yields an imprecise points-to set where `a` may point to `i` and `j` at the same time. However, if an analysis is flow

sensitive, we can observe that, at some point, `a` no longer points to `i`. To achieve flow-sensitivity, the program needs to be transformed to Static Single Assignment (SSA) form [56]. This is an intermediate representation (IR) of assignment which ensures each variable is assigned exactly once. In this case, the code presented in Figure 2.3 can be transformed by renaming variable `a` to `a_1` and `a_2` to represent its two stages (shown in Figure 2.4). IR can be generated using an Abstract Syntax Tree (AST) [102]. It represents the program syntax in a tree structure. Figure 2.5 shows an AST for a simple arithmetic expression: `3*5+1`.

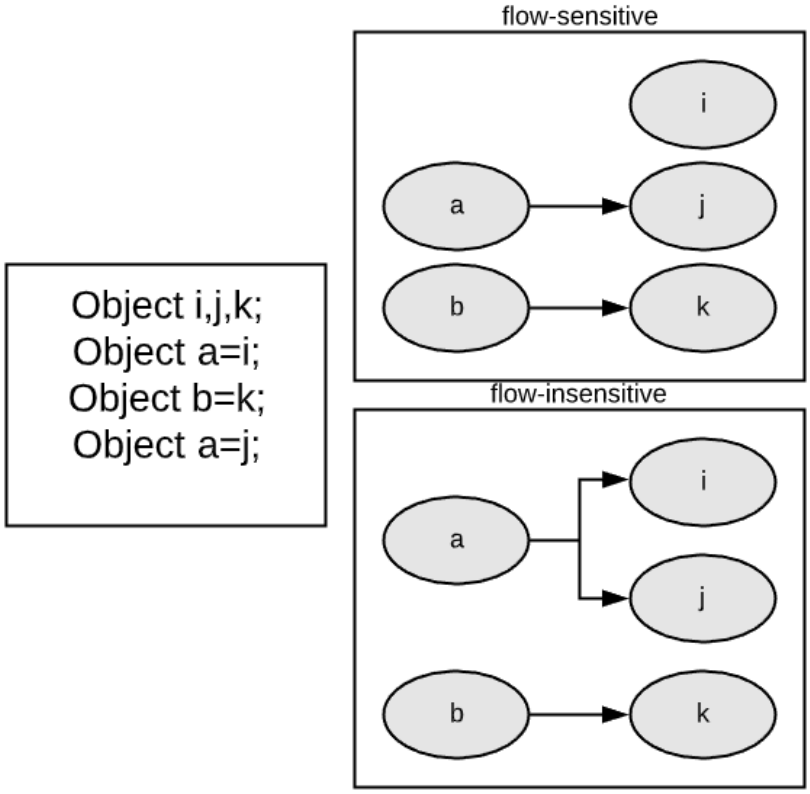


Figure 2.3: Flow-sensitivity

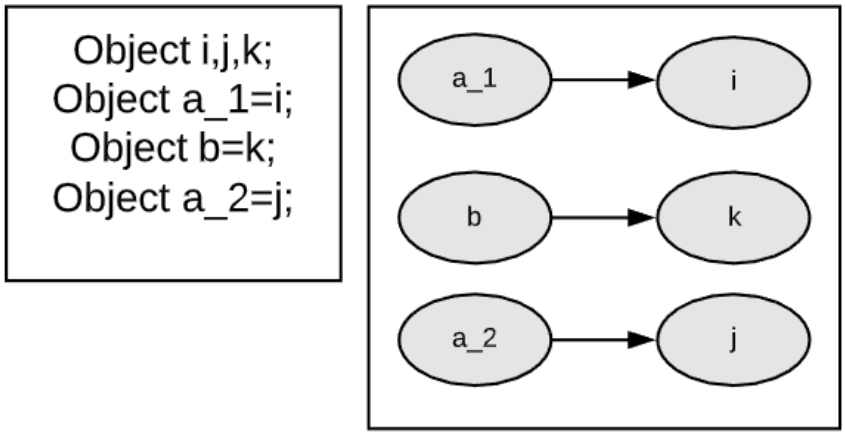


Figure 2.4: Static single assignment (SSA) form

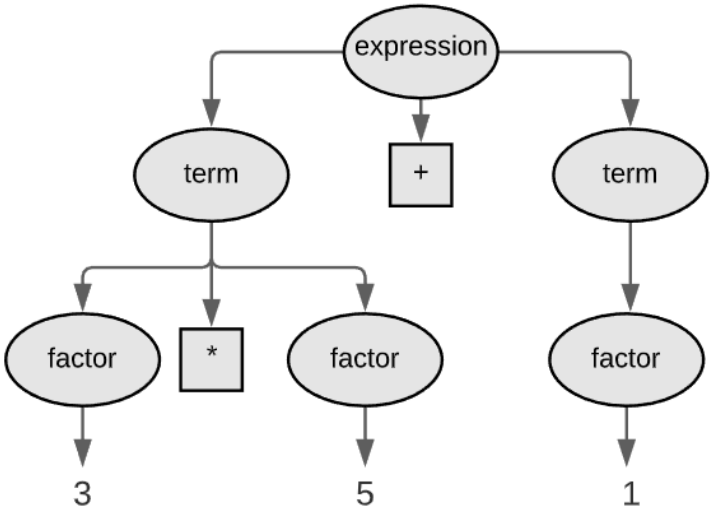


Figure 2.5: Abstract Syntax Tree (AST)

Data flow analysis was first introduced by Kildall [107] in 1973. It is a flow-sensitive analysis that is used to gather information about a set of possible values in a given program at various points. The notion of control flow [26] is applied here. The control flow indicates the program state and is represented in a graph structure. Figure 2.6 demonstrates a control flow graph of the code shown in Listing 2.2. If we set up two observation points A (at line 2), B (at line 6), the possible value set for variable `a` with respect to point A is [0] and for B is $[0, 1]$. Note that variable `c` is not alive at the observation point B as it is never used.

Listing 2.2: Forward Data Flow analysis

```
1 int a=0;
2 int b=1;//Observation Point A
3 if(b>0){
4  a=1;
5 }
6 int c=a*b;//Observation Point B
```

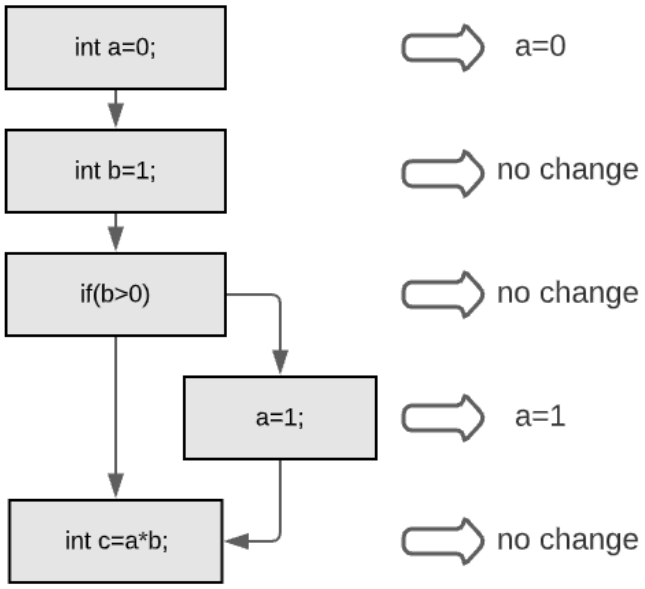


Figure 2.6: Control flow graph

The example analysis provided in Figure 2.6 is usually referred as forward data flow analysis [105]. The analysis is conducted along the direction of execution. On the other hand, backward data flow analysis [105] requires to propagate in a reverse manner. One of the classic static analysis problems is the live-variables problem [93]. It describes the problem that arises when using static analysis techniques to calculate the liveness of variables for a given program state. Backward data flow analysis does exactly this: an application of backward data flow analysis is compiler optimisation. During the compilation phase, the compiler can remove some unnecessary variable assignments and initialisations to avoid possible additional memory access. In the example shown in Listing 2.3, the analysis will try to decide which statements are unnecessary for the final state of the program (which is the statement `do(b);`). The statement at line 5 (`k=3;`) obviously cannot have an impact on variable `b` therefore can be eliminated.

Listing 2.3: Backward Data Flow analysis

```
1 int b=1;
2 if(k>0){
3   b=k;
4   if(k>2){
5     k=3;
6   }
7 }
8
9 do(b);
```

In contrast to flow analysis, context analysis focuses on execution context, especially on calling context. Context sensitivity [171, 166, 138] refers to differentiating between pointers in given contexts, such as call site and object allocation.

A call site is a method location where the method is invoked. As Figure 2.7 shows, a context insensitive analysis does not distinguish between the different contexts where the method `id()` is invoked. On the other hand, call site sensitivity requires the analysis to model the correct calling context. In this case, method `id()` belongs to two different call sites: `bar()` and `foo()`.

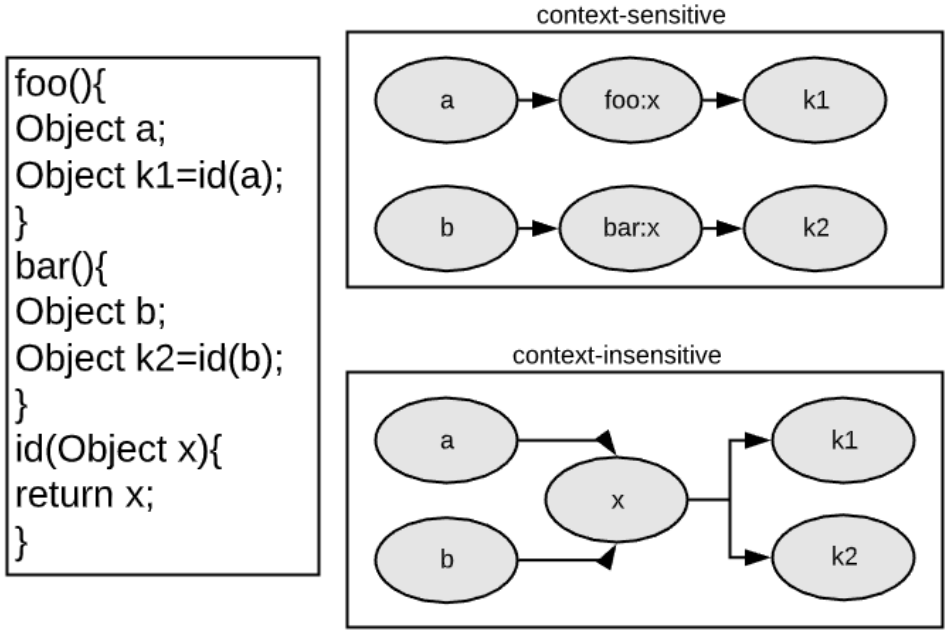


Figure 2.7: Callsite-sensitivity

If an analysis is sensitive to object allocation, it can distinguish between different receivers for various object allocation sites [171]. A typical example is shown in Listing 2.4, there are two allocation sites (marked as “alloc 1” and “alloc 2”) and they are not identical as each allocation references to a unique address in memory. Using an object

insensitive analysis, the two allocations could be merged. Therefore, object sensitivity analysis should figure out that the receiver `list` actually points to different allocation sites.

Listing 2.4: Object-Sensitivity

```
1 class Foo {
2
3   List list;
4
5   void bar() {
6     list.add(new Object()); // alloc 1
7     list.add(new Object()); // alloc 2
8   }
9 }
```

Yong et al. [187] firstly discussed a field-sensitive analysis to boost precision in obtaining points-to sets. Similar to object-sensitive analysis, field-sensitive analysis needs to consider a field access for an object. Andersen's analysis [31] is a field-insensitive analysis as it does not track object reference fields. Listing 2.5 demonstrates where an object is instantiated and assigned to a variable: `foo = new Foo()`, and the field access: `foo.a` needs to be in the points-to set. Field-sensitive analysis can provide a more precise modelling but the computation of each field can be extremely expensive [154].

Listing 2.5: Field-Sensitivity

```
1  class Foo{
2
3    String a;
4  }
5
6  class Main{
7
8    public static void main(Sting[] args){
9      Foo foo =new Foo();
10   }
11 }
```

The program calling context can be further categorised as either intraprocedural or interprocedural. Intraprocedural analysis is an analysis of a single procedure, whereas interprocedural analysis focuses on multiple procedures. Consider Listing 2.6, where we have three classes `A`, `B` and `Main` and both class `A` and class `B` contain the method `foo()`. An interprocedural analysis needs to decide which of the two `foo()` methods is being called – the variable `a` (at line 18 in Listing 2.6) points to the heap allocation of the object `A`, which indicates that the method `A#foo()` is being invoked.

Listing 2.6: Interprocedural analysis

```
1  class A{
2
3   foo(){
4    //calling other methods
5   }
6  }
7
8  class B{
9
10  foo(){
11   //calling other methods
12  }
13 }
14
15 class Main{
16
17   public static void main(Sting[] args){
18     A a= new A();
19     a.foo();
20   }
21 }
```

Listing 2.7: Intraprocedural analysis

```
1  class A{
2
3   foo(){
4     bar();
5   }
6
7   bar(){
8   }
9  }
```

Intraprocedural analysis studies the calling relationship within a single procedure. Listing 2.7 illustrates a calling relationship where the method `A#foo()` invokes the method `A#bar()`. We often store this calling relationship in a graph representation. A call graph is an important representation of program behaviours as many analysis algorithms build upon this model. Call graph construction is discussed in Section 2.2.1.3.

There are number of related studies in sensitivity analysis: An early work for flow analysis [104] studied the live-variables problem. Later Graham et al. [82] proposed a fast algorithm that runs in linear time. Distinctive work done by Allen [27] introduced control-flow to data flow analysis. [21] used the cartesian product of the types of arguments to improve precision. Interprocedural analysis is applied for pointer analysis

in [94, 68, 94] to elevate precision. The study [139] implemented a framework that uses object sensitive analysis. Wang et al. [181] proposed an algorithm that improves precision for constraint-based type inference. Reps et al. [159] managed to perform interprocedural data flow analysis in polynomial time. *Giga* [59] aims to improve the performance of computing point-to sets. It used a transitive closure data structure to deal with field-sensitive points-to analysis. Some noted works done for field-sensitivity based on constraint pointer analysis are: [163], [185] and *Spark* [119] (*Spark* is used in the experiment described in Section 4.2.2).

2.2.1.3 Call Graph Construction

The call graph is a directed graph where a vertex represents a call site and an edge represents a calling relationship. In fact, a call graph is a set of interprocedural calling relationships among procedures [164, 63]. Call graph analysis and points-to analysis are mutually dependent as a method invocation needs to know which object the method operates on and objects are connected via method invocations. A precise call graph can improve the precision of points-to sets and an accurate points-to set helps call sites to resolve receivers. This is also known as call graph construction on-the-fly.

There is a large body of research into call graph construction algorithms, mainly differing in achieving different trade-offs between precision and performance. Tip and Palsberg [179] conducted a large study on comparing the main approaches, such as Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA). The following list presents a number of algorithms and how they model the code example provided in Listing 2.8. This code example is constructed by using run-time polymorphism which determines method calls at run-time. In line 30, the super class reference variable `a` can refer to subclass object `X`. The challenge of it is that the method `a#foo()` is not resolved at compile-time.

Listing 2.8: Run-time polymorphism

```
1  abstract class A {
2
3    public abstract void foo();
4  }
5
6  class B extends A {
7
8    public void foo() {}
9  }
10
11 class X extends A {
12
13   public void foo() {}
14 }
15
16 class Y extends A {
17
18   public void foo() {}
19 }
20
21 class Z {
22
23   public void foo() {}
24 }
25
26 public class Main {
27
28   public static void main(String[] args) {
29     A a = new X();
30     a.foo();
31   }
32   public static void neverused(){
33     A a= new Y();
34     a.foo();
35   }
36 }
```

- Class Hierarchy Analysis (CHA) [58, 88] is a classic call graph algorithm that takes class hierarchy information into account. It assumes that the type of a receiver object (at run-time) is possibly any subtype of the declared type of the receiver object at the call site. CHA is able to eliminate method *foo()* in class `Z` because it is not a subclass of `A`. The call graph produced: `Main#main()`→`X#foo()`, `Main#main()`→`Y#foo()`, `Main#main()`→`B#foo()`. Note that Class Hierarchy Analysis is not a context nor a flow sensitive analysis, therefore imprecision is expected.

- Rapid Type Analysis (RTA) [35] is more precise. Meaning RTA produces a sub-graph of the graph produced by CHA. It analyses object types that are referenced. It produces the following call graph: `Main#main()`→`X#foo()`, `Main#main()`→`Y#foo()`. The instance of `B` is not created, therefore has been ignored.

- Variable Type Analysis (VTA) [175]. Unlike RTA, VTA analyses the allocation type of a variable. Method `foo()` can not be applied to `Y` because the type does not match the allocation site in `main()`.

- Control Flow Analysis of order k (k-CFA) [169]. The parameter k in k-CFA determines the context-sensitivity of the analysis. Roughly, higher context-sensitivity leads to better precision, at the expense of performance. For instance, 1- and 2-CFA analysis. These two analyses are considered heavy in performance [137, 44, 120]. Take 2-CFA for example, only the edge `Main#main()`→`X#foo()` is identified which is confirmed to a context-sensitive analysis.

Murphy et al. [142] presented one of the earlier empirical studies in call graph construction, which focused on comparing the results of applying 9 static analysis tools (including tools like GNU cflow) for extracting call graphs from three `C` programs. Lhoták [118] proposed tooling and an interchange format to represent and compare call graphs produced by different tools. *Judge* [157] builds upon [158], and also contains a case-study experiment on *xalan* in order to assess the recall of the static call graphs constructed by several static analysis tools. Karim and Lhoták studied the construction of call graphs for the application part of programs [24]. They used a methodology to assess the statically constructed call graphs against recorded program executions.

2.2.1.4 Static Analysis Tools

Static analysis tools are widely used in both industry and the research community. Tools like *SpotBugs*[1], *Checker Framework*, *Infer* and *PMD* focus on detecting a variety of bugs (including security violations), bad patterns and code smells [76]. They are all considered to be lightweight tools as they are often required to respond quickly to bugs during development. Other tools like *Soot*, *WALA* and *Doop* provide whole-program analysis and therefore are more suitable for research projects. They include various deep analyses such as sensitivity and data flow analysis to serve different needs. Some of these features are considered very expensive in term of computation. They also provide basic features for program analysis, such as computing call graphs and points-to set. Below is a brief discussion of the different known and widely used tools for Java.

[1] formally known as FindBugs

- *SpotBugs*[2] is the successor of *FindBugs* which was first introduced in 2006 [98]. The tool uses byte code analysis techniques to detect potential program defects (such as null pointer exceptions). It also supports identifying other violation patterns such as security violations (i.e., empty database password) and performance issues (i.e., explicit garbage collection). *SpotBugs* is available as a plug-in for popular IDEs such as Eclipse and IntelliJ for easy developer access. There are some famous *SpotBugs* users: GlassFish (Java EE framework), Sat4j (the boolean satisfaction and optimization library in Java) and Java Server Faces (Java-based web application framework).

- *Checker framework*[3] is developed by the MIT Program Analysis Group. It provides 24 different checkers (such as nullness checker, initialization checker, tainting checker for security issues) [152]. It takes advantages of annotation to describe additional information about Java types. For instance, annotating with `@NonNull` indicates the field/return value should not be null. *Checker framework* provides support for Android. It also can be configured via a build system such as Ant, Maven and Gradle.

- *Infer*[4] is developed and distributed by Facebook. It is a multi-language static analysis tool. *Infer* provides checks for null pointer exception, resource leaks, coding conventions for C++, C, iOS/Objective-C and Java/Android [46]. Bugs can be detected and presented to developers using a warning message. Many software companies, including Facebook, Amazon, Spotify, Uber, WhatsApps, Instagram and Mozilla, are currently using *Infer* in their development pipeline for early software defect detection [62].

- *PMD*[5] is a static code analyser that not only provides rules for Java but also supports JSP, Maven POM and XML. Developers are able to extend those rule sets by adding their *PMD* rules. Writing rules can be done by using either XPath query (defining the rule set directly in XML) or Java visitor (extending `AbstractRule` class). Most modern build systems, such as Ant, Maven and Gradle, have support for *PMD*.

- *Soot* [112] is a Java optimization framework which can provide call graph construction and points-to analysis. *Tamiflex* is an extension to *Soot* that complements *Soot*'s reflection support in a dynamic approach.

[2] `https://spotbugs.github.io/`, accessed 27 July 2020
[3] `https://checkerframework.org/`, accessed 27 July 2020
[4] `https://fbinfer.com/`, accessed 27 July 2020
[5] `https://pmd.github.io/`, accessed 27 July 2020

- *WALA* [17] is a tool developed by IBM that supports Java and JavaScript. It is also capable of providing call graph construction and points-to analysis. *WALA* provides support for reflection analysis.
- *Doop* [44] is based on Datalog which is a declarative logic programming language. It provides a fast pointer analysis as well as call graph construction.
- *Souffle* [103] provides high-performance execution models by compiling Datalog into optimised C++ code.

2.2.2 Dynamic Analysis

Static program analysis predicts program behaviours and captures **what could potentially happen during program execution.** Dynamic program analysis, on the other hand, reflects the actual program behaviour. It can provide a view of **what actually happened during a particular execution scenario**. The code shown in Listing 2.9 gives an example of a program with two possible execution scenarios. Depending on the input, the different methods might get called (either `foo()` or `bar()`). A static analysis will decide that both scenarios are **possible** (we refer to this as an over-approximation strategy). Dynamic analysis requires the program to run with an actual input (i.e., executing the program with **"`foo`"** as an input will lead to **exactly one** edge in the call tree: `main()`→`foo()`).

Listing 2.9: Strength of Dynamic analysis

```
1  class Main{
2
3    public static void main(String[] args){
4      if(args[0].equals("foo")){
5        foo();
6      }
7      if(args[0].equals("bar")){
8        bar();
9      }
10   }
11
12   public static void foo(){}
13
14   public static void bar(){}
15 }
```

Dynamic analysis has been widely used in debugging, software testing and program profiling. Typical dynamic techniques involve code instrumentation, symbolic execution and program slicing which are explained further in the following sections.

2.2.2.1 Code Instrumentation

Code instrumentation is the fundamental technique in dynamic analysis. Instrumentation allows code to be modified to facilitate external observers which can be used to record program behviours. Java code instrumentation can be done for two aspects: source code and byte code. Listing 2.10 demonstrates the instrumented source code provided in Listing 2.9. After running such a program, the print function will inform which input has been taken and which method has been correspondingly called.

Listing 2.10: Source code instrumentation

```
1  class Main{
2
3    public static void main(String[] args){
4      if(args[0].equals("foo")){
5        //added code
6        System.out.println("the input is foo");
7        foo();
8        //added code
9        System.out.println("foo() has been executed");
10     }
11     if(args[0].equals("bar")){
12       //added code
13       System.out.println("the input is bar");
14       bar();
15       //added code
16       System.out.println("bar() has been executed");
17     }
18   }
19
20   public static void foo(){}
21
22   public static void bar(){}
23 }
```

JVM provides a set of instructions which usually are referred to as byte code. It is the code that is compiled from Java source code and executed by the JVM. There are different types of instructions in Java byte code.

- Object creation. i.e., The `new` instruction allocates a new instance.
- Arithmetic operation. i.e., The `iadd` instruction adds up two integers.
- Reference operation i.e., The `aload` instruction loads reference onto the stack from a local variable.
- Control transfer. i.e., Jumping to another instruction uses the `goto` instruction.

- Method invocation. i.e., The `invokestatic` instruction is used to invoke a static method.

Byte code instrumentation can be done statically as well as dynamically. The `.class` files can be directly modified and later executed. Java also allows to modify byte code at run-time – Java agent [7] is the core Java feature introduced in Java 1.5. It hooks a `premain()` method to register a class transformer before the JVM loads the actual class. Listing 2.11 demonstrates a list of byte code instructions compiled from Listing 2.10. The injected statement shown at line 6 in Listing 2.10 is represented by a group of instructions: `getstatic`, `ldc` and `invokevirtual`.

There are a number of instrumentation frameworks available for Java, which are presented below:

- *Javassist*[6] provides a high level interpretation of byte code instrumentation. Byte code can be added in a form of source code and compiled at run-time. The instrumentation is also performed at run-time through a provided class loader [52].
- *ASM*[7] is a byte code manipulation framework that employs the visitor pattern [78] to traverse different types of statements and expressions. It is focused on simplicity of use and performance.
- *SUN* compiler library[8] also uses the visitor pattern to present a program's AST, but this library is not designed to rebuild or modify the AST.
- *JavaParser*[9] is a lightweight source code instrumentation library. It provides an AST parser as well as allowing to modify or create an AST from scratch.
- *AspectJ*[10] is an implementation for aspect-oriented programming (AOP) [106] for Java. AOP allows to add new behaviours without modifying the code in order to retain modularity.
- *DiSL*[11] is a domain-specific language for Java byte code instrumentation [132].

[6] `https://www.javassist.org/`, accessed: 22 June 2020
[7] `http://asm.ow2.org/`, accessed: 22 June 2020
[8] `https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/com/sun/source/util/package-summary.html`, accessed: 22 June 2020
[9] `https://github.com/javaparser/javaparser`, accessed: 22 June 2020
[10] `https://github.com/eclipse/org.aspectj`, accessed: 22 June 2020
[11] `https://disl.ow2.org/view/Main/`, accessed: 22 June 2020

Listing 2.11: Byte code instrumentation

```
1  class Main {
2    Main();
3    Code:
4      0: aload_0
5      1: invokespecial #1//java/lang/Object."<init>":()V
6      4: return
7
8    public static void main(java.lang.String[]);
9    Code:
10     0: aload_0
11     1: iconst_0
12     2: aaload
13     3: ldc           #2//Load the String:foo
14     5: invokevirtual #3//java/lang/String.equals:(Ljava/lang/Object;)Z
15     8: ifeq          30
16    11: getstatic     #4//injected print statement
17    14: ldc           #5//Load the String:the input is foo
18    16: invokevirtual #6//injected print statement
19    19: invokestatic  #7//invoke foo:()V
20    22: getstatic     #4
21    25: ldc           #8
22    27: invokevirtual #6
23    30: aload_0
24    31: iconst_0
25    32: aaload
26    33: ldc           #9//Load the String:bar
27    35: invokevirtual #3//java/lang/String.equals:(Ljava/lang/Object;)Z
28    38: ifeq          60
29    41: getstatic     #4//injected print statement
30    44: ldc           #10//Load the String:the input is bar
31    46: invokevirtual #6 //injected print statement
32    49: invokestatic  #11//injected print statement
33    52: getstatic     #4 //invoke bar:()V
34    55: ldc           #12
35    57: invokevirtual #6
36    60: return
37 }
```

Run-time profiling is one application where code instrumentation is applied. A profiling tool, such as Java VisualVM[12], provides a visual interface to inspect program execution at run-time. For instance, heap dump contains allocated objects in the memory. Heap dumps with allocation traces can forge call graphs for the purpose of inter-procedural analysis. A thread dump continues stack traces for all threads. This

[12] `https://visualvm.github.io/`, accessed: 22 June 2020

information can be harvested from Java VisualVM as well (Figure 2.8). An unconventional approach is discussed in Section 5.4.2.2 which uses stack traces mined from software repositories to construct call graphs.

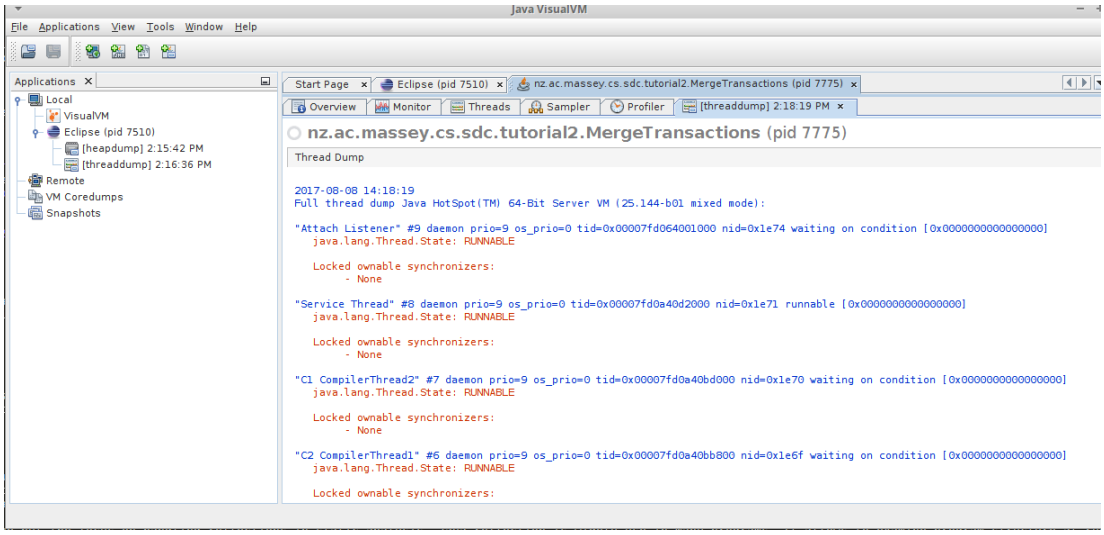


Figure 2.8: Thread stack by Java VisualVM

Code instrumentation is also used in fault location, such as "debugging", a term that describes the process of finding program defects. Modern debuggers allow to suspend the program to reveal its state. This process is achieved by code instrumentation. Java Platform Debugger Architecture (JPDA)[13] provides core functionality for the debugging process. It also allows to implement a custom debugger by access through the Java Debug Interface (JDI).

2.2.2.2 Program Slicing

Program slicing was first introduced by Weiser [182]. It is a method for reducing the program to a minimal state while maintaining its original behaviours. The reduced forms are referred as "program slices". Programs slices are generally used during debugging to reduce the size of program under analysis, thereby narrowing the source of errors. Originally it has been used as a static approach where data flow and control flow analysis are applied to construct program slices [183]. Listing 2.12 demonstrates a program that is to be sliced for the given variable `e`. Statements: `c=a+1` and `d=b+1` do not contribute to the variable `e` therefore are not included in the sliced program shown in Listing 2.13.

[13] `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html`, accessed: 22 June 2020

Listing 2.12: Before slicing

```
1 a=1;
2 b=2;
3 c=a+1;
4 d=b+1;
5 e=a+b;
```

Listing 2.13: After slicing

```
1 a=1;
2 b=2;
3 e=a+b;
```

Korel, B and Laski, J [111] developed the idea of dynamic program slicing in 1988. Unlike static slicing which is processed at compile-time, dynamic program slicing can construct slices for a known input at run-time which is convenient for specifying slices related to a particular execution. The work done by Agrawal et al. [22] extended dynamic slicing to relevant slicing which is used in incremental regression testing. Relevant slicing not only considers statements directly related to a variable, but also statements that could potentially be affected. Conditional slicing [47] produces program slices for a given set of execution paths. Hall et al. [91] proposed using union slicing to combine dynamic slicing algorithms for a set of test cases.

2.2.2.3 Testing

Testing can be also considered as a dynamic analysis method. A main goal of testing is to verify the outputs of the program (actual vs expected outcomes). Testing can not only improve software quality by writing test cases to locate faultiness, but also it has been proved to be a very efficient method in the area of project management [134, 36, 48, 99], such as in test-driven development (TDD), a software development methodology that uses test cases as requirements.

In general, three testing levels are considered: unit testing, integration testing, and system testing [42, Chapter 4]. Unit testing focuses on an individual module to be tested. Integration testing can test a particular functionality via a collection of unit tests. System testing requires complete testing of the whole system to meet user requirements.

Typical metrics used to measure the quality of tests include code coverage, lines of code (LOC) and cyclomatic complexity. Miller et al. [140] first introduced the code coverage method for systematic software testing. Nowadays, code coverage is widely used in software development. Well known coverage tools for Java include: *Jacoco* [96], *EMMA*[14], and *Atlassian Clover*[15]. Code coverage can be categorised as follows [143]:

- Statement coverage: the coverage of each program statement.
- Function coverage: the coverage of each method.

[14] `http://emma.sourceforge.net/`, accessed: 22 June 2020
[15] `https://www.atlassian.com/software/clover`, accessed: 22 June 2020

- Branch coverage: the coverage of each execution path.
- Condition coverage: reports the boolean outcome of each condition.

Lines of code (LOC) is used to indicate the size of a program. Some works [101, 70] argue that this metric provides weak support for software quality. McCabe and Thomas [135] developed cyclomatic complexity in 1963, It uses control flow graphs to represent program complexity. Each node indicates a program component, and directed edges link nodes based on decision points (i.e., a condition). For instance, if a program has no conditional statement then the cyclomatic complexity of such a program is 1.

Symbolic execution [108, 55] is a testing technique that uses symbolic input values instead of concrete values to evaluate program executions. Symbolic input values are used to execute a program symbolically to collect a symbolic constraint path. Consider the code provided in Listing 2.14. Let variable `x` have the value "λ". To reach line 3, the constraint must satisfy: λ `>` `6` and in order to reach the statement at line 8: λ `<=` `6`. The statement at line 5 will never be reached as the constraint: (λ `>` `6` $\wedge$ λ `<` `3`) is not satisfiable.

Listing 2.14: Symbolic execution

```
1  foo(x){
2    if(x>6) {
3      y=2;
4      if(x<3){
5        y=1;
6      }
7    }else{
8      y=0;
9    }
10 }
```

Mutation testing [33] is another testing technique. It modifies the test to change its behaviours to be different from the original version (the new version is referred as a "mutant"). In Listing 2.15, assuming `x = 1`, the original code can only cover the statement at line 4 , with a mutant demonstrated in Listing 2.16, the statement at line 2 can be covered as well.

Listing 2.15: Before mutating

```
1 if(x<0){
2   y=1;
3 }else{
4   y=0;
5 }
```

Listing 2.16: A mutant

```
1 if(x>0){ //mutation operator
2   y=1;
3 }else{
4   y=0;
5 }
```

Tests can be manually or automatically created. Manual testing is a process whereby developers construct test cases for programs by hand. This can be referred to as built-in tests. Generated tests are sets of tests that are produced programmatically. There are a number of Java test generation tools available. Different tools use different algorithms and techniques for test case generation. Some of the well-known tools that use testing techniques are listed below:

- *Evosuite* [77] is a whole-suite test generation framework. The goal is to produce test cases that can achieve a high branch coverage. *Evosuite* uses mutation testing to achieve certain coverage goals.
- *Randoop* [151] is an unit test generator for Java that can generate unit tests using feedback-directed random test generation [54, 150].
- *jCUTE*[16] is an automated test generation tool for Java programs. It uses symbolic execution and automatic constraint solving, together with randomised inputs to find many different execution paths for a given Java program.
- *Korat* [43] is a constraint-based generation tool. It automatically generates tests based on a given Java predicate (i.e., precondition) within a bound on the size of test inputs. Postcondition is then used to check the correctness of the test result.

2.3 Gaps in Static Analysis

In 2015, a manifesto [129] was published that raises a number of questions regarding the soundness of static analysis. Figure 1.1 in Section 1.1 illustrates a conceptual model of unsoundness issues (labeled as "gaps in static analysis") where modelling of program behaviours using static analysis is problematic. In the following sections, we are going to further explain the concepts of soundness/unsoundness and recall in detail, and then consider the gaps in existing static analysis work.

2.3.1 Soundness and Precision in Static Analysis

Rice's theorem [160] suggests: "*In a Turing machine, any non-trivial property of program behaviours is undecidable*". Thus, approximation of program behaviours is the best that can be done to approach decidable state. The assumption is that there is no perfect program analysis possible – we either over-approximate or under-approximate the actual program behaviour. When an over-approximation is applied, false program behaviours are expected to be included (i.e., **False Positive (FP)**). When an under-approximation is applied, some program behaviours are most likely neglected (i.e., **False Negatives**

[16] `https://github.com/osl/jcute`, accessed 30 August 2020

(FN)). We refer to program behaviour that can be correctly modelled as **True Positive (TP)**. We denote the set of behaviours predicted by static analysis as "SA" and the set of actual program behaviours as "APB". The set of TPs by a given static analysis can be described as follows:

$$TPs = SA \cap APB \tag{2.1}$$

The set of FNs is:

$$FNs = APB \setminus SA \tag{2.2}$$

The set of FPs is:

$$FPs = SA \setminus APB \tag{2.3}$$

Static program analysis must take into consideration both precision and soundness. A precise analysis aims to reduce the occurrence of FPs, while a sound analysis will aim to eliminate the occurrence of FNs (an unsound analysis will obtain FNs.) In a worst case scenario, we assume all program models are inter-related, therefore it is a sound but not a precise analysis. Modern static analyses aim to obtain both sound and precise analysis. However, in reality, this has been a key issue for static analysis as it has been a problem obtaining a sound analysis while maintaining precision [69]. To further quantify the term (un)soundness of static analysis, we borrow the concept of "recall" from the fields of data analysis and information retrieval. David L. Olson [149] explains recall and precision, where |TPs| is the number of true positives, |FNs| is the number of false negatives and |FPs| is the number of false positives.

$$recall = \frac{|TPs|}{|TPs| + |FNs|} \tag{2.4}$$

$$precision = \frac{|TPs|}{|TPs| + |FPs|} \tag{2.5}$$

To be more specific, in a given call graph produced by static analysis, the TPs can be seen as a set of calling relationships that are correctly modelled by static analysis. The FPs are a set of calling relationships that static analysis has incorrectly modelled. The FNs are a set of calling relationships that are overlooked by static analysis.

2.3.2 Dynamic language features in static analysis

More and more developers favour using dynamic languages due to their effectiveness and productivity [153]. Dynamically typed language such as Python, PHP, JavaScript are gaining popularity [14, 13, 12]. Statically typed languages generally enforce types and links at the compilation stage (i.e., Java, C and C++). They have also adopted the ideas of dynamic languages as a feature to enable flexibility during development. A typical example would be the use of reflection where new methods, variables or classes can be introduced at run-time. However, such dynamic features can cause problems for static analysis in the modelling of behaviours and, therefore, became the source of unsound analysis [129]. Many researchers have invested effort into improving the precision of static analysis (discussed in Section 2.2.1.2). The awareness of unsoundness, regarding the use of dynamic language features, has been addressed very recently [129]. The following sections present existing work on handling dynamic language features, such as reflection, invokedynamic instruction, and dynamic proxy. These approaches indeed claim that they can model such dynamic language features, but this thesis reveals that none of current static analysis tools are as sound as we expect. Besides, there are other dynamic language features that need to be handled, such as serialisation and the use of native libraries.

Reflection is one of the most widely used dynamic features in Java. The impact of reflection on static analysis has been a focal point of static analysis research [170, 129]. First introduced in LISP and Smalltalk [71, 173], reflection allows a program to dynamically create classes, fields and methods at run-time [72]. The classes, fields or methods information are mostly provided by string literals, which are not usually considered as part of program models. Therefore, tracking and reasoning about related string literals is crucial in reflection analysis. Several works have been presented with a focus on reflection analysis. Livshits et al. [130] uses points-to analysis to investigate reflective call sites by associating objects with them. [130] tracks strings that are supplied by reflection and interpreted as class names. Li et al. [123] have proposed *elf* to improve the effectiveness of handling reflection by adding additional rules to the *Doop* framework [44]. Smaragdakis et al. [170] adds substring and string flow analysis in *Doop* to improve reflection analysis for Java. *Tamiflex* uses dynamic analysis to support handling reflection analysis for *Soot*. *Tamiflex* runs a dynamic pre-analysis

by instrumenting the byte codes, and then logs all reflective calls and feeds this information into *Soot*. *WALA* employs a context-sensitivity policy to deal with reflections. *WALA* has built-in support for certain reflective methods such as `Class#forName()`, `Class#newInstance()`, and `Method#invoke()`. Landman et al. [114] have studied the challenges faced by state-of-the-art static analysers to model reflection in Java. They found that the parts of the reflection API that prove problematic for static analysers are widely used in practice. The authors used a lightweight static analysis based on detecting patterns in the abstract syntax tree (AST) of programs for their analysis.

The `invokedynamic` instruction is another dynamic language feature aimed at providing developers with more control over method dispatch, and is mainly used to compile lambda expressions defined in Java. Several studies have proposed static analysis support for modelling `invokedynamic` instructions. Bodden [39] provides a *Soot* extension that supports reading and rewriting `invokedynamic` byte codes. The *opal* static analyser also provides support for `invokedynamic` through replacing `invokedynamic` instructions using Java `LambdaMetaFactory` with a standard `invokestatic` instruction [5]. *WALA* also provides support for `invokedynamic` generated for Java 8 lambdas[17]. Like the different approaches to handle reflection, support for `invokedynamic` often does not address the language feature as such, but only particular usage patterns. In particular, the above-mentioned approaches assume that a certain bootstrap method is used to set up the invocation process. This works well as long as this is how the respective feature is used in real-world Java programs (e.g., making assumptions about the byte code emitted by the current Java compiler), but fails if byte code produced by non-Java or non standard Java compilers is analysed. This is a relevant problem as the JVM has become a polyglot platform. Support for `invokedynamic` has been recently added to *Doop* [74].

Dynamic proxy is a programming pattern that allows a dynamically created proxy object to act as a client object. Fourtounis et al. [73] have recently proposed the first analysis for dynamic proxies, based on the *Doop* framework. This analysis shows that there is a need for the mutually recursive handling of dynamic proxies and other object flows via regular operations (heap loads and stores) and reflective actions. Also, in order to be effective, static modelling of proxies needs full treatment of other program semantics, such as flow of string constants.

2.3.3 Hybrid analysis

Hybrid analysis aims to combine both static and dynamic analysis, in order to take advantage of both approaches. The aim is to combine both the relative soundness of

[17] `https://groups.google.com/forum/#!topic/wala-sourceforge-net/omsGtp_ow7I`, accessed 27 July 2020

static analysis and the accuracy of dynamic analysis [69, 144] in one single analysis. It involves using the results produced by static analysis to modify the code in order to perform dynamic checks. It can also be used in performing a dynamic pre-analysis to record executions, and then the information recorded can be fed into a static analysis. For instance, by modifying the original code or by creating a specification that can be used directly to augment a static analysis model. A typical usage would be generating additional facts for a datalog-based static analyser. Andreasen et al. [32] used a hybrid approach that combines soundness testing, blended analysis, and delta debugging for systematically guiding improvements of soundness and precision of TAJS – a static analyser for JavaScript[18]. Soundness testing is the process of comparing the analysis results obtained from a pure static analysis with the concrete states that are observed by a dynamic analysis, in order to observe unsoundness. Moser et al. [141] pointed out the limitation of static analysis especially in program security and suggested dynamic analysis is a necessary complement for static detection. Furthermore, program models obtained by dynamic analysis (i.e., log files) can be used to provide additional information for static analysis [57, 162]. Gupta et al. [90] proposed a hybrid approach to improve precision for program slicing. Dependence cache slicing [180, 176] is another hybrid approach where data dependencies are analysed statically and other dynamic information, such as invocations are harvested during execution [148]. Hybrid analysis is also proposed to analyse Android applications, such as *HybriDroid* [116].

In more recent years, hybrid analysis has become more popular when it comes to analysing dynamic features in Java. The work of Hirzel et al. on pointer analysis [95] combines both static analysis (analyse points-to set and propagating constraints) and dynamic analysis (generating new constraints when certain events occur). Bodden et al. [40] incorporated points-to analysis with flow-insensitive analysis to help to trace events that occur during program execution. The results suggest that this approach can significantly improve performance of run-time instrumentation. Bodden et al. [41] also developed *Tamiflex* which adopts a hybrid approach where a Java program is instrumented and method invocations, in particular reflective invocation, are recorded. The original code is then enriched with "unreflected" code. This approach has the advantage of being tool-agnostic. Grech et al. [83] have proposed *heapdl*. This tool is conceptually similar to *Tamiflex* but also uses heap snapshots to further improve recall. *Mirror* by Liu et al. [127] is a hybrid analysis specifically developed to resolve reflective call sites while maintaining precision.

[18] `https://www.javascript.com/`, accessed 27 July 2020

2.4 Summary

This chapter covers related work in program analysis techniques and frameworks. Both static and dynamic analysis have been discussed. Points-to sets and call graphs are two essential models for static analysis as they provide information about the relationships among program models. There are two fundamental points-to analyses: Steensgaard [174] analysis and Andersen [31] analysis. Each has its own merits. To combat imprecision in static analysis, sensitivity analysis is introduced to reduce false positives. Typical analyses include control flow and data flow analysis. We also present some classic algorithms in call graph construction such as CHA and RTA. Both industry focussed and research focussed tools have been discussed in Section 2.2.1.4.

Compared to static analysis, dynamic analysis delivers a sound analysis but struggles with completeness (i.e., missing execution branches due to low coverage). Here we listed techniques to harvest program behaviours at run-time, such as code instrumentation, program slicing and testing. Additionally, we discussed hybrid analysis, a combination of static analysis and dynamic analysis.

Chapter 3

Research Methodology

3.1 Introduction

The research presented in this thesis is empirical in nature. The conclusion we draw from the experiments presented in this thesis are based on observations and analysis of data collected from various open-source programs. The thesis follows an empirical analysis method. We conduct a number of experiments in order to answer the five research questions discussed in Section 1.2. Such an empirical approach has been widely followed in software engineering research [109, 186, 65].

Three main experiments have been conducted in this thesis: building a benchmark based on observations of dynamic language features in Java (Chapter 4), mining software repositories to harvest stack traces (Chapter 5) and quantifying the level of unsoundness on a set of real-world programs (Chapter 6). We explain our general research design in the following sections, but to improve readability, we discuss the specific methodology for each experiment in the relevant chapters.

3.2 Experimental Design

We first observed Java dynamic language features and then categorised those features based on usage patterns. We then conducted several empirical studies that aim to compare dynamically obtained program behaviours with statically built ones. Specifically, program behaviours can be represented by a calling relationship between methods, and modelled by the directed graphs drawn from these, known as **call graphs**. Call graph analysis is a fundamental analysis for code optimisations, program comprehension and many security analyses. We harvested call graphs from various sources: (1) stack traces mined from software repositories (2) the CVE database and (3) a set of real-world programs. We refer to these call graphs as oracles. These oracles are needed for quantitative measurements in terms of the proportion of program behaviours that are overlooked by

static analysis. All static analysis tools presented in this thesis (i.e., *Soot* [112], *Doop* [44] and *Wala* [17]) work at byte code level. There are a number of advantages for using byte code over source code, including (1) as an intermediate representation, it is easy to analyse (2) it is more stable than source code (byte code instructions have never been deprecated) and (3) it works with other program languages that also run on the JVM, such as *Groovy*, *Scala* and *Kotlin*.

3.2.1 Benchmark Construction

The manifesto of soundness in static analysis [129] highlights that the use of dynamic language features is a source of unsoundness in static analysis. We constructed a benchmark for the following categories of known dynamic language features in Java. These categories have also been used for the experiment presented in Chapter 6, which is conducted on a collection of real-world programs. Each feature contains a few usage patterns. Descriptions for each particular usage pattern are presented in Section 4.3.

- **Reflection** is a feature that allows to dynamically instantiate classes or to invoke methods.
- **Serialisation** is the process of converting (and storing) objects to a byte stream.
- **Dynamic class loading** allows to use custom class loaders.
- **Invokedynamic** is a Java byte code instruction used to control method dispatch.
- **Dynamic proxy** is a dynamically created object that serves as an intermediary object between a client and target objects.
- **Native code** is an implementation at the operating system level, such as C, C++ and assembly. It can be loaded by JVM to make it functional for Java programs. The callback from a native library triggers new invocations that must be modelled as well.

The benchmark consists of a number of JUnit[1] test cases. The benchmark has two parts: a harness and a simple program. The harness is responsible for exercising the program and serves as an entry point to each program. The simple program exhibits one particular Java dynamic language feature. We use the annotation `@Source` to mark the caller and `@Target` to mark the callee. We also include multiple callees to see which one can be modelled correctly. For instance, a correct callee is marked as `@Target(expectation = YES)`. To compute the static call graphs, we choose to use three well-known static analysis tools: *Soot* [112], *Doop* [44] and *Wala* [17] with various

[1] `https://junit.org/`, accessed 18 December 2020

analysis options (details are included in Section 4.2). The evaluation of these tools is done by looking at each combination of program and static analysis tool and a result state depending on the annotations found in the methods that are reachable. We refer to a pair of a caller and a callee that can be correctly reported by static analysis as a true positive ("TP"). We denote a set of methods predicted as reachable by static analysis as "PR" (predicted reachable) and a set of actual called methods as "ACM" (actual called methods). The following equation defines a set of TPs:

$$TPs = PR \cap ACM^2 \tag{3.1}$$

The set of FNs is:

$$FNs = ACM \setminus PR \tag{3.2}$$

The set of FPs is:

$$FPs = PR \setminus ACM \tag{3.3}$$

A result state can be one of the following: precision (TP), imprecision (FP), unsoundness (FN), or a state shows that the results of the static analysis are both unsound and imprecise (FN+FP).

3.2.2 Mining Stack Traces from Online Resources

We use a mining technique to acquire stack traces from well-known source code hosting and Q&A online communities: i.e., GitHub[3] and Stack Overflow[4]. Our mining technique involves crawling through the internet by sending HTTP requests to retrieve relevant pages from those sites. To prevent access being denied by the servers (i.e., a DDoS defense mechanism), a sleep timer is placed between each request. Both parsing stack traces from web pages and method pairs from stack traces are done by pattern matching (the regex expression is shown in Listing 5.8). A manual inspection approach is adopted for the purpose of verification. This is done by verifying the first 100 recorded stack traces against the original web page.

The static call graph is generated by *Doop*, and locating FNs is achieved by the following equation where "STM" is a set of methods extracted from stack traces and "PR" is a set of methods predicted as reachable by *Doop*:

$$FNs = STM \setminus PR \tag{3.4}$$

[2]The defination for TPs, FNs, FPs are the same as in Equation 2.1, 2.2 and 2.3 , but the names of the underlying sets have changed to fit this context.

[3]`https://github.com`, accessed 18 December 2020

[4]`https://stackoverflow.com`, accessed 18 December 2020

3.2.3 Quantifying Unsoundness in Static Analysis

The set of real-world programs is selected from our own dataset: the *XCorpus*, which is developed in our previous work leading to the work reported in this thesis [61]. The dataset provides an executable corpus for various Java programs. However, the *XCorpus* uses the JUnit framework which prevents static analysis from locating the entry point. We therefore adopt a systematic approach to rewrite program entry points. There are two sets of tests needed to be rewritten: (1) manually created tests and (2) automatically (programmatically) created tests. We use byte code instrumentation to inject an observer at run-time. As a result, the entire method stacks are recorded. These method stacks can then be used to assess the (un)soundness. The level of unsoundness is specified by the term "recall": the percentage of all methods in recorded method stacks to be reported in *Doop*. A set of recorded method invocations is denoted as a "CCT", a Context Call Tree. A set of statically modelled method invocations is denoted as an "SCG". Details are presented in Section 6.2.2. The following equation is used to measure the level of unsoundness in this experiment:

$$recall = \frac{|CCTs \cap SCGs|}{|CCTs|}_{5} \tag{3.5}$$

When building static call graphs, we also consider the impact, on recall, of different analysis strategies and setups, such as context-sensitive/context-insensitive, full/light-reflection support. We use descriptive statistics to compare the results and also use visualisation approaches (e.g., violin plots, sunburst graphs) to better compare the results of different analyses.

[5] The defination for recall is the same as in Equation 2.4, but the names of the set have changed to fit in this context.

Chapter 4

Benchmark Construction and Evaluation

4.1 Introduction

This chapter presents a benchmark comprised of a set of simple running programs that each exhibit one particular Java dynamic language feature and usage pattern. The purpose of this benchmark is to serve as a baseline for evaluating the soundness of static analysis tools with regard to modelling dynamic language features. Soundness requires that a static analyser models the entire program behaviour for all possible executions. The advantage of having a benchmark is that the programs are well designed and behaviours are easily modelled in the sense of viewing the source code. We explore several usage patterns based on these dynamic features. We argue that the word "patterns" can describe dynamic behaviours better than features, since many features could have more than one usage case being applicable. Take *reflection* for example, there are potentially two usage patterns: (1) target method name can be resolved via string concatenation that cannot be folded by the compiler. (2) target method name can be resolved from a file IO which is considered as an external resource.

The design goal of the benchmark is to be minimalistic and to use conventions-over-configurations to facilitate experiments. The programs are executable and are designed to have behaviour that is easy to observe. The construction of the benchmark focuses on call graph construction for Java programs. The call graphs are not obtained by executing the benchmark programs but are created manually, based on our complete understanding of the simple scenarios modelled.

The benchmark contains 34 programs from the following categories: reflection, serialization, unsafe, dynamic proxies, dynamic classloading, invokedynamic and Java Native Interface (JNI). Using this benchmark, we conducted an evaluation of how three state-of-the-art static analysers (*Soot*, *WALA* and *Doop*) handle these dynamic features.

In this chapter, we first demonstrate, in detail, dynamic language features in Java in order to construct the benchmark. We then present minimalistic example programs for each of these features (usage patterns). At the end, we discuss an experiment that is conducted with the benchmark to evaluate the capabilities (with regards to modelling those dynamic language features) of state-of-the-art static analysis frameworks.

4.2 Methodology

4.2.1 Benchmark Structure

The benchmark is organised as a Maven project using the standard project layout [2]. `src/main/java` contains benchmark source code and tests are placed under `src/test/java`. Resources are located at `src/main/resources`. The actual programs are organised in name spaces (packages) reflecting their category. As these programs are minimalistic, their behaviour is, in most cases, easy to understand for an experienced programmer by just looking at the source code. All programs have a `source()` method and one or more other methods, usually named `target(..)`. This refers to a calling relationship (from source method to target method).

Each program has an integrated oracle of expected program behaviour, encoded using standard Java annotations. Methods annotated with `@Source` are call graph sources: we consider the program behaviour triggered by the execution of those methods from an outside client. Methods annotated with `@Target` are methods that may or may not be invoked directly or indirectly from a call site in the method annotated with `@Source`. The expectation of whether a target method is to be invoked or not is encoded in the `@Target` annotation's `expectation` attribute that can be one of three values: `Expected.YES` – the method is expected to be invoked , `Expected.NO` – the method is expected not to be invoked, or `Expected.MAYBE` – exactly one of the methods with this annotation is expected to be invoked, but which one may depend on the JVM to be used. For each program, either exactly one method is annotated with `@Target (expectation = Expected.YES)`, or some methods are annotated with `@Target (expectation = Expected.MAYBE)`.

The benchmark contains a vanilla program that defines the base case: a single source method that has a call site where the target method is invoked using a plain `invokevirtual` instruction (invokes instance method based on the class). The annotated example is shown in Listing 4.1, this also illustrates the use of the oracle annotations.

Listing 4.1: Vanilla program source code

```
1  public class Vanilla {
2
3    public boolean TARGET = false;
4    public boolean TARGET2 = false;
5
6    @Source
7    public void source(){
8      target();
9    }
10
11   @Target(expectation = YES)
12   public void target(){
13     this.TARGET = true;
14   }
15
16   @Target(expectation = NO)
17   public void target(int o) {
18     this.TARGET2 = true;
19   }
20 }
```

In Listing 4.1, the target method changes the state of the object by setting the `TARGET` flag. The purpose of this feature is to make invocations easily observable, and to confirm actual program behaviour by means of executing the respective programs by running a simple client implemented as a JUnit test. Listing 4.2 shows the respective test for the vanilla program – we expect that after an invocation of `source()` by the test driver, `target()` will have been called after `source()` has returned , and we check this with an assertion check on the `TARGET` field. We also test for methods that should not be called, by checking that the value of the respective field remains `false`.

The main purpose of using the annotations is to facilitate the set up of experiments with static analysers. Since the annotations have a retention policy that makes them visible at run-time, the oracle to test static analysers can be easily inferred from the benchmark program. In particular, the annotations can be used to test for both soundness and precision. In this context, we measure the level of unsoundness as the number of actual calling relationships that are not reported. The missing calling relationships are referred to as False Negatives (FNs). The level of imprecision is the number of calling relationships that are mistakenly reported by static analysers. i.e., False Positives (FPs).

Listing 4.2: Vanilla test case

```
1  public class VanillaTest {
2
3    private Vanilla vanilla;
4    @Before
5    public void setUp() throws Exception {
6      vanilla = new Vanilla();
7      vanilla.source();
8    }
9
10   @Test
11   public void testTargetMethodBeenCalled() {
12     Assert.assertTrue(vanilla.TARGET);
13   }
14
15   @Test
16   public void testTarget2MethodHasNotBeenCalled() {
17     Assert.assertFalse(vanilla.TARGET2);
18   }
19 }
```

The experiments are set up as follows: for each benchmark program, we use a lightweight byte code analysis to extract the oracle from the predefined `@Target` annotations. Then the call graph is computed with the respective static analyser using the method annotated as `@Source` as the entry point, and the result is stored in PROBE call graph format [118]. Finally, using the static call graph, the FPs and FNs are computed with respect to the oracle, using the annotations as the ground truth.

4.2.2 Static Analysis Tools

We are particularly interested to see whether the benchmark examples were suitable to differentiate the capabilities of mainstream static analysis frameworks. We identified three state-of-the-art static analysis tools based on the following criteria:

1. Can provide a whole program analysis which fits our analysis scope. Tools like *SpotBugs*[1], *Checker framework*[2], Facebook's *Infer*[3] do not provide whole program analysis, but only serve a specific purpose, such as detecting Java NullPointerException.

2. Can support call graph construction.

[1] `https://spotbugs.github.io/`, accessed 27 July 2020
[2] `https://checkerframework.org/`, accessed 27 July 2020
[3] `https://fbinfer.com/`, accessed 27 July 2020

3. Are widely used by the program analysis research community, evidenced by citation counts of core papers, indicating that the respective frameworks are widely used, and therefore issues in those frameworks will have a wider impact on the research community.

4. Have some support for modelling dynamic language features, in particular reflection.

5. Under active development, indicating that the features of those frameworks will continue to have an impact.

Based on those criteria, we evaluated *Soot*, *Doop*[4] and *WALA*. For each tool, we considered a basic configuration (only the basic analysis such as CHA is enabled), and an advanced configuration (such as reflection support) to switch on support for advanced language features. All three tools have options to switch those features on.

To set up a **basic configuration**, we construct call graphs using a mid-precision, context-insensitive variable type analysis. Given the simplicity of our examples, where each method has at most one call site, we did not expect that context sensitivity would have made a difference. To the contrary, a context-sensitive analysis computes a smaller call graph, and would therefore reduce the FNs reported by the analyser. On the other hand, a less precise method like CHA could lead to reporting of more FPs, caused by the accidental coverage of non-target methods. **An advanced configuration** includes all options in the basic configuration, but with extra settings for handling dynamic features, if they are available. We explain the settings that we used for each of the analysers below:

WALA: we use the 0-CFA call graph builder. By default, we set `com.ibm.wala.ipa.callgraph.AnalysisOptions.ReflectionOptions` to `NONE`, which means reflection analysis is disabled. In the advanced configuration used, reflection analysis is enabled by setting it to `FULL`.

Soot: we use *SPARK* [119], a flexible points-to analysis framework that *Soot* supports. (`cg.spark=enabled,cg.spark=vta`). For the advanced configuration, the following options are being used:

- `"safe-forname"`. The class name is resolved when `Class#forName()` is used.
- `"safe-newinstance"` options. The newly created object is resolved when `Class#newInstance()` is used.

[4] as Doop does not release versions, we used a version built from commit 4a94ae3bab4edcdba068b35a6c0b8774192e59eb

- There is another option to support the resolution of reflective call sites, `typesfor-invoke`. Enabling this option leads to an error that was reported, but at the time of writing this issue has not yet been resolved [5].

Doop: we use the following options: `context-insensitive`, `ignore-main-method`, `only-application-classes-fact-gen` for the base analysis. For the advanced configuration[6], we also enable `reflection`, `reflection-classic`, `reflection-high-soundness-mode`, `reflection-substring-analysis`, `reflection-invent-unknown-objects`, `reflection-refined-objects`, `reflection-speculative-use-based-analysis`.

4.3 Benchmark of Dynamic Features in Java

Dynamic programming language features enable flexibility and productivity for developers writing programs. Features like *reflection* and *dynamic class loading* allow programs to be more generic and adaptable. In the following sections, various dynamic features in Java are discussed in detail.

4.3.1 Reflection

The benchmark examples reflect a range of usage patterns, from trivial to sophisticated patterns. Many programs overload the target method, this is used to test whether a static analysis tool achieves sound reflection handling at the price of precision.

Reflection is a widely used feature in many Java programs. It allows to examine the program's structure at run-time by inspecting classes, fields and methods [72]. With reflection, classes can be dynamically instantiated, fields can be accessed and manipulated, and methods can be invoked.

A common usage of reflection can be found in JUnit. Reflection can look for methods that are annotated with `@Test`, and will then invoke them. Service loader is another example of reflection. It uses `java.util.ServiceLoader` to load different implementation of services. Another use case of reflection is in dependency injection, where dependencies are supplied as a service (created only when it is needed). In order for the service to be registered, a text file is needed within the `META-INF/service` folder to specify the service provider.

To create an object or invoke a method, the relevant class name, method name and argument must be supplied. For static program analysis, reflection is difficult to handle because of its dynamic nature. Hirzel et al [95] pointed out that "It is not known statically which/when/where/whether class will be loaded". The analysis could

[5] `https://groups.google.com/forum/m/#!topic/soot-list/xQwsU7DlmqM`, accessed 6 May 2020

[6] the meaning of each configuration is discussed in Section 5.4.2.5

be over-approximated due to the class name or method name supplied as a run-time literal in the presence of dynamic class loading [170, 130]. Therefore, modelling reflective calls highly depends on the usage context. A reflective call site for `Method#invoke()` can be easily handled if the parameter at the method access site (i.e., the call site of `Class#getMethod()` or related methods) are known, for instance, if method name and parameter types can be inferred as shown in Listing 4.3.

Listing 4.3: Java reflection example

```
1  class Main{
2
3    public static void main(String[] args){
4      Method m = Main.class.getDeclaredMethod(
5                  "target",
6                  new Class[]{String.class});
7
8      m.invoke(this, "input");
9    }
10
11   public void target(String input) {
12   }
13 }
```

4.3.1.1 Reflection Variances

The data needed to accurately identify an invoked method might be supplied by other methods (therefore, the static analysis must be inter-procedural to capture this), only partially available (e.g., if only the method name can safely be inferred, a static analysis may decide to over-approximate the call graph and create edges for all possible methods with this name), provided through external resources (a popular pattern in enterprise frameworks like *Spring*, *service loaders*, or *JEE web applications*), or some custom procedural code.

The following Listing 4.4 shows an intraprocedural usage where the method name is provided via a series of transformations. Listing 4.5 demonstrates that reflection can happen in different procedures, i.e., the method name is supplied from an external file.

Listing 4.4: Java Intraprocedural Reflection

```
1  String methodName = new StringBuilder("TEGRAT").reverse().toString().
       toLowerCase();
2  Method m = Intraprocedural1.class.getDeclaredMethod(methodName, null);
3  m.invoke(this, null);
```

Listing 4.5: Java Interprocedural Reflection

```
1 BufferedReader br = new BufferedReader(new FileReader(this.getClass().
      getClassLoader().getResource("method.txt").getFile()));
2 String methodName = br.readLine();
3 Method m = Interprocedural1.class.getDeclaredMethod(methodName, null);
4 m.invoke(this, null);
```

4.3.1.2 Reflection with Ambiguous Resolution

We also considered other possible scenarios where a program is (at least partially) not generated by Java compiler `javac`. Since, at byte code level, methods are identified by a combination of name and descriptor, the JVM supports return type overloading, and the compiler may use this in order to support co-variant return types [81, section 8.4.5] by generating bridge methods. This raises the question how the methods in `java.lang.Class` are used to locate methods to resolve ambiguity as they use only name and parameter types, but not the return type, as parameters. According to the respective class documentation, *"If more than one method with the same parameter types is declared in a class, and one of these methods has a return type that is more specific than any of the others, that method is returned; otherwise one of the methods is chosen arbitrarily"* [7]. In case of return type overloading used in bridge methods, this rule still yields an unambiguous result, but one can easily engineer byte code where the arbitrary choice clause applies. The benchmark contains a relevant example, `dpbbench.ambiguous.ReturnTypeOverloading`. As Listing 4.6 shows, there are two target methods, one returning `java.util.Set` and one returning `java.util.List`. Since neither return type is a subtype of the other, the JVM is free to choose either of the two. In this case, we use the `@Target (expectation = MAYBE)` annotation to define the oracle. We acknowledge that the practical relevance of this might be low at the moment, but we included this scenario as it highlights that the concept of possible program behaviour used as ground truth to assess the soundness of static analysis is not as clear as it is widely believed. Here, possible program executions can be defined either with respect to all or some JVMs.

[7] `https://goo.gl/JG9qD2`, accessed 6 May 2020

Listing 4.6: Co-Variant Return Types

```
1  @Target(expectation=Expected.MAYBE)
2  public Set target(){
3    this.TARGET_SET =true;
4    return null;
5  }
6
7  @Target(expectation=Expected.MAYBE)
8  public List target() {
9    this.TARGET_LIST =true;
10   return null;
11 }
```

4.3.2 Dynamic Class Loading

Class instances can be loaded dynamically. Java distinguishes between classes and class loaders. This can be used to dynamically load, or even generate classes at run-time. As shown in Listing 4.7, the target class is not set in the class path. It is first compiled (at line 9), and then loaded at run-time (at line 3). This feature is widely used in practice, in particular for frameworks that compile embedded scripting or domain-specific languages, such as *Xalan*[8].

Listing 4.7: Dynamic class loading example

```
1  public void source() throws Exception {
2    CustomClassLoader classLoader = new CustomClassLoader();
3    classLoader.loadClass("dynamicClassLoading.Target").newInstance();
4  }
5
6  @Override
7  protected Class<?> findClass(String name){
8    byte[] content = new byte[0];
9    content = Utility.compile(this.getClass().getClassLoader(), name);
10   return defineClass(name, content, 0, content.length);
11 }
```

4.3.3 Dynamic Proxy

Proxy objects are intermediary objects between a client and target objects. Dynamic proxy can be used for many purposes: to facilitate distributed object frameworks like *CORBA* and *RMI*, as a transaction management for database connection (i.e., *Spring framework*), or in mocking framework for unit testing (such as *Mockito*). A proxy object forces method calls to happen indirectly through another object using

[8] `https://xalan.apache.org/`, accessed 6 May 2020

`java.lang.reflect.Proxy`. This where unsoundness could occur. Most static analysis tools assume an implementation in absence of proxy classes [114].

The benchmark contains an example program in the dynamic proxy category (shown in Listing 4.8). In this program, the source method invokes an interface method `foo()` through an invocation handler (`MyInvocationHandler`). In the invocation handler, method `target()` is invoked at line 11. The target method is overloaded in order to test the precision of the analysis (in this case, the target method with a `String` as an argument is taken into account).

Listing 4.8: Dynamic proxy example

```
1  public void source() {
2    MyInterface proxy = Proxy.newProxyInstance(MyInterface.class.
         getClassLoader(),
3    new Class[]{MyInterface.class}, new MyInvocationHandler());
4    proxy.foo("hello");
5  }
6
7  public class MyInvocationHandler implements InvocationHandler
8
9    @Override
10   public Object invoke(Object obj, Method m, Object[] arg){
11         target((String)arg[0]);
12         return null;
13   }
14 }
```

4.3.4 invokedynamic Instruction

The `invokedynamic` is a Java byte code instruction which gives a program power to control method dispatch by using a user-defined bootstrap method that computes the call target. A well known use case for `invokedynamic` in Java is lambda support which uses `java.lang.invoke.LambdaMetafactory` as a default bootstrap method for invokedynamic call sites. In OpenJDK 9, invokedynamic is also used for string concatenation. An example of lambda usage is shown in Listing 4.9. The benchmark provides four scenarios of using `invokedynamic`. It contains three programs with different uses of lambda. The fourth example is engineered from byte code and is an adapted version of the *Dynamo* compiler example from [100]. Here, `invokedynamic` is used for a special compilation of component boundary methods in order to improve binary compatibility. The intention of including this example is to distinguish between `invokedynamic` for particular usage patterns, and general support for `invokedynamic`.

Listing 4.9: Lambda example

```
1 Function<Integer, String> c = (i) -> target();
2 c.apply(3);
```

4.3.5 Serialisation

Serialisation is the process of converting (and storing) objects to a byte stream. The reverse process is called deserialisation. An object is deserialised from byte stream through method `java.io.ObjectInputStream#readObject()`.

The benchmark contains a program in this category that relates to the fact that deserialisation offers an extra-linguistic mechanism to construct objects, avoiding constructors. The scenario as shown in the Listing 4.10, constructs an object from a stream, and then invokes a method on this object. The client class is not aware of the actual type of the receiver object, as the code contains no allocation site.

Listing 4.10: Java deserialization example

```
1 ObjectInputStream ois=new ObjectInputStream(new ByteArrayInputStream(
      Utility.serialise(new Target())));
2 TargetInterface foo=(TargetInterface) ois.readObject();
3 foo.target();
4 ois.close();
```

4.3.6 JNI

The Java Native Interface (JNI) [125] is a framework that allows the JVM to load native code such as C, C++ and assembly. Static analyses usually do not consider native code as part of the analysis because the scope is limited to one language. This means it is not possible to capture the calls that flow into the native library and then back to the Java program.

There are two programs using JNI in the benchmark. The first scenario is demonstrated in the Listing 4.11, and uses a custom `Runnable` to be started by `Thread#start()`. In Java 8 (OpenJDK 8), `Runnable#run()` is invoked by `Thread#start()` via an intermediate native method `Thread#start0()`. The second program is a custom example that uses a callback through a native implementation (Listing 4.12).

Listing 4.11: Java JNI Thread example

```
1 java.lang.Thread t = new java.lang.Thread(this);
2 t.start();
```

Listing 4.12: Native Callback example

```
1  JNIEXPORT void JNICALL
2  Java_dpbbench_jni_Callbacks_source(JNIEnv *env, jobject obj)
3  {
4    jclass cls = (*env)->GetObjectClass(env, obj);
5    jmethodID mid = (*env)->GetMethodID(env, cls, "target", "()V");
6    if (mid == 0) {
7      return;
8    }
9    (*env)->CallVoidMethod(env, obj, mid);
10 }
```

4.3.7 sun.misc.Unsafe

The `sun.misc.Unsafe` in Java is used for accessing low-level programming. It was originally intended to facilitate the implementation of platform APIs, and to provide an alternative for JNI. This feature is now widely used outside the Java platform libraries [133]. Many developers treat this feature as a less constrained workaround [16]. For instance, Listing 4.13 illustrates a code example that does not need a constructor to be called for instantiating a class. It allocates an instance directly on the heap by using the method `Unsafe#allocateInstance()`.

Listing 4.13: Java Unsafe Allocation example

```
1  UnsafeInitialization.Inner inner=Utility.getUnsafe().allocateInstance(
       Inner.class);
2  inner.target();
```

The benchmark contains four patterns in this category, using unsafe to (1) load a class (`Unsafe#defineClass()`), (2) throw an exception (`Unsafe#throwException()`), (3) allocate an instance (`Unsafe#allocateInstance()`) and (4) swap references (`Unsafe#putObject()`, `Unsafe#objectFieldOffset()`).

4.4 Evaluation and Discussion

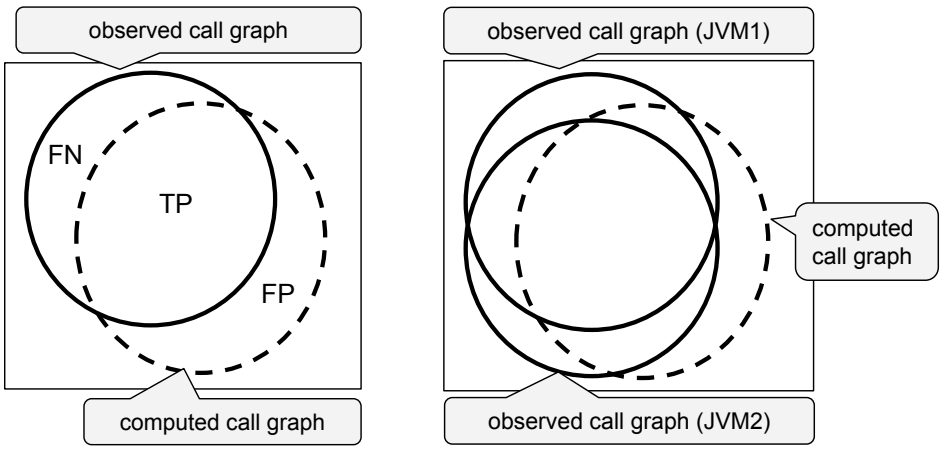
For each combination of benchmark program and static analyser, we compute a result state depending on the annotations found in the methods reachable from the `@Source` to an annotated method in the computed call graph as defined in Table 4.1. For instance, the state accurate (ACC) means that in the computed call graph, all methods annotated with `@Target (expectation = YES)` and none of the methods annotated with `@Target (expectation = NO)` are reachable from the method annotated with `@Source`. The `FP` indicates imprecision and `FN` indicates unsoundness, the `FN+FP` state shows that the results of the static analysis are both unsound and imprecise. Reachable means that

there is a path from a sink to a source method. This is slightly more general than looking for a single edge in the call graph and takes into account the fact that a particular JVM might use intermediate methods to implement a certain dynamic invocation pattern.

Table 4.1: Result for programs with consistent behaviour

Result state	Methods reachable from source by annotation	
	`@Target(expectation=YES)`	`@Target(expectation=NO)`
ACC	all	none
FP	all	some
FN	none	none
FN+FP	none	some

Figure 4.1 shows a conceptual illustration of what has been discussed in Section 4.3.1.2 – there are programs that use the `@Target (expectation = MAYBE)` annotation, indicating that actual program behaviour is not defined by the specification, and depends on the JVM implementation being used. This is illustrated in Figure 4.1(b), whereas Figure 4.1(a) shows a consistent behaviour across JVMs.



(a) Consistent behaviour across JVMs

(b) Inconsistent behaviour across JVMs

Figure 4.1: Observed vs computed call graph

For the programs that use the `@Target(expectation=MAYBE)` annotation, we modified this definition according to the semantics of the annotation: during execution, exactly one of these methods will be invoked, but it is up to the particular JVM to decide which particular method. We define result states as shown in Table 4.2. Note that the `@Target (expectation = YES)` and the `@Target (expectation = MAYBE)` annotations are never used for the same program, and that there is at most one method

annotated with `@Target (expectation = YES)` in each program.

This definition is very lenient – we assess the results of a static analyser as sound (`ACC` or `FP`) if it does compute a path that links the source with any possible target. This means that soundness is defined with respect to the behaviour observed with only some, but not all JVMs.

Table 4.2: Result for programs with behaviour that depends on the JVM

Result state	Methods reachable from source by annotation	
	`@Target (expectation = MAYBE)`	`@Target (expectation = NO)`
ACC	some	none
FP	some	some
FN	none	none
FN+FP	none	some

Tables 4.3, 4.4 and 4.5 report the number of patterns (also referred to as programs, since each pattern is independent and executable) with the respective result state for each framework (*Doop*, *WALA* and *Soot*, respectively). The cell in the tables are formatted as follows: **number of patterns obtained with basic configuration / number of patterns obtained with advanced configuration.** (an explanation of basic/advanced configuration is presented in Section 4.2.2). A discussion of the results from each analyser is presented in following sections.

4.4.1 Doop

There are significant differences between the basic and the advanced modes when reflection is involved – none of programs are handled by *Doop* when the basic configuration enabled. Four of them can be resolved under the advanced modes, however FPs are generated as well. For instance, `dpbbench.reflection.invocation.Interprocedural2` provides a scenario where the method name is supplied via a different procedure. A typical FN found under the reflection category is `dpbbench.reflection.invocation.Interprocedural1`. In this case, the method is stored in an external resource and *Doop* fails to resolve it (the source code can be found at Listing 4.5). *Doop* did not support dynamic proxy and JNI by the time that the experiment been conducted. The supports were added recently [73, 75].

Table 4.3: SCG evaluation results for Doop

Category	ACC	FN	FP	FN+FP
vanilla	1/1	0/0	0/0	0/0
reflection	0/0	12/8	0/4	0/0
dynamic class loading	0/0	1/1	0/0	0/0
dynamic proxy	0/0	1/1	0/0	0/0
invokedynamic	0/0	4/4	0/0	0/0
JNI	0/0	2/2	0/0	0/0
serialisation	0/0	1/1	0/0	0/0
Unsafe	0/0	2/2	1/1	1/1
reflection-ambiguous	0/0	2/1	0/1	0/0

4.4.2 Soot

Soot accurately reports the vanilla case as well as one of JNI patterns, (`Thread#start()` as shown in Listing 4.11), and the deserialisation pattern, (`dpbbench.serialisation.Deserialisation` as shown in Listing 4.10). With the advanced mode enabled, *Soot* is only able to handle 1/12 patterns which is the basic reflection pattern. The rest of reflection patterns are flagged as FNs. Dynamic class loading case is not supported by *Soot*.

Table 4.4: SCG evaluation results for Soot

Category	ACC	FN	FP	FN+FP
vanilla	1/1	0/0	0/0	0/0
reflection	0/1	12/11	0/0	0/0
dynamic class loading	0/0	1/1	0/0	0/0
dynamic proxy	0/0	1/1	0/0	0/0
invokedynamic	0/0	4/4	0/0	0/0
JNI	1/1	1/1	0/0	0/0
serialisation	1/1	0/0	0/0	0/0
Unsafe	0/0	2/2	1/1	1/1
reflection-ambiguous	0/0	2/2	0/0	0/0

4.4.3 WALA

WALA has support for certain usage patterns of other features: it models `invokedynamic` instructions generated by the compiler for lambdas correctly. However, it fails to resolve the user-defined bootstrap method. *WALA* also models the intermediate native call in `Thread#start()` and the dynamic proxy, when in advanced mode. This may be a reflection of the maturity and stronger industrial focus of the tool.

Table 4.5: SCG evaluation results for WALA

Category	**ACC**	**FN**	**FP**	**FN+FP**
vanilla	1/1	0/0	0/0	0/0
reflection	0/4	12/3	0/5	0/0
dynamic class loading	0/0	1/1	0/0	0/0
dynamic proxy	0/1	1/0	0/0	0/0
invokedynamic	3/3	1/1	0/0	0/0
JNI	1/1	1/1	0/0	0/0
serialisation	1/1	0/0	0/0	0/0
Unsafe	0/0	2/2	1/1	1/1
reflection-ambiguous	0/0	2/0	0/2	0/0

4.4.4 Discussion

In summary, none of the static analysers tested handled all features soundly, as expected. None of the frameworks handles any of the `Unsafe` scenarios well. There is one particular program where all analysers compute the wrong call graph edge (shown in Listing 4.14): the target method is called on a field that is initialised as `new Target()`, but between the allocation and the invocation of `target()` the field value is swapped for an instance of another type using `Unsafe#putObject`. While this scenario appears far-fetched, we note that `Unsafe` is widely used in libraries [133].

Listing 4.14: Unsafe type confusion example

```
1  @Source
2  public void source() throws Exception {
3    target = new Target();
4    Utility.getUnsafe().putObject(this,
5        Utility.getUnsafe().objectFieldOffset(UnsafeTypeConfusion.class.
             getDeclaredField("target")),
6        new Target2());
7    target.target();
8  }
9
10 public class Target {
11
12   @dpbbench.util.Target(expectation = NO)
13   public void target() {
14     TARGET = true;
15   }
16 }
17
18 public class Target2 {
19
20   @dpbbench.util.Target(expectation = YES)
21   public void target() {
22     TARGET2 = true;
23   }
24 }
```

A customised `invokedynamic` call site is unlikely to be supported by many tools. Jezek et al.[100] presents a case called *Dynamo* which we have included in the benchmark. *Dynamo* is designed to resolve linkage errors by creating a customised `invokedynamic` call site. Even though tools like *WALA* and *Opal* [67] can handle lambda by looking into the bootstrapped method, a customised bootstrapped method is often overlooked.

We note that *Soot* has better integration with *TamiFlex* [41]. *TamiFlex* is a dynamic analysis tool which requires the program to be executed to obtain program models, therefore it uses a fundamentally different approach to soundly model dynamic language features. How well a dynamic (pre-) analysis works depends a lot on the quality (coverage) of the driver. For the micro-benchmark we have constructed a perfect driver that has only one entry point to the program, and exercises the program as intended. Using *Soot* with *TamiFlex* with such a driver would have yielded the same results.

An interesting observation we made from experimenting with the benchmark programs, is that the behaviour of the program under analysis is not defined by the language specification nor by the JVM specification. We found that the behaviour varies across different JVM implementations (i.e., OpenJDK vs HotSpot). This raises a question about the ground truth of possible program behaviour. For instance, Oracle JRE

1.8.0_144 / OpenJDK JRE 1.8.0_40, on the one hand, IBM JRE 1.8.0_171, on the other, actually do select different methods. We have also observed that IBM JRE 1.8.0_171 chooses the incorrect method in the related `dpbbench.reflection.invoca-tion.ReturnTypeOverloading` scenario (as shown in Listing 4.15). In this scenario, the overloaded target methods return `java.util.Collection` and `java.util.List`, respectively, and the IBM JVM dispatches to the method returning `java.util.Collection`, in violation of the rule stipulated in the API specification. We reported this as a bug, and it was accepted and fixed[9].

Listing 4.15: Return type overloading

```
1  @Target(expectation=Expected.NO)
2  public Collection target(){
3    this.TARGET_COLLECTION =true;
4    return null;
5  }
6
7  @Target(expectation=Expected.YES)
8  public List target() {
9    this.TARGET_LIST =true;
10   return null;
11 }
```

We also observe that the call graphs differ depending on the JVM being used. For instance, in the program in Listing 4.16, the target method selected at the call site in `source()` is `target()` for both Oracle JRE 1.8.0_144 and OpenJDK JRE 1.8.0_40 , and `target2()` for IBM JRE 1.8.0_171.

[9] `https://github.com/eclipse/openj9/pull/2240`, accessed 6 May 2020

Listing 4.16: An example of reflection with ambiguous resolution

```
1  public void source() throws Exception {
2    for (Method method:Invocation.class.getDeclaredMethods()){
3      if (method.isAnnotationPresent(Method.class)){
4        method.invoke(this, null);
5        return;
6      }
7    }
8  }
9
10 @Method
11 @Target(expectation = MAYBE)
12 public void target() {
13   this.TARGET = true;
14 }
15
16 @Method
17 @Target(expectation = MAYBE)
18 public void target2() {
19   this.TARGET2 = true;
20 }
21
22 @Target(expectation = NO)
23 public void target3() {
24   this.TARGET3 = true;
25 }
```

Very recently, Reif et al. [158] have published a Java test suite designed to test static analysers for their support for dynamic language features, and evaluated *WALA* and *Soot* against it. While this is very similar to the approach presented here, there are some significant differences:

1. The authors of [158] assume that the tests (benchmark programs) provide the **ground truth**. In this study, we question this assumption, and propose an alternative notion that also take characteristics of the JVM and the platform used to execute the tests into account.

2. The study presented here also investigates *Doop*, which we consider important as it offers several features for advanced reflection handling.

3. While the construction of both test suites/benchmarks was motivated by the same intention, they are different. Merging and consolidating them is an interesting area for future research.

4.5 Summary and Conclusion

In order to answer RQ1: **What are the sources of unsoundness in static analysis?** a set of benchmark program is presented that describes the usage of dynamic language features in Java. We further sort these usage patterns into the following categories: *reflection*, *serialisation*, *dynamic class loading*, *invokedynamic*, *dynamic proxy* and the use of *native libraries*. An experiment is then conducted to assess RQ2: **Are state-of-the-art static analysis tools able to successfully model dynamic language features in Java?** The short answer is: no, they are not able to model all dynamic language features in Java. It is not surprising that the static call graphs miss edges in many cases, such as serialisation and unsafe. Even the widely used feature, reflection, has not been fully supported with many detailed usages.

The results indicate that it is necessary to differentiate between the actual features and a usage context for those features. For instance, there is a significant difference between supporting `invokedynamic` as a general feature, and `invokedynamic` as it is used by the Java 8 compiler for lambdas. Another example would be the use of reflection in general, and various ways of supplying meta data (class/method name) for the reflection. The benchmark design, and the results of the experiments, highlight this difference.

We do not expect that static analysis frameworks will support all of these features and provide a sound and precise call graph in the near future. Instead, many tools will continue to focus on particular usage patterns, such as support for reflection used in the Spring framework, which have the biggest impact on actual programs, and therefore should be prioritised. However, as discussed using examples throughout this thesis, more exotic usage patterns do occur, and can be exploited, so they should not be ignored. The benchmark provided here can provide some guidance for tool builders from here on.

An interesting insight coming out of this study is that notions like actual program behaviour and possible program executions are not as clearly defined as widely thought. This is particularly surprising in the context of Java (even in programs that do not use randomness, concurrency or native methods), given the strong focus of the Java platform on writing code once, and running it anywhere, with consistent program behaviour. This has implications for the very definitions of soundness and precision. We have suggested a pragmatic solution, but we feel that a wider discussion of these issues is needed.

Chapter 5

Oracle Generation from Stack Traces

5.1 Introduction

In this chapter, we present a technique of using stack traces to complement statically built call graphs. We treat invocations that are part of reported stack traces as soundness **oracles** as they represent observed program behaviours – they provide crucial information about the behaviour of a running program. This behaviour might even be particularly interesting and valuable in the sense that it has potentially led to exceptions, and therefore is likely to be a behaviour that has not been encountered during a routine dynamic analysis procedure, such as testing. Oracles are generated from: (1) Common Vulnerabilities and Exposures(CVE) [3], a well-known on-line platform for reporting vulnerabilities. We recreate some vulnerabilities and capture their stack traces in order to extract call chains. (2) Reflective call chains mined from GitHub[1] and Stack Overflow[2]. GitHub and Stack Overflow are two well-known communities for developers to post questions and issues they encounter. These online resources have practical implications for the usage patterns in the benchmark presented in Chapter 4 – they are either reported as real-world vulnerabilities or program issues that have been encountered by developers. As a result, we identified some patterns that are missed by static analysis, and also proved that the mining technique can be used as an alternative way to construct soundness oracles.

[1] `https://github.com`, accessed 18 December 2020
[2] `https://stackoverflow.com`, accessed 18 December 2020

5.2 Java Stack Traces Format

A typical stack trace consists of serial call traces produced by the program when unexpected behaviour occurs. This allows developers to locate the source of an issue, as it provides crucial information to diagnose a crashed program and, therefore, most modern languages have support to handle such a process. In Java, the keyword `try` defines a block to be executed and `catch` handles the exceptions. There are two types of exception: checked and unchecked exceptions. A checked exception is checked at compile-time. If the compiler detects an unhandled exception then it must be surrounded with a `try-catch` statement or use the `throw` keyword. An unchecked exception is thrown at run-time, which makes handling the exception optional. For instance, Listing 5.1 shows an example of an unchecked exception. A `NullPointerException` is thrown at line 13 as the variable `foo` has not been allocated. When a user encounters this exception, the stack trace shown in Listing 5.2 is generated. The exception type is shown in Line 1. The rest is the body of the stack trace. The format is defined as follows: `at` followed by a space, then package name (`org.example`), class name (`Main`) and method name (`foo`). The text in between brackets indicates the type name (Main.java) and its corresponding line number (8). Parameter type information is not available in a Java stack trace. As a result, methods with the same parameter type can not be differentiated. In this case, line numbers in source code can be used to distinguish methods with the same parameter type.

Listing 5.1: Unchecked exception

```
1  class Main{
2
3    public static void main(String[] args){
4      bar();
5    }
6
7    public static void bar(){
8      foo();
9    }
10
11   public static void foo(){
12     Object foo=null;
13     foo.toString();// throw NullPointerException
14   }
15 }
```

Listing 5.2: Java stack trace example

```
1 Exception in thread "main" java.lang.NullPointerException
2         at org.example.Main.foo(Main.java:8)
3         at org.example.Main.bar(Main.java:4)
4         at org.example.Main.main(Main.java:3)
```

The call trace is used to infer a calling relationship between two or more methods. From Listing 5.2, we can see that the root of the method is `org.example.Main#main()` at line 3. The exception is thrown at line 8: `org.example.Main#foo()`. A simple call graph can then be crafted bottom up, as shown below:

`org.example.Main#main()`→`org.example.Main#bar()`
`org.example.Main#bar()`→`org.example.Main#foo()`

Exceptions can also be linked. The "`Caused by:`" clause, as shown in Listing 5.3, indicates two linked stack trace bodies. The respective stack trace contains two different bodies, with the method that has the reflective call site in the first block just below the reflective call site (`Method#invoke()`), and the target at the bottom of the second block. This information can be used to infer the edge `b()`→`c()`. It is important to note that the call graph represents a code snippet that actually has been executed. In other words, it reveals the real program behaviour.

5.3 The Exception Caused by Reflective Invocation

Modern programming languages are full of dynamic features that are difficult to capture by static analysis, which results in unsoundness. Features such as reflection, proxies, dynamic class loading have been discussed in Chapter 4. In this experiment, we focus on the `InvocationTargetException` which is raised when a reflective invocation occurs. Previous work [61] suggests reflection is a widely used feature in practice (it is also presented in Table 6.1) and hence we expect that rich resources can be obtained online. Besides the `InvocationTargetException`, there are more exceptions that relate to reflective invocation which are shown below.

- InstantiationException: reflective instantiation via `Class#newInstance()`
- InvocationTargetException: reflective invocation via `Method#invoke()`
- UndeclaredThrowableException: used for dynamic proxy, reflective invocation via `InvocationHandler#invoke()`
- Javax.script.ScriptException: a reflective invocation via script evaluation.

InvocationTargetException is a checked exception and occurs when an invoked method throws an exception [81]. It usually can be observed under a reflection pattern being used. Consider again the program in Listing 5.4. A run-time exception is thrown at line 22. This will generate the following stack trace (Listing 5.3).

Listing 5.3: InvocationTargetException statcktrace

```
1  Exception in thread "main" java.lang.reflect.InvocationTargetException
2  ..
3  at java.lang.reflect.Method.invoke(Method.java:498)
4  at Foo.b(Foo.java:7)
5  at Foo.a(Foo.java:6)
6  at Foo.main(Foo.java:3)
7  ..
8  Caused by: java.lang.RuntimeException
9  at Foo.d(Foo.java:13)
10 at Foo.c(Foo.java:12)
11 ... 12 more
```

Listing 5.4: InvocationTargetException Example

```
1  class Foo {
2
3    public static void main(String[] p) throws Exception{
4      new Foo().a();
5    }
6
7    void a() throws Exception {
8      b();
9    }
10
11   void b() throws Exception {
12     Class c = Class.forName("Foo");
13     Method m = c.getDeclaredMethod("c",new Class[]{});
14     m.invoke(this, new Object[]{});
15   }
16
17   void c() {
18     d();
19   }
20
21   void d() {
22   throw new RuntimeException();
23   }
24 }
```

As we can see from the stack trace, an edge `b()`→`c()` which indicates a call from method `b()` to `c()`, via reflection, can be extracted. This edge is often overlooked by

static analysis clients due to its dynamic nature.

5.4 Methodology

5.4.1 Oracle Construction from CVE

One objective for the benchmark construction (Chapter 4) was to select features that are of interest to static program analysis, as there are known vulnerabilities that exploit those features. The discussed features allow bypassing of Java's security model, which relies on information-hiding, memory and type safety. Java security vulnerabilities which involve those uses have been reported that have implications ranging from attacks on confidentiality, integrity and the availability of applications. For instance, serialization attack [60, 155] is a form of DDOS attack. A famous attack being billion laughs [4] in which the attacker crafts a nested object for over-using computing resources. Between the years of 2015 and 2017, a number of attacks have been reported in the Common Vulnerabilities and Exposures(CVE) dataset. `CVE-2015-7450` is a well-known serialisation vulnerability in the *Apache Commons Collections* library. Attackers can invoke arbitrary commands though a crafted serialized Java object in the class `org.apache.commons.collections.map.LazyMap`. We reconstruct a number of serialisation-related attacks based on Chris Frohoff's ysoserial repository[3], we capture stack traces and therefore build call graphs. The call graph shown in Listing 5.5, demonstrates the methods involved along the call chain. The method in line 11 gives attackers a power to execute any commands (e.g., shutdown the system) that the system can provide and the method in line 10 (`java.lang.reflect.Method#invoke()`) grants the power to do so. Thus, this method is a critical section that needs to be monitored. A traditional defence mechanism would be running the entire program under a sandbox environment. However, the performance overhead in that case is also huge. Employing static program analysis can help to detect such unsafe call chains, preferably before the program is deployed.

[3] `https://github.com/frohoff/ysoserial`, accessed 18 December 2020

Listing 5.5: Unsafe deserialization call graph example [18]

```
1  java.io.ObjectInputStream.readObject()
2  ->java.util.HashSet.readObject()
3  ->java.util.HashMap.put()
4  ->java.util.HashMap.hash()
5  ->org.apache.commons.collections.keyvalue.TiedMapEntry.hashCode()
6  ->org.apache.commons.collections.keyvalue.TiedMapEntry.getValue()
7  ->org.apache.commons.collections.map.LazyMap.get()
8  ->org.apache.commons.collections.functors.ChainedTransformer.transform()
9  ->org.apache.commons.collections.functors.InvokerTransformer.transform()
10 ->java.lang.reflect.Method.invoke()
11 ->java.lang.Runtime.exec()
```

Use of reflection is common in vulnerabilities as discussed by Holzinger et al [97] where the authors discover that 28 out of 87 exploits studied utilised reflection vulnerabilities. An example is `CVE-2013-0431`, affecting the Java JMX API, which allows loading of arbitrary classes and invoking their methods. `CVE-2009-3869`, `CVE-2010-3552`, `CVE-2013-08091` are buffer overflow vulnerabilities involving the use of native methods. As for vulnerabilities that use the `Unsafe` API, `CVE-2012-0507` is a vulnerability in `AtomicReferenceArray` which uses `Unsafe` to store a reference in an array directly that can violate type safety and permit escaping the sandbox. `CVE-2016-4000` and `CVE-2015-3253` reported for Jython and Groovy are due to serialisable invocation handlers for proxy instances. While we are not aware of vulnerabilities that exploit `invokedynamic` instruction directly, there are several CVEs that exploit the method handle API used in the `invokedynamic` bootstrapping process, including `CVE-2012-5088`, `CVE-2013-2436` and `CVE-2013-0422`.

5.4.2 Oracle Generation from On-line Resources

GitHub and Stack Overflow are two well-known communities. GitHub is a project hosting site used by the open source community. It has a version control system, named Git, which allows a developer to manage source code and collaborate with others. GitHub also provides an issue tracking system for tracking problems during the development phase. Stack Overflow is one of the largest Q&A online forums. Developers can post a question and others can give answers. Data from both have been used to support our understanding about programs and developers. Developers post and share information in public so everyone in the community can contribute.

The mining process is shown in Figure 5.1. To obtain data from both platforms, we use a web crawler to gather HTML pages with keywords being searched for within Github issue tracker and Stack Overflow Q&A forum through HTTP requests. We then harvest relevant HTML pages and parse relevant information from each page. From

extracted stack traces, we then build call graphs from those stack traces. We then construct static call graphs using *Doop*. Both call graphs are then compared.

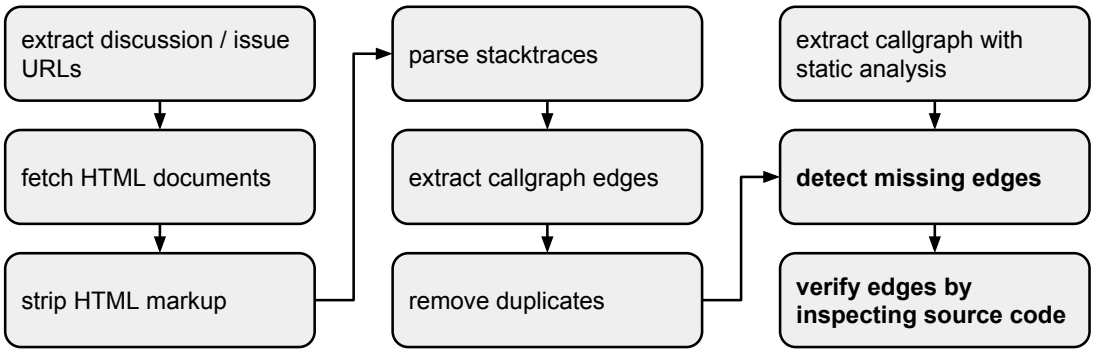


Figure 5.1: Mining stacktraces process

Static call graphs are constructed based on *DaCapo 2009* [38] – a Java benchmark that is widely used in programming language research. In addition to *DaCapo*, we also included the following programs that are known to use reflection: *log4j-2.1*, *antlr4-4.0*, *hbase-hbase-client-0.98.0-hadoop1*, *guava-11.0*, *spring-boot-loader-1.2.5* and *weld-core-impl-2.2.12* (by *jboss*).

5.4.2.1 Mining Technique

The idea of mining software repositories for useful knowledge has been gaining popularity in the past few years. Several mining techniques have been proposed and used in order to obtain information from on-line repository and knowledge platforms. We have explored a number of tools and techniques in order to retrieve the information that we need. Each of these techniques has some limitations. We list a brief description of each approach.

- Google Custom Search Engine[4]: Google provides APIs that allow users to use its powerful search engine to search through the Internet. The main limitation is that the freely available tool is limited to only 100 search queries per day.
- Ghtorrent[5]: Ghtorrent [19] provides offline data from GitHub. Users can import the data into a local MySQL database. Because it is a snapshot of the data, the data set is not up to date. Another limitation is availability of the issues from the issue tracker - issues entries do not contain the actual issue text.

[4] `https://developers.google.com/custom-search`, accessed 18 December 2020
[5] `https://ghtorrent.org`, accessed 18 December 2020

- GitHub API[6]: local GitHub data can be retrieved via the GitHub REST API in a JSON format. However, and similar to Ghtorrent, it does not provide the issues text from the issue tracker.
- HTTP+Jsoup[7]: Jsoup a HTML parser used in Java. HTTP GET requests are sent to retrieve any desired pages. Jsoup is then used to parse HTML DOM. The limitation of this approach is that a server may reject the requests, as such requests tend to consume network resources.

After experimenting with all of the above approaches, we found that the HTTP+Jsoup is the best option for the following reasons: (1) GitHub API and Ghtorrent do not provide full text of the reported issues as it appears in the issue tracker. (2) The free version of Google Custom Search Engine is limited to 100 search queries per day, where HTTP request does not have such a limitation. (3) The HTTP request always retrieves the latest information whereas Ghtorrent only provides a snapshot of the data from GitHub, which can be a few days back. We therefore decided to write a custom HTML client to search within the GitHub issue tracker and Stack Overflow Q&A sites for issues and discussions that include the text `java.lang.reflect.InvocationTargetException`. In order to resolve the request rejection issues, we added a time delay between each request. More details are discussed in the following Section 5.4.2.2.

5.4.2.2 Retrieving Web Resources

The custom HTML client that we wrote mimics a web browser session using the appropriate headers, and returns the URLs of the respective static issue web pages. We then downloaded the respective web pages, stripped HTML mark-up and extracted stack traces, instantiating the meta-model depicted in Figure 5.2.

HTTP is a stateless protocol. When exchanging information, servers need a cookie from users for the purpose of authentication. Therefore, the custom HTML client must encapsulate user's information which stored in a cookie together with a HTTP request. The following attributes are needed with the request: user agent(Browser), geolocation, user name, user session, time zone and the query.

Listing 5.6: Github query

```
https://github.com/search?type=issue&p=1&q=java.lang.reflect.
    InvocationTargetException
```

[6] `https://developer.github.com/v3`, accessed 18 December 2020
[7] `https://jsoup.org`, accessed 18 December 2020

Listing 5.7: Stackoverflow query

```
http://stackoverflow.com/search?page=1&tab=relevance&q=java.lang.reflect.
    InvocationTargetException
```

In the above queries, fields `page` and `p` refer to the page that the query will return. In GitHub, the field `type` defines the search scope, which, in our case, is the *issue tracking system*. We are only interested in the text within issues (and their subsequent discussion). For Stack Overflow, the field `tab` is to order results and the field `q` is the search query, which targets `java.lang.reflect.InvocationTargetException`. After sending a `GET` request to the server, 50 results in a single page are returned. Each result has a link to the issue/question page, another `GET` request is then dispatched to the server upon the given link. Once the issue/question page is received, all HTML mark-up will be stripped, making the text ready for parsing.

This method basically simulates a real user who navigates through a web page. Potentially, millions of results can be obtained in a second if requests are sent in parallel. In reality, many servers have anti-crawling mechanisms to prevent its resource from being overused [128]. So, we have to sacrifice time to process requests in a linear fashion by adding a sleep timer between each request. The client will pause for a random time between sending requests, which makes it harder for the server to predict whether the request has been sent by a human or by a bot.

5.4.2.3 Extract Stack traces

As shown in Figure 5.2, a stack trace can be modelled as the following: a stack trace can cause another stack trace. Each stack trace has multiple stack trace elements. A stack trace element consists of a line number, a class name and a method name.

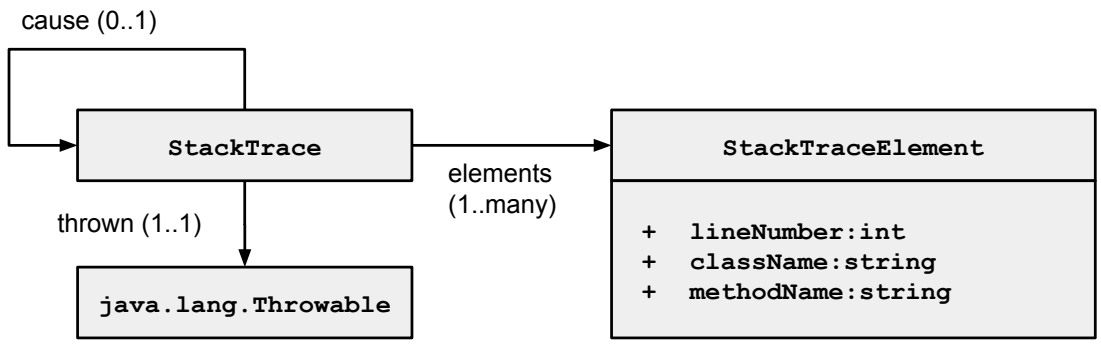


Figure 5.2: Stacktrace model

To extract the correct stack trace element from a text, the following regular expression (shown in Listing 5.8) is implemented. A stack trace always starts with `at`

(see Listing 5.3). In Java, package, class and method names should only contain ASCII letters and numbers [34]. A period dot (.) is used as a separator for each subpackage as well as for class and method names.

Listing 5.8: regular expression to parse stacktrace

```
at\\s(([a-z]+)\\.)(\\p{Alpha}|[0-9])+\\.(\\p{Alnum}|\\_|\\$|\\>|\\<|\\.)
    +\\((\\p{Alnum}|\\.|\\:|\\p{Space}|\\[|\\])+\\)
```

There are two processes required to correctly obtain a full stack trace from a plain text. Firstly, because a page could contain multiple independent stack traces that are all related to **InvocationTargetException**, we use the keyword `java.lang.reflect.InvocationTargetException` to divide the text into parts and process them independently. Secondly, each part is further separated to find **Caused By** clauses. Each separated piece can be seen as linked element in the order. The drawback of this method is when non-relevant texts that match the key are presented. Stack traces can be customised especially with a logging framework.

5.4.2.4 Validation

There are a few issues to be resolved in order to obtain correct results. Firstly, stack traces in many websites are lacking versioning information. The stack trace edges extracted may not be the same as in the version of the program we statically analysed. It is also difficult to extract this information as it has no fixed format and may result in a large amount of false results upon parsing. Secondly, the stack trace only contains method names, but is missing signatures and descriptor information, which is important for method overloading. Method overloading refers to multiple methods with the same name but different parameter types. This can also produce imprecise results, as shown in Listing 5.9. Both stack trace elements would point to the same call vertex without specifying parameter types: `example.foo`.

Listing 5.9: Method Overloading and its corresponding Stack trace element

```
foo(int i) -> example.foo(Foo.java:1)
foo(String i) -> example.foo(Foo.java:2)
```

To validate the correctness of our approach, two researchers conducted a manual cross-validation of the first 100 recorded stack traces. One would first read each stack trace and decide whether to include or exclude it, based on (1) whether or not it is a valid Java stack trace (2) whether or not it includes a reflective call site. The other researcher then cross-validated the results. The two would discuss any classification disagreements, until they reach an agreement. To overcome the method overloading issue and locate the accurate version, line numbers in both stack trace and source code are used. As result, we found 17 false results. Those stack traces were then eliminated.

We identified 15 unique edges that connect methods via reflection in the final result, and no noises were detected.

5.4.2.5 Static Call Graph Construction

Static call graphs are built using *Doop*. There are two analyses performed. 1): We have used the following *Doop* options to enable reflection analysis:

- `-reflection`. Enable logic for handling Java reflection.
- `-reflection-classic`. Enable (classic subset of) logic for handling Java.
- `-reflection-high-soundness-mode`. Enable extra rules for more sound handling of reflection.
- `-reflection-substring-analysis`. Allows reasoning on what substrings may yield reflection.
- `-reflection-invent-unknown-objects`. Create an object for unknown objects (of type `java.lang.Class`) [170].
- `-reflection-refined-objects`. Enable extra rules for more sound handling of reflection.
- `-reflection-speculative-use-based-analysis`. This analysis involves a back /forward-propagation mechanism [170] – gather information from reflective result to the original reflection and vice versa.

2): Without reflection analysis, we only used the option `context-insensitive`.

5.5 Result

In total, we mined a total of 18,431 pages (11,932 issues from GitHub and 6,499 posts from Stack Overflow). From those pages, we extracted a total of 12,329 stack traces. We then constructed a total of 11,920 reflective call graph edges (following the model explained in 5.4.2.3).

We observed that there are many cases in which stack traces that are reported on different pages were describing the same call graph edge. After filtering out those duplicates, we ended up with 4,747 edges. Among those, we were able to identify 495 reflective call sites that are matched in our data set[8]. To compare the results with the call graph edges generated statically with *Doop*, we further removed edges with

[8]DaCapo2009, log4j-2.1, antlr4-4.0, hbase-hbase-client0.98.0-hadoop1, guava-11.0, spring-boot-loader-1.2.5 and weld-core-impl-2.2.12 (by jboss)

targets outside the data set. Those removed cases are reflective calls that are in an additional client-specific library or program. In the end, we only found 15 unique edges (i.e., methods connected via reflection) that can be cross-referenced with the statically built call graphs. Not surprisingly, the number of relevant edges that we found was low, as the respective issues (in GitHub) and discussions (in Stack Overflow) cover a wide range of different projects, most of them not part of our data set. Without reflection analysis, *Doop* did not find any of the call graph edges we recovered from stack traces. With reflection analysis enabled, *Doop* found only 4 of the 15 edges. The following subsections discuss the four chosen edges in detail. Description of the rest of the edges can be found in Appendix A.1.

5.5.1 Apache Fop

Apache Fop[9] (Formatting Objects Processor) is a utility tool used to format console outputs. The `org.apache.fop.cli.Main` class specifies the target's class and method name in literals, as well as the parameter type (as shown in Listing 5.10). *Doop* can capture this edge if reflection analysis is enabled.

call site:	`org.apache.fop.cli.Main# startFOPWithDynamicClasspath`
target:	`org.apache.fop.cli.Main#startFOP`
stacktrace:	`https://goo.gl/bfFRhG`
source:	`https://goo.gl/JoXoam`

Listing 5.10: Reflective method invocation in fop

```
Class clazz=Class.forName("org.apache.fop.cli.Main",true,loader);
Method main=clazz.getMethod("startFOP",new Class[]{String[].class});
mainMethod.invoke(null,new Object[]{args});
```

5.5.2 Antlr

Antlr is a popular parser generator. In `org.antlr.v4.parse.GrammarTreeVisitor` class, `visitGrammar(GrammarAST)` method invokes `visit(GrammarAST,String)` method using the string literal **grammarSpec** as a second **ruleName** parameter. The rule name is then interpreted as method name in the method `visit(GrammarAST, String)`. *Doop* with reflection analysis enabled is able to identify the edge, since it tracks string literals and interprets as method names across procedures.

call site:	`org.antlr.v4.parse.GrammarTreeVisitor#visit`
target:	`org.antlr.v4.parse.GrammarTreeVisitor# grammarSpec`
stacktrace:	`https://goo.gl/BOZAOb`
source:	`https://goo.gl/g6JYdN`

[9] `https://xmlgraphics.apache.org/fop/`, accessed 18 December 2020

5.5.3 Hbase-client

Apache Hadoop is a popular framework for storing and processing big data. The target's method name: (**parseFrom**) and signature (**byte[].class**) are both defined within the `GrammarTreeVisitor#visit` method, but the class name is provided via a dynamic class loader that is configured with information read from project-specific configuration files. General-purpose static analysis tools are unlikely to precisely model this. However, the class must extend `org.apache.hadoop.hbase.filter.Filter`, and a possible approach is to ensure soundness by over-approximating the analysis. This can be achieved by adding edges to all `parseFrom(byte[])` methods implemented in subclasses of **Filter**. This means that soundness can be achieved by compromising precision. *Doop*, with reflection analysis enabled, will find this edge.

call site:	`org.apache.hadoop.hbase.protobuf.ProtobufUtil#toFilter`
target:	`org.apache.hadoop.hbase.filter.FilterList# parseFrom`
stacktrace:	`https://goo.gl/RUZO27`
source:	`https://goo.gl/bX3ffV`

Among the 15 edges, we found a particular edge, where the target is `org.apache.had -oop.hbase.filter.Filter#parseFrom()`. *Doop* can find this edge as well, with reflection analysis enabled.

5.5.4 Log4J

Log4j is a widely used logging framework. The use of reflection that creates the reflective call site is the most sophisticated we have encountered; however, this is common for frameworks that support plugins. Reflection is used to achieve loose coupling and sandboxing. In the `PluginBuilder#build()` method, the detection of the method invoked is delegated to the **findFactoryMethod**. This method detects the first static method it can find annotated with **@PluginFactory**. This semantic is almost impossible to capture with static analysis tools, and the only strategy possible here would be to over-approximate. *Doop* cannot find this edge, even with reflection analysis enabled. We discovered a usage pattern based on this particular log4j scenario, which we then included in our benchmark (shown in Listing 4.16).

call site:	`org.apache.logging.log4j.core.config.plugins.util.PluginBuilder#build`
target:	`org.apache.logging.log4j.core.appender.RollingFileAppender#createAppender`
stacktrace:	`https://goo.gl/Ohg7lo`
source:	`https://goo.gl/o885Hw`

5.5.5 Stack Traces from CVE

We have set up a script[10] that demonstrates vulnerabilities provided in Table 5.1. These vulnerabilities can provide a good understanding of how dynamic features are being used in a sense of program security. Throughout our analysis of stack traces, we were able to gain insight into the benchmark presented in Chapter 4, in particular, how serialization operates with other dynamic features, such as reflection and dynamic proxy. Table 5.1 contains a set of 19 programs with vulnerabilities. Note that not all of the vulnerabilities are obtained from existing CVEs, entries labeled as `none` in the related column in Table 5.1 are obtained from Chris Frohoff's ysoserial repository[11].

Table 5.1: CVE results

program	versions	related CVEs	dynamic features
BeanShell	2.0b5	CVE-2017-5586,CVE-2016-2510	dynamic proxy
C3P0	0.9.5.2	none	dynamic proxy
clojure	1.8.0	none	reflection
commons-beanutils	1.9.2	CVE-2014-0114 ,CVE-2016-4385	reflection
commons collections	3.1 and 4.0	CVE-2015-4852,CVE-2015-7501, CVE-2015-8765,CVE-2017-15708, CVE-2017-5586,CVE-2017-10932, CVE-2016-4373,CVE-2016-4372 , CVE-2016-4369,CVE-2016-4368, CVE-2016-3642,CVE-2016-2170, CVE-2016-2009,CVE-2016-2003, CVE-2016-2000,CVE-2016-1999, CVE-2016-1998 ,CVE-2016-1997, CVE-2016-1986,CVE-2016-1985, CVE-2016-1114,CVE-2015-7450, CVE-2015-6934,CVE-2015-6420	reflection,dynamic proxy
commons-fileupload	1.3.1	CVE-2016-1000031,CVE-2013-2186 CVE-2016-7462,CVE-2016-6793	File operation
JRE	Java 7 update 6, Java 6 update 18 Java 5.0 Update 23, and 1.4.2_25, JRE 1.7u21	CVE-2012-4681, CVE-2010-0840	reflection(Field),dynamic proxy
groovy	2.3.9	CVE-2016-6814, CVE-2015-8103 CVE-2015-3253	dynamic proxy
hibernate-core	4.3.11.Final	none	reflection
jboss-interceptor-core	2.0.0.Final	none	reflection
json-lib	2.4	none	dynamic proxy
javassistWeld1	javassist 3.12.1.GA and weld-core 1.1.33.Final	none	reflection
jboss application server	4.xx and 5.xx	CVE-2013-4810	reflection
jython-standalone	2.5.2	CVE-2016-4000	dynamic proxy
rhino-js	1.7R2	none	reflection
rome	1.0	none	reflection
spring-core	4.1.4.RELEASE	CVE-2011-2894	dynamic proxy
URLDNS	java.net.URL	none	dns look up
wicket-util	6.23.0	CVE-2016-6793	File operation

All detailed critical edges and stack traces can be found at [8]. We also report

[10] `https://bitbucket.org/Li_Sui/java-vulnerabilities`, accessed 18 December 2020
[11] `https://github.com/frohoff/ysoserial`, accessed 18 December 2020

which dynamic features that the vulnerability applies to. GONDVV (also known as Java Facepalm) [53] is a known vulnerability in early Java 7 update 6. It allows remote attackers to bypass the security manager. Similar vulnerabilities were identified in Java 6 update 18, 5.0 Update 23, and 1.4.2_25 as well – attackers are able to disable the security manager via reflection. Another use of reflection to execute an abstract command would be the case found in *commons collections*. The stack trace is shown in Listing 5.5 and the critical edge can be presented as follows:

```
java.lang.reflect.Method#invoke() -> java.lang.Runtime#exec()
```

Eight out of 19 programs are identified as exploiting through dynamic proxy. For instance, the following critical edges in *spring-core:4.1.4.RELEASE*:

```
com.sun.proxy.$Proxy1#newTransformer() ->..InvocationHandler[12]#invoke()
```

5.6 Threats to Validity

There are a few potential issues with the extraction process. We might have missed some stack traces that are formatted in unusual ways - for instance, stack traces produced by log frameworks that allow custom stack trace formatting. This might have given us some more results.

There are several issues that could have caused false positives. Firstly, stack traces only contain method names, but neither signatures nor descriptors. This can introduce false positives when methods are overloaded. Secondly, imprecise parsing could have produced false positives. We sampled 100 results to validate the correctness of parsed stack traces, 17 false positives were found. Thirdly, stack traces lack version information, although in some cases version information can be found on the enclosing web sites. This means that we might have extracted edges not present in the version of the program we statically analysed. We have mitigated this issue by running a script that matched the line numbers found in stack traces against program versions, and then selected the best matched version for analysis.

We addressed all of the issues related to precision by manually checking the 15 edges obtained against the version of the source code of the program we analysed with *Doop*.

[12] org.springframework.beans.factory.support.AutowireUtils$ObjectFactoryDelegatingInvocationHandler

5.7 Summary and Conclusion

RQ3: **Can information obtained from stack traces improve the soundness of static analysis?** The short answer is yes. We found 15 unique edges that link methods via reflection, and 11 of them are missed by the static analysis tool, even with reflection analysis enabled, *Doop* will find only 4 of the edges we extracted. We argue that while our analysis does not provide a large number of call graph edges that can enhance the static analysis, it is useful to retrieve interesting (and in this sense, high-quality) edges that can point to the weaknesses of static analysis tools. For example, we have included the *Log4J* scenario in our benchmark, which can be used to assess static analysis tools on inconsistent behaviours across JVMs. Moreover, stack traces reflect program failures which can add practical meaning to those dynamic language features. It seems more likely that a programmer encounters an exception or error if the software is used in a way that was not intended by the programmer, e.g., by bypassing program invariants or boundary checks, but those are exactly the cases of interest to static analysis as it has the ambition to discover those cases in order to reveal bugs and vulnerabilities.

One could argue that the use of hybrid analysis [41, 84] addresses problems with (un-)soundness. The main challenge is to create drivers (harnesses) that exercise the unsound parts of a program. The use of test case generation/fuzzing techniques for this purpose is promising [61]. It seems that hybrid analysis techniques can mitigate, but not solve the problem. Extending this study by cross-referencing the call graphs with the call graphs produced by *Tamiflex* or similar tools is an interesting and relevant topic.

An extended study with a larger dataset is an interesting topic for future research. One particularly interesting issue is the study of call graphs that cover multiple projects and libraries, including frameworks known for their heavy use of reflection (plugin-based systems, dependency injection) and the Java core libraries. We have noticed a large number of reflective invocations where the call site and target were in different libraries. One potential problem here is that it is still challenging to build comprehensive and sufficiently precise static models for real-world programs that include all library dependencies, although new algorithms and tools are under development to address scalability issues [59].

Chapter 6

Recall of Static Call Graph Construction

6.1 Introduction

In Chapter 4, we presented a micro-benchmark of a number of dynamic language features (and usage patterns) in Java. Those features and patterns were used to identify certain program behaviours that can be potentially missed by standard static analysis frameworks such as *Soot*, *WALA* and *Doop*. As discussed in previous chapters, modelling dynamic language features in Java via static analysis is a challenge, even for comprehensive static analysers. In this chapter, we present the results of an in-depth empirical study into the unsoundness of static program analysis. The main objective is to provide a **quantitative measure** of this unsoundness using various static analysis algorithms and configurations for **real-world** programs. Rather than considering the outcome of an analysis as sound or unsound in a binary form, we aim to measure the "degree of soundness" – we use the term **recall**, which measures the percentage of all known method invocations present in the statically constructed call graph. We refer to the known method invocations as the **oracles** as they are obtained by a dynamic analysis (i.e., testing). The known method invocations that are not present in the static call graph are referred to as **false negatives (FNs)**. Two sets of oracles are used to measure the recall level: 1) manually created tests and 2) automatically (programmatically) created tests. Furthermore, we also investigate whether gaining precision (by employing a context-sensitive analysis) could increase the level of recall.

6.2 Methodology

The study is empirical in natural. 31 real-world programs are analysed. We have considered a number of options for a suitable program corpus, such as *DaCapo 2009* and

SPECjvm 2008, however, we determined our previous dataset, the *XCorpus*, to be the ideal match for our experiment. Reasons for choosing *XCorpus* are discussed in Section 6.2.1. The experiment consists of a comparison study of statically constructed call graphs and dynamically generated call graphs. Dynamically generated call graphs are obtained by means of instrumentation, injecting an observer into a running program. Statically constructed call graphs are generated via a combination of analysis strategies and setups: context-sensitive/context-insensitive, full/light-reflection support. We proposed two input configurations: (1) pack the program and its dependencies in one .jar file. (2) separate them in different .jar files. The reason to have a such setup is to gain insight into how dependencies are handled under whole-program analysis. We have designed a traversable and context-sensitive data format for the oracles. This is presented in Section 6.2.2.

6.2.1 Dataset Selection

There are several datasets available to assist empirical studies into programming language and software engineering research. One of the most widely used benchmarks is *DaCapo 2009* [38] – a set of open source, real-world Java programs with non-trivial memory loads. *DaCapo* provides a customizable harness to execute the respective programs. *DaCapo* contains 11 programs: *antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd and xalan.* The key purpose of this benchmark is for comparison of results of empirical studies, e.g., to compare the performance of different JVMs.

SPECjvm 2008 [168] is a multi-threaded Java benchmark focusing on the performance of the JRE. It includes 3 executable synthetic datasets: Java Grande [178], Ashes [64] and Jolden [45]. Both *DaCapo 2009* [1] and *SPECjvm 2008* have not been updated for a very long time and therefore the versions of programs in these datasets are outdated (prior to 2009 and 2008 respectively).

Qualitas Corpus[2] [177] provides a larger set of curated Java programs intended to be used for empirical studies on code artefacts. It consists of 112 programs, 754 versions in total. However, there is no harness provided to exercise them (some programs contain test cases).

XCorpus is developed by Dietrich et al. [61]. It provides an executable version of the *Qualitas Corpus*. The dataset was designed with the goal of providing high code coverage. The dataset contain a variety of real-world programs – it contains 70 programs that

[1] DaCapo has updated the benchmark recently.`https://github.com/dacapobench/dacapobench` [accessed:15/06/2020]

[2] QualitasCorpus version 20130901

were ported from *Qualitas Corpus*[3] and it also provides support to extend the dataset by adding new programs. The current version has 6 additional programs [4] which have been selected based on popularity and the use of dynamic language features. Programs have been chosen from different domains – networking programs (Apache JMeter and tomcat), collection libraries (common-collection and Guava), logging framework (log4j), template engine (Velocity), bug detection tool (findbugs), bytecode engineering tools (Aspectj and ASM) and data mining tool (weka). Among the programs in the dataset, there are 28 programs that pack synthetic tests with them, which are created manually by developers. We refer to them as "built-in tests".

XCorpus also contains tests generated programmatically. We refer to them as "generated-tests". We use *Evosuite* [77] – a search-based test code generation framework. It is guided by coverage criteria when generating tests [161]. As mentioned before, the goal of the dataset is to achieve a high level of coverage. The average branch coverage of generated tests of the 75[5] programs in the dataset is 55.86%. For the 28 programs that have built-in tests, the average branch coverage is 34.42%. This can be considered a reasonable level, but not high. In comparison, the average branch coverage for the programs in *DaCapo 2009* [38] is only 16.10%.

We decided to use the *XCorpus* dataset in our experiment for the following reasons:

1. It is based on the widely used *Qualitas Corpus* that consists of a large curated (and representative) set of real-world Java programs.

2. It has programs with built-in and generated tests with high coverage.

3. Programs in the *XCorpus* use several dynamic language features.

4. The dataset has been recently used in other related works [157, 73, 75].

The file structure of *XCorpus* is shown in Figure 6.1. The `/data` folder contains two datasets: programs from the *Qualitas Corpus* and *corpus-extension* which is an extension from the original *Qualitas Corpus*. Each program has two sub-folders: 1) `../.xcorpus` contains build scripts, reports and generated-tests in source code, and 2) `../project` includes program binaries and program resources. The `../.xcorpus/exercise.xml` is an Ant[6] script to run tasks such as compiling and running built-in/generated tests. Program dependencies are managed by Ivy[7]. The corresponding script is `../.xcorpus/ivy.xml`. Its task is to download dependencies from the Maven

[3] QualitasCorpus version 20130901

[4] xcorpus-extension version 20170313

[5] ASM is removed from the dataset as we use it to process byte code.

[6] Ant is an automating build tool for compiling and running Java applications `https://ant.apache.org/` [Accessed: 20/06/2020]

[7] `https://ant.apache.org/ivy/`, accessed 20 June 2020

repository[8]. The folder `/tools` contains scripts for adding new programs, exercising the entire corpus and dependencies for the *XCorpus*. The *DaCapo 2009* dataset is also included for the purpose of comparing branch coverage, a script is used to calculate the branch coverage of the dataset (`/misc/dacapo-9.12/build.xml`).

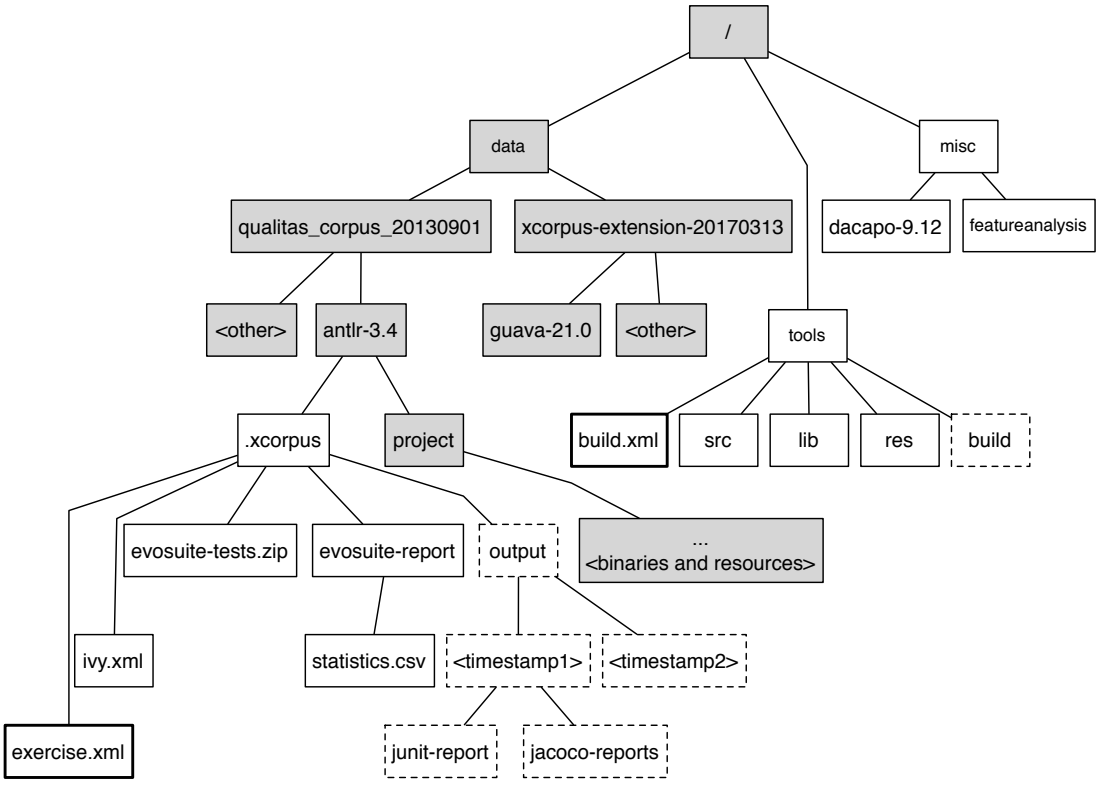


Figure 6.1: XCorpus structure

Programs in the *XCorpus* are chosen from a variety of domains, therefore they provide a wide range of programming features that are valuable for program analysis. We included a tool in `/misc/featureanalysis` that can be used to identify programming features presented in the *XCorpus*. These features are extracted by means of byte code analysis. We group these features under 11 categories:

[8] `https://mvnrepository.com/`, accessed 20 June 2020

- dynamic proxy
- reflection
- dynamic class loading
- invokedynamic instruction
- Java generics
- weak reference
- threads
- annotation
- system functions
- misc (synthetic/bridge method)

Each category includes a number of features. The features were selected based on the following criteria:

- It must be possible for a bytecode analyser to locate the feature. For instance, the use of auto-boxing or annotations with a `SOURCE` retention policy cannot be detected in the byte code.
- The currentness of a feature. For instance, the use of lambdas/invokedynamic is of interest, as lambdas were only introduced in Java version 8.
- The point of interest. In this experiment, we are interested in the impact of dynamic features for static analysis.

We distinguish between programs from the *Qualitas Corpus* and programs from the extension (i.e., newly added programs), and report the total number of programs in which a feature occurs ("p.count" in Table 6.1), and the average of occurrences for all programs that contain the respective feature at least once ("avg"). From Table 6.1 we can see that 62/75 programs have reference to `java.lang.reflect.Method#invoke` and 13/75 program have reference to `java.lang.reflect.Proxy#newProxyInstance`. This indicates that they contain reflection and dynamic proxy features. None of the programs in *Qualitas Corpus* contain the `invokedynamic` instruction, but 3 programs in the corpus-extension have reference to `invokedynamic`. They are *guava*, *jasperreports*, and *drools*.

Table 6.1: Features in programs included in the XCorpus

		Qualitas Corpus		corpus-extension	
category	feature	p.count	avg	p.count	avg
dynProxies	java.lang.reflect.InvocationHandler	10	1.40	3	1.67
dynProxies	java.lang.reflect.Proxy#newProxyInstance	10	1.60	3	1.33
reflection	java.lang.reflect.Method#invoke	47	15.60	5	9.00
reflection	java.lang.reflect.Constructor#newInstance	37	5.05	5	6.60
reflection	java.lang.reflect.Field#get*	29	3.83	4	6.25
reflection	java.lang.reflect.Field#set*	12	3.75	3	3.00
reflection	java.lang.Class#newInstance	58	11.50	4	20.75
reflection	java.beans.Introspector#*	13	3.62	2	4.00
reflection	java.util.ServiceLoader#*	0	n/a	1	2.00
reflection	java.io.ObjectInputStream#*	39	22.79	5	26.40
reflection	java.beans.XMLDecoder#*	5	4.60	0	n/a
classloading	java.lang.ClassLoader#*	51	19.51	6	21.00
classloading	java.security.SecureClassLoader#*	2	2.50	0	n/a
classloading	java.net.URLClassLoader#*	20	4.80	1	2.00
classloading	java.rmi.server.RMIClassLoader#*	0	n/a	0	n/a
invokedyn.	java.util.function	0	n/a	1	12.00
invokedyn.	java.util.function.*#*	0	n/a	2	59.50
invokedyn.	call site of java.util.invoke.*#*	0	n/a	0	n/a
invokedyn.	invokedynamic call site	0	n/a	3	430.67
generics	generic method signature	34	446.26	4	2662.50
generics	generic type signature	32	92.78	4	498.50
generics	generic field signature	33	223.27	4	536.75
generics	generic local variable signature	27	944.85	4	4941.75
dynlang	javax.tools.JavaCompiler#*	0	n/a	1	2.00
dynlang	javax.tools.ToolProvider#*	0	n/a	1	1.00
dynlang	javax.script.*#*	0	n/a	1	37.00
reference	java.lang.ref.WeakReference#*	23	13.04	4	7.50
reference	java.lang.ref.SoftReference#*	11	13.55	2	4.00
reference	java.lang.ref.PhantomReference#*	3	1.00	1	3.00
reference	java.util.WeakHashMap#*	19	10.42	2	1.00
threads	java.lang.Thread#*	47	12.96	5	4.40
threads	subclass of java.lang.Thread	39	10.31	1	6.00
threads	java.lang.Runnable	44	26.18	5	13.60
threads	java.util.concurrent.Executors#*	7	1.71	3	3.67
annotation	declares annotation	10	23.30	3	8.33
annotation	uses type annotation	16	78.38	4	235.25
annotation	uses field annotation	10	140.60	4	97.25
annotation	uses method annotation	20	196.10	4	484.25
annotation	uses type use annotation	0	n/a	0	n/a
annotation	uses type parameter annotation	0	n/a	0	n/a
system	native method definition	4	73.25	0	n/a
system	java.lang.Runtime#*	40	13.23	4	6.00
system	sun.misc.Unsafe#*	0	n/a	1	30.00
misc	synthetic method definition	68	233.71	6	993.00
misc	bridge method definition	35	115.43	4	1001.50

Of the 75 programs in the *Xcorpus*, we studied selected programs that have both built-in and generated-tests. Only 31 programs are included in our study (each contains at least one built-in test)[9]. The list of programs are shown in Table 6.2.

Table 6.2: Selected Programs from the XCorpus

program	**version**	**program**	**version**	**program**	**version**
castor	1.3.1	jFin_DateMath	R1.0.1	openjms	0.7.7-beta-1
checkstyle	5.1	jfreechart	1.0.13	oscache	2.4.1
commons-collections	3.2.1	jgrapht	0.8.1	pmd	4.2.5
drools	7.0.0.Beta6	ApacheJMeter_core	3.1	quartz	1.8.3
findbugs	1.3.9	jrat	0.6	tomcat	7.0.2
fitjava	1.1	jrefactory	2.9.19	trove	2.1.0
guava	21.0	log4j	1.2.16	velocity	1.6.4
htmlunit	2.8	lucene	4.3.0	wct	1.5.2
informa	0.7.0-alpha2	marauroa	3.8.1	weka	3.7.9
javacc	5.0	nekohtml	1.9.14	mockito-core	2.7.17
jena	2.6.3				

We use JACOCO[10], a code coverage tool, to measure branch coverage of all programs. Figure 6.2 shows the branch coverage results for each program with different test sets: built-in, generated and combined tests. The violin plot in Figure 6.3 demonstrates the distribution shape of branch coverage. In general, generated tests show a better coverage than built-in tests. However, combining both generated and built-in tests resulted in a significant increase in the overall coverage. This indicates that generated and built-in tests may exercise different parts of the programs.

[9] ASM has built-in test but it is the tool we used to process byte code, therefore cannot be included.
[10] `https://www.eclemma.org/jacoco/`, accessed 20 June 2020

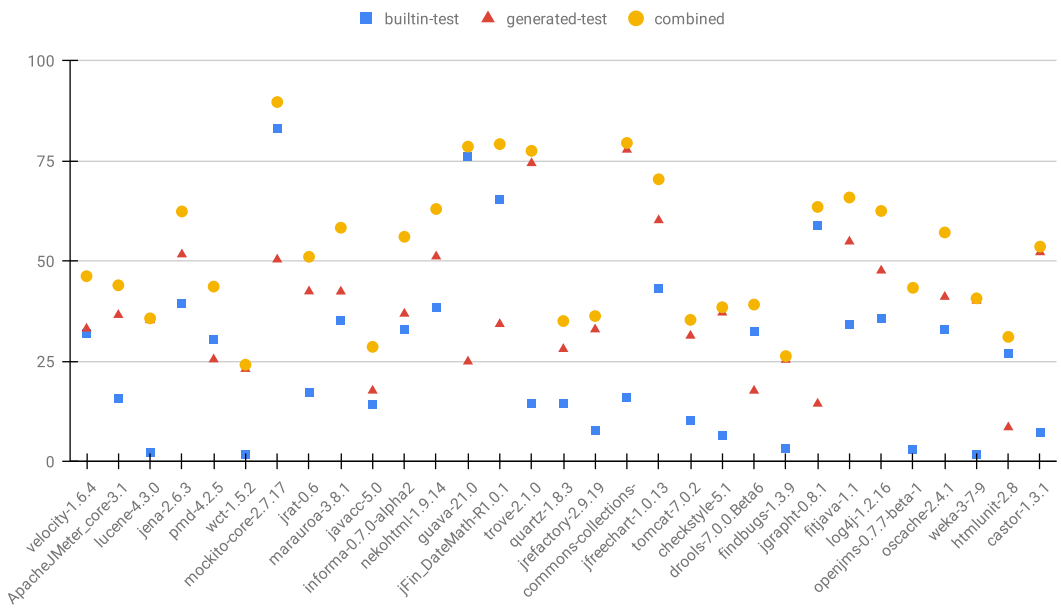


Figure 6.2: Branch coverage obtained for each programs by executing built-in, generated and combined tests in percentage

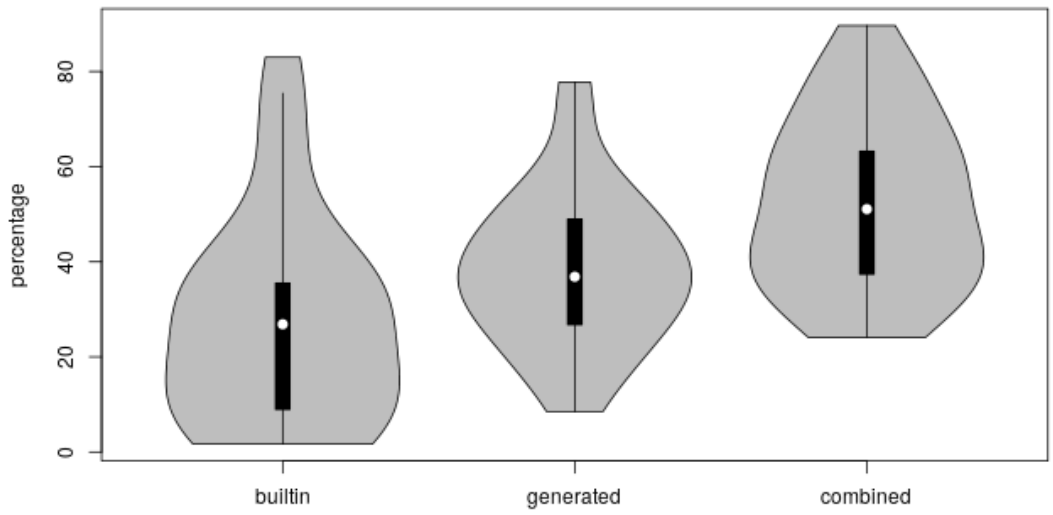
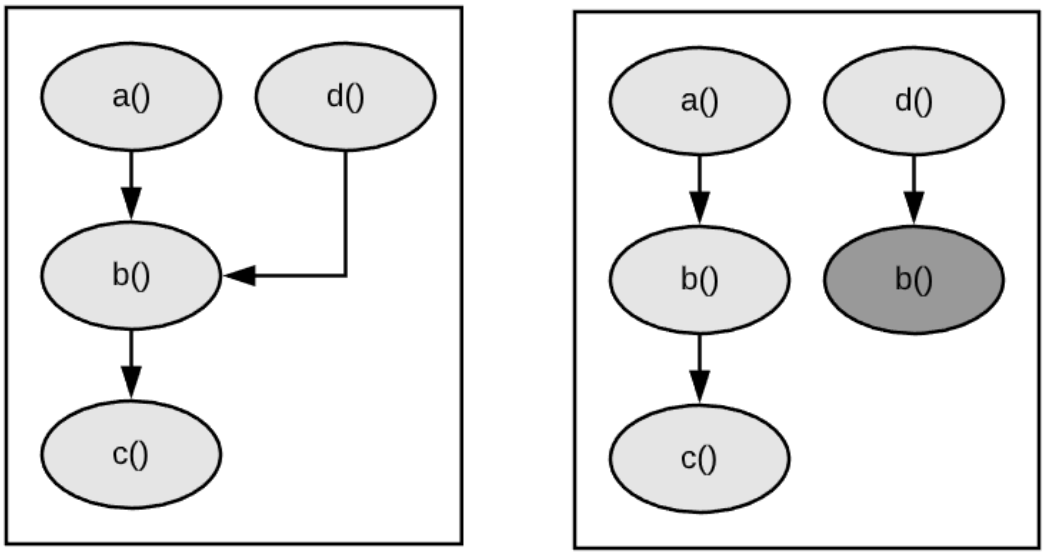


Figure 6.3: Branch coverage obtained for all programs by executing built-in, generated and combined tests in percentage

6.2.2 Context Call Tree (CCT)

We use a tree-like structure (i.e., CCT) [29] to represent program behaviours harvested at run-time. A vertex in the CCT is a method invocation. The root of this tree is an entry point, such as the static "`main`" method of Java programs. Initially, we chose to present the oracle in a call graph fashion – based on pairs of vertices and an edge in between. However, we quickly realised that transitive closure over a graph would lead to an imprecise analysis. In the case of a method being invoked twice, Figure 6.4(a) illustrates the graph that is constructed without distinguishing two calling contexts: the method `c()` is reachable by `a()`, `b()` and `d()`, whereas `c()` should not be reached by `d()`. We can compute the correct relationship by identifying the calling context as shown in Figure 6.4(b), the invocation of `b()` from `d()` is differentiated from `a()` to `b()`. Using a CCT, it is easier to reconstruct the method stack which enables us to precisely model method invocations. The downside of using CCTs is that the call tree size will expand quickly as every execution path must be recorded. An example of such a case is a loop which invokes a method multiple times.



(a) without context (b) with context

Figure 6.4: Call Graph construction for edges (a,b) (b,c) and (d,b)

The recorded data format is defined in the Table 6.3. We use *commas* and *tabs* as separators. The field "`kind`" indicates the method is either a native or a none-native invocation (to be discussed in Section 6.2.4.1). The "`tag`" reflects which invocations we tagged – features like reflective calls, dynamic allocations, dynamic access are tagged (more details are provided in Section 6.2.4.6). In multi-classloader environments the

plain name of a class does not unambiguously identify a class [6]. We use the field "`classLoaderName`" to differentiate between classes loaded by different class loaders. The fields "`threadID`" and "`threadObjectHash`" are used to deal with concurrent programs – as each vertex is written to a file sequentially, it is important to group together methods that belong to different threads. The field "`depth`" refers to position on the method stack. Note that 0 indicates the root of the CCT.

Table 6.3: CCT format definition

field	**description**	**example**
classLoaderName	the current class loader name	sun.misc.Launcher$ AppClassLoader@18b4aac2
className	fully-qualified class name	nz.ac.massey.Foo
methodName	the name of the method	putList
parameterType	the method takes java.util.ArrayList as an argument	(Ljava/util/ArrayList;)
returnType	no return value, otherwise the type is the same as parameterType	V
kind	not a native invocation	none-native
tag	not tagged by object allocation or filed/array access	noTag
threadID	returned by java.lang.Thread#getId()	11
threadObjectHash	returned by java.ang.System #identityHashCode()	1304359947
methodID	hash code returned by boost#uuid	1f6cb0d5-ddbf-468c-8c0a-012eb2979228
threadName	returned by: java.lang.Thread #getName()	main
depth	the position on the method stack	2

6.2.3 Confirming False Negatives

In order to quantify unsoundness of static analysis, we measure the recall of the analysis with respect to an oracle. For both the static call graphs (SCGs) and an oracle (set of context call trees), we can extract sets of reachable methods with a single traversal: for the SCG, this is just the set of vertices, and for the CCTs, this is the set of methods that occur in any invocation in any of the CCTs. For a given program, let SCGs and CCTs be those sets. The **recall** can then be defined as follows:

$$recall = \frac{|CCTs| \cap |SCGs|}{|CCTs|} \tag{6.1}$$

A sound analysis has a recall value of 1, whereas the presence of false negatives (i.e., methods that are observed when the program executes, but not computed as reachable by the static analysis) lowers the recall value. It is important to note that the recall measured here is relative to the oracle. The oracle itself is unsound as it does not reflect all *possible* program behaviours. We believe that this is the only practical approach to the problem as it is practically impossible to construct a driver that triggers all possible program behaviours except for trivial micro-benchmarks (note that: we have constructed a micro-benchmark for such a purpose in Chapter 4).

The metrics used to identify false negatives are based on call graph vertices (reachable methods), not on edges (an edge represents a calling relationship between two methods). We opted for this approach for the following reasons:

1. We have encountered method invocations by the JVM and they do not have visible call sites. A vertex-based approach allowed us to capture those methods as sources of unsoundness.

2. The recall measured with a vertex-based approach can be higher than the recall that would have been obtained with an edge-based approach. This makes our measurements conservative. The main takeaway of this experiment is that the recall observed in practice is relatively low, and this observation is very likely to remain valid even if we switched to an edge-based approach.

3. A vertex-based approach to study call graphs has been widely used in previous studies, examples include [121], [24] and [157].

4. There are several analysis clients relying on a vertex-based call graph reachability analysis, including dead code elimination [110] and static regression test selection [167].

6.2.4 Experimental Procedure

An overview of the process is shown in Figure 6.5. This study is based on the comparison of two models – a static model computed by means of a static analysis (SCG), and a dynamic model (CCT) that is constructed by observing a running program. The experiment consists of the following steps: pre-analysis, driver generation, program instrumentation and exercise, CCTs processing, SCG generation, CCTs tagging, statistics collection and graph construction. Each of these steps is explained in the following subsections:

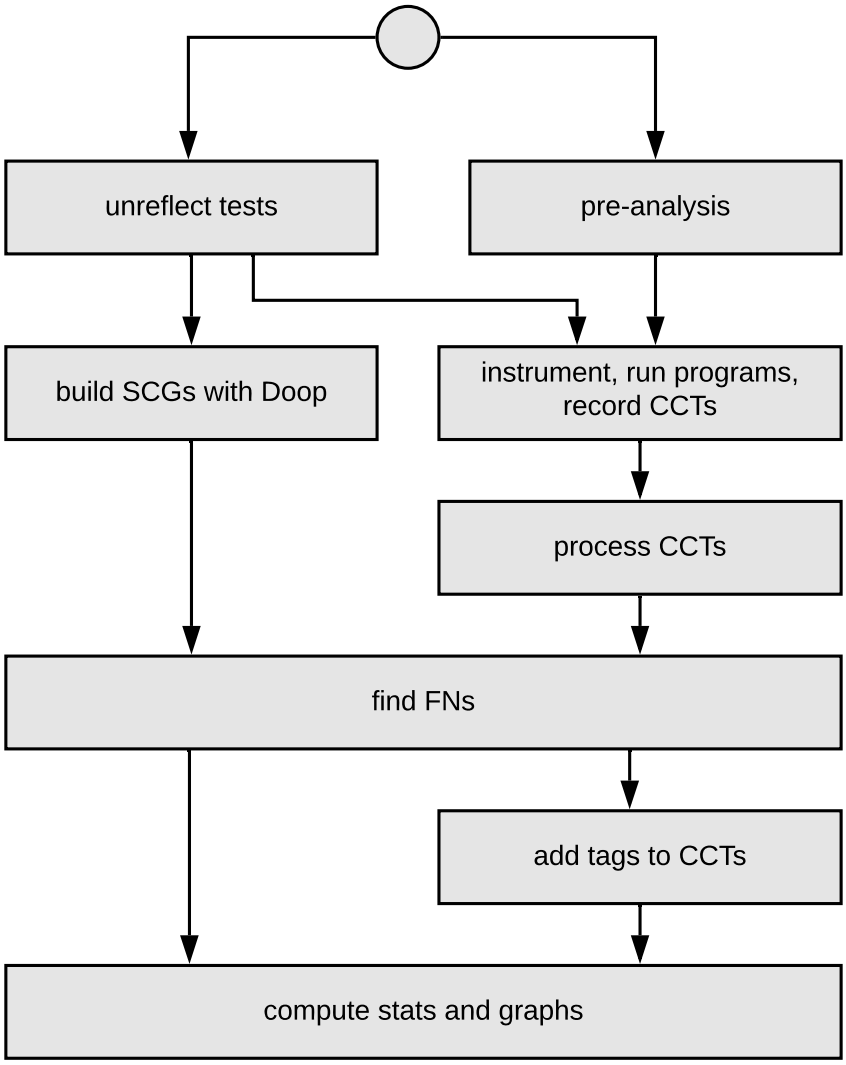


Figure 6.5: Study setup overview

The experiments conducted were extremely resource-intensive, both in terms of runtime and memory required. We attempted to mitigate the resource issues as follows:

- "horizontal" sampling: The scripts replicating each step of our experiments are implemented so that a parameter can be used to set whether to execute the script for all programs, a selected set of programs, or only for a single program.
- "vertical" sampling: The scripts rely on each other as some scripts require the data produced by others as input, the dependencies are part of the overall process illustrated in Figure 6.5. The dependencies are listed in Table 6.4 . In cases where running times are particularly long or a large memory size was required, we provide cached data that can be used to check the validity of a step without performing the entire processing pipeline for all input programs up to that point.

Table 6.4: Process dependencies and Resources requirement

process	**prerequisite process**	**expected run time**	**expected memory**
pre-analysis	none	10mins	16GB
unreflect tests	none	10mins	16GB
instrument, run program, record CCTs	pre-analysis, unreflect-tests	24days	16GB
process CCTs	pre-analysis, unreflect-tests, instrument-run-program	45days	256GB
build SCGs with Doop	unreflect-tests	18days	384GB
find FNs	pre-analysis, unreflect-tests	10mins	16GB
add tags to CCTs	pre-analysis, unreflect-tests, instrument-run-program,process-CCTs, find-FNs	12hours	64GB
compute stats and graphs	all above processes	12hours	64GB

6.2.4.1 Pre-analysis

Capturing native methods is tricky as the native methods in Java do not contain a method body, therefore, there is no corresponding byte code representation. The instructions used to invoke such methods are the same as for other normal method invocations such as `invokevirtual`, `invokespecial`, `invokestatic` or `invokeinterface`. Since there is no specific mechanism for capturing native invocations, we attempt to find a way to relate the call site to the native invocation – a pre-analysis is performed to record such call sites prior to the instrumentation. For example, in Listing 6.1, the call site `dispatchCall` at line 5 is associated with the native method `dispatchCall()` at line 12. We perform a static byte code analysis on the code to obtain information

about native methods such as package/class/method names and method descriptors. With such unique information, native call sites can be identified and resolved during the instrumentation, as shown in Listing 6.2. In rare cases, a virtual method can be resolved to both non-native and native methods – when multiple versions of the same method are presented. This could result some vertices not being recorded in the CCT.

Listing 6.1: Java Native method example

```
1  package nz.ac.masssey;
2  class Test{
3
4    public static void main(String[] args){
5      dispatchCall();
6    }
7
8    //a native definition:nz.ac.masssey.Test+dispatchCall+()V
9    public static native void dispatchCall();
10 }
```

Listing 6.2: Resolve Native call site

```
1  package nz.ac.masssey;
2  class Test{
3
4    public static void main(String[] args){
5      //this call site matches the native definition:
6      //nz.ac.masssey.Test+dispatchCall+()V
7      observer.push("dispatchCall");//instrumentation
8      dispatchCall();
9    }
10
11   //a native definition:nz.ac.masssey.Test+dispatchCall+()V
12   public static native void dispatchCall();
13 }
```

All information about native methods are recorded in a file in pre-analysis which will be loaded during the instrumentation phase, The data structure format is defined as follows: `packageName.className+methodName+(paramterType)ReturnType`. An example of the data extracted from the code in Listing 6.1 would be: `nz.ac.masssey.Test+dispatchCall+()V`.

6.2.4.2 Driver Generation

Xcorpus provides set of executable programs but the harness is built upon the *JUnit*[11]- a testing framework for Java. In general, *JUnit* test cases are detected (depending on the

[11] `https://junit.org/junit4/`, accessed 20 June 2020

version) by either the presence of annotations (*JUnit4*) or by naming patterns (*JUnit3*). *JUnit* then uses reflection to invoke respective methods. This is a significant issue as it is the very problem that we are investigating: how reflection is being handled by static analysis tools. If the static analysis tool fails to detect the entry point for a program, then the entire program model will be neglected. We also consider the *JUnit* framework not to be relevant to actual program behaviour (for the programs under analysis). It is likely that the analysis will be impacted as *JUnit* will add additional vertices and edges to the oracle. In particular, we remove invocations of `assert*` methods from tests as this invokes methods belonging to *JUnit*. We also remove *Evosuite* dependencies such as *Evosuite* scaffolding classes and annotations of `org.evosuite.*`.

We refer to the process for creating drivers as "unreflecting" the tests (*JUnit* dependency has been removed therefore no reflection is being used in the tests). Both *JUnit3* and *JUnit4* conventions are supported. Listing 6.3 and 6.4 demonstrate *JUnit3* and *JUnit4* test fixture, respectively. Listing 6.5 shows the corresponding driver code. The unreflected code for each test case runs in its own exception handler to ensure that test cases resulting in exceptions or catchable errors will not prevent the execution of the subsequent tests. This would fail if these tests resulted in uncatchable throwables, such as out of memory errors, preventing following unreflected tests to execute.

Listing 6.3: JUnit4 test case

```
1  public class Test42 {
2    private T tested = null;
3    @BeforeClass
4    public void beforeClass() {};
5
6    @Before
7    public void setUp(){
8      tested = new T();
9    }
10
11   @Test
12   public void test(){
13     tested.foo();
14   }
15
16   @After
17   public void tearDown(){
18     tested = null;
19   }
20
21   @AfterClass
22     public void afterClass() {};
23 }
```

Listing 6.4: JUnit3 test case

```
1  public class Test42  extend junit.framework.TestCase{
2
3    private T tested = null;
4
5    public void setUp(){
6      tested = new T();
7    }
8
9    public void test(){
10     tested.foo();
11   }
12
13   public void tearDown(){
14     tested = null;
15   }
16 }
```

Listing 6.5: Unreflected JUnit3&4 test case

```
1  public class Driver_Test42 {
2
3    public static void main(String[] args){
4    try{
5      Test42 test = new Test42();
6      test.beforeClass();
7      try{
8        test.setup();
9        test.test();
10       test.tearDown();
11     }catch(Throwable e){
12       System.err.println(e.getMessage());
13     }
14     test.afterClass();
15     }catch(Throwable e){
16       System.err.println(e.getMessage());
17     }
18   }
19 }
```

JUnit has some features that are difficult for any analyser to capture by means of static byte code analysis. For instance, `Rules` is a mechanism that extends test functionality. *JUnit* provides a wide range of rules to control test execution, such as timeout, external resource and expected exception. Implementing those rules means that we have to interfere with the *JUnit* main functionalities, which is against the main idea of creating a light-weight driver generator. Furthermore, *JUnit* allows custom

rules to be created by implementing `org.junit.rules.TestRule` interface. This makes capturing such behaviour even harder. We also exclude the support for `Runner` which also can be customised by users. In summary, our driver generation technique supports the following *JUnit* features.

- Test methods annotated with `@Test`.
- Non-static fixtures annotated with `@Before` or `@After`.
- Static fixtures annotated with `@BeforeClass` or `@AfterClass`.
- Tests annotated with `@RunWith(Parameterized.class)`. An example is shown in Listing 6.6. The idea is to iterate parameters directly (by calling method `data()`) and then to pass them to the constructor where fields are to be initialised. The corresponding unreflective driver code can be found in Listing 6.7.
- Test methods in subclasses of `junit.framework.TestCase` complying to *JUnit3* test method conventions.
- Fixtures in *JUnit3*. For instance, `setUp()` and `tearDown()` are implementations in subclasses of `junit.framework.TestCase`.

Listing 6.6: JUnit4 parameterized test case

```
1  public class Test43 {
2
3    public int a;
4    public int b;
5
6    public Test43(int a, int b){
7      this.a=a;
8      this.b=b;
9    }
10
11   @Test
12   public void test(){}
13
14   @Parameters
15   public static Collection data(){}
16 }
```

Listing 6.7: Unreflected JUnit4 parameterized test case

```
1  public class Driver_Test43 {
2
3    public static void main(String[] args){
4    try{
5      for(Object[] data:Test43.data()){
6        Test43 driver= new Test43((int)data[0],(int)data[1]);
7        try{
8            driver.test();
9        }catch(Throwable e){
10         System.err.println(e.getMessage());}
11     }
12   }catch(Throwable e){
13     System.err.println(e.getMessage());}
14   }
15 }
```

6.2.4.3 Instrumentation and Program Exercising

In Section 2.2.2.1, we discussed the use of several instrumentation tools for Java. We chose to use ASM[12] for byte code instrumentation. The goal is to instrument all loaded classes (core JDK classes as well as application classes) to get a full view of the program behaviours. However, due to the complexity associated with this process, we have encountered the following problems when instrumenting all loaded classes.

1. Instrumenting all methods may end up in a loop. e.g., the instrumented code gets called, calls back into the type that instrumented, which in turns calls back into the instrumented code.

2. Native method has no method body and therefore it cannot be instrumented directly.

3. “noise" produced by the instrumentation agent. e.g., Method `Instrumentation.#retransformClasses()` retransforms classes that are already loaded by JVM. This retransformation triggers the `Instrumentation#transform()` method which does an installation of the new definition of the class. As a result, the call tree spawned by method `Instrumentation#transform()` is added to the oracle and impacts the result. This behaviour is part of the observer, therefore, it cannot be seen as part of the program behaviour.

4. Throwing an exception can disrupt the logging process – i.e., current method is never popped from the method stack when an exception occurs.

[12] `http://asm.ow2.org/`, accessed 20 June 2020

To deal with the call loop issue, we implement an observer in C++. In fact, the entire logging function has been written in C++ to present call back to the instrumented code. The observer is responsible for logging method stacks. We created 4 functions, each has its corresponding Java API defined below:

- `push(String classLoaderName,String className,String methodName,String descriptor,String invocationID,int kind,int threadObjectHashCode,long threadID,String threadName,int objectHashCode)`: push the current method to method stack.
- `pop(int threadObjectHashCode,long threadID)`: pop the current method from method stack.
- `clear(String classLoaderName,String className,String methodName,String descriptor,String invocationID,int threadObjectHashCode,long threadID)`: clear method stack when an exception occurs.
- `addAllocationHash(int objectHashCode,int allocationType)`: record tagged objects. See section 6.2.4.6 for detail discussion.
- `flush()`: output to a file when program exits. A `ShutdownHook` is registered to do this job.
- `getInvocationID()`: An unique ID is produced for each method invocation for the purpose of tracing context.

We also created a blacklist of classes and methods that could be considered to cause a loop call. The list includes the following classes and methods:

- the observer classes and their dependencies: `nz.ac.massey.cs.instrumentation*`, `org.objectweb.asm.*`, `module-info*` (ASM classes).
- JDK methods (referenced by the observer): `java.lang.Thread#getName()`, `java.lang.Thread#getId()`, `java.lang.Thread#<init>`, `java.lang.Thread#<init>`, `java.lang.Object#<init>`. Note that `java.lang.System#identityHashCode()` is used by the observer as well, but we did not include it in the list as it is a native method, and therefore it cannot actually make a call loop (no method body).

Threads are operated based on independent method stacks. Therefore, each thread is assigned to a unique method stack. In order to differentiate thread context (a method could be invoked multiple times within different thread groups), we chose to use `java.lang.System#identityHashCode(Thread.currentThread())` to create an identifier for the method based on the current running thread.

Native methods cannot be instrumented directly. Therefore, we inject the observer for native invocations directly around the call site, instead of at the method entry/exit. As shown in the Listing 6.8, the observer is injected for method "main" at the entry point at line 4 and exit point at line 8, whereas injection points for the native invocation are at lines 5 and 7.

Listing 6.8: Instrumenting a call aite for Native invocation

```
1  class Main{
2
3    public static void main(String[] args){
4          observer.push("main");//instrumentation
5          observer.push("dispatchCall");//instrumentation
6          dispatchCall();
7          observer.pop("dispatchCall");//instrumentation
8          observer.pop("main");//instrumentation
9    }
10
11   public static native void dispatchCall();
12 }
```

We consider the calls that are not related to the program as "noise". For instance, the instrumentation agent produces extra calls to the instrumentation API. We chose to remove the call tree with a root invocation of `sun.instrument.InstrumentationImpl #transform()`. Furthermore, we added a check to start logging at the program entry point (static main method with a descriptor that matches `[Ljava/lang/String;)V`). This program entry point is defined in Section 6.2.4.2 and it is a fixed point for all programs.

We have to treat exceptions as a special case, because it is possible that an exception will be raised before popping the method from the stack. We therefore inserted a statement to keep popping the method from the stack up to the point where the exception is being handled. In the example shown in Listing 6.9, method `clear()` is injected at line 11. When an exception is raised within method `b()` at line 12, the catch block is then being executed and method `b()` is popped from the stack before continuing to method `d()`. Note that our approach does not track unhandled exceptions. While this can be done by instrumenting the `uncaughtException` method in all classes implementing an exception handler `Thread.UncaughtExceptionHandler`, this was not necessary in our case, since the JVM specification states that "If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the thread in which the exception was thrown is terminated" [126, sect 2.10]. For example, there is no further manipulation of the respective stack that may lead to the recording of any additional invocations.

Listing 6.9: Dealing with exception

```
1  class Main{
2
3    public static void main(String[] args){
4      a();
5    }
6
7    public static void a() {
8      try {
9          b();
10     }catch(Exception e){
11         clear();//instrumentation
12         d();
13     }
14   }
15
16   public static void b(){
17     throw new Exception();
18   }
19 }
```

There is another scenario where an exception could be thrown within a recursive call (as shown in Listing 6.10). The problem arises in that the method stack does not actually know which method `recursion()` should be popped first. Therefore, it is important to label all methods, as a single method that could be invoked multiple times. In Listing 6.10, we use `getInvocationID()` at line 11 to create a unique ID and then reference the context to determine which "`recursion`" the method currently is at.

Listing 6.10: Instrumenting the exception within a recursion

```
1  class Main{
2
3    static int count=0;
4
5    public static void main(String[] args){
6      recursion();
7      foo();
8    }
9
10   static void recursion() {
11     int id=getInvocationID();//instrumentation
12     count++;
13     if(count<=3) {
14         try {
15           recursion();
16         } catch (Exception e){
17           clear(id);//instrumentation
18         }
19     }
20     if (count==4){
21         count++;
22         throw new Exception();
23     }
24  }
25 }
```

6.2.4.4 Removing Duplicate Branches

One of the limitations of using CCTs is that they can quickly grow in size and become very large. In the case of our experiment, the raw CCTs data (before reduction) occupied 600GB in space. We therefore investigated several approaches to help reduce the size of the CCTs.

One approach we used is through implementing a simple loop reduction. When methods are invoked in loops, a new branch is created for each iteration. Often, these branches are isomorphic and therefore redundant: for each branch, the same methods are invoked in the same order. More precisely, we can define two branches with roots $v_1 = (method_1, id_1)$ and $v_2 = (method_2, id_2)$ as isomorphic as follows: if the vertices don't have successors, they are isomorphic if and only if $method_1 = method_2$. Otherwise, they are isomorphic if and only if $method_1 = method_2$ and the ordered lists of children are element-wise isomorphic. We remove redundant branches caused by loops. We use the following simple algorithm: traverse the tree to compute structural hashes from the invoked methods and the hashes of the successors, for all vertices, and then look

for siblings with identical hashes. For those candidate roots of isomorphic branches, a recursive structure is compared to avoid hash collision. In most cases, this reduced the size of CCTs dramatically, by an order of magnitude. Removing branches with certain roots does reduce the number of methods being recorded, and therefore has an impact on metrics computed later. However, these branches are not caused by method invocations that are visible to the static analysis, and therefore, not computing them cannot be considered a shortcoming of the analysis. We also encountered branches spawned by our instrumentation, with invocations of `sun.instrument.InstrumentationImpl#transform()` as root. Those branches were removed to ensure that the experimental setup did not bias the results.

6.2.4.5 Static Call Graph Generation

The static model is the static call graph (SCG), a directed graph (V, E) consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. Vertices represent methods, while edges represent invocation relationships. For our study, we used the *Doop* framework with different configurations to construct the call graph. *Doop* implements a wide range of algorithms including support for context sensitivity and several dynamic language features. This support is comparable to, or exceeds, similar features available in alternative frameworks such as *Soot* [112] and *WALA* [17], as demonstrated in Chapter 4 and a recent benchmark-based comparative study [158]. The *Doop* version used was `4.14.4`. We proposed to have two categories to study the impact of context sensitivity on recall: context-sensitive and context-insensitive analysis. Under context-insensitive analysis we include: base analysis (no advance analysis enabled), reflection analysis (*Doop* claims to handle reflection, dynamic proxies, method handles and native methods) and reflection lite analysis (a light-weight reflection analysis).

- base analysis: `context-insensitive`
- context-sensitive analysis: `1-call-site-sensitive`
- reflection analysis: `context-insensitive -reflection -reflection-classic -reflection-dynamic-proxies -reflection-method-handles -simulate-native-returns`
- reflection lite analysis `context-insensitive -light-reflection-glue -distinguish-all-string-constants -reflection-dynamic-proxies -simulate-native-returns`

In all cases, the `-main` option was used with the generated entry point as argument (discussed in Section 6.2.4.2). In order to run the static analysis, we needed the byte code of the program and the library the program depends on. This required us to first resolve the symbol references to dependencies in the *XCorpus* programs. For each

program, we used the `ivy resolve` in Ant task to fetch dependencies from Maven and, in some cases, local repository (under the folder `project` as shown in Figure 6.1), and made local copies of these libraries available for the static analysis. A crucial decision to be made when setting up the static analysis is the handling of libraries. We ran the static analysis in two modes:

1. **superjar mode**: all library classes are part of the analysis, this was done by building a single "super" jar containing all program classes as well as all library classes.

2. **library mode**: library code is handled differently by only representing the parts of the library used by the program. This is supported by *Doop*, but introduces some additional unsoundness. The main reason for this is that *Doop* relies on the facts *Soot* generates from (library) code, and if library code is only accessed through reflection or similar means, those fact sets will be incomplete. Even if *Doop* is used with reflection support, the analyses may still fail to generate some call graph edges.

Handling libraries and the main program differently is a widely used technique in static program analysis [25, 28, 156]. By investigating both settings, we are in a position to measure the impact that this has on the analysis recall.

The data structure that *Doop* produces does not confirm with our CCTs. We follow the ASM format [11] which, for example, defines a type for `ArrayList` as `Ljava/util/ArrayList;`. However in *Doop*, this is how it is presented: `java.util.ArrayList`. We implemented a parser to covert to our CCTs format. Table 6.5 summarises the types that have been converted.

Table 6.5: Type Format for CCT and Doop

type	**CCT**	**Doop**
object	Ljava/util/ArrayList;	java.util.ArrayList
primitive type	I	int
	B	byte
	J	long
	F	float
	D	Double
	S	short
	C	char
	Z	boolean
array	[[	[][]
other	V	void

6.2.4.6 Tagging CCTs

We analyse which particular language features are used to spawn branches within a CCT that contain invocations of methods which are not reachable in the statically constructed call graphs (false negatives). For some features, such as method invocations through reflection, this is a straightforward process: remove vertices corresponding to invocations of `Method#invoke()` from the CCT, and then count the removed vertices labeled as false negatives (with respect to a static analysis). This process is illustrated in Figure 6.6. It provides a measure of the actual impact that the presence of `Method#invoke()` has on the recall of the static analysis. We refer to dynamic features that can be detected through the presence of certain methods in the CCT as `dynamic invocations (DI)`. The main idea here is to **tag** dynamic invocations with a label corresponding to the language feature (such as `Method#invoke()`), and then to measure the percentage of invocations corresponding to false negatives in the CCT dominated [117] by the tagged vertices.

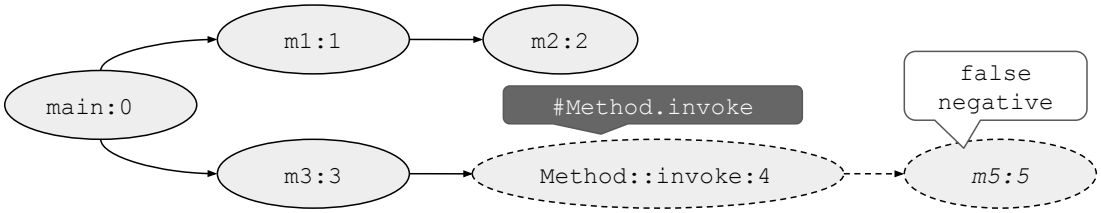


Figure 6.6: CCT cause analysis for dynamic invocations

Another common dynamic invocation pattern occurs when lambdas are compiled and the `invokedynamic` instruction is used. We track those invocations by taking advantage of naming patterns used by the OpenJDK compiler [80].

While we initially expected that dynamic invocations would explain most analysis false negatives, this was not the case. It turns out that `dynamic allocations (DALL)` also have a significant impact. An example is the use of `Class#newInstance()` (as shown in Listing 6.11). This dynamically creates an object `obj` of some type `T`, and the static analysis has to track method invocations `v#foo()` with `v` pointing to `obj`. If the object is not correctly tracked, then devirtualisation would not be modelled correctly, and the analysis result may contain false negatives.

Listing 6.11: Invocation of a method with a dynamically allocated object

```
1 void m3(String clsName) throws Exception {
2   T foo = Class.forName(clsName).newInstance();
3   foo.bar();
4 }
```

It became quickly apparent that tracking those allocations required additional instrumentation to enrich the CCT with additional information in the form of vertex labels. Figure 6.7 illustrates our approach used here, using the code snippet from Listing 6.11. Objects returned by dynamic allocation are hashed by the method `java.lang.System#identityHashCode(foo)` and the observer (presented in Section 6.2.4.3) uses the method `addAllocationHash(123)` to log these in a hash table. Whenever a method is pushed to the method stack, we hash the receiver (`foo` at line 3 in Listing 6.11) to see whether it points to the object returned by `Class#newinstance()`.

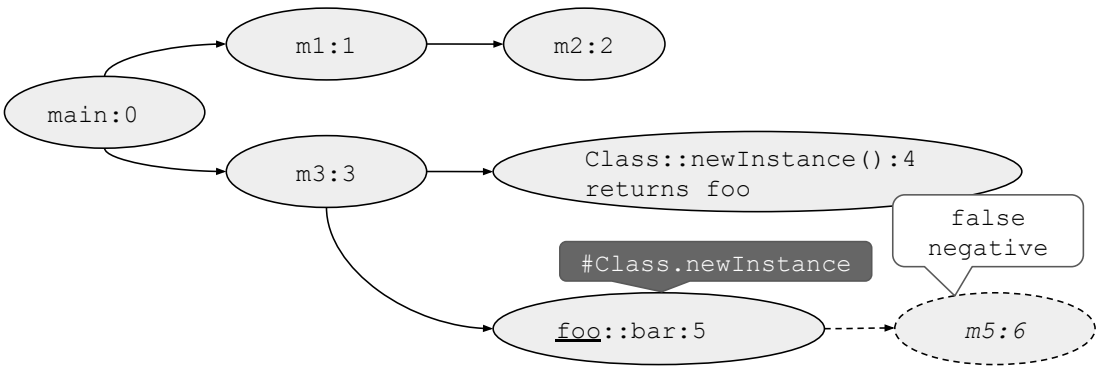


Figure 6.7: CCT cause analysis for dynamic allocations

This tagging process can be considered as a form of lightweight dynamic taint analysis [146], whereby objects are considered tainted when they are dynamically allocated. Note that we tracked the last dynamically allocated object. In particular, this matters when considering that `Class#newInstance()` calls `Constructor#newInstance()`. If an object has been created by `Class#newInstance()`, it is already marked as being created by `Constructor#newInstance()`. Therefore, when we tag an invocation with `Constructor.newinstance()`, this means that the application has created the object by invoking `Constructor#newInstance()` directly, not indirectly via the intermediate `Class#newInstance()`.

A situation similar to dynamic allocation arises when an object is accessed via reflection or similar means, such as a reflective heap access via `Field#get()`. We refer to this pattern as `dynamic access (DACC)`, and model it like dynamic allocation by tracking objects returned by the invocations of these methods. We also track objects

returned by native methods. They are also included in the `DALL` category, however, we do not track whether the objects returned are actually newly allocated objects, or are already known objects. So, there could be some cases of dynamic access in this category.

The next pattern we have encountered are false negatives caused by invocations without matching call sites in the program. There are methods that are only invoked by the JVM, in particular life cycle-related methods such as `ClassLoader#loadClass()`. For each invocation, we check whether there is a call site for this method in the parent method (i.e., the method of the parent vertex in the CCT) and if this is not the case, we tag the method with `nocallsite`. Note that tagging `nocallsite` is a post analysis which occurs after processing the CCTs (labelled as "add tags to CCTs" in Figure 6.5). Other tags are created during the program exercising phase. Closely related to `nocallsite` are methods that are called from system threads calling back into application code. Examples are invocations of `Object#finalize()` and user interface event handlers. There are a number of system threads that can be recognised by name. We use a special tag `systemthread` to tag the roots of the CCTs generated for these threads. We track the following threads: `Signal Dispatcher`, `AWT-EventQueue-0`, `Reference Handler`, `AWT-Shutdown`, `Finalizer` and `DestroyJavaVM`. Note that the naming of these system threads depends on the particular JVM implementation used in the experiments as it is not defined by the JVM specification. We categorise the invocations tagged with either `nocallsite` or `systemthread` as `system (SYS)`. Table 6.6 lists the tagged invocation patterns and their respective categories.

Table 6.6: Dynamic invocation, allocation and access patterns used for tagging

invocation pattern	**tag**	**category**
java.lang.reflect.Method#invoke	method.invoke	DI
lambda$	lambda	DI
java.lang.reflect.InvocationHandler#invoke	dynproxy.invoke	DI
java.lang.invoke.MethodHandle#invoke*	handler.invoke	DI
java.lang.Class#newInstance	class.newinstance	DALL
java.lang.reflect.Constructor#newInstance	constructor.newinstance	DALL
java.io.ObjectInputStream#readObject	deserialize	DALL
sun.misc.Unsafe#getObject	unsafe.getobject	DALL
objects returned by native methods	nativeallocation	DALL
java.lang.reflect.Field#get	field.get	DACC
invocations without call sites in program	nocallsite	SYS
roots of system threads	systemthread	SYS

Note that there might be multiple possible causes for a method not to be reachable in the SCG. If an invocation corresponds to a static analysis false negative, there might be multiple tagged invocations on the path connecting it to the root, offering multiple explanations as to why the respective method is unreachable. In fact, this does not

necessarily indicate that this classification yields falsely reported vertices as there might actually be multiple root causes that prevent the static analysis from computing a method as reachable.

6.3 Results

The main goal of the experiments conducted, that we present in this chapter, is to provide a quantitative measure of the recall of various SCG construction techniques with respect to different oracles. This led to a combinatorial explosion in the number of possible experiments: we have three types of static analyses configured in *Doop* (context-insensitive, context-insensitive with reflection support, and context-sensitive), three possible oracles (constructed based on built-in, generated, and combined test cases) and the additional parameter of whether to run the analysis in library or whole program (super jar) mode (see Section 6.2.4.5). This implies that 18 computationally expensive experiments had to be conducted and reported for each of the 31 programs, making both the execution and reporting challenging. To deal with this, we prioritise experiments as follows:

- Measure the recall of the baseline context-insensitive (base) analysis with respect to the oracles provided by built-in, generated and combined test cases, for both the library and the super jar configuration. The results reveal the recall with respect to different oracles and are reported in Section 6.3.2.
- The impacts of context sensitivity and reflection support were then assessed and are reported on in Sections 6.3.3 and 6.3.4.
- False negatives are further investigated in detail in Section 6.3.5.
- We restricted the experiments to always use the combined set of generated and built-in tests, and the library analysis mode.

6.3.1 Time and Resources

For all experiments, we used Java 1.8.0_144-b01 (Java HotSpot(TM) 64-Bit Server VM, build 25.144-b01, mixed mode), running on a Ubuntu 18.04. The heap size of the JVM was set to 16GB for the CCT recording, 256GB for the CCT reduction and 384GB for the static analyses. We report the running times of the respective experiments in Table 6.4. Full running time information for each program is provided in Appendix A.2. While the analysis of performance was not our main goal, performance is an important part of the trade-off that is being made when choosing a static analysis framework. Performance also did have an impact on our methodology with regards to measuring the recall. Note

the high cost of running the instrumented tests (not taking into account the already very high cost of generating tests, reported in [61]), and of running the static analysis with reflection support, with only 20 programs avoiding time outs (set to 6 hours). The timeout of 6 hours chosen is at the upper end of the time outs used in related work: [157, 121] – 90 mins, [124, 172, 122] - 3 hours, [73, 85] – 4 hours, [86] – 6 hours, [87] – 7 hours. The programs that did not time out with reflection support are shown in Table 6.7.

Table 6.7: Programs that did not time out with reflection support

program	**version**
checkstyle	5.1
commons-collections	3.2.1
findbugs	1.3.9
fitjava	1.1
informa	0.7.0-alpha2
javacc	5.0
jena	2.6.3
jFin_DateMath	R1.0.1
jfreechart	1.0.13
jgrapht	-0.8.1
jrat	0.6
jrefactory	2.9.19
marauroa	3.8.1
nekohtml	1.9.14
openjms	0.7.7-beta-1
oscache	2.4.1
pmd	4.2.5
quartz	1.8.3
trove	2.1.0
velocity	1.6.4

6.3.2 The Recall of Static Program Analysis

We measured recall for context-insensitive analysis. The recall values for this analysis are depicted in the violin plot in Figure 6.8. While in general the recall values (combined tests, the static analysis uses the lib setup) were high with a median of 0.884, the "unsoundness" gaps were still significant, indicating that the static analysis typically misses around 11% of the known reachable methods in regards to the 31 programs

analysed. We also computed the recall with respect to the oracles obtained by the built-in and generated tests separately. The recall with respect to the oracle obtained with built-in tests was significantly lower (median 0.859) than the recall obtained using the generated test oracle (median 0.904). This suggests an interesting characteristic of built-in tests - they are potentially better at penetrating code that uses dynamic language features than the generated tests. Note that this result was obtained with tests generated with one particular test generation framework - *Evosuite*. The likely explanation is that test case generators (in the case of *Evosuite*) are less likely to exercise dynamic language features, therefore the recall can be higher than built-in tests.

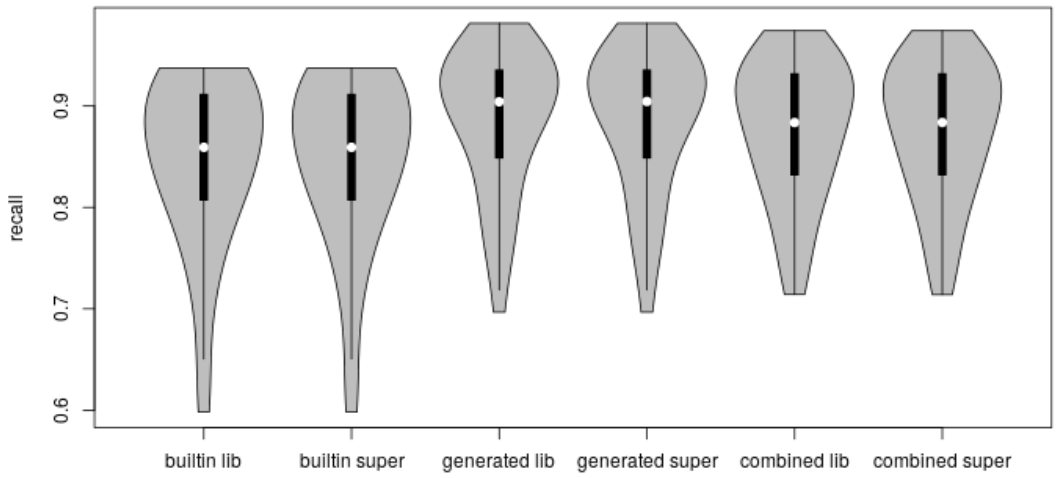


Figure 6.8: Recall of the base static analysis with respect to different oracles and configurations

Figure 6.8 also indicates that there is no significant difference between the library and the super jar analysis mode. This indicates that dynamic language features are not used at component boundaries where methods could flow from the program to the library through a dynamic invocation. The use of a plugin-like model in *JDBC 4* with service locators is an example of such a model [30, Section 9.2.1]. We note however that there are programming patterns that do exactly this, but none of the programs in our dataset use these.

We also investigated whether the false negatives are due to methods declared in core Java (methods that are declared in classes within `java.*` packages), extended Java (other official packages that are part of the Java Runtime Library, such as `javax.*`,

`org.omg.*`), Java private (`sun.*`, `com.sun.*`, `com.oracle.net`) or application-defined (everything else, including application and third-party library packages). The average percentages of false negatives in the respective categories are as follows: 22.25% Java core, 10.51% Java extended, 46.50% Java private and 21.47% application. The high number of methods defined in Java-private classes stands out. This is consistent with the results of the cause analysis discussed in Section 6.3.5.

6.3.3 The Impact of Context-Sensitivity

We measured recall with respect to the oracles created by executing all tests for both the base (context-insensitive) analysis and a context-sensitive analysis as described in Section 6.2.4.5. The results are depicted in the second column of Figure 6.9, the numbers in brackets indicate the size of the dataset used, the base analysis data are provided for both the full dataset (column 1) and the reduced dataset (column 3). The median recall is 0.880. It turns out that gaining precision has very little impact on recall. To be more specific, eliminating edges that are falsely reported by static analysis (false positives) does not contribute much to the recall level. There were very few false negatives that were covered by the false positives of the less precise context-insensitive analysis.

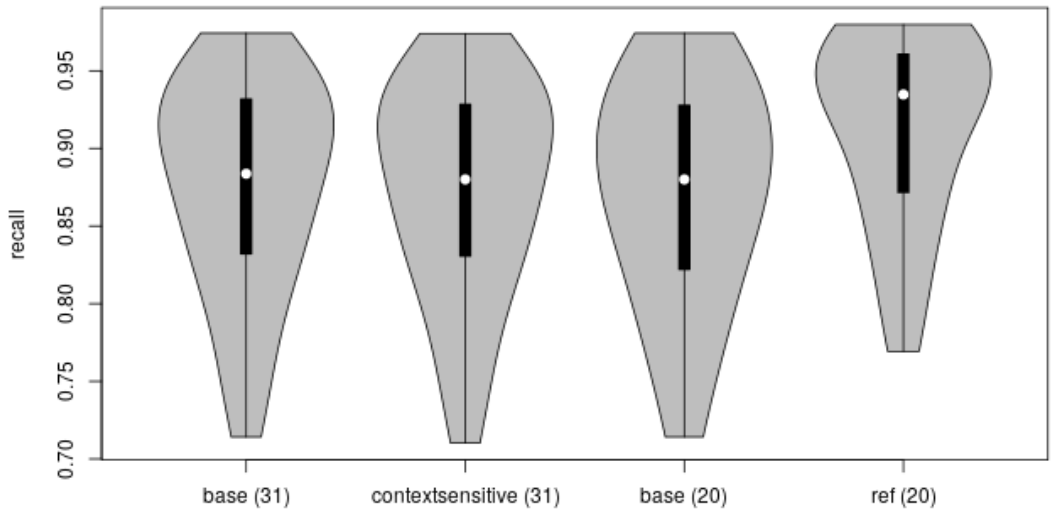


Figure 6.9: Recall of base vs context-sensitive analysis and reflection analysis

6.3.4 The Effectiveness of Dynamic Language Feature Support in Static Analysis

We compared the recall obtained by the base analysis with the recall obtained by analyses with reflection support being enabled. This allowed us to measure the effectiveness of state-of-the-art support for reflection and similar dynamic language features. Unfortunately, the additional reasoning *Doop* has to perform is resource-intensive and timed out for several programs, as detailed in Table 6.4. Therefore, the results summarised in columns 3 and 4 in Figure 6.9 were obtained with a smaller dataset consisting of only 20 programs[13]. In general, the reflection support in *Doop* is very effective - the median recall increases significantly from 0.884 to 0.935.

6.3.5 Quantifying the Causes of Unsoundness

Tagged vertices are removed from the CCTs to measure the percentage of false negatives (with respect to a given analysis) still reachable. The method has been described in Section 6.2.4.6. Table 6.8 shows the detailed classification of the false negatives left when running the static analysis with base and reflection support. Note that only 20 programs are under analysis due to time out in reflection support in others. The detail of each category is presented in Table 6.6.

Table 6.8: Detailed classification of FNs for the base analysis of the full dataset

category (tag)	**base (31)**			**base (20)**			**reflection**		
	avg	stdev	No.P	avg	stdev	No.P	avg	stdev	No.P
method#invoke	8.08	8.88	29	7.14	8.44	18	4.22	4.53	16
lambda	0.10	0.16	16	0.10	0.18	10	0.01	0.04	1
handler#invoke	1.45	1.98	24	1.73	2.42	14	2.98	4.13	14
dynproxy#invoke	0.42	0.80	14	0.21	0.56	5	0.26	0.81	2
class#newinstance	21.36	14.31	29	19.62	14.12	18	20.64	17.03	18
constr#newinstance	5.97	6.41	28	4.76	5.53	17	5.61	6.82	17
deserialize	0.01	0.06	3	0.02	0.08	2	0	0	0
unsafe#getobject	0.15	0.30	9	0.18	0.35	6	0.27	0.52	6
nativeallocation	40.00	12.67	31	41.81	12.7	20	36.24	14.41	20
field#get	0.08	0.40	2	0	0	0	0	0	0
nocallsite	52	22.51	31	47.27	19.34	20	52.49	20.57	20
systemthread	2.57	3.25	31	3.41	3.8	20	4.82	4.88	20
other	17.94	11.39	31	17.96	11.14	20	13.83	10.74	20

The results for the base analysis are presented in Figure 6.10. The figure uses the

[13] *checkstyle-5.1, commons-collections-3.2.1, informa-0.7.0-alpha2, findbugs-1.3.9, fitjava-1.1, javacc-5.0, jena-2.6.3, jFin_DateMath-R1.0.1, jfreechart-1.0.13, jgrapht-0.8.1, jrat-0.6, jrefactory-2.9.19, marauroa-3.8.1, nekohtml-1.9.14, openjms-0.7.7-beta-1, oscache-2.4.1, pmd-4.2.5, quartz-1.8.3, trove-2.1.0, velocity-1.6.4*

aggregated as well as detailed categories, also showing statistical variation. It summarises the percentages of false negatives that can be explained by the presence of the respective classes of language features across the dataset. It turns out that dynamic invocations are only a minor source of false negatives. In particular, the presence of `Method#invoke` can only explain less than 10% of the false negative cases. However, invocations triggered by methods invoked by the JVM and different types of dynamic allocations can explain the majority of false negatives. Note that the dataset consists of programs that were released before lambda support was provided for Java, and the lambda feature is likely to be under-represented (Table 6.1 contains an overview of the language features used by the *XCorpus* programs). The only programs where we found false negatives caused by dynamic access are *wct-1.5.2* and *guava-21.0*. Other categories that have overall little impact are allocations when objects are deserialised (0.01%), dynamic proxies (0.42%), and `Unsafe#getObject` (0.15%). However, there are 14 programs that use dynamic proxies and 9 programs that use `Unsafe#getObject`. More generally, we detected at least some usage of each of the features/patterns investigated when executing the programs in the dataset.

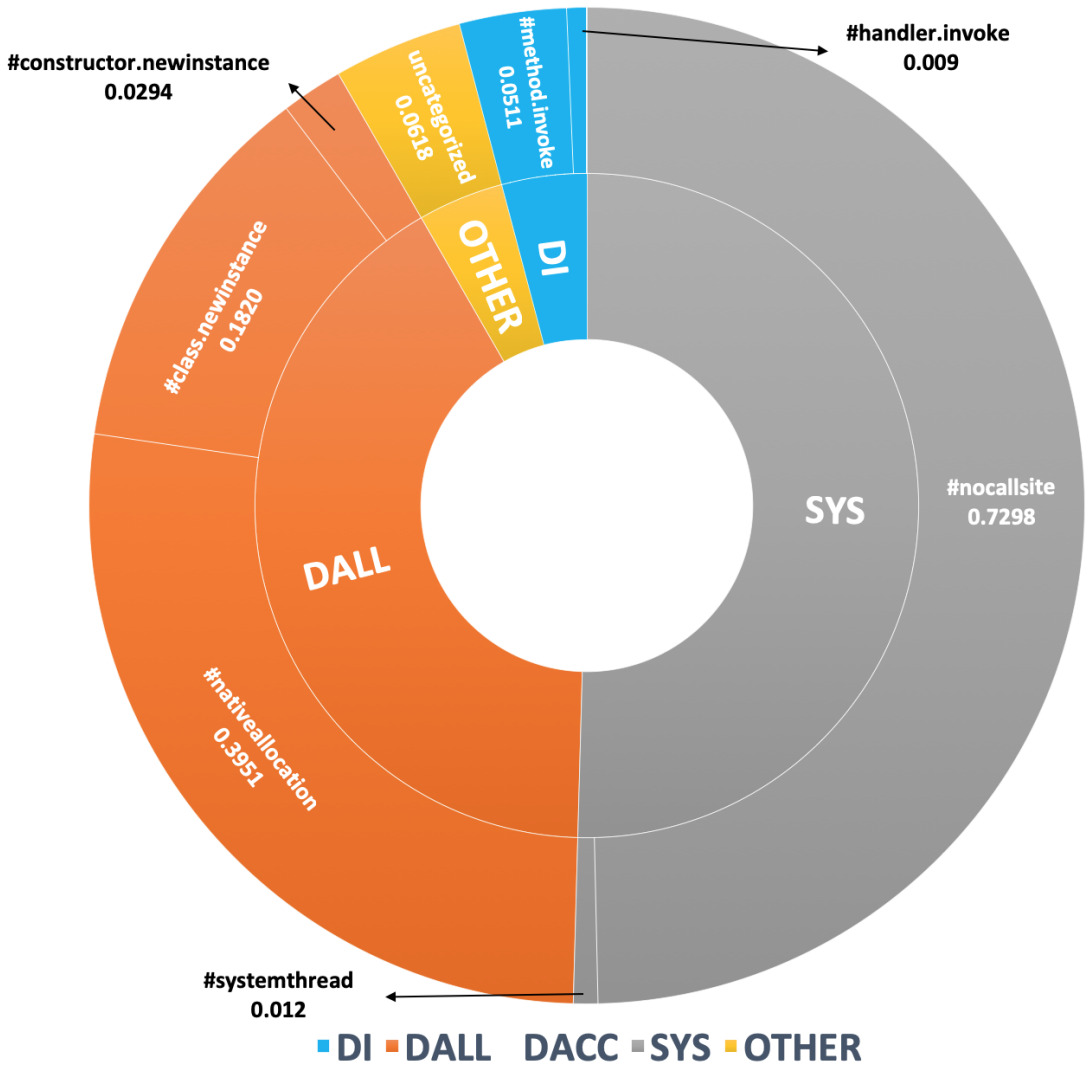


Figure 6.10: Cause of FNs in the static analysis with base support

The base analysis for the reduced dataset (20 programs that did not time out with reflection support) is also included in order to make the base and the reflection data comparable. Figure 6.11 shows the variation of recall values across the dataset. We observed that reflection support for the context-insensitive analysis addresses a significant share of false negatives caused by `Method#invoke`. It addresses all false negatives caused by dynamic proxies (invocation handlers) in 3/5 programs, and all false negatives caused by allocation via deserialisation (although only two programs *quartz-1.8.3* and *trove-2.1.0* are in this category). For the system category, the percentage increases, indicating that *Doop* reflection support is relatively ineffective for these categories.

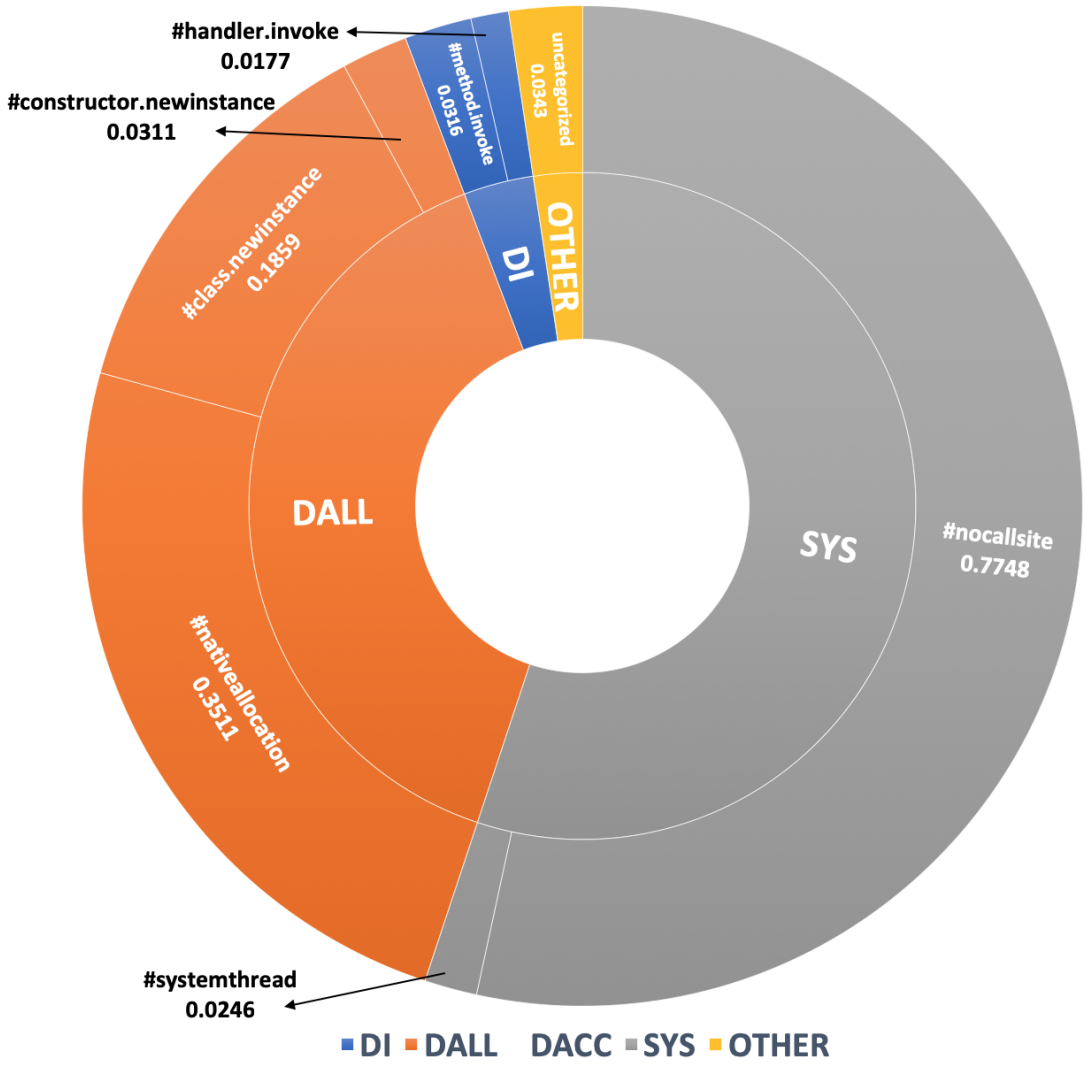


Figure 6.11: Cause of FNs in the static analysis with reflection support

We analysed the percentage of false negatives left after all tagged vertices were removed from the CCTs – this is the number of uncategorised false negatives in the Other category in both Figure 6.10 and 6.11. A cross-validation was carried out by two other researchers. We cross-validated false negatives in the *Other* category. We have randomly selected 373 vertexes for the base and 344 vertexes for the reflection analysis (the sample size is determined by a confidence level of 95% and a confidence interval of 5%, of the total number of false negatives from the *Other* category). We then extracted the respective number of CCT paths from the respective CCT root to an uncategorised false negative. It turns out that these are dominated by a single pattern we refer to as *double-reflective factory*, which we discuss in some more detail below This pattern

accounted for 54.4% of the uncategorised false negatives in the base analysis and 51.5% of the uncategorised false negatives in the analysis with reflection support. The following paragraph describes the pattern that we detected.

The double-reflective factory is a particular use of the factory design pattern [79] in conjunction with reflection, used to manage character sets. To illustrate this, consider the stack trace caused by an invocation of `System.out.println()` in Listing 6.12. Using both the base and the reflection analysis, `encodeLoop` is unreachable in the statically computed call graph. The encoder is created by the `Charset` (`sun.nio.cs.UTF`) which is created via reflection (`Class#forName` and `Class#newInstance`) by a `CharsetProvider` (`sun.nio.cs.FastCharsetProvider`) which is in fact itself also created using reflection by a service loader from jar manifest meta data. This is a triple factory, with two of the factories using reflective allocation. This is a good example of the framework complexity Java is known for. While our analysis tags the factories as dynamically allocated, it does not do this to the objects created in those factories using plain object allocation with `new`.

Listing 6.12: Stacktrace created by the invocation of `PrintStream#println()`

```
1 sun.nio.cs.UTF_8$Encoder::encodeLoop(Ljava/nio/CharBuffer;Ljava/nio/
      ByteBuffer;)Ljava/nio/charset/CoderResult
2 java.nio.charset.CharsetEncoder::encode(Ljava/nio/CharBuffer;Ljava/nio/
      ByteBuffer;Z)Ljava/nio/charset/CoderResult
3 sun.nio.cs.StreamEncoder::implWrite([CII)V
4 sun.nio.cs.StreamEncoder::write
5 java.io.OutputStreamWriter#write([CII)V
6 java.io.BufferedWriter#flushBuffer()V
7 java.io.PrintStream::newLine()V
8 java.io.PrintStream::println(Ljava/lang/String;)V
9 net.sourceforge.pmd.util.designer.MyPrintStream::println(Ljava/lang/String
      ;)V
```

As for the *nocallsite* and *systemthread* categories, we focus on false negatives defined in applications or third-party libraries. It turns out that for the base analysis, 9/31 programs have such false negatives in the *systemthread* category, and 28/31 programs have such false negatives in the *nocallsite* category. Using the same sampling procedure as described above, we found that 86% of the application false negatives in the *nocallsite* category are caused by static initialisers (`<clinit>` methods) invoked by the JVM. Another example are invocations of `Runnable#run` methods through native dispatch from `java.security.AccessController#doPrivileged`. Reflective method invocations are also classified in this category due to the native dispatch in `sun.reflect.NativeMethodAccessorImpl#invoke0`. In the sampling set, this accounted for 4.8% of cases. Sampling application-defined methods classified as *systemthread* reveals that all of these can be explained by invocations of `finalize` in application classes

in the `Finalizer` thread.

6.4 Threats to Validity

There are still a number of validity threats to the results. We discuss them in the following paragraphs.

The test unreflection process described in Section 6.2.4.2 has limitations, since *Junit* features, such as tests with rules and custom runners, were ignored, i.e., not unreflected. This has reduced the coverage of the oracle.

Test flakiness [131] is a known issue that affects test outcomes. It will affect the replication of results in the sense that, in some cases, slightly different results will be obtained. We have mitigated this by using docker, to provide an environment as close as possible to the one used to conduct the original experiments. We expect the variations in the number of reachable methods to be less than 1%. We found such non-deterministic coverage in 14 of the 31 programs between executions. They are: *ApacheJMeter_core-3.1, jena-2.6.3, marauroa-3.8.1, guava-21.0, quartz-1.8.3, jrefactory-2.9.19, jfreechart-1.0.13, tomcat-7.0.2, drools-7.0.0.Beta6, fitjava-1.1, log4j-1.2.16, oscache-2.4.1, weka-3-7-9 and htmlunit-2.8.* Figure 6.2 uses averages from five runs, the oracle used was generated by a single run. The reason for this decision is the high cost of oracle generation (see Table 6.4). In some cases, a slightly larger oracle could have been obtained by running the instrumented tests multiple times and merging the constructed CCTs.

The tagging of lambdas relies on naming patterns used by the OpenJDK compiler. There is a possibility that some of the library code within the analysis scope was compiled with a different compiler using a different convention, such as Oracle JDK or IBM JDK. This would have resulted in more false negatives that are not classified.

Tagging with `nocallsite` relied on a static pre-analysis to collect the call sites in methods. For libraries, this depends on the library version used. There is a chance that in some cases programs use custom class loaders, choosing a different version of the class. This would have resulted in methods being incorrectly tagged as `nocallsite`, and in an over-reporting of false negatives in this category.

6.5 Summary and Conclusion

The main contribution of this study is to provide quantitative evidence for better understanding the unsoundness of static program analysis, especially in relation to real-world programs. The experiments consist of two parts in general: dynamic analysis and static analysis. For the dynamic analysis part, we adopt an approach to model program behaviours during execution to gain insight into program behaviours. For

the static analysis part, we chose a modern framework *Doop* to build static models. The level of recall is computed based on respective static analyses. The results answer RQ4: **What is the level of recall achieved by state-of-the-art static analysis tools?** – around 11% of program models are not reported by static analysis. The gaps in static analysis are still significant as many methods that are known to be reachable are missed. While state-of-the-art analysis with reflection support can significantly improve recall, its high cost renders it impractical for many practical applications. The results further indicate that some language features suspected of being a major cause of unsoundness (in particular the usage of reflection: `Method#invoke`) play only a minor role. The classification of static analysis false negatives answers RQ5: **Which particular language features cause unsoundness in real-world programs?** This classification is also useful for static analysis tool builders to guide them where to best focus efforts to improve the recall for their analysis: to include the analysis of native methods and the JVM itself. We note that *Doop* models some native methods, including `Object#clone()` and some methods in `java.lang.System`, `sun.misc.Unsafe`,`java.io.UnixFilesystem`, `java.lang.Thread`, `java.lang.ref.Finalizer`, and `java.security.AccessController`. However, these models are still unsound and, as Grech et al. noted, such manual modelling "*... is hard. Extra native operations get added in every release of the JDK and analysis authors typically do not keep up with them*" [83].

The fact that dynamic analysis reveals a significant number of false negatives in the static analysis also indicates that hybrid techniques can be very effective. In particular, generated tests can be used to discover program behaviour that is out of reach of static analysis. However, there are limitations: like static reflection analysis, test generation is expensive [61], and our study has demonstrated that it is not as effective in discovering dynamic program behaviour as manually written tests.

Chapter 7

Conclusions

7.1 Introduction

This thesis has investigated the unsoundness of static program analysis in Java. Since static analysis always over-approximates program behaviours, it is expected, in theory, to be sound (all program behaviours will be predicted and considered). However, there are some program behaviours that may be neglected due the use of dynamic language features, such as *reflection*, *serialisation*, *dynamic class loading*, *invokedynamic*, *dynamic proxy* and the use of *native libraries*.

This thesis has made following key contributions to discover this unsoundness in static program analysis:

- A benchmark of dynamic language features in Java, with multiple usage patterns that can be used to assess static analysers' ability in modelling particular dynamic features.
- An investigation into the usefulness of generating oracles from mined stack traces.
- An empirical assessment of recall, conducted on real-world programs, which indicates an average of 11% of program behaviours are missed by a state-of-the-art static analysis tool.

7.2 Conclusion Remarks

We designed a series of experiments to explore the unsoundness of static program analysis. Firstly, we provided a categorisation of dynamic language features. They are *reflection*, *serialisation*, *dynamic class loading*, *invokedynamic*, *dynamic proxy* and the use of *native libraries*. This categorisation answers **RQ1: What are the sources of unsoundness in static analysis?**. To answer **RQ2: Are state-of-the-art static**

analysis tools able to successfully model dynamic language features in Java? we then constructed a micro-benchmark which consists of a number of usage patterns for each of these categories. An evaluation was performed against well-known static analysis tools, i.e., *Soot*, *Wala* and *Doop*, to examine how soundly they can handle those categories. The results suggest that none of the tools can soundly model the benchmark, even where it is claimed that some dynamic features are being handled. For instance, all three tools support reflection analysis, but none of them can fully model all twelve reflection usage patterns that we created in the benchmark. In fact, static analysis tools may consider a particular usage of a feature (i.e., a plain reflection as shown in Listing 4.3), but often overlook how it has been used in different contexts (i.e., an interprocedural reflection as shown in Listing 4.5). We note that our benchmark includes an interesting case where we observed inconsistent program behaviour across the Java platform, which raises a question: "If the actual program behaviours are indeterminate, how can static analysis model them soundly?"

Secondly, we constructed program oracles from real-world programs to further assess static analysis tools. Oracles are constructed by means of program execution. They reflect the actual program behaviours. We have proposed an unconventional way to obtain such oracles – stack traces, which are produced by the program when unexpected behaviour occurs. This gives us an insight into how dynamic language features are applied when abnormal program behaviour occurs. Stack traces are mined from on-line resources such as GitHub and Stack Overflow. The results show that *Doop* misses a high proportion of edges in the call graphs that it creates (11 out of 15 call edges). This experiment has shown that it is possible to use stack traces to assess the soundness of static analysis and answered **RQ3: Can information obtained from stack traces improve the soundness of static analysis?**

Lastly, to further explore and quantify the impact of dynamic language features, we conducted a large empirical study on real-world programs. Unlike the stack trace study, we used tests to drive the programs to obtain program oracles. The novelty of this study is we have considered multiple dimensions under analyses: two analysis settings (libraries versus super jar), various static analysis techniques (context-sensitive, context-insensitive, full reflection, light reflection support) and two different program oracles (built-in versus generated tests). We used quantitative measurements in different combinations with the above analyses to quantify missing program behaviours. The results indicate the gap between dynamically generated oracles and statically generated program models is substantial, an average of 11% of program behaviours are overlooked, which answers **RQ4: What is the level of recall achieved by state-of-the-art static analysis tools?** In addition, we provided a classification of such missed program behaviours, which answers **RQ5: Which particular language features cause**

unsoundness in real-world programs? – Most of them are related to native code. The recent work on *Doop* of Fourtounis et al. [75] has improved the tool so that it can better handle native code, under the guideline of our classification.

In conclusion, state-of-the-art static analysis frameworks generally lack support for dynamic language features. The classification we created for dynamic language features will help researchers using static analysis to quickly navigate through the causes of unsound analyses. This thesis also addressed different aspects of static analysis tools and brought clearer meaning to the term unsoundness, by looking into sensitivity settings and the choice of program oracles. Dynamic analysis techniques such as testing and hybrid analysis [41, 84] can potentially mitigate some of the limitations of static analysis, but cannot solve the unsoundness issue entirely – the main challenge is to create drivers (harnesses) that exercise the unsound parts of a program. The use of test case generation for this purpose is promising.

7.3 Future Work

There are a number of possible avenues for future work. An extended study with a larger dataset is an interesting topic for future research. One particularly interesting line of work would be the study of call graphs that cover multiple projects and libraries, including frameworks known for their heavy use of reflection (plugin-based systems, dependency injection) and the Java core libraries. We have noticed a large number of reflective invocations where call sites and targets were each located in separate libraries. The experiment described in Chapter 6 is based on Java byte code. Other non-Java programs that compile into JVM byte code (e.g., Scala, Kotlin etc) could also be considered for future study. Of course, studying unsoundness of static analysis for other program languages is important.

As we mentioned in Section 6.3.1, the amount of time and resources we invested into this experiment means that the analysis is not practical for industry needs. So the potential problem here is that it is still challenging to build comprehensive and sufficiently precise static models for real-world programs, to be embedded into the development process. Moreover, this thesis focuses on analysing open-source programs, and commercial programs have more demands on bug and vulnerability detection. Future work involves developing a plugin for known bug detection tools that targets a particular dynamic language feature. Our benchmark can then help with assessing its soundness.

Relying on naming patterns when tagging invocations (discussed in Section 6.2.4.6) is not a scalable approach as some of the names can be compiler dependent. For instance, we followed the OpenJDK naming convention for detecting lambda expressions. An improvement can be made to trace the `invokedynamic` instruction in the constant pool and to retrieve the relevant index to extract arguments used by the lambda method

factory.

Test flakiness [131] (i.e., tests with non-deterministic outcomes) is a known issue that affects test outcomes (discussed in Section 6.4). We have observed non-deterministic behaviours, in terms of different branch coverages and test outcomes in 14 of the 31 programs, across different executions. There is an increasing research interest in test flakiness [131, 113, 23, 66]. Potential future research directions in this space include: 1) study language features and patterns that cause tests to be flaky, 2) explore the relationship between coverage strategies and flaky tests and how synthesised tests perform with regards to flakiness compared with automated tests.

Another potential approach is to study the relationship between branch coverage and recall level. There is an hypothesis that the relationship is positive: as the branch coverage goes up, the recall level rises. However, there is currently no empirical evidence to back up this hypothesis.

References

[1] Airbus issues software bug alert after fatal plane crash. https://www.theguardian.com/technology/2015/may/20/airbus-issues-alert-software-bug-fatal-plane-crash. Accessed: 10-08-2017.

[2] Apache maven project standard directory layout. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html, accessed 20 July 2020.

[3] Common Vulnerabilities and Exposures the standard for information security vulnerability names. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java. Accessed: 19-04-2020.

[4] CVE-2003-1564 billion laughs. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564. Accessed: 19-04-2020.

[5] Invokedynamic rectifier / project serializer. http://www.opal-project.de/DeveloperTools.html, accessed 14 Jan 2019.

[6] Java class identity. https://www.jacoco.org/jacoco/trunk/doc/implementation.html. Accessed: 23-06-2020.

[7] Java programming language agents api. https://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html. Accessed: 05-08-2017.

[8] java-vulnerabilities repository. https://bitbucket.org/Li_Sui/java-vulnerabilities/, accessed 15 July 2020.

[9] Novopay botch-ups cost 45m to fix. http://www.stuff.co.nz/national/education/66349800/Novopay-botch-ups-cost-45m-to-fix. Accessed: 10-08-2017.

[10] Novopay wikipedia. https://en.wikipedia.org/wiki/Novopay. Accessed: 10-08-2017.

[11] org.objectweb.asm.classvisitor api. https://asm.ow2.io/javadoc/org/objectweb/asm/ClassVisitor.html#visitMethod-int-java.lang.String-java.lang.String-java.lang.String-java.lang.String:A-, accessed 12 July 2020.

[12] stack overflow developer survey 2016. https://insights.stackoverflow.com/survey/2016#technology-_-programming-scripting-and-markup-languages. Accessed: 6-08-2020.

[13] stack overflow developer survey 2018. https://insights.stackoverflow.com/survey/2018#technology-_-programming-scripting-and-markup-languages. Accessed: 6-08-2020.

[14] stack overflow developer survey 2019. https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages. Accessed: 6-08-2020.

[15] Tiobe index for august 2020. https://www.tiobe.com/tiobe-index/. Accessed: 6-08-2020.

[16] Understanding sun.misc.unsafe. http://www.javaworld.com/article/2952869/java-platform/understanding-sun-misc-unsafe.html. Accessed: 13-08-2017.

[17] WALA watson libraries for analyses. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis. Accessed: 31-07-2017.

[18] ysoserial a proof-of-concept tool for generating payloads that exploit unsafe java object deserialization. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections6.java. Accessed: 12-07-2017.

[19] Ghtorrent services, 2020. https://ghtorrent.org/services.html [accessed 12 March 2020].

[20] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[21] Ole Agesen. The cartesian product algorithm. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 1995.

[22] Hiralal Agrawal, Joseph Robert Horgan, Edward W Krauser, and Saul A London. Incremental regression testing. In *1993 Conference on Software Maintenance*, pages 348–357. IEEE, 1993.

[23] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. *arXiv preprint arXiv:1906.00673*, 2019.

[24] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proc. ECOOP'13*. Springer, 2013.

[25] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proc. ECOOP'13*. Springer, 2013.

[26] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[27] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[28] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proc. SOAP'15*, pages 13–18. ACM, 2015.

[29] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[30] Lance Andersen. Jdbc™4.0 specification. *JSR*, 221:1–126, 2006.

[31] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[32] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proc. SOAP'17*. ACM, 2017.

[33] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[34] Ken Arnold, James Gosling, David Holmes, and David Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 2000.

[35] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[36] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363, 2006.

[37] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE, 2013.

[38] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[39] Eric Bodden. Invokedynamic support in soot. In *Proc. SOAP'12*. ACM, 2012.

[40] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*, pages 525–549. Springer, 2007.

[41] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[42] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[43] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133. ACM, 2002.

[44] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

[45] Brendon Cahoon and Kathryn S McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291. IEEE, 2001.

[46] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[47] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[48] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 364–371, 2006.

[49] Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–146. ACM, 1999.

[50] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[51] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[52] Shigeru Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[53] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 7–12, 2015.

[54] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[55] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[56] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[57] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[58] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[59] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *ACM SIGPLAN Notices*, volume 50, pages 535–551. ACM, 2015.

[60] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: Dos attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[61] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. Xcorpus–an executable corpus of java programs. *JOT*, 16(4):1:1–24, 2017.

[62] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[63] Thomas J. Watson IBM Research Center. Research Division and FE Allen. *Interprocedural data flow analysis*. 1973.

[64] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 149–168, 2003.

[65] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[66] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.

[67] Michael Eichberg and Ben Hermann. A software product line for static analyses: the opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[68] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994.

[69] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[70] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[71] Brian Foote and Ralph E Johnson. Reflective facilities in smalltalk-80. In *Proc. OOPSLA'89*. ACM, 1989.

[72] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[73] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. In *Proc. ISSTA'18*. ACM, 2018.

[74] George Fourtounis and Yannis Smaragdakis. Deep static modeling of invokedynamic. In *Proc. ECOOP'19*, 2020.

[75] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. Identifying java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 388–400, 2020.

[76] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[77] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[78] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[79] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proc. ECOOP'93*. Springer, 1993.

[80] Brian Goetz. Translation of lambda expressions, 2012. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.

[81] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification: Java se 8 edition. oracle america. *Inc., Redwood City, California, USA*, 2015.

[82] Susan L Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM (JACM)*, 23(1):172–202, 1976.

[83] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. In *Proc. OOPSLA'17*. ACM, 2017.

[84] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. In *Proceedings OOPSLA'17*. ACM, 2017.

[85] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proc. ISSTA'18*. ACM, 2018.

[86] Neville Grech, George Kastrinis, and Yannis Smaragdakis. Efficient reflection string analysis via graph coloring. In *Proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.

[87] Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. In *Proc. OOPSLA'17*. ACM, 2017.

[88] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[89] Rajiv Gupta, DA Benson, and Jesse Zhixi Fang. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113. IEEE, 1997.

[90] Rajiv Gupta and Mary Lou Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. *ACM SIGSOFT Software Engineering Notes*, 20(4):29–40, 1995.

[91] Robert J Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

[92] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[93] Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.

[94] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[95] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. *ECOOP 2004–Object-Oriented Programming*, pages 96–122, 2004.

[96] Marc R Hoffmann et al. Jacoco java code coverage library, 2014.

[97] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proc. CCS'16*. ACM, 2016.

[98] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, 2007.

[99] David Janzen and Hossein Saiedian. Does test-driven development really improve software design quality? *Ieee Software*, 25(2):77–84, 2008.

[100] Kamil Jezek and Jens Dietrich. Magic with dynamo–flexible cross-component linking for java with invokedynamic. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[101] Capers Jones. Software metrics: good, bad and missing. *Computer*, 27(9):98–100, 1994.

[102] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, page 26, 2003.

[103] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[104] Ken Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3(1-4):5–15, 1972.

[105] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2017.

[106] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[107] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[108] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[109] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[110] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[111] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[112] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[113] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–111, 2019.

[114] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[115] Frank Law, KP Chow, and YH Mai. Understanding computer forensics requirements in china via the" panda burning incense" virus case. *The Journal of Digital Forensics, Security and Law: JDFSL*, 9(2):51, 2014.

[116] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: Static analysis framework for android hybrid applications. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 250–261. IEEE, 2016.

[117] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[118] Ond Lhoták et al. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42. ACM, 2007.

[119] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[120] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):3, 2008.

[121] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. In *Proc. OOPSLA'18*. ACM, 2018.

[122] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proc. ESEC/FSE'18*. ACM, 2018.

[123] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for java. In *Proc. ECOOP'14*. Springer, 2014.

[124] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):7, 2019.

[125] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[126] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification: Java se 8 edition, 2015. 2015. `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

[127] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. Reflection analysis for java: Uncovering more reflective targets precisely. In *Proc. ISSRE'17*. IEEE, 2017.

[128] Yi Liu, Zhengqiu Yang, Jiapeng Xiu, and Chen Liu. Research on an anti-crawling mechanism and key algorithm based on sliding time window. In *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pages 220–223. IEEE, 2016.

[129] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[130] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, pages 139–160. Springer-Verlag, 2005.

[131] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[132] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250, 2012.

[133] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: the java unsafe api in the wild. In *Proc. OOPSLA'15*. ACM, 2015.

[134] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569. IEEE, 2003.

[135] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[136] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[137] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In *ACM Sigplan Notices*, volume 45, pages 305–315. ACM, 2010.

[138] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 1–11. ACM, 2002.

[139] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[140] Joan C Miller and Clifford J Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[141] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[142] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. An empirical study of static call graph extractors. *ACM TOSEM*, 7(2):158–191, 1998.

[143] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[144] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[145] Sun Developer Network. The java hotspot performance engine architecture. *Sun Microsystem*, 2007.

[146] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS'05*. Internet Society, 2005.

[147] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[148] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 422–431. IEEE, 1999.

[149] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[150] Carlos Pacheco and Michael D Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European conference on Object-Oriented Programming*, pages 504–527. Springer-Verlag, 2005.

[151] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[152] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

[153] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[154] David J Pearce, Paul HJ Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):4–es, 2007.

[155] Shawn Rasheed, Jens Dietrich, and Amjed Tahir. Laughter in the wild: A study into dos vulnerabilities in yaml libraries. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 342–349. IEEE, 2019.

[156] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proc. FSE'16*, pages 474–486. ACM, 2016.

[157] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proc. ISSTA'19*. ACM, 2019.

[158] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Proc. SOAP'18*. ACM, 2018.

[159] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[160] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[161] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Proceedings SSBSE'15*. Springer, 2015.

[162] Kevin A Roundy and Barton P Miller. Hybrid analysis and control of malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 317–338. Springer, 2010.

[163] Atanas Rountev, Ana Milanova, and Barbara G Ryder. Points-to analysis for java using annotated constraints. *ACM SIGPLAN Notices*, 36(11):43–55, 2001.

[164] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[165] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[166] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. 1978.

[167] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Marinov, and Owolabi Legunsen. Reflection-aware static regression test selection. In *Proc. OOPSLA'19*. ACM, 2019.

[168] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. Specjvm2008 performance characterization. In David Kaeli and Kai Sachs, editors, *SPEC Benchmark Workshop*. Springer, 2009.

[169] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, PhD thesis, Carnegie Mellon University, 1991.

[170] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[171] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, 2011.

[172] Yannis Smaragdakis and George Kastrinis. Defensive points-to analysis: Effective soundness via laziness. In *Proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.

[173] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proc. POPL'84*. ACM, 1984.

[174] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[175] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. *Practical virtual method call resolution for Java*, volume 35. ACM, 2000.

[176] Tomonori Takada, Fumiaki Ohata, and Katsuro Inoue. Dependence-cache slicing: A program slicing method using lightweight dynamic information. In *Proceedings*

10th International Workshop on Program Comprehension, pages 169–177. IEEE, 2002.

[177] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *Proc. APSEC'10*, 2010.

[178] George K Thiruvathukal. Java grande forum report: Making java work for high-end computing. 1998.

[179] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. OOPSLA'00*. ACM, 2000.

[180] Fumiaki Umemori, Kenji Konda, Reishi Yokomori, and Katsuro Inoue. Design and implementation of bytecode-based java slicing system. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 108–117. IEEE, 2003.

[181] Tiejun Wang and Scott F Smith. Precise constraint-based type inference for java. In *European Conference on Object-Oriented Programming*, pages 99–117. Springer, 2001.

[182] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[183] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[184] John Whaley and Monica Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. *Static Analysis*, pages 79–99, 2002.

[185] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, 2004.

[186] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[187] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *ACM SIGPLAN Notices*, 34(5):91–103, 1999.

Appendices

A.1 Mined Stack Traces Results

call site: `org.dacapo.harness.Eclipse#iterate`
target: `org.eclipse.core.runtime.adaptor.EclipseStarter#run`
stacktrace: `https://github.com/dvyukov/data-race-test/issues/32`

call site: `org.dacapo.harness.Batik#iterate`
target: `org.apache.batik.apps.rasterizer.Main#execute`
stacktrace: `https://github.com/dvyukov/data-race-test/issues/32`

call site: `doppio.JarLauncher#main`
target: `org.python.util.jython#main`
stacktrace: `https://github.com/plasma-umass/doppio/issues/381`

call site: `org.apache.hadoop.hbase.mapreduce.Driver#main`
target: `org.apache.hadoop.util.ProgramDriver#driver`
stacktrace: `http://stackoverflow.com/questions/14421493/import-text-file-to-hbase-using-importtsv`

call site: `com.google.common.base.internal.Finalizer#cleanUp`
target: `com.google.common.collect.MapMaker$SoftEntry#finalizeReferent`
stacktrace: `https://github.com/icella/guava-libraries/issues/252`

call site: `org.apache.hadoop.hbase.protobuf.ProtobufUtil#toFilter`
target: `org.apache.hadoop.hbase.filter.Filter#parseFrom`
stacktrace: `http://stackoverflow.com/questions/23805959/protobuf-error-with-custom-filter`

call site: `org.jboss.weld.injection.producer.DefaultLifecycleCallbackInvoker#invokeMethods`
target: `jersey.repackaged.com.google.common.base.Preconditions#checkState`
stacktrace: `https://github.com/IQSS/dataverse/issues/2628`

call site: `org.springframework.boot.loader.MainMethodRunner#run`
target: `org.springframework.boot.loader.PropertiesLauncher#getArgs`
stacktrace: `https://github.com/spring-projects/spring-boot/issues/1037`

call site: `org.springframework.boot.loader.MainMethodRunner#run`
target: `org.springframework.boot.loader.PropertiesLauncher#main`
stacktrace: `http://stackoverflow.com/questions/25428454/what-is-wrong-when-i-get-a-unable-to-determine-code-source-archive`

call site:	`org.springframework.boot.loader.MainMethodRunner#run`
target:	`org.springframework.boot.loader.JarLauncher#main`
stacktrace:	`https://github.com/spring-projects/spring-boot/issues/1037`

call site:	`org.apache.logging.log4j.core.config.plugins.util.PluginBuilder#build`
target:	`org.apache.logging.log4j.core.layout.JsonLayout#createLayout`
stacktrace:	`https://github.com/elastic/elasticsearch/issues/22103`

A.2 Performance: CCT and SCG Build Time

program	builtinTest	generatedTest
lucene-4.3.0	89	1,001
guava-21.0	644	96
fitjava-1.1	4	18
castor-1.3.1	220	1,944
checkstyle-5.1	1,282	212
commons-collections-3.2.1	42	57
informa-0.7.0-alpha2	92	146
javacc-5.0	4	46
jFin_DateMath-R1.0.1	41	18
jgrapht-0.8.1	66	14
jrat-0.6	32	375
log4j-1.2.16	63	239
marauroa-3.8.1	38	68
nekohtml-1.9.14	5	25
openjms-0.7.7-beta-1	2	484
oscache-2.4.1	71	49
pmd-4.2.5	256	338
quartz-1.8.3	11	223
trove-2.1.0	14	44
velocity-1.6.4	51	119
wct-1.5.2	46	622
findbugs-1.3.9	66	1,309
htmlunit-2.8	2,000	1,030
jena-2.6.3	2,279	2,285
tomcat-7.0.2	94	778
weka-3-7-9	30	3,201
jfreechart-1.0.13	1,423	1,996
mockito-core-2.7.17	500	152
drools-7.0.0.Beta6	2,245	2,318
jrefactory-2.9.19	133	2,493
ApacheJMeter_core-3.1	367	1203
total running time	12,210	22,903

Figure 1: Performance:CCT build time (in minutes)

program	base-lib	callsite-lib	ref-lib	refLite-lib	base-super	callsite-super	ref-super	refLite-super
lucene-4.3.0	6	36	360	360	17	88	360	360
guava-21.0	19	88	360	360	21	92	360	360
fitjava-1.1	1	2	7	13	1	2	7	13
castor-1.3.1	6	27	360	360	7	28	360	360
checkstyle-5.1	2	4	24	69	2	5	328	115
commons-collections-3.2.1	4	29	176	69	4	29	176	69
informa-0.7.0-alpha2	1	2	8	22	3	3	360	55
javacc-5.0	1	3	7	17	1	3	7	17
jFin_DateMath-R1.0.1	1	2	8	16	2	2	9	21
jgrapht-0.8.1	2	4	15	35	2	4	18	34
jrat-0.6	2	4	67	47	2	4	317	49
log4j-1.2.16	2	5	360	71	2	5	360	76
marauroa-3.8.1	2	3	11	30	2	3	221	63
nekohtml-1.9.14	1	2	8	27	1	2	29	31
openjms-0.7.7-beta-1	2	6	282	120	3	6	360	225
oscache-2.4.1	2	2	9	35	3	4	360	71
pmd-4.2.5	6	21	96	150	6	22	360	190
quartz-1.8.3	2	3	17	59	2	4	360	110
trove-2.1.0	2	2	12	22	2	2	12	22
velocity-1.6.4	2	3	17	48	2	4	360	76
wct-1.5.2	4	9	360	360	14	20	360	360
findbugs-1.3.9	4	20	128	253	5	20	360	360
htmlunit-2.8	6	21	360	360	8	23	360	360
jena-2.6.3	4	15	41	189	4	16	360	240
tomcat-7.0.2	7	28	360	292	8	30	360	360
weka-3-7-9	25	130	360	360	26	129	360	360
jfreechart-1.0.13	9	35	92	233	9	35	360	256
mockito-core-2.7.17	3	8	360	94	3	8	360	112
drools-7.0.0.Beta6	31	234	360	360	39	260	360	360
jrefactory-2.9.19	13	104	342	360	13	99	360	360
ApacheJMeter_core-3.1	10	60	360	360	22	77	360	360
total running time	182	912	5,327	5151	236	1,029	8,684	5,805

Figure 2: Performance:SCG build time (in minutes)