Title
Adapting Static Analysis Tools to Meet User Expectations

Author
Utture, Akshay Anand

Publication Date
2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Adapting Static Analysis Tools to Meet User Expectations

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Akshay Anand Utture

2023

ABSTRACT OF THE DISSERTATION

Adapting Static Analysis Tools to Meet User Expectations

by

Akshay Anand Utture
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Jens Palsberg, Chair

Traditionally, static analysis tools for catching program errors and security vulnerabilities were designed as verification tools. Hence, soundness, the criteria of never missing an error, was the primary goal. In practice, however, most users are significantly more concerned about false-positives, analysis times, and repairability than unsoundness; they expect the tools to have fewer than 20% false-positive warnings and take at most a few minutes to run. Since most static analysis tools are optional to run, users give up on tools that don't meet their expectations. To meet these expectations, a few newer tools have been designed from the ground up to prioritize these three criteria, but they require a redesign for every static analysis; they cannot use existing, mature, soundness-focused tools. So the question is: can we adapt existing soundness-focused static analysis tools to meet user expectations?

This thesis shows that the answer to this question is a yes, by introducing three new tools, CGPruner, QueryMax, and RLFixer, to address the three criteria users care about the most: false-positives, analysis time, and automated repair suggestions. The central idea underlying each of these tools is identifying opportunities where a little soundness can be traded off for large improvements in these three criteria. These three new tools are designed

as pre-processors and post-processors to a black-box static analysis, and hence are applicable to many analyses. Our experiments show that they significantly improve the results of several existing soundness-focused static analysis tools on the three critical user criteria.

The dissertation of Akshay Anand Utture is approved.

Harry Xu

Miryung Kim

Todd Millstein

Jens Palsberg, Committee Chair

University of California, Los Angeles

2023

To my mom and dad, who have supported me through everything.

To all my teachers, for everything they have taught me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are many people who have shaped my journey through the Ph.D. and supported me through its challenges. I would like to properly acknowledge their contribution and support here.

First and foremost is my advisor, Jens. The list of things he has taught me and the things I am thankful for could probably fill a few pages, but I'll talk about just a few of them here. Jens' optimism kept me going through the many failures that one invariably goes through during a Ph.D. After a bad paper review or a failed experiment, I would want to scrap the research direction and throw it all away. But Jens' encouragement gave me the faith to fix the problems in our project instead of quitting, and to keep working on making my work better everyday. His faith in my judgment, both technical and otherwise, has given me the confidence that I can do independent research. In addition to all the technical lessons about static analysis, he also taught me how to think of new research problems, how to design an experimental evaluation, and how to position my research in the context of existing work. But, I think the most important thing Jens taught me is to write and communicate about my research effectively. When I look back at my paper drafts from my first year versus now, I am amazed to see just how far I have come. Working with him, I have come to realize that writing and presenting research well is just as important as its technical aspects. I am truly grateful that I have such a great advisor.

I would also like to thank my other committee members Todd, Miryung, and Harry, who've given me insightful feedback about my work and from whom I have learned a lot through the courses I've taken with them.

My journey through the Ph.D. would have been rough without some great mentors: Aishwarya, Saswat, Siva, Christian, and Parthe. Aishwarya listened to all my ideas, both good and stupid, and spent a lot of time giving me detailed feedback on my work. And she was there to encourage and support me whenever I lost faith in my own ideas and research.

Saswat taught me a lot about formal verification, and programming languages in general. He also mentored me through my Amazon internship. Saswat and Siva both helped me a lot with career advice, and making major decisions through grad school. Christian helped me a lot in my first year when I really needed the guidance. I knew that I could rely on him when I didn't understand some static analysis concept, or was stuck with a new Java analysis tool. He was also a co-author on the call-graph pruning paper. Parthe took a lot of interest in my work even though he works on machine learning, and he has always inspired me to take more ambitious steps.

And I don't think I would have made it to the Ph.D. in the first place if it wasn't for my undergraduate research advisors, Krishna Nandivada and Meghana Nasre. They both took a chance with me even though I didn't have much research experience then. I learned my first lessons in research from them, and they also gave me the confidence that I had it in me to become a researcher. In fact, it was Professor Meghana who finally convinced me to do a Ph.D. I was worried about committing to such a narrow field for the rest of my life, and she told me that the Ph.D. isn't just about becoming an expert in a narrow domain. It is about learning to tackle hard problems in a scientific way, and I would take that with me to whatever field I moved to.

I had shorter interactions with Dana and Wontae, but they had an impact. Dana, my writing class teacher, got me to consciously pay attention to many aspects of writing that I had overlooked. Wontae mentored me through my Google internship, and taught me about the interesting differences between building static analysis tools in academia and industry.

Grad school wouldn't have been as fun without my friends and labmates, who I've also had many interesting discussions with. From our research group there was Zeina, Shuyang, Micky, and John. Zeina was there help me laugh off all my mistakes and failures. Shuyang always took the initiative to organize our (research) reading groups, and she was also a co-author on the call-graph pruning paper. Micky was always there to discuss interesting ideas and keep up the optimism in our lab. Then there were other people in the department:

Arjun, Vidushi, Pradeep, Amita, Ana, Poorva, Rathin, Neil, and Jefferey. And finally, to some friends outside of the department: Kshitij, Vishal, Navjot, Aditya, Bijoy, Pratik, Michael, Ha, and Sumit.

I would also like to acknowledge the ONR (Office of Naval Research) and NSF (National Science Foundation) grants that supported my research.

Finally, I am dedicating this thesis to my mom (Alpana) and dad (Anand) for supporting me through all my years of education, right up to now. They have encouraged me to give my best at work, but also to balance work and play. Knowing that they always have my back has allowed me to take the risks that I wouldn't have been able to take by myself, including undertaking the Ph.D. I am truly thankful for everything they have done.

VITA

2016	Software Engineering Intern, Microsoft, Hyderabad
2018	B.Tech. and M.Tech. (Dual Degree) in Computer Science and Engineering, Indian Institute of Technology Madras
2018	UCLA Graduate Dean's Scholar Award
2019-2021	Teaching Assistant (Compiler Construction), UCLA
2020	Ph.D. Software Engineering Intern, Google, Sunnyvale
2022	Applied Scientist Intern (Automated Reasoning Group), Amazon, Boston

PUBLICATIONS

Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. "Striking a Balance: Pruning False-Positives from Static Call Graphs." In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p. 2043–2055, New York, NY, USA, 2022. Association for Computing Machinery.

Akshay Utture and Jens Palsberg. "Fast and Precise Application Code Analysis Using a Partial Library." In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p. 934–945, New York, NY, USA, 2022. Association for Computing Machinery.

Akshay Utture and Jens Palsberg. "From Leaks to Fixes: Automated Repairs for Resource Leak Warnings". In submission at FSE '23.

CHAPTER 1

Introduction

The growth in code-base sizes has accelerated rapidly over the last few decades. For example, the Linux kernel has over 28 million lines of code [Lin23], the Facebook website has over a 100 million [DFL19], and Google's monolithic code repository has over 2 billion [PL16]. The frequency of code changes has grown just as rapidly. An empirical study of open source developers reports that the average developer makes at least one commit per day [KRS13]. Likewise, Google reports more than 20,000 commits a day to their monolithic code-base [SSC18].

Maintaining code quality and correctness in this turbulent ocean of software is becoming increasingly challenging. We need automated tool support to catch software errors that could lead to crashes or expose security vulnerabilities. Over the years, automated code-quality tools like static analysis tools, dynamic analysis tools, fuzzers, symbolic execution tools, and model checkers have received increasing adoption, with static analysis being one of the most popular categories. Static analysis tools can analyze code without running it and can catch a variety of errors such as security vulnerabilities [TPF09a], memory-access errors [HJP08], resource-leaks [KSS21a], and concurrency bugs [BGO18]. Some prominent examples of static analysis' popularity can be seen with the Coverity Scan tool, which has seen over 600,000 of its warnings fixed by various open-source projects [Syn17], and the Infer tool [CDD15], which has seen over 100,000 of its warnings fixed by developers at Meta alone [DFL19].

So what are the primary design criteria for a static analysis tool? Traditionally, static analysis tools for catching program errors were designed as verification tools. In other words,

the main goal was to give a mathematical proof that a program was free of a certain error or security vulnerability. Hence, whenever some runtime behavior is infeasible to capture statically (as is often the case [Ric53]), these tools choose to over-approximate. An over-approximation includes the effects of all possible executions, but it may also include the effects of some impossible executions. This over-approximation criteria, known as *soundness*, ensures that we never miss an error, but it could produce false-positive error warnings. Such sound verification tools are important in understanding what kind of theoretical guarantees we can give about programs. They are also very valuable in practice for safety-critical software in space-shuttles and power-plants, where a single bug can be catastrophic.

A minority of developers working on safety-critical applications treat static analysis as a verification tool and care deeply about soundness. But what about large static analysis deployments, where users apply these tools in their daily programming tasks? Static analysis teams at Google [SAE18a], Meta [CDD15] and Coverity [BBC10a] find that, in practice, the most important goal for a static analysis tool is for users to trust the static analysis tool and like using it. In the words of Sadowski et. al [SAE18a] from Google,

> *For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.*

This is especially true because, in practice, most static analysis tools are optional for the user; the user could apply them in their development process, but they aren't obligated to. So if a user doesn't like a static analysis tool, they simply won't use it. Thus, user expectations trump all other criteria.

User Expectations from Static Analysis tools

We want to build static analysis tools that users trust and like. But what does the average static analysis user really care about in practice? And what are their expectations from static analysis tools? To answer this question, Christakis and Bird conducted a developer survey

Pain Points
Wrong checks are on by default
Bad warning messages
Too many false positives
Too slow
No suggested fixes
Difficult to fit into workflow
Bad visualization of warnings
No suppression of warnings
No ranking of warnings
Can't selectively turn off analysis
Complex user interface
Can't handle all language features
No support for custom rules
Misses too many issues
Not cross platform

Figure 1.1: Pain points reported by users in Christakis and Bird's developer study [CB16a], in decreasing order of importance

[CB16a] with hundreds of users of various static-analysis tools at Microsoft and compiled a list of fifteen pain points they faced. Fig. 1.1 lists these pain points in decreasing order of importance. After the first two user-interface based issues, the top three pain points were the false-positive rate, analysis time, and lack of suggested fixes. Missing errors (i.e. soundness) appeared at position 14 on the list. To get a better sense of user-expectations on the false-positive rates and analysis times, they asked users what the maximum threshold of each they were willing to tolerate. Over 75% of users said that they would not tolerate more than 20% false-positives or a couple of minutes of analysis time. And as far as repairability is concerned, 54% of developers expected repair suggestions and were willing to sift through as many as 10 repair suggestions. Other independent empirical studies on user-expectations of static analysis tools [BBC10a, JSM13c, SAE18b] have arrived at very similar conclusions as Christakis and Bird [CB16a]. Thus, in practice, a majority of developers have very high expectations on the false-positive rate, analysis time and repairability, and soundness is a low-priority issue.

Traditional soundness-focused static analysis tools put the very things that users care about the most as secondary to soundness: the false-positive rate, analysis time and fix suggestions. Simultaneously achieving soundness, zero false-positives, low analysis time and fix suggestions is not possible, since there is an inherent trade-off between soundness and the three user criteria. Tools that chase perfect soundness sacrifice the three user criteria. For example, several soundness-focused static analysis tools take hours to run on their largest benchmarks [LTM18a, BKL20a, GFF18a, SWF20, FWS19, SZ20]. Other soundness-focused tools report false-positive rates [ULK22, KSS21b, BKL20b] beyond 50%. Such static analysis tools fall short of user expectations and often get in the way.

To better understand why a soundness-focused static analysis gets in a user's way, imagine the following scenario. A software developer has several features to roll out, and tight deadlines to meet. They could optionally use a static analysis tool to improve their development process. Now imagine Tool A, a sound static analysis that takes two hours to run, has over 50% false-positive warnings, and no repair suggestions. The developer gets frustrated with first waiting so long for the analysis to complete, then with the time wasted in examining false-positive warnings, and finally the few true warnings have no hints for repairs. The frustrated developer simply ignores the tool's warnings, checks that their hand-written tests pass, and then goes ahead to upload their code. In fact, the next time around, the developer doesn't even bother to run this optional static analysis tool; it was frustrating and a waste of their time. Instead, imagine Tool B, an unsound static analysis that runs in a minute, has almost entirely true warnings, and has fix suggestions for these as well. The developer uses Tool B regularly, takes its warnings seriously, and fixes most of its warnings. In summary, even though Tool A was sound and Tool B was not, Tool B results in more errors being fixed because users actually like using it and as a consequence use it regularly. Thus, meeting user-expectations on the false-positive rate, analysis speed and repairability ultimately leads to better quality software in practice.

Tools that are tailored to User Expectations

Having observed these user-expectations in practice, some newer tools like Infer [CDD15], Tricorder [SGJ15], and NullAway [BCS19] have been designed from the ground up to prioritize low false-positive rates and fast analysis times. The limitations of such an approach is that we need to redesign every kind of analysis from the ground-up; we can't use existing soundness-focused analyses which are well understood and have several mature tools. For example, there is currently no call-graph analysis that prioritizes low false-positive rates and fast analysis times. At the same time, a soundness-focused call-graph analysis is a very well studied topic with several existing, mature tools. It would be great if we can adapt existing soundness-focused call-graph tools to meet user expectations. So the question we would like to answer is:

Can we adapt existing soundness-focused tools to meet user-expectations on the criteria that matter the most to them: the false-positive rate, analysis time, and automated repair suggestions?

Our approach: Adapt existing soundness-focused tools to meet user expectations

This thesis introduces three new tools, *CGPruner*, *QueryMax*, and *RLFixer*, to adapt existing soundness-focused static analyses to meet the three criteria that users care about the most: false-positives, analysis time, and automated repair suggestions, respectively. The central idea underlying each of these tools is identifying opportunities where a little soundness can be traded off for large improvements in one of these three criteria.

Figures 1.2 and 1.3 give an overview of how the three new tools function in a static analysis pipeline. They are designed as pre-processors (*QueryMax*) or post-processors (*CGPruner* and *RLFixer*) to a black-box static analysis. This design makes them applicable to many existing analysis tools. *QueryMax* pre-processes the input program to select a partial-program, thereby reducing the input size for the existing static-analysis. The post-

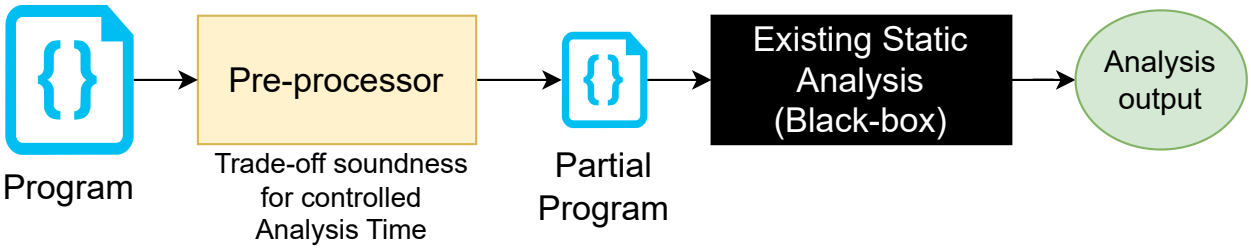


Figure 1.2: Preprocessor workflow (*QueryMax*)

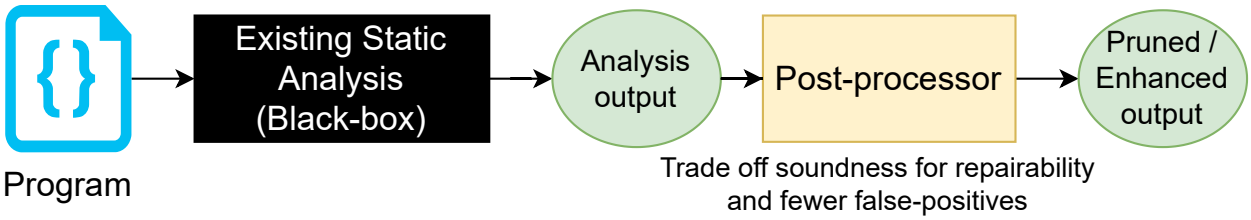


Figure 1.3: Post-processor workflow (*CGPruner* and *RLFixer*)

processors, *CGPruner* and *RLFixer*, prune and enhance the output from existing static analyses.

For each of these new tools, we would like their design to be driven by the following four *Desirable Properties*:

1. **Black-box:** Treat the existing static analysis as a black-box.
2. **Tunability:** Trade-off between soundness and the three user criteria should be tunable.
3. **Preservation:** An improvement in one of the three user criteria shouldn't come at the cost of the other two.
4. **Worthwhile trade-off:** Reduction in false positive rate and analysis time should be significant compared to the traded-off soundness.

The black-box property requires that our solution not be tied to any particular analysis algorithm or tool implementation, and should be applicable to any tool for a given language; this gives our solution generality across tools. Further generality can be achieved by being

applicable to different kinds of static analyses like null-pointer analysis, taint-analysis, cast-check analysis, etc. Tunability is important because different analyses or use-cases require different trade-offs between soundness and the three user criteria, and so a user should be able to control this trade-off. Preservation of all three user criteria are important; hence, optimizing for one shouldn't deteriorate the others. Finally, the trade-off of soundness for lower false-positives and analysis times should be worthwhile, since soundness is still an important criteria. We discuss how these four properties apply to each tool in the later chapters of the thesis. The handful of past research works that try to pre-process an analysis to improve scalability [AL13] or post-process it to prune false-positives [RPM08, TGP14], often fail to achieve most of these properties.

Given the problem setting and my approach to tackling this problem, I formulate the following thesis statement, which I substantiate through the rest of the thesis.

Thesis Statement

Existing soundness-focused static analysis tools can be pre-processed or post-processed to systematically trade off some soundness for significant improvements in criteria that users care about the most: fewer false-positives, controlled analysis time, and automated repair suggestions.

Organization of the thesis

The rest of the thesis is split into four chapters: the first gives some background on static analysis tools, and the remaining three describe the three new tools, *CGPruner* [ULK22], *QueryMax* [UP22], and *RLFixer*.

Chapter 2] Background. In the next chapter, I will introduce the technical details of how static analysis can be used to catch errors and security vulnerabilities, as well as some of the challenges in the field.

Chapter 3] CGPruner: Pruning False-Positives from Static Call-graphs. In this chapter, I introduce a new tool, *CGPruner*, to improve the false-positive rate in existing static analysis tools. *CGPruner* (or call-graph pruner) achieves this by pruning the static call graph that sits at the core of many static analyses. Specifically, static call-graph construction proceeds as usual, after which a call-graph pruner removes many false-positive edges but few true edges. The call-graph pruner is generated through an automatic, ahead-of-time learning process on a training dataset of call-graphs. We added such a call-graph pruner to a software tool for null-pointer analysis and found that its false-positive rate decreased from 73% to 23%.

Chapter 4] QueryMax: Fast and Precise Application Code Analysis using Partial Libraries. This chapter introduces *QueryMax*, a new tool to control analysis time for an application code analysis without introducing more false-positives. *QueryMax* acts as a pre-processor to an existing analysis tool to select a partial library that is most relevant to the analysis queries in the application code. The selected partial library plus the application is given as input to the existing static analysis tool, with the remaining library pointers treated as the bottom element in the abstract domain. *QueryMax* catches, relative to a whole-program analysis, 87% of its errors, with a 0% false-positive rate and a geometric mean speedup of 10x.

Chapter 5] RLFixer: Automated Repairs for Resource Leak Warnings. In this chapter, I discuss the design of *RLFixer*, a specialized repair tool that generates high-quality fixes for resource leaks identified by any resource-leak static analysis. A major challenge for the resource-leak repair problem is that it is at least as hard as compile-time object

deallocation, a well-known hard problem for compilers. *RLFixer* tackles this challenge using a new analysis, called *resource escape analysis*, that separates out the decidable fixes and generates correct repairs for them. *RLFixer* is demand-driven and hence only analyzes statements relevant to the leak, thereby keeping overhead low. When applied to five popular Java resource-leak detectors, *RLFixer* can generate repairs for a majority (66%) of their warnings with a 14 second repair time and a fix-correctness of 95%.

CHAPTER 2

Static Analysis Background

In this chapter, I shall introduce some basic concepts about static analysis and give a broad overview of the main challenges in the field. Readers who already understand static analysis well can skip to the next chapter.

2.1 Basic Concepts

We start off this chapter by discussing some basic concepts about static analysis.

Over-approximation and under-approximation. A static analysis used in the context of code-quality improvement typically aims to statically compute all information about a single kind of error across the program. For example, a null-pointer analysis would compute all pointers that could point to null and be dereferenced. However, Rice's theorem [Ric53] shows that most non-trivial properties that static analysis tools reason about, such as null-pointer information, are undecidable. Even in programs where it is decidable, computing perfectly accurate information is often prohibitively computationally expensive. Hence, static analysis tools can resort to either over-approximating or under-approximating or both. An over-approximation includes the effects of all possible executions and hence never misses an error, but it may also include impossible executions, and this could produce false-positive error warnings. An analysis that perfectly over-approximates is said to be *sound*. An under-approximation only includes effects of executions that are actually possible, but it may miss out on some of the possible executions, and hence could miss some errors (false-negatives).

Precision and Recall. The over-approximation and under-approximation of a static analysis are quantified using *precision* and *recall* respectively.

Precision is the ratio of true-positive errors to the total number of errors output by the static analysis. Precision can also be thought of as true-positive rate. *Recall* is the ratio of true-positive errors output by the static analysis to the ground-truth errors. A perfectly sound analysis has 100% recall.

$$\textit{Precision} = \frac{|\text{True-positive errors}|}{|\text{Total errors by analysis}|} \qquad \textit{Recall} = \frac{|\text{True-positive errors}|}{|\text{Ground-truth errors}|}$$

Getting an accurate ground-truth is typically challenging. Researchers have used manual programmer annotation, results from running the program, or the results of a reference tool as approximations for the ground-truth. Each of these have their own advantages and problems, and the best source of ground-truth may depend on the specific benchmark-set and analysis.

Inter-procedural and Intra-procedural analyses. A static analysis is intraprocedural if it analyzes a method in isolation. A static analysis is interprocedural if it models effects and dependencies across method calls. A special case of an interprocedural analysis is a whole-program analysis, which analyzes the whole program as one and captures all the inter-procedural effects in the program. Researchers have shown that an interprocedural analysis is often needed for accurate static analysis results on most real-world programs.

Data-flow and Control-flow. Control-flow is the order in which statements of an imperative program are executed. An intra-procedural Control-Flow Graph (CFG) represents this control flow as a graph with the instructions as nodes and edges between instructions a and b if b appears immediately after a in some execution. For example, for code of the form

(`S0; if(a) S1; else S2;`), there will be one edge from $S0$ to the if statement, and two outgoing edges from the if statement: one to $S1$ and the other to $S2$. An inter-procedural control-flow graph needs to additionally depict the targets of method calls.

Data-flow represents the flow of data through the nodes of the control-flow graph. The kind of data we are interested in depends on the analysis. For example, for a null-pointer analysis on the program (`x = null ; y = x`), the data we are interested in is the null value and it flows from x to y.

Both data-flow and control-flow are typically over-approximated in static analysis. Hence, if control can flow in two directions depending on the input, the control-flow graph shows that both are possible. Similarly, if two data values flow into a given node, we assume that the node can potentially get both values.

Static Single Assignment. Static Single Assignment or SSA is a property of an IR (Intermediate Representation) that it requires each variable to be assigned once. Static analysis tools often transform code into SSA form before analyzing it because it simplifies the analysis passes significantly. Source-code variables that have multiple assignments are split into multiple variables in the SSA form, each with a single assignment. Instructions in SSA also often have a single operation with operands being variables and constants (no complex expressions).

2.2 Applications of Static Analysis

Here are some of the applications of static analysis:

1. **Security:** Security is the application that has gained the most interest recently. Static analysis is especially useful here since security vulnerabilities will not show up during unit testing. Some of the most common and severe security vulnerabilities such as SQL-Injection, Cross-site scripting, etc. can be identified with a static taint anal-

ysis [TPF09a]. Some other kinds of vulnerabilities can be cast as static type-state analysis [FYD08] problems.

2. **Preventing program crashes:** Errors such as null-pointer warnings, resource leaks, buffer-overflows, etc. can cause a program to crash, and occasionally also cause security vulnerabilities. Static analysis can identify and warn programmers of such errors at compile time [BCS19, KSS21a].

3. **Automated program repair:** Static analysis results can enable automated repair tools for certain kinds of errors such as null-pointer errors [LHO22, XSY19], memory leaks [HLL20, GXM15, LHO18], and buffer overflows [SDH14]. The category of errors here has a large overlap with the category of errors that a static analysis can catch.

4. **Program Optimization:** A program could potentially be optimized based on the results of a static analysis. For example, static analysis can help identify dead-code, common sub-expressions, optimal register allocation, loop optimization opportunities, etc. This application, however, is the oldest and most well-studied and hence will not be explored in this thesis.

5. **Others:** There are several other applications of static analysis such as program understanding, greybox fuzzing, etc. that we don't discuss in detail here.

For this thesis, we focus our experiments on errors that can cause program crashes such as null-pointer errors and cast-check errors, as well as automated program repair. However, our techniques are not tied to a particular application; they benefit the other applications as well.

2.3 Example

While most of the concepts in this discussion apply to other programming languages, we specialize to Java static analysis for the rest of this chapter to keep our discussion concrete.

```
1  class A {
2      void foo(int option) {
3          Scanner sc = new Scanner(System.in);
4          String userId = sc.nextLine();
5          B x = null;
6          if (option == 1){
7              x = new B();
8          }
9          else if (option == 2) {
10             x = new C();
11         }
12         x.bar(userInput);
13     }
14 }
15
16 class B {
17     void bar(String userId) {
18         System.out.println("UserID:" + userId)
19     }
20 }
21
22 class C extends B {
23     void bar(String userId) {
24         String query = "select * from T where user_id =" + userId
25         ResultSet rs = SqlUtil.stmt.executeQuery(query);
26         SqlUtil.printResultSet(rs);
27     }
28 }
```

Figure 2.1: Java program snippet with a potential SQL Injection vulnerability and null-pointer error. We can catch these using static analysis tools.

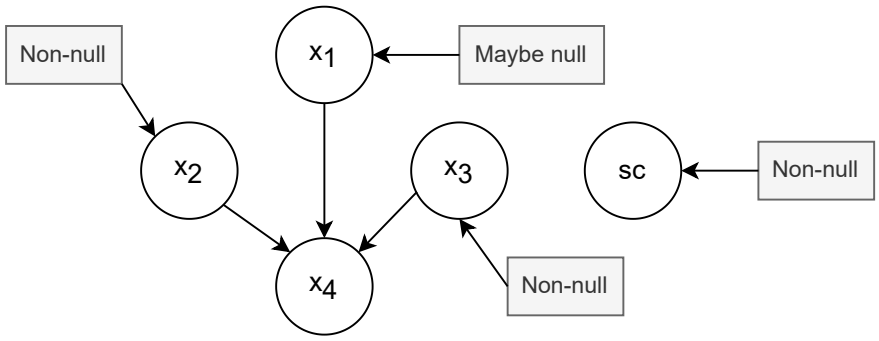


Figure 2.2: Data-flow graph of a null-pointer analysis for method *foo* in Fig. 2.1.

We will use the Java code-snippet from Fig. 2.1 as the running example in this chapter to illustrate how we analyze code using static analysis. The figure has 3 classes A, B and C, with C being a subclass of B. Execution starts at the `foo` method, where a user-id is read and passed in the method call to `bar`. Classes B and C both have a `bar` method. In `B.bar` we just print the user-id, whereas in `C.bar` we make an SQL query with the user-id and then print the result.

Let us now look at three analyses, a null-pointer analysis, a call-graph analysis and a taint analysis, for the code snippet from Fig. 2.1

Null Pointer Analysis. A null-pointer analysis tracks the flow of null values through the program, and flags method calls or field accesses where the receiver object could be null. [1] For example, in Fig. 2.1 the variable x is initially assigned to null. Then, if *option* is either 1 or 2, x gets assigned to some object. However, if *option* has any other value, x remains null, and the method call on line 12 throws a null pointer exception.

Fig. 2.2 gives the data-flow graph for an intra-procedural null-pointer analysis of *foo*. There is one node for *sc*. There are 4 nodes for x, (x_1, x_2, x_3, x_4), one for each of the 4 definitions of x on lines 5, 7, 10, and 12 respectively. (In SSA form, these would anyways be 4 different variables). We leave out the *userId* node for brevity. There are only two possible

[1] It additionally tracks null values originating from uninitialized fields, but we will ignore these kind of null values for now.

values in this analysis: *non-null* and *maybe-null*. Only nodes that are guaranteed to not be null get *non-null*; all others get *maybe-null*. *sc*, x_2 and x_3 clearly get assigned a non-null value, whereas x_1 gets assigned a maybe-null value. $x4$ potentially gets its value from x_1, x_2 or x_3, depending on the value of *option*; hence we add an edge from each of these to $x4$. Solving for the data-flow here simply involves taking a transitive closure over the graph. Here are the results from the transitive closure.

- x_1 : *maybe-null*, x_2 : *non-null*, x_3 : *non-null*, x_4 : *maybe-null*, *sc* : *non-null*

The results for x_1, x_2, x_3 and *sc* are trivial. x_4 gets its value from both, *maybe-null* and *non-null* values. Therefore, there are executions where x_4 could be null, and since we are performing an over-approximation, we set its final value as *maybe-null*. In the general static analysis framework, the set of values are organized in a lattice, and to over-approximate, the result of merging two values is given by the common ancestor of the values in the lattice. A more in-depth discussion of this lattice structure can be found here [ALS06].

Finally, since x_4 (i.e. x at line 12) can be null, the method call *x.bar(userInput)* is flagged as a null-pointer error.

Call Graph Analysis. A call graph is a graph of all possible method calls in the program. It is necessary for any inter-procedural analysis. Call-graph analysis in Java is hard because of Java's *dynamic dispatch* feature (also called virtual method calls). Java programs are object-oriented, with a class hierarchy where a subclass inherits all the methods and fields of its super class. If a subclass implements the same method as its super class (called function overloading), we get *dynamic dispatch* for method calls, where the target of the call depends on the type of the receiver object. For example, in Fig. 2.1, the receiver object in the method call to `bar` is x. The declared type of x is B, but at runtime x could point to either a B object, or a C object, and they both implement `bar`. Hence, the runtime target of the method call to `bar` on line 12 depends on the runtime type of the receiver object x. An

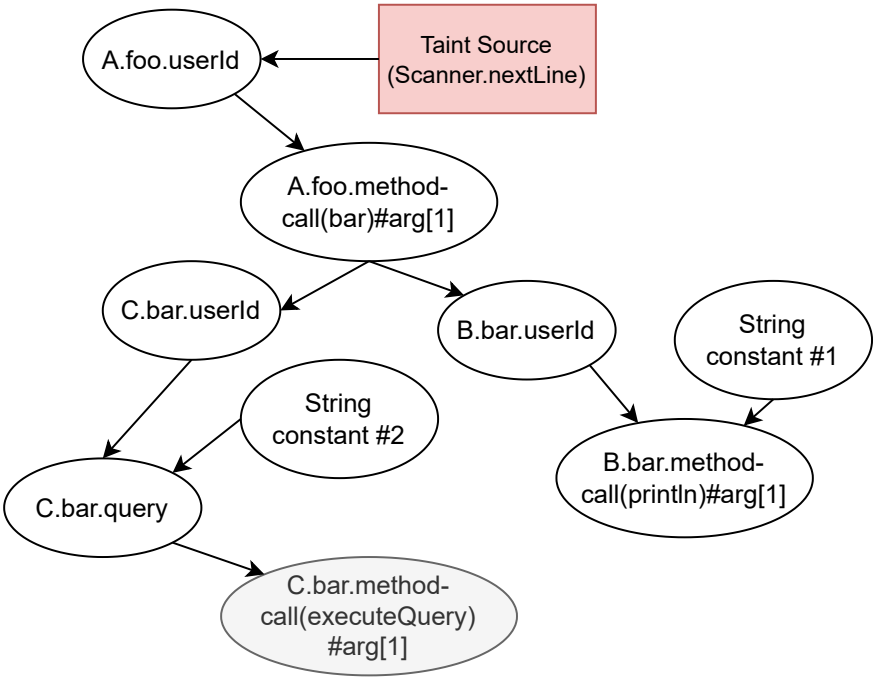


Figure 2.3: Data-flow graph of a static taint analysis for the code in Fig. 2.1.

accurate call-graph analysis estimates a tight over-approximation on the runtime types in order to get accurate method call targets and build an accurate graph of method calls.

Static Taint Analysis. Next, let us look at an inter-procedural taint analysis for the SQL-Injection vulnerability in Fig. 2.1. A static taint-analysis takes as input an untrusted source, such as the user input on line 3, and a vulnerable sink, such as the SQL query on line 25. It then outputs whether there is a path in the program from the source to the sink without any sanitization code in between. The taint source and sink have to be specified, but the set of potential sources and sinks for a given language and framework are fixed; hence the analysis writer can define this upfront.

Fig. 2.3 gives the data-flow graph for the analysis (also called taint-propagation graph). In addition to modeling the intra-procedural edges, this graph also includes the two call-graph edges from the *bar* method call to *C.bar* and *B.bar*. There are edges to both methods because the call-graph analysis could not statically estimate the target, and an over-approximation

strategy requires us to add both edges. The only taint source here is the call to *sc.nextLine()*. Solving for the data-flow here simply involves taking a transitive closure over the graph. Since there is a tainted-path to the *executeQuery* sink node, the transitive closure will mark the *executeQuery* sink as tainted, and the analysis gives an alarm for a possible SQL injection vulnerability.

We explained some examples of data-flow and control-flow analysis very briefly and informally here. For a more comprehensive and formal discussion, refer to the following textbook [ALS06].

2.4 Challenges and Limitations

We touched on some of the main challenges in static analysis briefly in the last section, but we discuss them in more detail here.

Soundness

Soundness is the ability to perfectly over-approximate the set of possible executions (i.e. 100% recall). It is valuable for verification because if we can be sound in our analysis of an error, and there are no static analysis warnings, we can guarantee the absence of that error in all possible program executions. While it is possible to achieve perfect soundness in theory, there are a few obstacles to achieving it in practice.

1. **Reflection:** Reflection allows a developer to examine or modify the behavior of methods and classes at runtime. For example, a user can call methods by their name, create objects given their class names, and access object fields given their name. The challenge is that this dynamic behavior is hard to model statically. If the reflection uses user-input, it may even be impossible to model this behavior statically.

   Nevertheless, there are two major approaches to partially resolve reflective calls. The

first is static resolution, which uses points-to and type-cast information to narrow down the possible reflection targets [LWL05]; the targets can be further narrowed using some user annotations. The second method involves recording the method calls dynamically (i.e. at runtime), and augmenting the static analysis with any missed edges [BSS11]. The first method is an over-approximation which introduces many false-positive edges, whereas the second method is an under-approximation which may miss some reflection edges (i.e. false negatives).

2. **Native code:** The Java Native Interface (JNI) allows a Java program to call native code (external methods from a binary). Since this native code is often unavailable at compile time or is hard to analyze with the Java code, the static analysis may miss out on its effects.

3. **Dynamic class loading:** Java programs can load classes at runtime, using a feature called *dynamic class loading*. Since these classes may be unavailable at compile time, it is difficult to model their behavior statically.

Given that these features make a perfectly sound static analysis (i.e. 100% recall) infeasible, practitioners opt for a *soundy* analysis [LSS15a], which is a sound analysis except for these features.

This inability to reach 100% recall adds indirect support for our idea of trading off recall (i.e. soundness) for the criteria that users care about. Not only is perfect soundness not important to users, but it is also infeasible to achieve in practice.

High Precision (fewer false-positives)

Christakis and Bird's developer survey [CB16a] shows that static analysis tool designers should aim for at least 80-85% precision. Since most vanilla static analysis implementations don't meet this goal, designers often use one or more of these strategies to improve precision:

1. **Improving Sensitivity:** The most well-studied strategy is to improve sensitivity, which means having a more fine-grained modeling of some aspect of the program. Field-sensitivity [LH03a] models field accesses more precisely, context-sensitivity [Shi91] models the calling context more precisely, flow-sensitivity [DD12] models the control flow more precisely, and path-sensitivity [DDA08] models branch conditions more precisely. We elaborate on each of these sensitivities later in this section. Improving sensitivity often improves precision, but it comes at the cost of (often significant) increased analysis time. The advantage of this precision-improving strategy over the rest is that it preserves soundness. No recall is lost. All other techniques will trade off some recall.

2. **Machine-learning based methods:** Several researchers have proposed using machine learning to prune false-positives from static analysis results [RPM08, TGP14, FSS18, ULK22]. The typical workflow is as follows: collect static analysis results, get ground-truth labels for the results (using manual annotation, dynamic analysis, etc.), design a feature-set for the results (or learn the features using neural networks), and then train a classifier on these labeled results to identify false-positives.

3. **Combining with dynamic analysis:** This third strategy involves using the results from a dynamic analysis (see Dynamic Analysis in Section 2.5) to refine the results of a static analysis. The dynamic information is used to either replace the static information for a part of the analysis [GFF18b], or it can be used to prioritize alarms [CHR21, AKG07]. The main drawback of this approach is the same as that faced by dynamic analysis tools: we may not have test inputs, the inputs may take too long to run, or the inputs may not have good coverage of the code.

4. **Alarm correlation:** Two static analysis alarms can be correlated if they share a root cause. Correlated alarms can introduce false-positives because the second alarm examined in a correlated pair is often already fixed (because of the fix for the first

alarm) and hence becomes a false-positive. Identifying alarm correlations can help reduce these false-positives, because one can show just one alarm for every correlated pair [LLK17, LS10].

5. **Heuristic pruning:** One could come up with heuristic rules for pruning false-alarms. These heuristics are often specific to a language, platform and error type, but they are simple to implement and can work well in practice.

Let's take a more detailed view of context-, flow-, path-, and field-sensitivity.

- **Context-sensitivity:** A context-sensitive analysis considers the calling context of a function when analyzing it. The two main types of sensitivity are call-string sensitivity and object-sensitivity. Call-string sensitivity [Shi91] maintains a separate context for every unique caller method. For example, in Fig. 2.1 we would analyze a separate copy of *foo* for every different caller. This would prevent the merging of information from different callers. Object sensitivity [MRR05] maintains a separate context for every unique receiver object. Object sensitivity is especially effective for analyzing libraries (such as the Collections library) in Java. Since context-sensitivity significantly increases analysis time, researchers have attempted to selectively apply context-sensitivity to the methods that need it the most [LTM20].

- **Flow-sensitivity:** A flow-sensitive analysis takes into account the order of statements in a program, whereas a flow-insensitive analysis simply treats a method as a set of statements with any possible order. Code, when analyzed in SSA form, automatically gets partial flow-sensitivity.

- **Path-sensitivity:** A path-sensitive analysis records the branch condition when analyzing code in a branch. For example, in the following code (`if(x != null) x.toString();`), when analyzing the branch with `x.toString`, a

path-sensitive analysis tracks that `x` is not null, and hence rules out a null-pointer error.

- **Field-sensitivity:** A field-sensitive analysis models field accesses in a fine-grained fashion. In the data-flow graph, it maintains a separate node for every instance's field. Hence, if there are I instances of a class which has F fields, it would maintain $I * F$ nodes for the fields of that class. Field insensitive analyses are split into two types: field-based analysis and field-independent analysis. A field-based analysis maintains just one node per field, and merges information from all the instances that use the field. A field-independent analysis distinguishes between different instances, but merges the information from all the fields of a given class or struct. This saves analysis time in C/C++, where fields are accessed by offsets from the object pointer. In Java, however, since fields are accessed by their names, it makes no sense to use a field-independent analysis. Hence, a field-insensitive in Java often specifically refers to a field-based analysis.

Data-structures such as arrays and hash-maps present another challenge to high-precision analysis. The main challenges with array is that it could have an unbounded number of elements (could be unknown statically), and we need to accurately identify the relationship array indices and the actual size of the array. A first approach is to solve this is *array smashing* [BCC02], where we over-approximate the whole array with a single element in the analysis; any read or write could go to any index in the array. This solution is sound but it compromises on precision because information from different array indices gets merged. Some advanced techniques improve precision by partitioning the array and using a different element in the analysis for each partition [GRS05a]; this is more precise than array smashing, but it still merges some information. More complex data-structures such as hash-maps, linked-lists and trees are even harder, and have no good solutions apart from a corresponding version of array smashing.

Analysis time

Depending on when the static analysis runs there could be different timing expectations: in an IDE it needs to run in seconds, in a code-review platform it needs to run in minutes, and in a nightly-build scenario, it needs to run in hours. However, Christakis and Bird's developer survey [CB16a] show that most users expect the static analysis to run in the IDE or code-review platform, thereby giving us strict time requirements. Computation speeds have increased over the last two decades, but so have codebase sizes and library dependencies; analysis time continues to be a concern. It is especially a concern for large programs with millions of lines of code, or a large collection of programs, like an app-store with a million apps. Without efficient analysis strategies, these large cases could take days or even years of compute time. There are 4 popular strategies to improve analysis time:

1. **Efficient analysis:** Efficient set propagation algorithms, efficient set implementations and propagation-graph simplifications significantly improve analysis speeds [LH03b].

2. **Decrease analysis precision:** Decreasing analysis precision by using more coarse-grained abstractions generally decreases time complexity, and hence analysis time. For example, one can merge the sets used to represent different allocation sites, or use static declared types instead of types computed using a data-flow analysis. Dropping context-, flow-, path-, and field-sensitivities also trade-off precision for a quicker analysis. The main disadvantage of this approach is the reduced precision (i.e. increased false-positive rate). Researchers have tried to mitigate this by selectively reducing sensitivity only when it doesn't significantly affect precision [LTM20, RRL99].

3. **Demand-driven analysis:** A demand-driven analysis [SGS05a, HT01] only computes the analysis results for a single query (i.e. demand). Since this analysis computes much less information than a whole-program analysis it is significantly faster when the number of queries are small. However, it adds overhead which is proportional to the

number of queries, eventually making it slower than a whole program analysis. Thus it is most effective for quick results on a single or small number of queries.

4. **Partial-program analysis:** A partial program analysis analyzes only a part of the program isolation, while using a model or some assumptions to account for the effects from the rest of the program [AL13, RSX08, CC02, UP22].

5. **Incremental analysis:** Codebases often evolve incrementally, and rerunning an analysis after every update can do redundant analysis on the unchanged code. An incremental analysis [AB14, LH22] gets rid of this redundancy by caching the results of a static analysis and updating its results to incorporate incremental code changes. The assumption here is that if the incremental code change is small, the analysis result update will also be relatively small and much cheaper than a fresh whole program analysis.

The issues of soundness, precision and analysis time are intertwined because trying to maximize two of the metrics often comes at the cost of the third. The challenge is to strike the right balance between these three.

Automated Repair

As Christakis and Bird's developer survey [CB16a] pointed out, users expect automated repair suggestions for static analysis warnings, and it was the third most important point on the list. However, there are two major hurdles to achieving this. Firstly, we need to track the chain of variables that propagated an error. This chain, called the provenance information, is often computationally expensive to generate. Secondly, we often do not have an accurate specification for what the correct program behavior should be. For example, a null-pointer warning was given at the call to *bar* in Fig. 2.1. What should the correct fix be? There are at least 3 options:

1. Place a non-null check before the call to *bar*.

2. Add an else branch in *foo*, where x is assigned to a new object.

3. Add an else branch in *foo*, with a user-defined exception.

Picking the correct option depends on the intention of the programmer, and could be hard to guess correctly. An automated tool would need some kind of specification to pick the correct option, and writing such a specification is often quite hard for a programmer; it is often harder than making the fix itself.

There are some partial solutions to this repair problem for some kind of static analysis warnings [HLL20, LHO22] but for the most part, this is an open research problem.

Ranking of Warnings

All warnings given by the static analysis are not equally important; ranking the alarms by importance will allow users to examine the most important ones first. This was reported as the ninth most important feature in the list of pain-points from Fig. 1.1. One simple approach to this problem is to rank warnings by their severity. In this case, an expert gives a severity score for each analysis, and the warnings from multiple analyses are ranked by this severity score. Another approach to rank warnings is by their probability of being false-positives [KE03, JKS05]. Taking this a step further, one can incorporate the user's feedback in real time about which of the examined warnings were false-positives, and re-order the rankings of the remaining warnings accordingly [RKH18a, MZN15a]. For example, if warning-1 was marked as a false-positive by the user, and warning-2 has the same root cause as warning-2, we can downgrade warning-2 because it is also likely to be a false positive.

Dealing with Libraries

An important challenge for Java is its extensive ecosystem of libraries. The standard library and third party libraries are very large, often making the whole-program size several fold bigger than just the application code. Furthermore, library methods often use Java Generics and get called from multiple application methods; this necessitates over-approximation, which in turn can lead to more false positive warnings.

There are two main approaches to minimize the increase in analysis time and false-positive rate caused by libraries. The most effective approach is to manually write *stubs* for common library methods that capture the effects of analyzing the library. The stubs are usually quite small and context-sensitively analyzing them prevents loss of precision. The downside, however, is the manual effort by a static analysis expert in writing correct stubs. A second approach is to create an analysis summary for the library, and then use this library summary instead of the actual library for the analysis [AL13, RSX08, UP22].

Android Static Analysis.

Static analysis has been used extensively to analyze Android applications. Getting Java static analysis tools to work on Android requires us to additionally model the life-cycle methods in Android (such as the `onStart` or `onPause` methods) as well as event call-backs (such as when a user presses a button). Today, most popular Java analysis frameworks such as WALA, Soot or Infer provide support to automatically model these aspects of Android programs. These frameworks build a call-graph that incorporates these additional features, and we can build our tools on top of this call-graph. Hence, even though this thesis focuses on the analysis of regular Java programs, its techniques can easily translate to Android programs using the support provided by these analysis frameworks.

Other programming frameworks such as Java enterprise applications or Java web applications require similar modeling of features outside of the code that can affect the data flow

and control flow.

Limitations of the technique

Static analysis tools only catch a certain class of errors, and cannot ensure that the program is free of other kinds of errors. They generally target the kind of errors that are common to all programs, and don't require the tool to know the logic of the programmer's intention. For example, in a banking application, a static analysis tool can catch all null-pointer errors and resource leaks, but it cannot check if the programmer's code deducts the correct amount of bank fees for a transaction.

2.5 Other Program Verification Techniques

Next, I discuss some other program verification approaches to give the reader a broader understanding, as well as distinguish static analysis from these other approaches.

1. **Symbolic execution:** Symbolic execution [PR10] involves running an interpreter for the program with symbolic values for inputs rather than actual inputs. The values of expressions and branch conditions can then be expressed in terms of these symbolic values. Finally, one can generate a query for an error condition by asking whether there exist any symbolic values that could satisfy the error condition. Typically, this query is solved using a model checker [Cen05], which uses a SAT [MMZ01] or SMT [DB08] solver in the backend.

   The advantage of this approach is that it models all properties of a program and can be used to verify arbitrary properties and assertions. However, the main downside of this approach is that it scales very poorly; often, analyzing even a thousand lines of code times out. Furthermore, verifying correctness with this approach involves writing accurate mathematical specifications, which is hard for both users as well as automated

tools.

2. **Linters:** Linters [web01] parse a software to catch stylistic errors or simple bugs. They are typically very local and they only analyze a single line or a couple of lines at a time. They are often simple to write and run very quickly, but unlike static analysis tools, they often cannot catch errors that span multiple lines or methods.

3. **Type checking:** Type checkers verify and enforce type-based constraints over values. For example, a type-checker can statically check that a variable declared to be an integer type only holds integer values. More complex type systems can check a broader set of type-based conditions.

   Type checkers are often incorporated into standard compilers because they are efficient, accurate, and their errors are easily understood by programmers. However, the type errors that most popular type-checkers catch have only a small intersection with the kind of errors tackled by static analysis tools.

4. **Dynamic Analysis:** A dynamic analysis executes the program on concrete inputs, and collects runtime information about the program execution. It then flags an error if the runtime information does not match with an expected value. Popular dynamic analysis techniques include unit testing, debugging, runtime error detection [NS07], and fuzzing [PLS19].

   The main disadvantage of dynamic analysis is that it can only test a finite number of input-output pairs. It will leave out all the program execution paths that are not covered by these finite inputs. A secondary disadvantage is that it is hard to apply in cases where inputs are not available or the program takes very long to execute.

Note that some researchers use *static analysis* to refer to any analysis which does not run the program, including symbolic execution, linters, static type-checking, etc. I, however,

use *static analysis* in this thesis to specifically refer to the kinds of analyses discussed before Section 2.5.

CHAPTER 3

CGPruner: Pruning False-Positives from Static Call Graphs

In this chapter, we introduce the first tool, *CGPruner* [ULK22], which tackles the first biggest pain point for users: the large false-positive rate in existing static analysis tools. *CGPruner* achieves this by pruning the static call-graph that sits at the core of all inter-procedural static analyses. To demonstrate the impact of this call-graph pruning on a downstream analysis, we added *CGPruner* to a null-pointer analysis, and decreased its false-positive rate from 73% to 23%.

3.1 Overview

The Problem. Christakis and Bird [CB16a] interviewed developers about program analysis tools and they concluded:

> *Program analysis design should aim for a false-positive rate no higher than 15–20%.*

Other empirical studies have found similar results [BBC10a, JSM13c, SAE18b]. Until now, this goal has been particularly hard to achieve for *static* analyses.

As a motivating experiment, we tried Wala [WAL15], which is one of the best tools for static analysis of Java bytecode, on a subset of the NJR-1 benchmark suite [PL18]. For each benchmark, we compared the edges in the static call graph with the edges found by

executing the benchmark. With a context-insensitive analysis, Wala has a false-positive rate of 76%, while with a better but also much slower context-sensitive analysis, the false-positive rate is 70%. Those results are disappointing though we must emphasize that call graphs are usually fed to client tools rather than directly to developers. So, we did a second experiment to see how the high false-positive rate of call-graphs affects client tools. Specifically, we implemented a version of a static analysis for warning about null-pointer problems [HJP08] that is a client of the context-insensitive call graphs produced by Wala. We ran this tool on the same subset of NJR-1 and again had disappointing results: 60 bugs among 223 warnings, on average, so a false-positive rate of 73%. We can easily imagine how a developer will tire of investigating warnings that in nearly three of every four cases are false alarms. The false alarms have several causes, but an important cause is the high false-positive rate in the underlying static call graph. Hence, we can also see a glimmer of hope: if we can reduce the false-positive rate of static call-graph constructors, we may be able to move client tools closer to the goal of a false-positive rate of 15–20%.

Our Idea. Our approach stems from another conclusion by Christakis and Bird [CB16a] who reported a preference of developers:

> *When forced to choose between more bugs or fewer false positives, they typically choose the latter.*

This quote inspired our idea for how to improve the false-positive rate: we will report *fewer* bugs but also *much fewer* false positives. Indirect support for this idea comes from previous work that showed that practical static analyses aren't totally sound [LSS15a, SDT20] and therefore may miss bugs. Thus, developers expect bug reports to be incomplete so reporting fewer bugs seems acceptable.

We want to reduce the false-positive rate in a modular way that leaves existing call-graph constructors unchanged. This brings us to our idea of a *call-graph pruner* that statically

post-processes a static call graph by removing *many* false-positive edges but *few* true edges. The challenge is to strike a balance between being aggressive in removing false-positive edges but not so aggressive that no true edges remain. Additionally, we have to do better than removing edges at random because random removals will leave the false-positive rate unchanged.

How can we design a call-graph pruner?

Our Approach. We execute an automatic, ahead-of-time learning process on results from both a static and a dynamic call-graph constructor. The outcome is a call-graph pruner that works as follows. The call-graph pruner determines the probability that an edge in the call graph is a false positive, and if this probability is above a threshold, then the call-graph pruner removes the edge. We can vary this threshold and thereby tune the call-graph pruner.

In contrast to previous work on using a dynamic analysis to improve a static analysis [GFF18b, AKG07, CHR21], we use the dynamic call-graph constructor only in an ahead-of-time training phase and only on a training set of programs. Once the training phase has produced a call-graph pruner, the combination of the call-graph constructor and the call-graph pruner is itself a static analysis, as illustrated in Figure 3.1.

Our Contributions and the Rest of the Chapter. We begin with an example of how a call-graph pruner works (Section 3.2) and then we detail our contributions:

- We present the design (Section 3.3) and implementation (Section 3.4) of a tool that produces call-graph pruners.
- We show experimentally (Section 3.5) that adding a call-graph pruner to a client tool can significantly decrease the false-positive rate, in one case from 73% to 23%. Specifically, we added a call-graph pruner to the tool for warning about null-pointer problems,

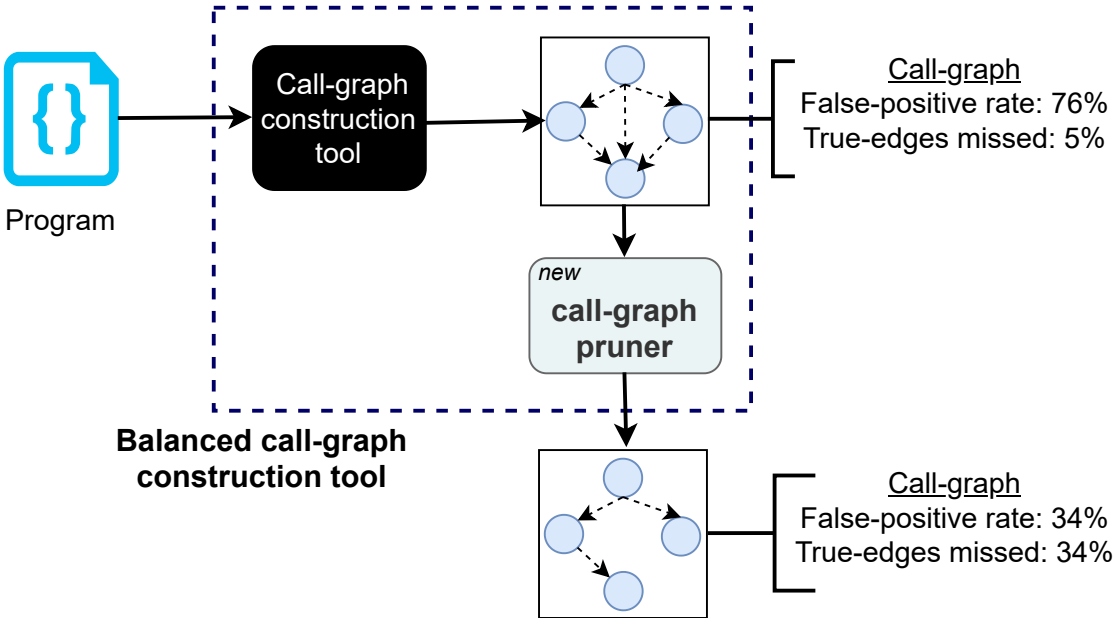


Figure 3.1: Overview of our technique

after which we got 15 bugs among 20 warnings, on average. Thus we reported 45 fewer bugs but also 158 fewer false positives.

- We show experimentally (Section 3.5) that the overhead of adding a call-graph pruner is 18% of the original call-graph analysis time.

We end with a discussion of related work (Section 3.6).

Significance. Call-graph pruners improve static call-graphs significantly and thereby make client tools more useful to developers.

3.2 Example

Now we give an example of a call-graph pruner, how it works on a example call graph, and how it affects a client analysis for warning about null-pointer problems. Our example program fragment in Figure 3.2, has three classes `A`, `B`, `C`, each of which has a method `foo`,

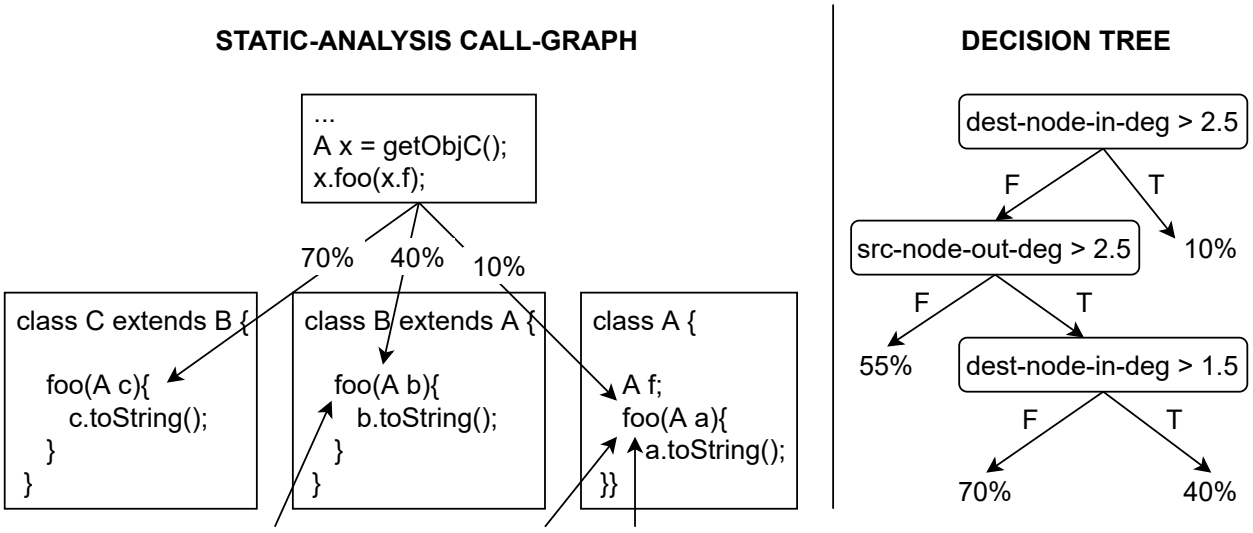


Figure 3.2: Example call graph and call-graph pruner

and a main method that contains a method call `x.foo(x.f)`. The call to `getObjC()` returns an object of type `C`, which is then assigned to the variable `x`. On the next line, the access `x.f` happens, but the field `A.f` may be uninitialized hence null. Thus the call `x.foo(x.f)` may pass null as an argument to `C.foo`, which, in turn, at the call `c.toString()`, may throw a NullPointerException. The program has two additional methods, including `getObjC`, that we omitted from Figure 3.2.

Null-Pointer Warnings. As we mentioned in Section 3.1, we implemented a version of a static analysis for warning about null-pointer problems. This analysis finds null-pointer problems that stem from uninitialized fields, like the problem with `c.toString()` that is caused by the uninitialized field `A.f`. If we run this tool on the example program, we get *three* warnings, one for each call of `toString` in the `foo` methods. One of them is a true warning but the other two are false alarms. Let us investigate how that could happen and what a call-graph pruner can do about it.

Call Graph. The null-pointer tool uses a static call-graph constructor that built the call graph shown in Figure 3.2. In a call graph, each node is a method, and each edge is a directed edge from one method to another. Such an edge represents a call that *may happen* during the execution of the program.

The call-graph constructor uses a data-flow analaysis to analyze the entire program, including the methods that we omitted from Figure 3.2. We skip the details of how this works and instead we focus on the constructed call graph. Specifically, in Figure 3.2 we focus on the four nodes for the main method, `A.foo`, `B.foo`, and `C.foo`. The call graph has an edge from the main method to each of `A.foo`, `B.foo`, and `C.foo`, as well as an edge some other method to `B.foo` and a couple of edges from some other methods to `A.foo`. The edge from main to `C.foo` is a true edge, while the edges from main to `A.foo` and from main to `B.foo` are false positives. The false call-graph edges from main to each of `A.foo` and `B.foo` can arise from some of the challenges discussed in Section 2.4.

The Null-Pointer Analysis in more Detail. Based on the call graph in Figure 3.2, the null-pointer analysis derives that `x.foo(x.f)` may call any of `A.foo`, `B.foo`, and `C.foo`. Then the null-pointer analysis uses the rule that if a field is not initialized by the end of a constructor, it is marked as *Uninitialized*; and if an *Uninitialized* field is dereferenced, the analysis gives a null-pointer warning. Thus, the analysis concludes that each of the `foo` methods may be passed null as an argument, and thus it issues a warning for every one of those methods.

Call-Graph Pruner. The goal of a call-graph pruner is to remove edges from the call-graph, preferably many false-positive edges and few true edges. The key component of a call-graph pruner is a classifier that computes the probability that a call-graph edge is a true-positive. Based on that probability, a call-graph pruner will decide whether to keep or to remove the edge. Figure 3.2 shows a classifier that is represented as a decision tree.

Each internal node of the decision tree asks a true-false question about a call-graph edge. The recursive decision process begins in the root of the decision tree; if the answer to the question at the root is false, we move to the left subtree, while if the answer is true, we move to the right subtree. When we reach a leaf, we find the probability that the call-graph edge is a true-positive. The probabilities computed for each call-graph edge in this fashion are marked on the call graph in Figure 3.2. Based on these probabilities, we will decide whether to keep or remove the call-graph edge.

The decision tree in Figure 3.2 has three internal nodes that are labeled with questions about *dest-node-in-deg*, which is the in-degree of the destination node of the edge, and about *src-node-out-deg*, which is the out-degree of the source node of the edge. For example, the edge from main to `C.foo` has destination-node in-degree 1 and source-node out-degree 3. This gives us the path false-true-false, which assigns the edge the probability 70%. Similarly, the edges from main to `A.foo` and `B.foo` get probabilities 10% and 40%, respectively. The call graph in Figure 3.2 shows those three probabilities.

Let us set a threshold of 50% for when we deem an edge to be a false-positive: if the probability of being a true-positive is below 50%, we remove the edge. Then the call-graph pruner will remove the edges from main to `A.foo` and `B.foo`. Hence, the null-pointer analysis will issue just a single warning, and indeed a true warning, namely for the call of `toString` in `C.foo`.

3.3 Call-Graph Pruners

Now we describe how we use machine learning to produce a call-graph pruner.

3.3.1 Overview

We will use Program to denote the set of Java bytecode programs.

A call graph $G \in \mathsf{CallGraph}$ is a multi-graph in which each node represents a method and each edge represents a potential transfer of control at a method call. Two nodes can have multiple edges between them because of multiple method calls. Each edge has a label that identifies the method call site.

We distinguish between two kinds of call-graph constructors that have the same type:

$$\begin{aligned} \mathsf{StaticCallGraphConstructor} &= \mathsf{Program} \to \mathsf{CallGraph} \\ \mathsf{DynamicCallGraphConstructor} &= \mathsf{Program} \to \mathsf{CallGraph} \end{aligned}$$

Here, an element of StaticCallGraphConstructor constructs a call graph without running the program, while, in contrast, an element of DynamicCallGraphConstructor runs an instrumented version of the program on one or more inputs and examines the output from the instrumentation.

The key component of each call-graph pruner is a classifier. A classifier $C \in \mathsf{Classifier}$ is a function that maps a vector of feature values for an edge to a probability that the edge is a true-positive. In our case, such a vector has 11 elements that we will define in Section 3.3.3.

Our tool for generating classifiers implements a function of this type:

$$\begin{aligned} \mathsf{classifier\ generator} \ : \ & (\mathsf{StaticCallGraphConstructor} \times \\ & \mathsf{DynamicCallGraphConstructor} \times \\ & \mathsf{Set[Program]}) \\ & \to \mathsf{Classifier} \end{aligned}$$

Our classifier generator executes an automatic, ahead-of-time learning process on results from running both a static and a dynamic call-graph constructor on a training set of programs. The dynamic call graphs serve as ground-truth for the learning process. We will detail this learning process in Section 3.3.2.

```
Inputs: CallGraph G, Classifier C, Threshold T
let G' be a copy of G
for every edge e in G do v = the feature values for e
    if C(v) < T then
        remove e from G'
    end if
end for
Output G'
```

Figure 3.3: Call-graph Pruner

Once we have a classifier, we can use it in a call-graph pruner of this type:

$$\textsf{call-graph pruner} : (\textsf{CallGraph} \times \textsf{Classifier} \times \textsf{Threshold}) \rightarrow \textsf{CallGraph}$$

The algorithm in Fig. 3.3 shows how a call-graph pruner works. Intuitively, a call-graph pruner uses a classifier to determine the probability that an edge in a static call graph is a true-positive. If that probability is below a given threshold $T \in \textsf{Threshold}$, the call-graph pruner removes the edge.

The threshold parameter enables us to explore different levels of aggressiveness in removing edges. For our example in Figure 3.2, we discussed a threshold of 50% in Section 3.2, which led to the removal of two edges. We could also use a lower threshold of 20%, which would lead to the removal of a single edge, namely the one from main to `A.foo`. The challenge is to strike a balance between removing many false-positive edges and keeping many true-positive edges. In Section 3.5 we will show results from an experimental investigation of how to choose a good threshold.

Notice that we use a static call graph constructor, a dynamic call graph constructor, and the training set of programs for the sole purpose of generating a classifier, while those items are no longer needed when we use the call-graph pruner.

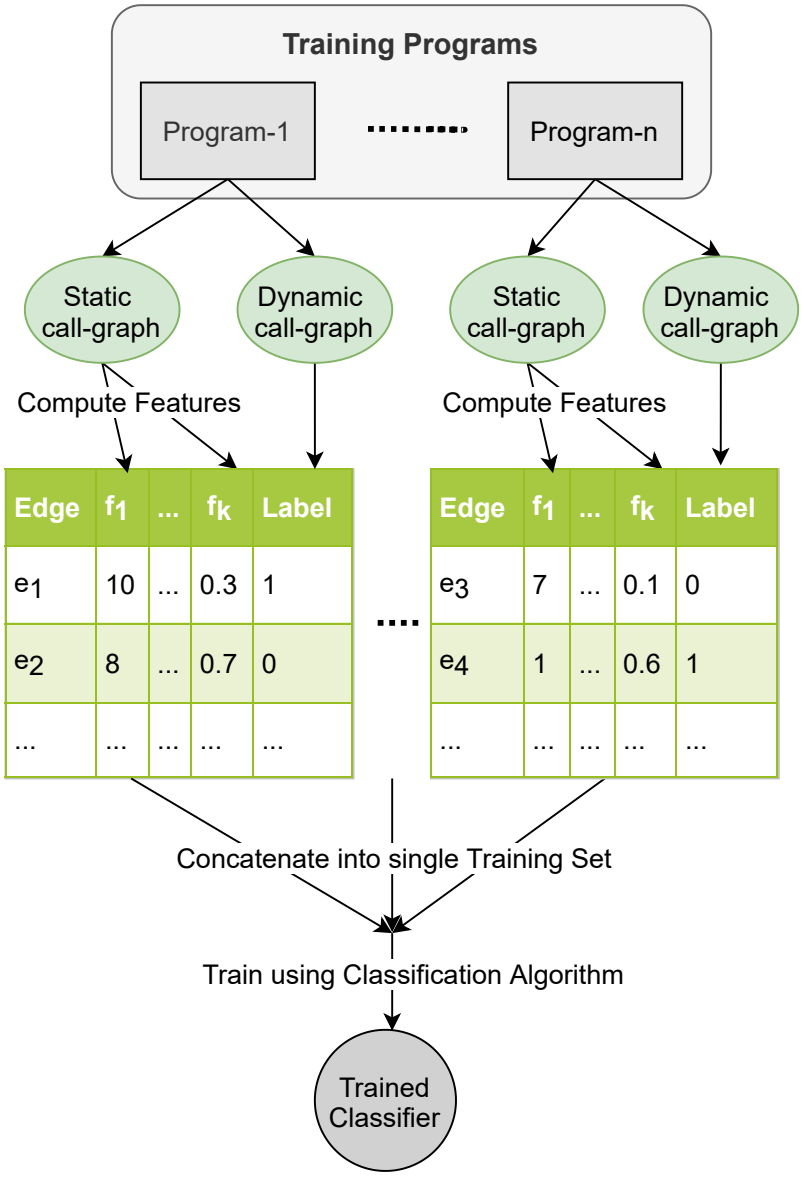


Figure 3.4: Classifier Generator workflow

3.3.2 Classifier Generator

We cast the edge-pruning problem as a classification problem for which learning a classifier can be done with machine learning. We proceed in three steps, as shown in Fig. 3.4.

In the first step, we run existing static and dynamic call-graph constructor tools on every program in the training set (the dataset of programs is described in Section 3.4). The result is a set of pairs of call graphs: each pair consists of a static call graph and a dynamic call graph. We use the dynamic call graph as an approximation of the ground truth: if a static call-graph edge is also present in the dynamic call graph, we view it as a true-positive, and otherwise as a false-positive.

In the second step, for each program, we construct a table in which each row represents a static-call-graph edge. Fig. 3.4 illustrates this table. The last column in each row (titled *Label* in Figure 3.4) contains a label of 1 or 0, based on whether the edge exists in the dynamic call graph. The remaining columns (titled f_1 to f_k) represent the set of *features* of the static call-graph edge. The example in Figure 3.2 uses two features: *dest-node-in-deg* and *src-node-out-deg*; we will discuss other features below. We can view each row in the table as a vector of feature-values. Concatenating the tables of each individual program gives us a single large training dataset of call-graph edges with ground truth labels. This training dataset consists of a large number of pairs (x_e, y_e), where x_e is a vector of feature values corresponding to a static call-graph edge, and y_e is a prediction of whether it is a false-positive or not. Our problem is now expressed in a format where it can be cast as a machine-learning classification problem [Kot07].

In the third step we run an off-the-shelf machine-learning tool on the table constructed in second step. The result is a classifier that for any edge assigns a probability that it is a true-positive. We picked *random forests* [Ho95] (ensembles of Decision Trees). One might try other approaches, which we leave to future work. Our goal with this step is to show that an off-the-shelf machine-learning tool is sufficient to get good results.

For readers unfamiliar with machine-learning classification, let is a brief overview of the binary classification problem. Given a large number of examples, $R = (x_i, f(x_i))|i \in 1..n$, where x_i is a data point (typically a vector of real values), and $f(x_i)$ is a binary class label for the data-point x_i, we need to learn a function f' that approximates f as closely as possible. This learned function f' is called the classifier. The difficulty in learning a good f' is that we cannot observe the function f we are approximating; we only see the training examples from R. Different classification algorithms use different assumptions on the space of functions from which they will choose f'. For example, a decision tree classifier picks the best f' from the space of all decision trees. Most classifiers, in addition to providing a predicted label $f'(x)$, also provide a likelihood score for $f'(x)$ being a 0 or 1. Finally, once the classifier f' is learned, it can be applied a new unseen datapoint y to predict $f'(y)$ and its likelihood score. See [Kot07] for a more detailed discussion on classification.

Getting back to our classifier generator: our classifier generator can take any static call-graph constructor as input. For example, we have used the call-graph constructors WALA [WAL15], Doop [BS09], and Petablox [MZN15b] as inputs and generated a call-graph pruner for each one.

The complexity of generating a classifier based on a training set with n edges is $O(n \log n)$ [Ho95].

3.3.3 Feature set

Now we describe how we designed the feature set that both our classifier generator and our generated call-graph pruners use.

A *feature* is information about a static-call-graph edge that may help predict whether the edge is a true-positive. We would like our feature set to capture important context and semantic information about a call-graph edge. Encoding important semantic information as features is a common machine learning practice for incorporating domain knowledge into the

learning process. For example, since dynamic dispatch is likely to affect the false-positive probability of a call-graph edge, we should add features that capture information about the targets of a method call. Using the context information of a graph edge has been useful for the related task of selective context and heap sensitivity in pointer-analysis [JLO20], and we consider it a good criteria for picking features. Context information can be local by describing the neighborhood of the edge, or global by describing the call graph that the edge is a part of. In addition to capturing context and semantic features, we identify three criteria that we want our feature set to satisfy:

1. linear-time computation complexity,
2. interpretable and generalizable, and
3. black-box.

The time-complexity guideline is particularly important given that some of our benchmarks can have several hundred thousand call-graph edges. Interpretability gives us an understanding of which call-graph edges are being dropped, and generalizability ensures that what is learned for the training edges also applies to call-graph edges of unseen programs. The black-box criterion implies that the features should only be designed on the output call graph, and not on some internal state or representation of a tool. This allows us to post-process the results without being specific to a particular algorithm or tool. Using these criteria, we arrived at the following features for an edge.

Figure 3.5 presents our feature set for an edge in a static call graph G, where the edge is from a caller method *caller* to a callee method *callee*. The node for the main method in G is *main*. The first seven features describe local information while the last four describe global information. Note that the L-fanout of an edge is the number of outgoing edges at the call-site of that particular edge, whereas src-node-out-deg is the number of outgoing edges from all the call-sites of an entire source method.

Feature	Description
src-node-in-deg	number of edges ending in *caller*
src-node-out-deg	number of edges out of *caller*
dest-node-in-deg	number of edges ending in *callee*
dest-node-out-deg	number of edges out of *callee*
depth	length of shortest path from *main* to *caller*
repeated-edges	number of edges from *caller* to *callee*
L-fanout	number of edges from the same call-site
node-count	number of nodes in G
edge-count	number of edges in G
avg-degree	average src-node-out-deg in G
avg-L-fanout	average L-fanout value in G

Figure 3.5: Our feature set

Our selection process started with a much longer list of features that all satisfy the three criteria listed above. We picked from that list the ones that helped the most with removing false-positives. Our process used the training set as case studies to find the main reasons why tools give false positives. The result was the eleven features in Figure 3.5.

3.4 Implementation and Dataset

Static Call-Graph Constructors We used the static call-graph constructors WALA [WAL15], Doop [BS09], and Petablox [MZN15b]. In each case we used the default setting, which implements 0-CFA for methods that are estimated to be reachable from the main method and without any special handling of reflection. Those tools produce significantly different call graphs and so we generate a separate call-graph pruner for each tool.

Reflection In preliminary experiments, we found that enabling special handling of reflection in the static call-graph constructors introduces many false-positive edges in the call graphs. Our generated classifiers tend to assign each of those edges a low probability of being a true-positive, and therefore our call-graph pruners will correctly remove most of them. Therefore, special handling of reflection presents no additional challenge for call-graph prun-

ing and we decided to go with the default setting of each static call-graph constructor.

Dynamic Call-Graph Constructor We used the open-source tool Wiretap [KP18] to instrument the Java bytecode and thereby enable dynamic call-graph construction. Next, we ran the instrumented bytecode and collected data about the run, particularly about the method calls.

Standard Library The Java standard library is large and has the potential to dominate the measurements for every benchmark,
which is counterproductive. So, when we do our measurements and training, we omit nodes from the standard library as well as edges between standard library nodes. We preserve aspects of the edges to and from the standard library in the following way. For every path of the form

$$v \to \langle \dots \text{ standard library nodes } \dots \rangle \to w$$

where v, w are nodes outside the standard library, we create a single edge from v to w.

Random Forest Classifier Our classifier generator uses the Random Forest algorithm [Ho95] implemented with the Scikit-Learn [PVG11] library (v0.21.3). The Random Forest algorithm works as follows:cgpruner- it trains several decision-trees using Bagging [Bre96], and makes predictions by a "majority vote" across the decision trees. The training took 4 minutes. We tuned the hyper-parameters using Random Hyper-Parameter Search [BB12] with 4-fold cross-validation on the training set. The chosen hyper-parameters are listed in Fig. 3.6.

Dataset Our dataset consists of 141 programs from the NJR-1 benchmark suite [PL18], of which we used 100 programs for generating three call-graph pruners and the remaining 41 programs for our evaluation. We selected those 141 programs from the 293 NJR-1 programs according to the following criteria:

Hyperparameter	Value
Number of Trees	1000
Maximum Depth	10
Bootstrapping	False
Minimum samples for split	2
Maximum features for split	sqrt(feature count)
Minimum samples for leaf	1
Split quality criterion	Entropy
Other hyper-parameters	Library default

Figure 3.6: Hyper-parameters for Random-Forests

- consists at least 1,000 methods and at least 2,000 static call-graph edges according to Wala,
- executes at least 100 distinct methods at runtime, and
- has high coverage: executes a large percentage of the methods that are reachable from the main method according to Wala; for our benchmarks, the coverage is 68%, on average.

Each program consists of 560,000 lines of code, on average (not counting the standard library). In more detail, each program consists of the main application, which is 8,000 lines of code, on average, in addition to third-party libraries which account for an estimated 552,000 lines of code, on average.

The total number of static-call-graph edges (not counting the standard library) that are reachable from the main methods of the 141 programs is 1.3 million. For our classifier generator, each edge from 100 of those programs is a data point, which is 860,000 edges. Note that manual creation of ground truth about those 860,000 edges infeasible.

Large Benchmarks The histogram in Figure 3.7 gives the distribution of the edge counts in the training programs. The X-axis is plotted on a logarithmic scale due to the skew in the distribution. Among the 100 training programs, 7 of them have a very large number of

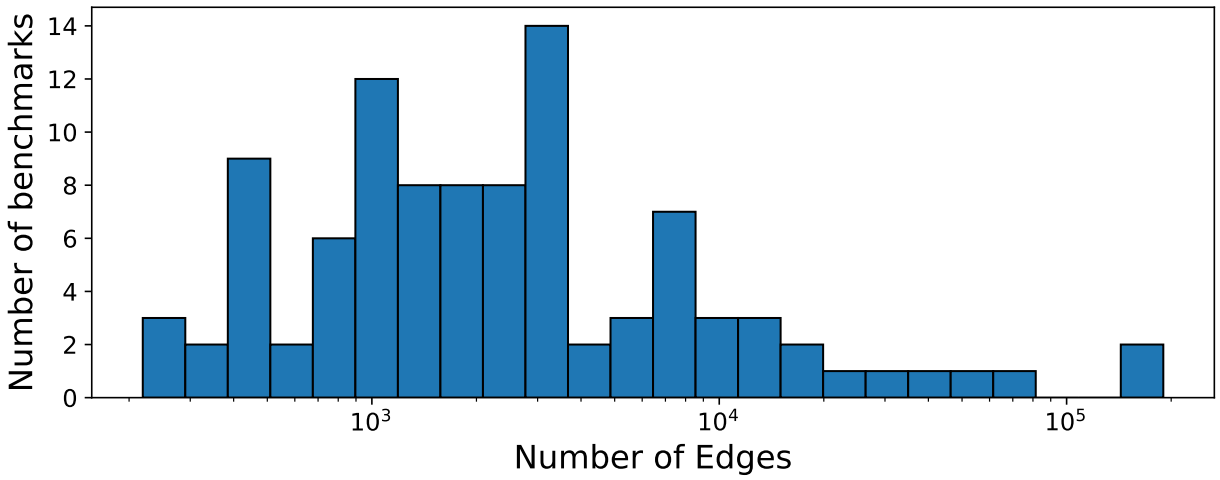


Figure 3.7: Histogram of Edge-counts in the 100 Training Programs.

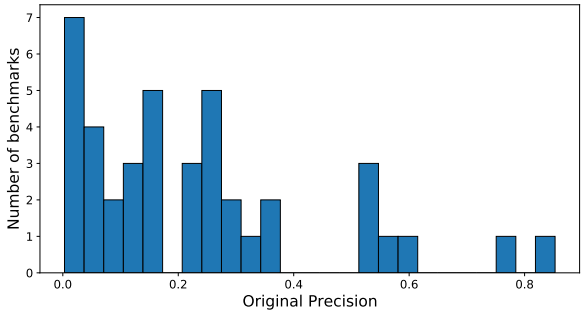
call-graph edges ($>$ 20,000). This gives them the potential to dominate how the classifiers work. To overcome this, we randomly sample 20,000 edges from the edge-sets of these 7 programs. Notice that this sampling is done only during generation of call-graph pruners, while we use all the edges from the 41 programs that we use for evaluation.

Analysis Time Running the three static call-graph constructors and the dynamic call-graph constructor on all the programs takes four days of compute time.

Precision and Recall We estimate the quality of a static call graph using the standard notions of *precision* and *recall*. In our setting, if S is the edge set produced by a static call-graph constructor, and W is the edge set produced by Wiretap, then:

$$Precision = \frac{|S \cap W|}{|S|} \qquad Recall = \frac{|S \cap W|}{|W|}$$

The rate of false-positives is $(1 - Precision)$. We compute the average precision and recall values for the entire test-set by taking the arithmetic mean over the precision and recall values of individual programs.



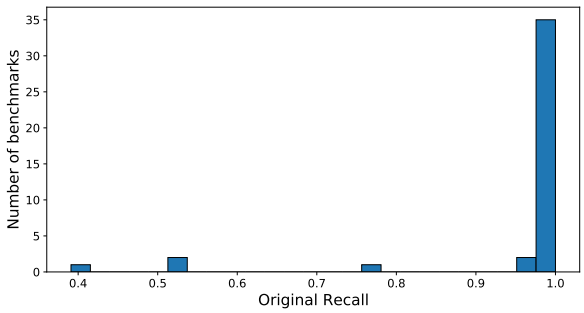
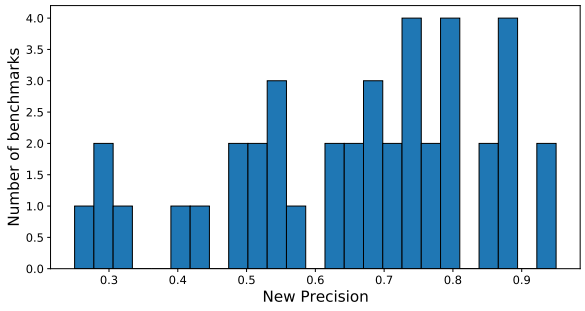


Figure 3.8: Precision and recall for 41 test programs.



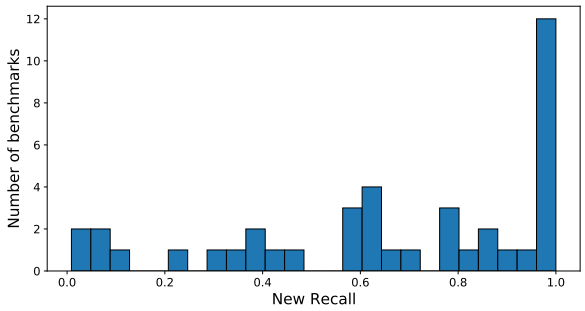


Figure 3.9: Precision and recall after call-graph pruning.

Figure 3.8 shows a histogram of the original precision and recall scores for WALA on the 41 individual programs of the test set. Note that the precision values vary significantly, but almost all programs get below 40% precision. Hence, there is a lot of scope for improving the precision. The recall is close to 100% for most programs, but low for some due to heavy use of reflection, dynamic class-loading or native code.

3.5 Experimental Results

In this section, we discuss our experimental results that validate the following claims.

1. Our generated call-graph pruners for WALA, Doop, and
   Petablox produce call graphs with balanced 66% precision and 66% recall.

2. For precision-sensitive clients, our generated call-graph pruners are significantly better at boosting precision than context-sensitive analyses, and have a much smaller

overhead.

3. The precision improvement is consistent across the test set.

4. The call-graph pruner enables a monomorphic call-site client to balance its skewed 52% precision and 93% recall to a more balanced 68% precision and 68% recall.

5. The call-graph pruner enables a null-pointer analysis to reduce its average warning count from 223 to 20, while increasing precision from 27% to 77%.

All experiments are run on a separate test set of 41 programs which were not used during training. The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 Gb RAM. A minimum RAM size of 32Gb is essential for ensuring that the static analyses run in reasonable time. The artifact for the chapter is available here [UKL21] and the NJR-1 dataset can be downloaded from [UKL20].

3.5.1 Main Result

Figure 3.10 gives the main result of the chapter: a call-graph pruner can be successfully used to boost precision and to balance the goals of precision and recall for the 0-CFA call-graph analysis of WALA, Doop and Petablox. The plot is used to represent the precision and recall values of various tools, wherein all precision and recall values are reported as averages over the test-set programs. The black triangle marks the WALA 0-CFA analysis (23.8% Precision, 95.3% Recall), the green triangle marks the Doop 0-CFA analysis (23.1% Precision, 92.6% Recall) and the blue triangle marks the Petablox 0-CFA analysis (29.8% Precision, 88.8% Recall). They all have close to perfect recall, but poor precision. The red plus sign marks the WALA 1-CFA analysis (29.6%. 95.4%). The black curve represents the precision-recall trade-off points obtained when a call-graph pruner is applied to the WALA 0-CFA output. The original WALA-0CFA output is a single point on the precision-recall graph, but the call-graph pruner gives a curve instead. This is because the call-graph pruner

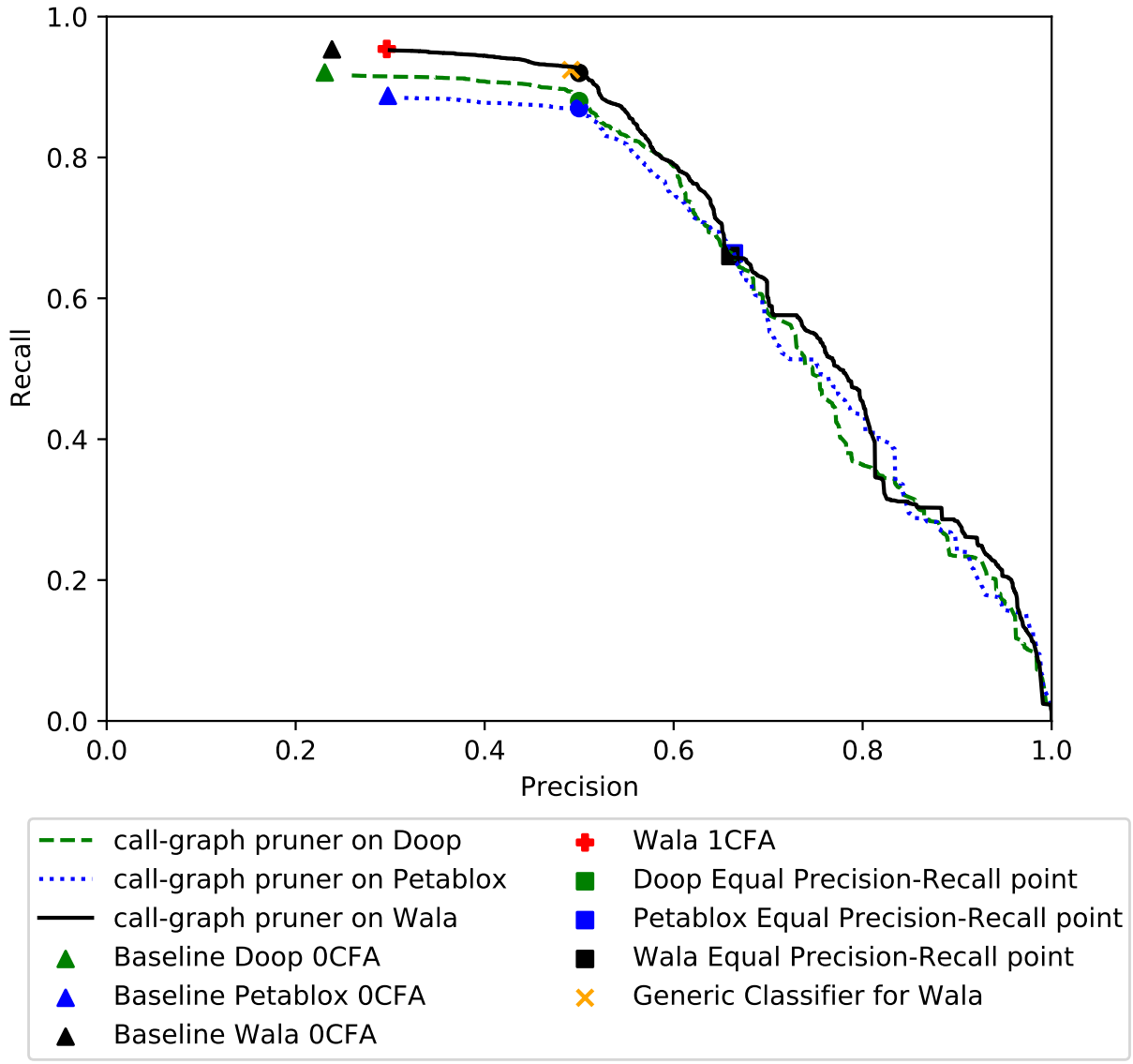


Figure 3.10: Main Result for the WALA, Doop and Petablox static analysis tools. The baseline precision-recall values for the 3 tools, along with the precision-recall curve obtained after applying a call-graph pruner (averaged over all test programs)

gives a probability score for each edge being in the ground-truth call-graph, and by setting different thresholds (i.e. cutoffs below which an edge is removed), we can obtain different points on the precision-recall curve. Joining all these different points gives us the black curve in the figure. Setting a low-probability threshold for accepting an edge, gives us points near the left end of the black curve, because we accept a large percentage of edges, thereby giving us higher recall but lower precision. Setting a high-probability threshold gives us points near the right end of the curve because we accept only very few edges which are very likely to be in the ground-truth call-graph, and this gives us high-precision and low recall. The green and blue curves represent the precision-recall trade-off obtained by applying the call-graph pruner to the Doop and Petablox call-graphs respectively, and the case is very similar to the black WALA curve.

These curves which trade-off recall for precision show that the classifier has assigned probabilities meaningfully. In contrast, a tool that randomly assigns probabilities to edges would result in a curve that goes straight down to zero recall without improving any precision. This is because it results in a random removal of edges, which keeps the ratio of true-positives (i.e. precision) the same. Boosting precision requires the ratio of false-positive edges in the removed edge set to be higher than the rest of the edges.

There are 2 particularly interesting points on the black (WALA) curve in Figure 3.10. The first is the one marked by the black (WALA) square (66.0% Precision, 66.0% Recall), which represents the point with balanced precision and recall. Such a point will be useful to a precision-sensitive client analysis. As compared to the original WALA 0-CFA (black-triangle), this point has over 72% of the edges from the original call-graph removed, and out of the removed edges, less than 10% are true positives. This point is at a 0.45 probability threshold. Similar points for Doop and Petablox, marked by a green square (hidden behind the black square) and blue square (also hidden behind the black square) respectively, are at (66.2% Precision, 66.2% Recall) and (66.4% Precision, 66.4% Recall) respectively. A second interesting point is the right-most point on the curve after which recall starts dropping faster,

represented by a black circle (50% Precision, 92% Recall). Such a point would be useful for a client analysis that needs to increase a little precision, without losing much recall. Similar points for Doop and Petablox are marked by the green circle (50% Precision, 88% Recall) and blue circle (50% Precision, 87% Recall) respectively.

Both these points give larger precision boosts than the 1-CFA analysis. However, in general, the best precision-recall trade-off point is decided by the needs of the client of the call graph. Precision-sensitive clients would benefit more from our call-graph pruner since it gives a larger precision boost, but clients that need high recall may prefer the 1-CFA call graph.

Our call-graph pruner adds an overhead of 18% to the WALA 0-CFA analysis, whereas moving to a 1-CFA analysis adds 292% overhead. Prior research also finds that context-sensitivity increases analysis time by many folds [LTM18b].

For completeness, we also ran this experiment for WALA's RTA implementation and it gets similar results. Since the three tools show similar characteristics, we only present numbers for the WALA 0-CFA call graph in the rest of this section.

Picking a Cutoff value We picked the balanced precision-recall point because it gave good results for a null-pointer analysis client, but different precision-recall trade-off points may be suitable for different client analyses. Figure 3.11 helps a user pick the right trade-off point for their client. It plots the probability cutoff values on the X-axis, and the Precision, Recall and F-score on the Y-axis. The graph shows what values each of these metrics takes at every probability cutoff value, as well as what the expected cutoff would be for a given target Precision, Recall or F-score. For example, by looking at the figure, we can say that to obtain an expected Precision of 60%, we can set a cutoff value of 0.4. At this point we would get a Recall of approximately 75% and F-score of around 65%. This graph also shows that the balanced precision-recall point is also very close to the point with maximum F-score.

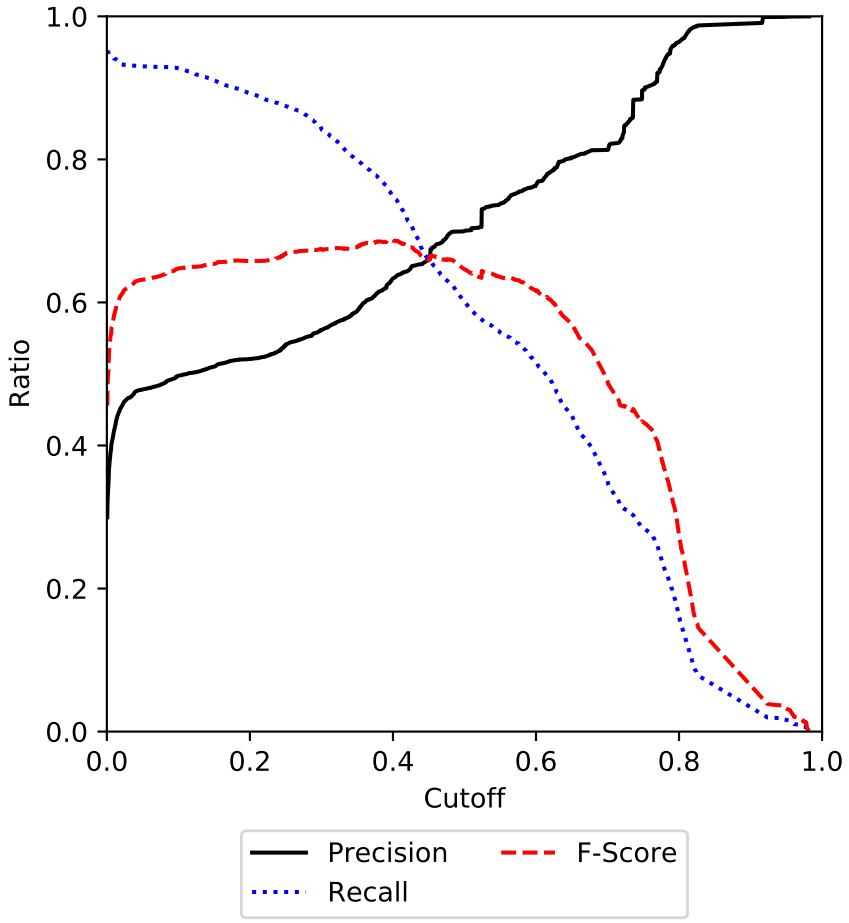


Figure 3.11: Probability cutoff plotted vs Precision, Recall and F-score curves for WALA

Feature Importance Figure 3.12 gives the impurity-based importance [Sci] for each feature used in the random-forest in descending order. The *L-fanout* and *dest-node-in-deg* are the most important features and the four global features are the least important. Dropping the four global features decreases the area under the precision-recall curve from Figure 3.10 by 6%.

Human-Interpretable Explanation of the Classifiers. We can give a human-interpretable explanation of the main aspects of the Random Forest classifiers that were learned in the

Feature	Importance
L-fanout	0.182
dest-node-in-deg	0.114
src-node-out-deg	0.094
repeated-edges	0.092
src-node-out-deg	0.090
depth	0.084
dest-node-out-deg	0.079
node-count	0.071
edge-count	0.067
avg-L-fanout	0.036
avg-degree	0.028

Figure 3.12: Importance of each feature in the Random Forest Classifier in descending order.

experiment. In each case, the top-level decisions center around the following generic classifier:

$$\text{if } ((\text{L-fanout} > m) \wedge (\text{dest-node-in-deg} > n)) \text{ then 0 else 1}$$

The above expression says that if an edge has *L-fanout* greater than m and *destination-node in-degree* greater than n, then the probability that it is a true edge is 0, and otherwise 1. For each of the static call-graph constructors, we can identify the constants m and n:

WALA:

$$\text{if } ((\text{L-fanout} > 3.5) \wedge (\text{dest-node-in-deg} > 9.5)) \text{ then 0 else 1}$$

Doop:

$$\text{if } ((\text{L-fanout} > 3.5) \wedge (\text{dest-node-in-deg} > 16.5)) \text{ then 0 else 1}$$

Petablox:

$$\text{if } ((\text{L-fanout} > 3.5) \wedge (\text{dest-node-in-deg} > 20.5)) \text{ then 0 else 1}$$

The orange cross (49% precision, 92% recall) in Figure 3.10 gives the precision-recall trade-off when using the generic classifier for WALA. This generic classifier has a slightly worse trade-off and is much less tunable than the black line (WALA with call-graph pruner).

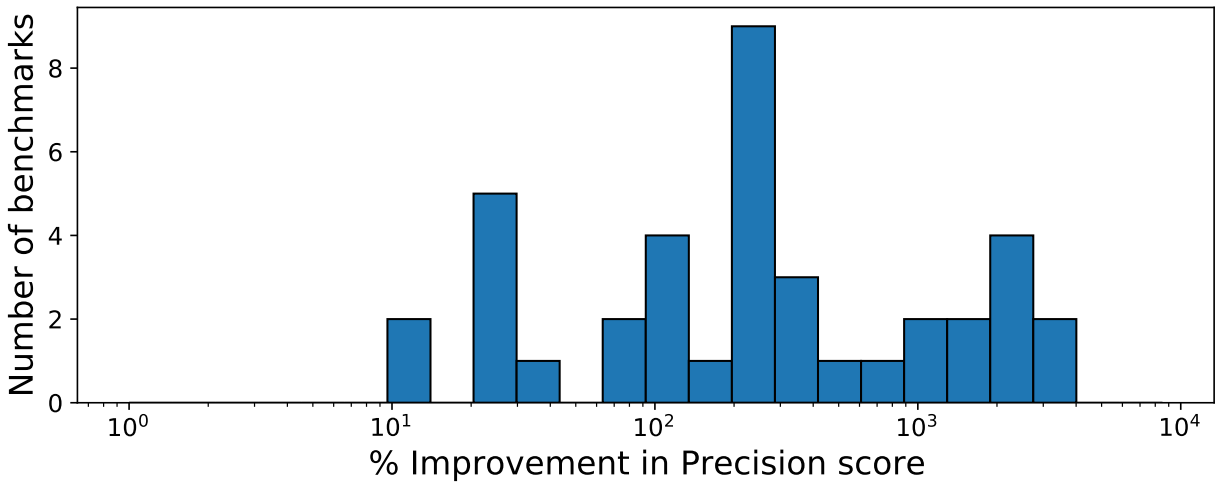


Figure 3.13: Historgram of Percentage Improvement in Precision scores for individual programs.

However, its pruning rules are also much simpler and easily understandable. The use of *L-fanout* and *dest-node-in-deg* in the generic classifier aligns with the fact that these are the most important features according to Figure 3.12.

3.5.2 Distribution of Precision and Recall for individual programs

Figure 3.9 gives a histogram of the precision and recall scores of individual programs when a call-graph pruner is used to prune the WALA call graph at the balanced precision-recall point (marked by the black square in Figure 3.10). Most of the programs get at least 50% precision, and a several even reach the 70% precision goal. Contrast this to the precision in Figure 3.8 where almost all programs fail to cross the 40% precision point.

As expected, the recall scores from Figure 3.9 dropped as compared to Figure 3.8. However, most programs still get at least 50% recall, implying that they retain a good portion of their true edges. Note that it is impossible to improve recall using a call-graph pruner since it cannot find new edges that WALA did not find.

The histogram from Figure 3.13 illustrates the percentage improvement in precision

Call-graph tool	Precision	Recall
WALA 0-CFA	51.8%	92.6%
WALA 0-CFA + call-graph pruner	67.7%	68.4%

Figure 3.14: Impact of improved call-graph precision on a monomorphic call-sites client

scores. The X-axis is plotted on a logarithmic scale. By using a call-graph pruner, 30 out of the 41 programs have their precision score boosted by at least 2 times their original precision score. All but 2 programs have their precision score boosted by at least 20%. No benchmark gets a worse precision. Thus, a significant majority of the individual programs consistently get a large precision improvement without loosing too much recall, and achieve a better precision-recall balance.

3.5.3 Effect on Client Analyses

Next, we look at the effect of improved call-graph precision on the monomorphic call-site detection and null-pointer analysis clients.

Monomorphic call-site client. This client is based on the WALA-generated 0-CFA call graph, and it uses the dynamic analysis as the ground-truth. Figure 3.14 give the precision and recall of a monomorphic call-site client with and without the call-graph pruner. The call-graph pruner helps the client boost precision from 52% to 68% and balance its goals of precision and recall.

Applications of the monomorphic call-sites client include devirtualization and inlining. Since the call-graph analysis is never sound in practice [LSS15a], these applications require some safety checks, resulting in overheads. For example, if devirtualization is used for optimization, run-time checks need to be inserted to ensure correctness [IKY00]. Higher precision for the monomorphic call-sites client implies that more of the call-sites declared monomorphic by the static analysis actually turn out monomorphic in the ground-truth.

ID	Warnings		True-Positives in a sample of 10	
	Before	After	Before	After
B1	137	12	2	10
B2	365	31	4	5
B3	190	15	2	8
B4	308	44	7	10
B5	204	16	0	10
B6	429	42	0	7
B7	404	136	7	10
B8	70	10	0	0
B9	231	10	0	9
B10	102	34	5	8
Average			2.7	7.7

Figure 3.15: Total warning counts and a manual classification of a sample of 10 warnings for the null-pointer analysis before and after applying a call-graph pruner

This in turn implies that whenever we incur the overhead of inlining or devirtualization, we are also more likely to realize its benefits.

Null pointer analysis. This analysis is based on the paper by Hubert et al. [HJP08]. It is implemented in WALA, and is used to find null-pointer errors originating from uninitialized instance fields. The analysis is context-insensitive, field-insensitive and flow-sensitive. It only reports potential null-pointer dereferences in application code, and not for the standard library.

The original WALA call graph gives us, on average, 223 null pointer warnings per program. The high volume of warnings makes it cumbersome for developers to manually inspect and in practice this results in developers ignoring the tool output entirely [JSM13c, BBC10a]. Using the call graphs produced after pruning gives us much fewer (on average 20 per program) warnings.

Two of the authors manually inspected a random sample of 10 null-pointer warnings from 10 of the 41 test programs when used with and without the call-graph pruner. The 10

programs were chosen with the criteria that they had at least 10 warnings both with and without the call-graph pruner, and the ratio of warnings with and without the call-graph pruner was close to (20/223). Figure 3.15 gives the total warning counts as well as the true-positive counts (from a sample of 10 warnings) for each of these 10 programs. The use of a call-graph pruner helped the null-pointer analysis improve its precision from 27% to 77%

The criteria for marking a warning as a true-positive was that the author could trace the backward slice of a dereference to an instance field which was uninitialized by the end of a constructor. Warnings that either could not be verified in 10 minutes, ran into another exception before triggering the null exception, or otherwise unverifiable by the authors, were considered as false-positives. Reachability from the main method was not considered because it is hard to verify manually.

We leave to future work to try other clients, including other approaches to null-pointer analysis such as NullAway [BCS19].

3.5.4 Threats to Validity

The first threat is the use of a dynamic analysis as a proxy for the call-graph ground truth. It assumes good coverage of the true ground-truth call-graph and affects the precision-recall calculations. If the dynamic analysis had higher coverage, more of the static analysis edges would be in the dynamic call-graph. As a consequence, both the baseline precision scores as well as the pruned-call-graph precision scores would be higher. In contrast, we expect the recall scores to remain similar. However, improving dynamic analysis coverage is a non-trivial and orthogonal problem and any techniques improving coverage will automatically improve our technique and evaluation. Symbolic execution [KPV03] is one option to improve coverage, but it doesn't scale to the size of our programs. Instead, we use a subset of the NJR-1 benchmark set which gets good coverage. Note that this threat does not affect the evaluation of the null-pointer analysis.

The second threat is the manual inspection of the null-pointer warnings, which are vulnerable to human errors. The authors inspecting the errors have a limited familiarity with the code-bases of the examined program. This could lead to misclassification of both true and false errors, and affect the precision score accordingly. Further, the precision scores are reported for a sample of 10 programs.

The third threat to validity is the generalizability of the results to programs outside the NJR dataset. Our assumption is that our learning and evaluation results generalize to other programs outside the dataset.

The fourth threat to validity is that programs in the training set and evaluation set share some third-party libraries. On average (geometric mean), 3.6 percent of the methods of a program in the evaluation set also occur in some training program. We believe that this overlap is low enough to not significantly affect the conclusions of our evaluation.

3.6 Related Work

Our technique is the first to apply machine learning to boost call-graph precision. In our discussion of related work, we focus on three areas: combining static and dynamic analyses, applying machine learning to remove static-analysis false-positives, and improving the precision of call-graph construction.

Combining static and dynamic analysis Prior research has used a dynamic analysis to improve the precision of a static analysis. Grech et. al [GFF18b] generate dynamic heap information and use this as a drop-in replacement for the heap modeling part in an existing static analysis tool to improve its precision. Artzi et. al [AKG07] use a dynamic analysis to confirm the mutability information computed by a static analysis. Chen et. al [CHR21] use the information from test-executions to prioritize the alarms given by a static analysis. The main drawback that these tools face is that they need the dynamic analysis to be run every

single time the tool is run. In contrast, our technique needs the dynamic analysis only for generating a call-graph pruner. After that, a call-graph pruner is purely a static tool, and hence does not suffer from the usual drawbacks of a dynamic analysis like long execution times or finding good inputs.

Applying machine learning to improve static-analysis by removing false-positives The technique of filtering static-analysis false-positives by casting it to a classification problem with hand-picked features has been used for static bug-analysis tools [RPM08, HW09, TGP14, YS13, FSS18]. Each of these works follows the same workflow: collect static analysis error-reports, get a programmer to label them as true or false-positives, design a feature-set for the error reports, and then train a classifier on these labeled error-reports to identify false-positives. However, they have minor differences among themselves in terms of the feature-set chosen, the bug-reporting tool used and the benchmarks used for the training data. Ruthruff et. al [RPM08] use the FindBugs [HP04] bug-reporting tool and the set of Java programs at Google as their dataset. Heckman and Williams [HW09] also use FindBugs reported bugs on 2 open-source Java projects. Yuksel and Sozer [YS13] classify bug-alerts for a digital TV software. Flynn et al. [FSS18] combine the bug-alerts from multiple tools, in addition to using the hand-picked features. Tripp et. al [TGP14] work with a JavaScript security checker's warnings from popular Web sites as its dataset.

Our work differs in three ways: it uses an estimate of ground-truth produced by dynamic analysis, it has a generalizable approach to picking a feature set, and it has a tunable precision-recall trade-off, as we discuss next.

The key bottleneck faced by each of these prior works was that they relied on the collection of human-labeled ground-truth, which does not scale. This restricted their dataset to a handful of projects and a couple of thousand data-points (bug reports) at best. In fact, for each type of error, there is typically less than a few hundred bugs in each of the datasets. In contrast, our technique uses an estimate of ground-truth produced by dynamic analysis,

which allows it to scale to a much larger number of programs with a million data points (call-graph edges).

The second major difference is in the choice of the feature-set. This is partly a consequence of the fact that the previous work focuses on static-analysis error report data, which is different from the graph output generated by call-graph construction tools. Hence some of the common features used in these works are the bug-priority level, file-modification-frequency, coding-style metrics, and lexical features (like method or package names). These features, though appropriate, violate generalizability and black-box guiding principles listed in Section 3.3.3. Non-black-box features like bug-priority level will not generalize across different tools or algorithms, and non-generalizable features like lexical features are unlikely to generalize to programs outside the dataset. In contrast, we use a systematic approach to selecting features, as described in Section 3.3.3, and as a consequence, our approach generalizes easily across multiple programs and multiple call-graph construction tools.

The third difference is that these prior works, except for [TGP14], provide a single precision-recall point. [TGP14] provide eight different precision-recall points, by varying the classifier used. Instead, our approach has a tunable precision-recall trade-off by predicting edge-probabilities and pruning edges with probability lower than a threshold. Further, we only use a single classifier (Random Forests) since it achieves superior precision-recall trade-offs than the classifiers used in [TGP14].

Another area that uses machine learning for filtering false positive is the work by Raghothaman et al. [RKH18a]. They predict the probabilities of static-analysis alarms using Bayesian inference and update these as the user resolves alarms as true or false positives. This paradigm of online learning, where the model is learned and improved as the user gives feedback, is quite different from our fully-automated offline learning paradigm, where we do a one-time training on a large dataset of static and dynamic analysis outputs and require no user input.

Recently data-driven techniques have also been used to selectively apply context- and flow-sensitivity [JJC17, CJO18] to methods that will benefit it the most. These techniques

can potentially provide the precision improvement of a 1-CFA at a lower overhead, but as seen in Figure 3.10, this improvement is still much lower than what is achieved by our call-graph pruner.

Improving the precision of call-graph construction Lhotak [Lho07] designed an interactive tool to qualitatively understand the root cause of differences between different static and dynamic analysis tools. This is then used in a case study to understand the main cause of imprecision in a static analysis tool as compared to its corresponding dynamic analysis output. In contrast, our classifier generator is fully automated, using machine learning, and doesn't require a skilled programmer to use an interactive tool to figure out the cause of the imprecision.

Sawin and Rountev [SR11] propose certain heuristics to deal with dynamic features like reflection, dynamic class loading and native method-calls in Java, which helps to improve call-graph precision of the CHA algorithm without sacrificing much recall. Similarly, a call-graph pruner trades of a little recall for a large boost in precision, but it achieves this through automated machine learning on a dataset of call graphs instead, and is able to boost precision by a much larger amount. Additionally, we work with a 0-CFA baseline (with no handling of dynamic features like reflection), which already has a large precision gain over a CHA algorithm with reflection handling.

Zhang and Ryder [ZR07a] create precise application-only call graphs by identifying which edges from the standard library to the application are really false-positive. This is similar to the precision boost we gain for the edges that go via the standard library. However, we generate a classifier that learns this on its own from data, and we use the classifier in a call-graph pruner that is able to boost precision even further.

The patent by Reif et. al [LWR20] uses probabilities to quantify analysis imprecision. Each analysis constraint is assigned a probability heuristically or via user configuration, and the probabilities for call-graph edges are derived from these using a type-propagation graph.

In contrast, our call-graph pruner learns all its edge probabilities from data about static and dynamic call-graphs. Further, while their technique calls for a new static analysis, our call-graph pruner works as a black-box post-processor for existing call-graph construction tools.

More distantly related is the work by Blackshear et. al [BCS15], which prunes control-flow edges representing interleavings between events in an event-driven system. This pruning task is different from our task which focuses on pruning call-graphs edges for sequential code.

There has also been prior work that uses a dynamic analysis to evaluate call-graph related static analysis tools [SDT20, AL12, RKG04, BW09, GFF18b]. Our tool additionally uses the dynamic analysis results as training labels to prune the result from a static call-graph construction tool.

CHAPTER 4

QueryMax: Application Code Analysis using Partial Libraries

In this chapter, we introduce the second tool, *QueryMax* [UP22], which tackles the second biggest pain point for users: long analysis times for existing static analysis tools. *QueryMax* achieves a speedup by selecting a partial library (instead of the whole library) to analyze with the application code. It catches, relative to a whole-program analysis, 87% of its errors, with a 0% false-positive rate and a geometric mean speedup of 10x.

4.1 Overview

Motivation. Long analysis times are a key bottleneck for the widespread adoption of whole-program static analysis tools. Several recent papers for both Java [LTM18a, BKL20a, GFF18a] and C/C++ [SWF20, FWS19, SZ20] report that a whole-program analysis on their largest benchmarks can take several hours. Analyzing a large collection of benchmarks like an app-store takes even longer, with a total compute time of many years for the largest app-stores. Hence, a speedup in analysis time can save significant compute time and energy, and enable us to use more precise and expensive algorithms.

Whole-program analyses may be slow, but a user is often only interested in finding errors in the application code [ZR07b], which constitutes a small fraction of the whole program. In the NJR-1 dataset [UKL20], application code (excluding third-party libraries) constitutes less than 1% of the whole program on average. Hence, an application-focused analysis has

the potential for a large speedup.

Ideally, an application-focused analysis should compute the same set of errors for the application-code as a whole-program analysis. However, this is hard to achieve because errors can both originate in or propagate through the library. We use the singular *library* to refer to the aggregate of the third-party libraries and the standard library. The quality of an application-focused analysis tool's results can be quantified using *precision* and *recall*. *Precision* is the ratio of true-positives in the tool's results, with the whole-program analysis results serving as the ground-truth. *Recall* is the ratio of whole-program analysis errors caught by the tool. Thus, any application-focused analysis tool can be judged by its performance on the three metrics of precision, recall and speedup.

The current best tool for an application-focused analysis is Averroes [AL13]. Averroes overapproximates the effect of the library with a compact summary. The overapproximation ensures high recall and the small size of the summary compared to the whole library gives a large speedup. However, this summary is created by merging the analysis information from all the library pointers into a single set, resulting in significantly worse precision than the whole program analysis. In our experiments, Averroes gets an average precision of 59% relative to the whole-program analysis. This precision drop is problematic because empirical studies show that users have a very high bar for precision.

For example, Christakis and Bird [CB16b] find that, in practice, static analysis users care much more about precision than recall. They conclude that practical analysis tools must aim for a minimum of 80% *user-perceived precision*. Failing to meet this value results in users ignoring the tool output entirely. Other empirical studies [BBC10b, JSM13a] also arrive at similar conclusions. Whole-program analyses themselves often get much less than 80% *user-perceived precision* [BKL20a, BCS13, RKH18b]. Hence, an application-focused analysis that gets less than 100% precision relative to a whole-program analysis will almost certainly fail to meet the 80% *user-perceived precision* target. This defines the goal for this chapter.

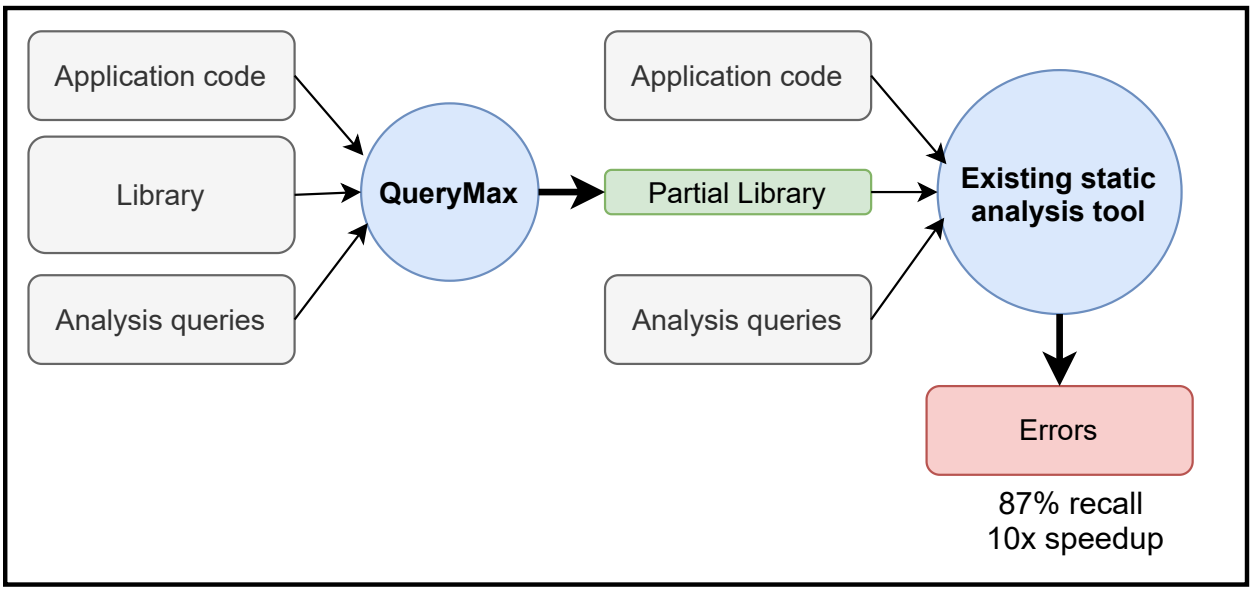


Figure 4.1: Overview of the QueryMax workflow

Our goal is to capture the speedup potential of an application-focused analysis, while maintaining 100% precision relative to the whole-program analysis.

Our technique. In this chapter, we introduce a new application-focused analysis tool called *QueryMax*, that achieves our goal of 100% precision and gets both good speedup and good recall. Figure 4.1 gives an overview of the workflow. *QueryMax* acts as a pre-processor to an existing static analysis by selecting a small subset of the library (i.e. partial library) which is relevant to the set of analysis queries in the application. To decide which part of the library is most relevant, *QueryMax* uses a new static analysis called the *external source analysis*. Once *QueryMax* picks the partial library, the existing static analysis tool is run on the application code plus the partial library, with all external library pointers treated as the bottom element in the abstract domain.

The analysis queries used in Figure 4.1 are exactly like the queries in a demand-driven analysis [SGS05a] and they represent all the instructions of interest in the application code.

For example, in a cast-check analysis, the analysis queries would be all the down-cast instructions in the application code.

The complexity of *QueryMax* is $O(a^3 + p^2)$ where a is the size of the application-code and p is the size of the (application-code + partial-library). This is much less than the complexity of a whole-program analysis like 0CFA, which has complexity $O(n^3)$ where n is the size of the whole program. Here we assume $(n > p)$ and $(n >> a)$, both of which are true for our benchmarks.

Our experiments focus on Java bytecode programs from the NJR-1 dataset [UKL20], but our approach applies to other object-oriented languages as well. We implemented *QueryMax* in Wala [WAL15] and ran experiments on it with an existing cast-check analysis and null pointer analysis.

Our contributions.

- We introduce a new static analysis, the *external source analysis*, which computes the set of external library pointers affecting each pointer in the application code.
- We describe the *QueryMax* tool which uses the external source analysis and picks a partial library which is small yet sufficient to yield a good recall.
- We show experimentally that *QueryMax* successfully speeds up two different analyses. In a particular configuration, *QueryMax* achieves a 97% recall (on average, relative to a whole-program) and an 8.7x geometric-mean speedup for a cast-check analysis, and a (79% recall, 11.2x speedup) for a null pointer analysis. Both analyses get 100% precision.

Significance. The impact of this research contribution is that the 10x analysis speedup without any loss in precision will help us meet user expectations on both speedup and precision. Further, the speedup will enable us to use expensive and precise analysis algorithms

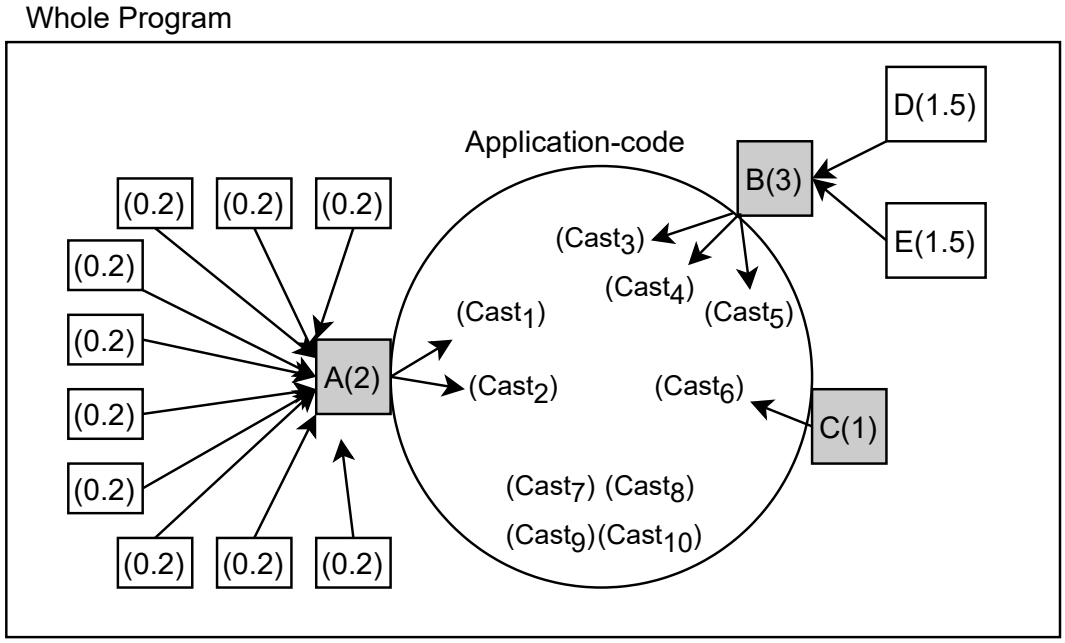


Figure 4.2: Schematic of a cast-check analysis on application-code

as well as analyze large programs or large collections of programs (like an app-store) that previously couldn't be analyzed in a reasonable amount of time.

4.2 Example

In this section, we show an example of how *QueryMax* picks a partial library to analyze, and compare this with Averroes' approach. We also discuss two other baselines which can be adapted to provide a speedup over a whole-program analysis:querymax- a demand-driven analysis [SDA16, SGS05a] and an application-only analysis.

Figure 4.2 shows the schematic of a program we wish to analyze for cast-errors. The application code, represented by the circle, is the part in which we wish to catch the cast errors, and everything outside is the library. The grey boxes (labeled A, B, C) on the edge of the circle show library methods with pointers that influence the value of cast instructions in the application code. The accompanying number in the grey box tells us how many

cast instructions are affected by that method. The application code has a total of 10 cast instructions and each cast instruction is considered an analysis query. We say that an application-focused analysis *covers* a cast-query if it overapproximates the result of that query. In other words, a query *covered* by a tool is guaranteed to mark it as a cast-error if the whole-program analysis does.

The first baseline technique is to run a demand-driven analysis for every analysis query in the application. The demand-driven analysis exhaustively traces the backward slice of all 10 cast instructions. Casts numbered 7-10 at the bottom of the application circle get their value from inside the application, and hence are answered quickly. The casts affected by B and C (casts numbered 3-6) are also answered quickly because the backward slices have only 2 and 0 caller-methods respectively. However, the demand-driven analysis faces a significant slowdown when answering the two cast queries influenced by A ($Cast_1$ and $Cast_2$). Their backward trace involves the 10 callers of A, each of which could result in a long trail, making this approach expensive because of these two queries. In total, the demand-driven analysis analyzes all the 15 library methods in the figure. It gets 100% precision and covers all 10 cast instructions since its output is identical to the whole-program analysis. Note that the demand-driven analysis is the only one which requires a new demand-driven design of an existing inter-procedural analysis; the others use the existing interprocedural analysis as is.

The second baseline is an application-only analysis. Such an analysis analyzes the code inside the application circle in isolation and assumes the bottom element of the abstract domain for all library pointers outside. Hence it analyzes zero library methods and only *covers* the 4 casts that get their values from inside the application (that is, the casts numbered 7-10). The application-only analysis gets 100% precision because its errors are the subset of the whole-program errors that do not involve the library.

Averroes [AL13] improves upon the application-only analysis by modeling the whole library with a small summary. In Figure 4.2, everything outside the application circle is represented using this summary. The summary primarily consists of a single summary-pointer

to represent all library pointers, and a single summary-node to perform all the object initializations and application call-backs. A usual inter-procedural cast-analysis is performed on the application-code plus this summary. Averroes's summary is sound for some analyses, the cast check being one them. Hence, it covers all 10 cast instructions while only analyzing the summary. However, the analysis information merged in the common summary-pointer and summary-node drops precision relative to the whole-program analysis.

QueryMax's approach differs from Averroes primarily in that it selects a small part of the library to fully analyze instead of modeling the library using a summary. *QueryMax* keeps expanding the partial library to be used until it reaches some stopping criterion. Let us assume that we use *QueryMax* with a stopping criterion of 80% *query coverage*. This means that we will have to pick a *fragment* consisting of the application-code plus a partial library, such that at least 8 of the 10 queries (i.e. casts) are *covered* within this fragment.

QueryMax starts out by performing an *external source analysis* on the application code to find out which library pointers affect the 10 cast instructions. This information is marked by the arrows inside the application circle. *QueryMax* then assigns priorities to each external library method based on the number of casts it affects. In Figure 4.2, this is denoted by the numbers in the grey boxes. Next, *QueryMax* expands on the method with the highest priority (method B) to look at its callers, callees and field-reads. Method B has 2 callers, D and E. We estimate that each of D and E affects half as many casts as B, and hence each of them get half its priority (i.e. 1.5 each). Now, the method with the highest priority is A, which on expansion leads to 10 different caller methods, and we assign a priority of (2 / 10) to each of them. The next methods with the highest priority are D and E, followed by method C. Each of these methods are expanded in turn.

At this point, our *fragment* consists of the application code plus a partial library consisting of methods (A, B, C, D, E). Performing another *external source analysis* on this fragment shows that now 8 of the casts (casts numbered 3-10) are covered within this fragment. Recall that we started *QueryMax* with a stopping criterion of 80% query coverage,

Analysis Tool	**Casts covered**	**Lib Methods analyzed**	**Precision**
Application-only	4	0	100%
QueryMax	8	5	100%
Demand-driven	10	15	100%
Averroes	10	Summary	Low

Figure 4.3: Number of casts covered, library methods analyzed, and Precision (relative to the whole program analysis) for each of the competing tools

or in other words, we would like to terminate when 8 of the 10 casts (i.e. queries) are covered. Hence, *QueryMax* stops expanding at this point, and an existing inter-procedural cast-check analysis is now performed on this *fragment*. By terminating the expansion early, *QueryMax* avoided exploring the 10 callers of method A, and their subsequent callers which could potentially expand large sections of the program, while only answering the queries for $Cast_1$ and $Cast_2$. In total, by using *QueryMax*, we analyzed only 5 library methods and covered 8 casts. *QueryMax*, just like an application-only analysis, reports a subset of the whole-program errors, thereby getting 100% precision.

Figure 4.3 summarizes the number of library methods analyzed (less is better), the cast-instructions covered (more is better), and precision (more is better) for each of the four techniques. *QueryMax*, the demand-driven analysis and the application-only analysis each get 100% precision. For the other two metrics, *QueryMax* obtains a useful trade-off point in between the application-only analysis and the demand-driven analysis. Note that the differences in library methods analyzed is rather small for this example, but the differences are much larger in real programs. Averroes covers all casts and analyzes just the small summary, but gets low precision, thereby falling short of our 100% precision goal.

This example illustrates the core insight underlying *QueryMax's* speedup: few queries in the application code require large sections of the library for their analysis (like $Cast_1$ and $Cast_2$), whereas the remaining queries need a much smaller subset of the library. By identifying these expensive queries and assigning them a low priority, *QueryMax* can pick a

small partial library that is sufficient to cover all the remaining queries. The downstream client can now use this partial library in its analysis, which is a fraction of the size of the whole library. The trade-off is that the few expensive queries (like $Cast_1$ and $Cast_2$ in the example) are not fully covered by the partial library, resulting in a few missed errors.

4.3 Approach

In this section, we describe in detail how *QueryMax* works to pick the partial library to analyze.

4.3.1 Overview

QueryMax picks its partial library by finding the library classes mostly likely relevant to the queries in the application code. *QueryMax* accomplishes this by using a new static analysis called an *external source analysis*. *QueryMax* expands its partial library in a greedy fashion to maximize the number of queries answered in the application code until some stopping criterion is reached. We discuss two stopping criteria: a *class-budget* if the user wants to set a limit on the number of classes analyzed (proxy for analysis time), and a *query-coverage* if the user wants to set a goal for the number of queries covered (proxy for recall).

4.3.2 External Source Analysis (ESA)

The external source analysis, or *ESA* for short, takes a program and a subset of its classes called the *fragment*, and computes, for every pointer in the fragment, the set of external pointers that pass values to it. For example, defining the application code as the *fragment* would make the library pointers the external pointers, and an ESA would tell us which library pointers directly pass values to each pointer in the application code. An example of applying the ESA was illustrated in the example in Figure 4.2, where we computed the

No	Stmt	Condition	Constraint
1	x = y	x is not an array	ext(y) $\subseteq$ ext(x)
2	x = y	x is an array	ext(y) $\subseteq$ ext(x) and ext(x) $\subseteq$ ext(y)
3	x = y.f	field f is internal	ext(f) $\subseteq$ ext(x)
4	y.f = x	field f is internal	ext(x) $\subseteq$ ext(f)
5	x = foo(z)	target foo(p){.. ret q} is internal	ext(q) $\subseteq$ ext(x) and ext(z) $\subseteq$ set(p)
6	x = y.f	field f is external	{f} $\subseteq$ ext(x)
7	y.f = x	field f is external	(No constraint)
8	x = foo(z)	target foo(p){.. ret q} is external	{q} $\subseteq$ ext(x)
9	N/A	foo(x) has an external caller y.foo(z)	{z} $\subseteq$ ext(x)

Figure 4.4: Constraints for the External Source Analysis

library methods affecting cast-instructions in the application code.

The ESA is designed to be context-, flow- and field-insensitive because it's primary application is partial-program analysis, which is time-sensitive. Any overhead of performing an ESA during partial program analysis eats into the speedup that we may get over a whole-program analysis.

Figure 4.4 outlines the core constraints used for ESA. The second column lists a statement, the third column lists an accompanying condition, and the fourth column gives the corresponding constraint. The third column in the figure uses the words *internal* and *external*. A pointer is considered internal if it is within the fragment, and external otherwise. The abstract domain for the ESA consists of all possible subsets of external pointers. Hence, the notation $ext(y)$ in the fourth column represents the set of external pointers passing values to the fragment pointer y. This is different from the notation $\{z\}$ which is a singleton set consisting of the external pointer z.

Rows 1-5 in Figure 4.4 are identical to a standard context, flow and field-insensitive pointer analysis such as [SHR00], and we assume that the reader understands them well. Rows 6-9 deal with the different types of external pointers: external fields, external return values, and external function-arguments. The constraints for these rows are similar to what

one would expect for a *new* statement in a pointer analysis. Row 6 says that for the read of an external field f, the external field f should be added to the *ext* set of the assigned variable x. Row 7 says that writes to external fields produce no constraint. Row 8 says that for every external target of a method call, the return pointer of the target should be added to the *ext* set of the assigned variable x. There are no constraints for the arguments in this case. Row 9 says that if a method in the fragment has a caller outside the fragment, then the external caller's argument should be added to the *ext* set of the method's parameter.

The generated constraints can be solved using standard static-analysis constraint solving techniques. The complexity of solving the ESA constraints on a fragment of size p is $O(p^3)$. The complexity calculations are very similar to that of a context-insensitive pointer analysis.

In addition to the ESA, we define a faster version of it called the *fast-ESA*, with the primary change being to the abstract domain. Instead of maintaining the set of external sources for every fragment pointer, *fast-ESA* only maintains whether or not the set is non-empty. Hence there are only two elements in the *fast-ESA* abstract-domain: the top element is used when the fragment pointer may be passed a value by an external source, and the bottom element is used when the pointer is guaranteed to not get any values from external sources. The constraints are the same as in Figure 4.4, except for Rows 6-9 using the Top element instead of the external pointer names. Due to the smaller size of the abstract domain, the complexity of *fast-ESA* on a fragment of size p is $O(p^2)$, which is lesser than the cubic complexity of ESA. Hence, *fast-ESA* allows us to compute whether a fragment pointer is affected by external sources much quicker than an ESA.

4.3.3 QueryMax Algorithm

The *QueryMax* algorithm is used to pick a *fragment* to analyze, consisting of the application and the partial library, with a best effort to catch as many of the whole-program errors as possible. The example in Section 4.2 showed how *QueryMax* runs for one particular case. Here, we describe the algorithm (given in Figure. 4.5) in detail. The figure has three main

```
1: procedure QueryMax(appClasses, allClasses, queries)
2:     fragment ← appClasses
3:     visited ← new Set()
4:     pQueue ← new PriorityQueue()
5:     esa ← ESA(allClasses, appClasses)
6:     extLibPtrs ← computeAffectedQueries(esa, queries)
7:     for ExternalLibraryPointer e in extLibPtrs do
8:         pQueue.setPriority(e.method, e.affectedQueries)
9:         visited.add(e.method)
10:    end for
11:    while not (pQueue.empty() ∨ Criterion) do
12:        Method m ← pQueue.poll()
13:        analysisFragment.add(m.declaringClass)
14:        methodSlice ← getmethodSlice(m)
15:        newPriority ← m.priority / methodSlice.size
16:        for Method n in methodSlice do
17:            if visited.contains(n) then
18:                pQueue.addToOldPriority(n, newPriority)
19:            else
20:                pQueue.setPriority(n, newPriority)
21:                visited.add(e)
22:            end if
23:        end for
24:    end while
25:    return fragment
26: end procedure
27:
28: procedure BudgetCriterion(fragment)
29:    percentAnalyzed ← (fragment.size / allClasses.size)
30:    return (percentAnalyzed ≥ budget)
31: end procedure
32:
33: procedure CoverageCriterion(fragment, queries)
34:    coveredQueries ← fastESA(allClasses, fragment, queries)
35:    coverageRatio ← coveredQueries / fragment.totalQueries
36:    return (coverageRatio ≥ goal)
37: end procedure
```

Figure 4.5: QueryMax algorithm

procedures: the main algorithm, the class-budget stopping criterion and the query-coverage stopping criterion.

The main algorithm (line 1) takes as input the application classes, set of all classes, and the queries to be answered. For internal bookkeeping, *QueryMax* uses the set $fragment$ to mark the classes that are to be analyzed finally, a *visited* set for the methods, and a priority-queue *pQueue* to keep track of the priorities of the external (library) methods to be explored. The intuition behind the priority values is that they represent the estimated number of queries answered by that method, and *QueryMax* will explore methods with a higher priority earlier.

The main algorithm starts off by performing an ESA (line 5), with the application classes as the $fragment$. The ESA computes the set of external library pointers affecting each pointer in the application classes. Using the ESA result, we compute its inverse information: the number of queries affected by each of the external library pointers (line 6). Now, the method of each of the external library pointers is added to *pQueue* with a priority equal to the number of queries it affects. For external field pointers, we add the methods which write to that field. Each of the external library pointers' methods are added to the *visited* set. After this initialization phase, we move into the main algorithm loop.

The main algorithm loop starts at line 11. It keeps looping until either *pQueue* is empty or we satisfy the stopping criterion (described below). Inside the loop, we remove the method m with the maximum priority in *pQueue*, and add its class to the $fragment$. This step is a greedy move to expand the class that is expected to affect the largest number of queries. The next step is to find the *method-slice* of m (line 14). This is similar to computing one step in the backward slice of a pointer, but is performed at the granularity of methods instead of pointers to reduce the overhead. The *method-slice* consists of callers and callees of m, as well as methods which write to fields that are read in m. Each method in the method-slice gets a new priority which is the priority of m divided by the size of its method-slice. The intuition behind this priority assignment is that if m affects k queries and has t callers/callees, then

each caller/callee is expected to affect k/t queries. If a method from the method-slice is already in *pQueue* we add the new priority to its old priority, else we add the method to *pQueue* with the new priority. Finally, once the loop has terminated, the $fragment$, which has the set of classes to be analyzed, is returned. An existing inter-procedural static analysis is performed on the set of classes returned, with all external pointers assumed to be the bottom element.

QueryMax uses a stopping criterion to know when to stop expanding the fragment and return, and we experiment with two such criteria: *class-budget* and *query-coverage goal*.

Class budget. The class budget stopping criterion (line 28) is used when the user wants a handle on the analysis time. The class budget is a proxy for a time budget, and we prefer to use the number of classes instead of analysis time because it can be accurately computed in advance without running the actual analysis. This criterion simply checks if the percentage of classes used in the fragment is greater than a certain budget. The budget is assumed to be specified as a global variable for readability. For this chapter, we experiment with a 3%, 10% and 30% class-budget. A budget of under 2% will have no space for library methods in some programs, and a budget of over 40% will analyze a large partial library, resulting in only a small speedup.

Query-coverage goal. The query-coverage criterion (line 33) is used when the user wants a handle on the recall. Query-coverage is a proxy for recall, because the number of errors found is expected to be proportional to the number of queries covered. The query-coverage criterion uses a *fast-ESA* (line 34) to find the number of queries covered by the fragment classes, and computes a *coverage-Ratio* which is the percentage of queries covered. Finally, if the *coverage-Ratio* exceeds the query-coverage goal, then we return true. The goal is assumed to be specified as a global variable for readability. The coverage criterion is not used at every iteration of the main loop because the *fast-ESA* adds significant overhead. Instead, we only

evaluate this criterion at some set checkpoints. For this chapter, we experiment with 70% and 90% query-coverage goals. A goal of less than 60% gives recall close to that of a application-only analysis, and a goal of greater than 95% requires too many classes to be added to the partial library, thereby resulting in too small a speedup.

The overall complexity for *QueryMax* is $O(a^3 + p^2)$ where a is the size of the application-code and p is the size of the (application-code + partial-library). The $O(a^3)$ term comes from the ESA performed on the application-code on line 5, and the $O(p^2)$ term comes from the *fast-ESA* performed for the coverage-criterion on line 34.

4.3.4 Applicability of QueryMax to Client Static Analyses

Now that we understand how *QueryMax* works as a preprocessor to select a partial library, we can discuss what kind of client analyses *QueryMax* can be applied to.

Firstly, since *QueryMax* trades off recall for analysis speedup, its client analysis should be able to afford to lose some recall. For example, compiler optimization clients that prefer the static analysis be sound (or soundy [LSS15b]), will not use *QueryMax*. Secondly, *QueryMax* is restricted to client analyses that only care about errors manifesting in the application code. It cannot speed up a client analysis that aims to catch errors manifesting in both the application code and the library.

On the plus side, *QueryMax* makes no assumptions about the flow-, context- and field-sensitivity of the client analysis that it is preprocessing for. Hence it can be applied regardless of the client analysis' sensitivities. Further, unlike [AL13], it makes no assumptions about the demarkation between application and library code. Hence, the user can choose any subset of classes as the application code to focus on and get everything outside the subset treated as the library.

Figure 4.6 lists some analysis clients that *QueryMax* could be applied to and shows the corresponding analysis queries for such a client analysis. This is not an exhaustive list of

Client Analysis	Analysis Queries
Cast-check analysis [SHR00]	Cast instructions
Null-pointer analysis [HJP08]	Method calls and field accesses
Taint Analysis [LL05]	Taint sink instructions
Type-state analysis [FYD08]	State-change instructions
Pointer analysis [LH03c]	Client analysis queries

Figure 4.6: Analaysis Queries for different Client Analyses

client analyses, and its main purpose is to give examples of what the analysis queries would be for different kinds of client analyses. Typically, an analysis query would be any instruction in the application code where a particular kind of error could potentially manifest. For example, for a cast-check analysis the queries are cast instructions. For a null-pointer analysis they are all dereference instructions, including method calls and field accesses. For a taint-analysis which is defined in terms of vulnerable source-sink pairs, the analysis queries would be all the sinks. For a type-state analysis, like one that checks for the correctness of file-operations, all the state-change operations (like file-open, file-close, etc.) will be the analysis queries. A pointer analysis itself does not have any statements or variables of interest, and hence cannot define analysis queries for itself. However, if the pointer analysis is used by a particular client (like cast-check or taint analysis), we can define its analysis queries as the queries of that client.

4.4 Implementation

The WALA [WAL15] framework for Java bytecode analysis is used to implement *QueryMax* and the ESA analysis. The actual analysis is performed on the WALA IR, which is in SSA form and hence automatically grants partial flow-sensitivity. We use the CHA-callgraph for all the analyses, since computing a whole-program 0-CFA call-graph would defeat the purpose of doing a partial library analysis. We ignore call-graph edges involving a single call-site with more than 10 targets, since the likely root cause of this is severe imprecision, and it

results in mostly false-positives. We also exclude the *java/util* package since it is well known for introducing too many false-positives unless one uses high context-sensitivity [TPF09b].

Client Analyses *QueryMax* accepts any inter-procedural analysis to run with as long as the analysis can be run on a subset of the classes in the program. We experiment with two such analyses: a cast-check analysis and a null-pointer analysis. The cast-check analysis is based on the VTA algorithm [SHR00] for pointer analysis. The null-pointer analysis (based on [HJP08]), focuses on catching null-pointer exceptions resulting from uninitialized instance fields. The two analyses vary significantly in their constraints, abstract domains, design decisions, number of analysis queries, and number of errors per program. Hence, the two analyses offer considerable diversity for experimentation. We leave to future work to experiment with other client analysis, including other implementations of cast-check and null-pointer analysis, such as NullAway [BCS19].

For the analysis sensitivities, we choose to be context-, flow- and field-insensitive as far as possible. The cast-check analysis is insensitive on all three axes. The null-pointer analysis is context- and field-insensitive but flow-sensitive because a flow-insensitive version of the analysis trivially marks all fields as null. Our choice of sensitivities are different from other papers such as [SDA16, Spo11, SB06a], because their task is to improve precision, whereas ours is to improve analysis speed. For the task of improving precision, a flow-, context- and field-sensitive analysis is the hardest baseline because it is the most precise. In contrast, for our task of improving analysis speed, a context-, field- and flow-insensitive analysis is the hardest baseline because it is the fastest.

Demand-driven analysis We choose to write our own demand-driven cast-check instead of using an existing tool like [SDA16] or [SGS05a]. This ensures that the whole-program analysis and demand-driven analysis are identical in their various sensitivities, analysis design decisions, constraint solvers and errors generated. This normalization helps to make a fair

timing comparison between the demand-driven analysis, and other techniques like *QueryMax*, Averroes and the application-only analysis. For the demand-driven cast check, we implement caching across queries to reuse computations done for a previous query.

Most prior research on demand-driven analysis deals with pointer analysis which can be used to implement the cast-check. However, a design of the demand-driven version of the null-pointer analysis [HJP08] is not publicly available and is non-trivial to design from scratch. Hence, for the demand-driven analysis, we only report experiments for the cast-check analysis.

Averroes Averroes takes as input the original Jar file and the set of application classes, and produces modified Jar files consisting of the application classes and the library summary. We do not count the time taken to produce the modified Jar files since it is a one-time cost which is amortized across all client analyses. The Averroes library summary also has the *java/util* package excluded from it. Finally, the same null-pointer and cast-check analyses described above are run on the modified Jar files, thereby making a fair comparison between Averroes and the other techniques.

Reflection We do not use WALA's inbuilt reflection support for the client analyses because this would worsen the analysis time of the baseline, thereby making *QueryMax* look better. Further, we also do not use reflection support for the ESA. While reflection support may help the ESA find external sources reachable through reflection, its overhead is too high and this reduces the effective speedup provided by *QueryMax*.

Precision, Recall and Speedup To measure the quality of an analysis using *QueryMax* or any of the baseline techniques like Averroes, demand-driven analysis, etc., we evaluate it on the three axes of speedup, precision, and recall. Here are the standard formulae for computing these metrics:

Statistic	Mean	Std-dev
Lines of application code	9911	12689
Number of application classes	97	91
Number of 3rd party library classes	2608	5220
Percentage of application classes	0.33%	0.33%

Figure 4.7: Statistics about the benchmark programs

$$Speedup = \frac{Whole\text{-}program\ analysis\ time}{Application\text{-}focused\ analysis\ time}$$

$$Precision = \frac{|A \cap W|}{|A|} \qquad Recall = \frac{|A \cap W|}{|W|}$$

where A is the set of errors given by *QueryMax* and W is the set of errors given by the whole-program analysis (which we consider as the ground-truth).

4.5 Dataset Description

We use the NJR-1 dataset (available here [UKL20]), as our benchmark-set. We chose NJR-1 because its 293 Java bytecode programs run successfully with WALA, and each program explicitly lists its set of application and third-party library classes. Out of the 293 programs we remove 68 programs that crash the Averroes tool. The crash reports have been filed with the developers. Another 4 programs which run out of memory for the whole-program null-pointer analysis are removed, leaving us with a total of 221 programs.

Figure 4.7 lists some statistics about the benchmark programs. On average, each benchmark program has almost 10k lines of Java source code in the application, with an average of almost 100 classes each. The third-party library classes are much larger, with an average of 2608 classes per benchmark, and these correspond to an estimated 250,000 lines of Java source code. The application classes constitute just 0.33% of the program, with the

Statistic	Cast-check	Null-pointer
Total number of programs	221	221
Mean Errors per program	4.4	37
Std-dev Errors per program	27	56
Programs with non-zero errors	58	177
Mean Analysis time	27 sec	293 sec
Std-dev Analysis time	41 sec	142 sec

Figure 4.8: Statistics about the whole-program cast-check and null-pointer analysis on the benchmark set

remaining being the Java standard library and third party library classes. The large standard deviation for all these metrics implies that they vary significantly across benchmarks. Among the 221 benchmarks, 63 use reflection in the application code and 130 use reflection in the third-party libraries.

Figure 4.8 lists some statistics about the benchmarks when analyzed with a whole-program null-pointer analysis and the cast-check analysis. The cast check analysis gets 4.4 errors per program on average, whereas the null pointer analysis gets 37. This large difference is expected, since down-casting is rare, whereas method calls and field accesses are common.

The table also shows that only 58 of the 221 programs have non-zero cast errors and only 177 of them have non-zero null-pointer errors. The programs with zero errors in the whole program analysis are a problem for the evaluation because their recall is undefined for all of the techniques. Hence, the experimental results are reported in two parts:querymax- those with zero errors and those with non-zero errors. We report the recall and speedup for the non-zero error cases and only speedup for the zero error cases.

The analysis times for the two analyses also vary widely, with the cast-check taking 27 seconds per program and the null-pointer analysis taking 293 seconds per program. The standard deviation for analysis times is large, especially for the cast-check analysis, implying that a few outliers have large analysis times.

4.6 Experimental Results

In this section, we discuss our experimental results which validate the following claims.

1. **C1:** *QueryMax* gets a significant speedup, full precision and reasonable recall as compared to the whole-program analysis, with trade-off points that none of the existing techniques can achieve.

2. **C2:** The distribution of speedups and recall-scores are uniform across the benchmarks.

The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 GB RAM. For the JVM, the default heap size of 32GB, and default stack size of 1MB, was used. The artifact for the paper is available here [UP21].

The first two sub-sections validate the claims made, and these experiments focus on the programs with non-zero errors. The third subsection evaluates the programs with zero errors, the fourth examines the *QueryMax* analysis time split-up, the fifth compares the correlation between class-budget and analysis time, and the sixth subsection outlines the threats to validity.

4.6.1 C1: Main Result

Figures 4.9 and 4.10 show the various recall and speedup trade-off points for the cast-check analysis and null-pointer analysis respectively. The X-axis gives the recall plotted on a linear scale and the Y-axis gives the speedup plotted on a logarithmic scale. There is actually a third axis for precision, but we do not show it because all the techniques except for Averroes, get a 100% precision. We mark Averroes' precision directly in the figure.

Whole-program analysis The whole-program analysis (marked by the black circle) is considered as the ground-truth and the reference for all speedup calculations. Hence it trivially gets 100% recall and 1x speedup.

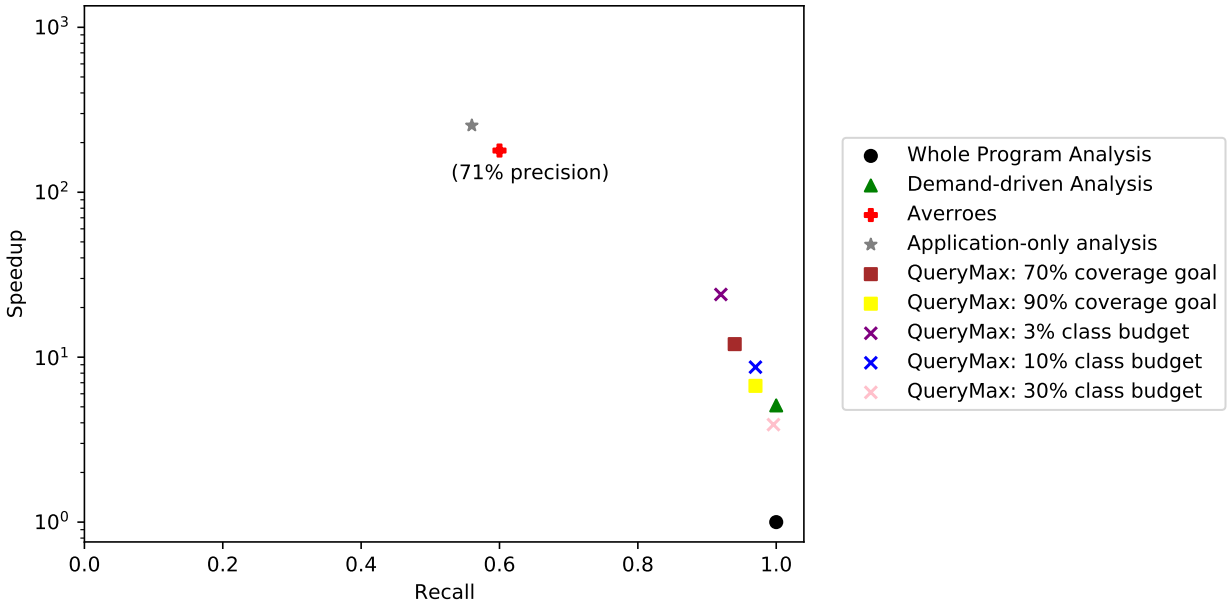


Figure 4.9: Recall and Speedup for the various techniques for the cast-check analysis

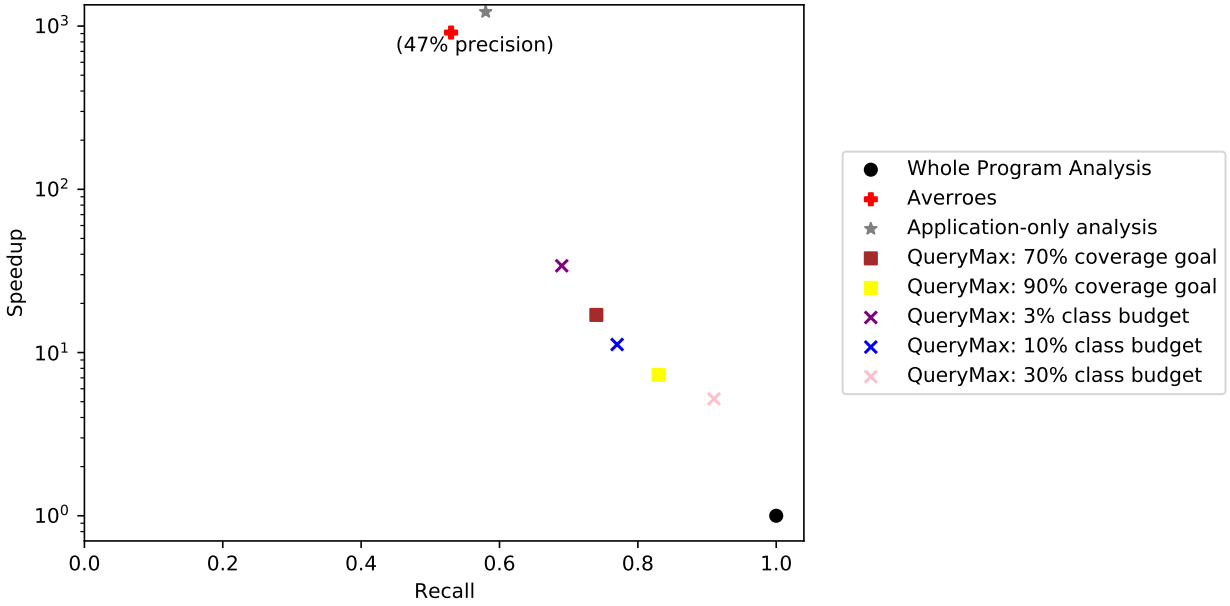


Figure 4.10: Recall and Speedup for the various techniques for the null-pointer analysis

Demand-driven analysis The demand-driven analysis (marked by the green triangle) computes the same result as a whole-program analysis and hence gets 100% recall, but it manages a 5.1x geometric mean speedup for the cast-check analysis because it avoids analyzing the whole program. This mean speedup is not representative of the average benchmark. One portion of the benchmarks get a large speedup because they analyze a small part of the program, while others experience a slowdown because they analyze a large section of the program and the demand-driven analysis adds some overhead. The reason for this difference in speedups is that some programs either have expensive queries like the example in Section 4.2, or a larger number of queries, and others don't. This observation is in line with previous experiments on demand-driven analyses [HT01]. A demand-driven version of the null-pointer analysis does not exist (see why in Section 4.4), but we expect it to perform worse than in the cast-check analysis because there are significantly more queries in the null-pointer analysis and the demand-driven analysis works on a per-query basis.

Application-only analysis At the other end of the spectrum is the application-only analysis (marked by a grey star), which is orders of magnitude faster, but gets a significantly lower recall. For the cast-check analysis it gets a 254x speedup and a 56% recall, whereas for the null-pointer analysis it gets 1222x speedup and 58% recall. The large speed-up is attributed to the fact that the application constitutes only 0.33% of the whole program on average (Figure 4.7). An application-only analysis is a good option for use-cases where analysis speed is significantly more important than recall, but when both are important, it doesn't strike as good of a balance between the two.

Averroes The point closest to this is Averroes (marked by a red plus), which gets a (179x speedup, 60% recall, 71% precision) for the cast check analysis, and a (913x speedup, 53% recall, 47% precision) for the null-pointer analysis. This is the only tool for which we report the precision because the other tools get 100% precision.

The massive speedup of Averroes is attributed to the fact that its summary is tiny compared to the size of the library. However, the tiny size is also what causes analysis information to be merged and precision to drop. The 47% and 71% precision values are significantly lower than our target of 100% precision.

Averroes should theoretically get 100% recall for the cast-check, but not for the null-pointer analysis because its library summary includes information about object-initialization but not about field-initialization. The observed recall is lower than expected because of a bug in its dealing of inner-classes which causes any error propagating through a Java inner-class to be dropped. The bug has been reported to the developers.

QueryMax Finally, *QueryMax* gives some points in between these two extremes. The points marked with crosses are for the class-budgets and the points marked with with squares are for the query-coverage goals.

For the cast-check analysis (Figure 4.9) *QueryMax* performs very well. The 3% budget (purple cross) gets a 24x speedup and 92% recall, and this strikes a really useful balance between the two metrics. The 10% budget (blue cross) gets an 8.7x speedup and a 97% recall, thereby favoring the recall a little more than the speedup, but still a great trade-off between the two metrics. The 30% budget (pink cross) gets 3.9x speedup and a 99.6% recall.

The query-coverage stopping criterion (represented by the squares) for the cast-check analysis gets similarly good results. The 70% goal (brown square) gets (12x speedup, 94% recall) and the 90% goal (yellow square) gets (6.7x speedup, 97% recall). The speedups for the coverage goals are slightly lower than the class budgets. For example, the yellow square in Figure 4.9 is directly below the blue cross. This happens because calculating the query-coverage involves the overhead of at least one *fast-ESA*, which the class-budget version avoids. However, the coverage-goal gives a guarantee on the number of queries covered, which could be more valuable than a guarantee on the number of classes analyzed.

For the null-pointer analysis (Figure 4.10), we see a similar speedup vs recall trade-off for

QueryMax. The 3% class-budget, marked by the purple cross gets (34x speedup, 69% recall), the 10% class budget marked by the blue cross gets (11x speedup, 77% recall), and the 30% class-budget, marked by the pink cross gets (5.2x speedup, 91% recall). The query-coverage points (marked by squares) lie in between these three points. Unlike the cast-check analysis, the coverage-goal variants are not much worse than the class-budget variants for the null pointer analysis. We discuss the reason for this observation in Section 4.6.4

Comparing figures 4.9 and 4.10 shows that *QueryMax* gets much better recall for the cast-check than the null-pointer analysis. The main reason for this is that some dereference instructions get a high-priority from *QueryMax*, but are often never null-pointer exceptions. For example, in any given program, the *println()* call occurs many times, and in all cases gets its value from the field *java/lang/System.out*. Since this field affects several dereference instructions, it ends up getting a high-priority and that part of the library gets added to our partial library first, even though the *println()* calls never cause null-pointer exceptions. A similar case happens to some other common dereference instructions.

To sum up, *QueryMax* with either stopping criterion provides a useful analysis design point in-between the application-only analysis and the demand-driven analysis, just like in the example from Section 4.2. Further, unlike Averroes, it achieves this speedup without sacrificing precision, and thus continues to meet the high-precision expectation of its users.

4.6.2 C2: Distribution of Recall and Speedup

We now understand the recall and speedup trade-off points for *QueryMax*, but we would also like to know their distribution across the benchmark programs. Figures 4.11 and 4.12 use a histogram to show the distribution of the recall and speedup for *QueryMax* with a 70% query coverage. The X-axis gives the speedup or recall, with the values split into bins, and the Y-axis gives the number of programs in each bin. Just like figures 4.9 and 4.10, we use a logarithmic scaling for speedup here. The recall is still plotted on a linear scale.

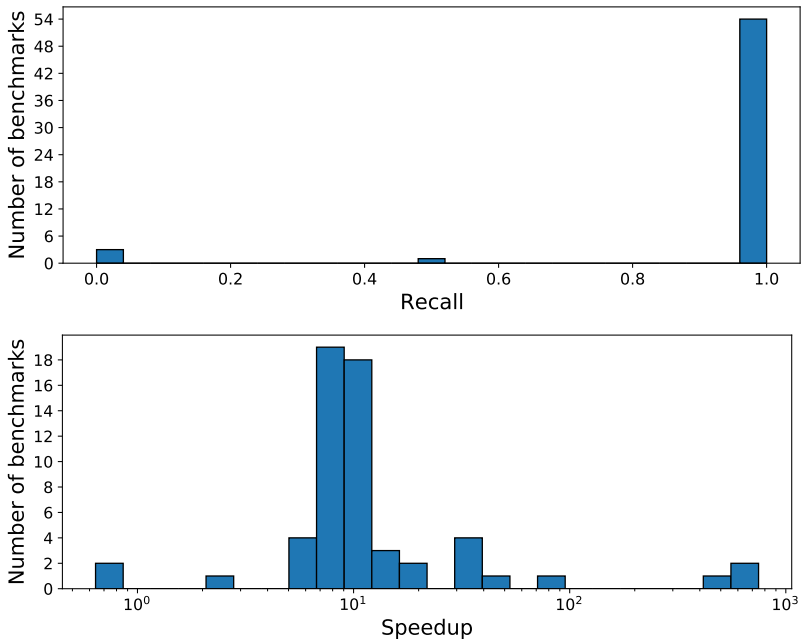


Figure 4.11: Speedup and Recall histograms for *QueryMax* (70% query coverage) on the cast-check analysis

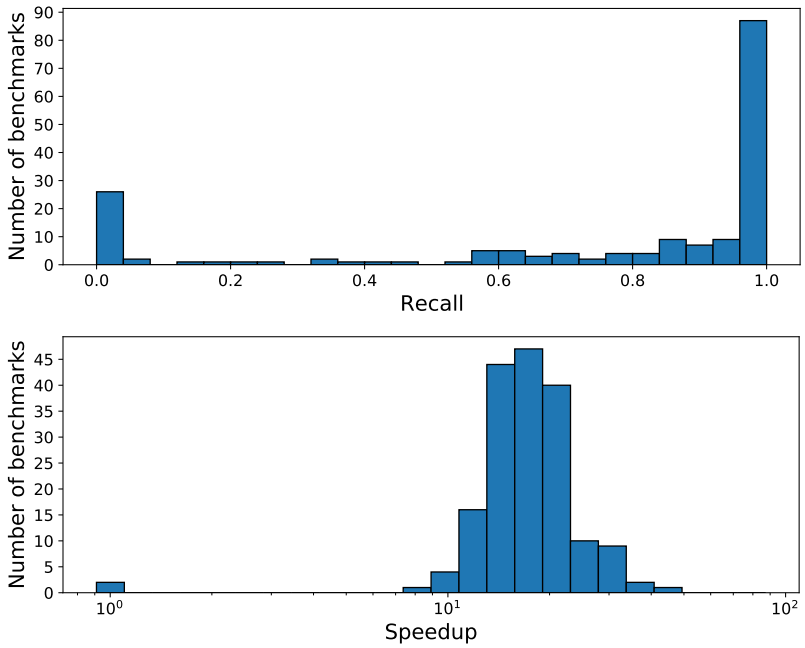


Figure 4.12: Speedup and Recall histograms for *QueryMax* (70% query coverage) on the null-pointer analysis

Analysis	Cast-check	Null-pointer
Application-only	395x	2196x
Averroes	230x	1744x
QueryMax 3% class-budget	30x	84x
QueryMax 10% class-budget	13x	33x
QueryMax 30% class-budget	6.4x	18x
QueryMax 70% query coverage	16x	20x
QueryMax 90% query coverage	12x	10x
Demand-driven	42x	N/A

Figure 4.13: Speedup for the various analysis techniques for the Zero-error benchmarks

The recall for *QueryMax* with the cast-check analysis (Figure 4.11) is close to 100% for most of the programs, with only a couple of programs getting lower scores. Two programs get a 0 recall. These programs had just 1 and 2 errors each and missing those errors meant a recall of 0. The null-pointer analysis (Figure 4.12) has a similar story for recall, but it has a larger number of programs with 0 recall. In most of these cases, the null-errors are very few and highly related, and hence missing one library method could cause all the null-errors to be missed.

The speedups for both analyses are consistent, with most programs getting close to the mean speedup value. The cast-check has 2 programs that get less than a 1x speedup. This happens because if *QueryMax* cannot guarantee that 70% coverage has been reached by the time its chosen fragment expands to 30% of the program, it simply falls back to picking the whole program, thereby resulting in no speedup.

4.6.3 Zero-Error Benchmarks

The results so far focused on the programs with non-zero errors. Figure 4.13 lists the speedup for programs with zero errors in the whole-program analysis. The speedups for *QueryMax* are on average twice as much as the non-zero error benchmarks. The demand-driven cast-check however, gets a 42x speedup here as compared to the 5.1x speedup on the non-zero error benchmarks. This high speedup for the demand-driven analysis on these benchmarks

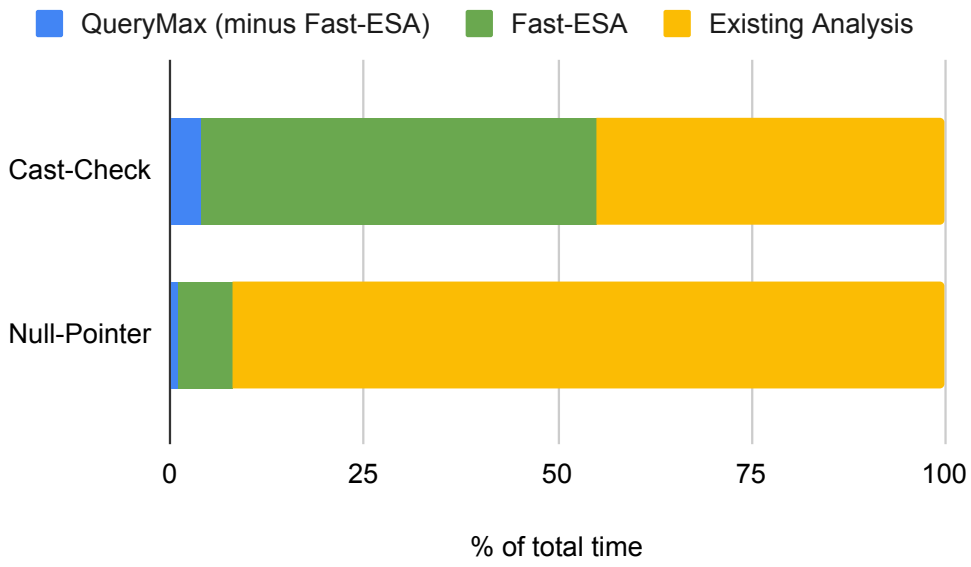


Figure 4.14: Split up of the time taken by each component for an analysis using *QueryMax* with the query-coverage goal

stems from the fact that these programs have much fewer down-cast instructions than the non-zero error benchmarks. Thus, when there are very few analysis queries, a demand-driven analysis gets a higher speedup.

4.6.4 Split-up of Analysis Time

Recall the workflow of *QueryMax* from Figure 4.1. We first run *QueryMax* with either a query-coverage goal or a class-budget. For query-coverage, *QueryMax* includes the additional overhead of the *fast-ESA*. Finally, we run the existing analysis. Figure 4.14 gives a split-up of the time between *QueryMax* (minus the *fast-ESA*), the *fast-ESA*, and the existing static analysis, for the query coverage goal.

For the cast-check, the *fast-ESA* takes 51% of the time, whereas the other *QueryMax* part takes just 4%. This explains why the query-coverage criterion from Figure. 4.9 is slower than the class-budget one; computing the query-coverage needs the *fast-ESA*, but computing

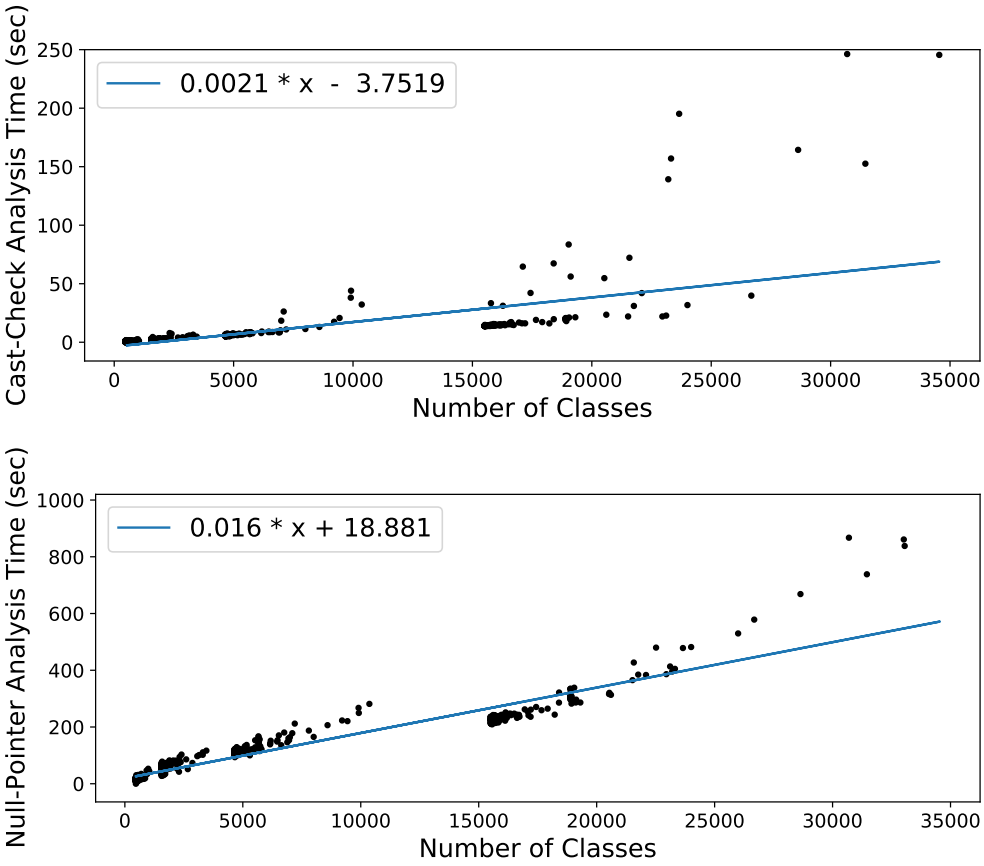


Figure 4.15: Class-budget and analysis time relationship.

the class-budget does not.

For the null-pointer analysis, both the *fast-ESA* and the other part of *QueryMax* take up a small percentage of the time (8% totally). The contribution of *QueryMax* and *fast-ESA* to the total analysis time is larger for the cast-check than the null-pointer analysis. The reason for this is that existing null-pointer analysis has a longer absolute analysis time than the cast-check, but the absolute *fast-ESA* time is similar in both cases.

4.6.5 Analysis-time vs Number of Classes

As a minor result, we show the relationship between the class-budget and the analysis time, to justify our use of the former as a proxy for the latter. Figure 4.15 compares the number

of classes analyzed on the X-axis with the analysis time on the Y-axis for both analyses. Each point represents one analysis of *QueryMax* with a class-budget. For both analyses, the analysis time is almost linear, but the cast check has more outliers, which explains the high-standard deviation for its analysis time (see Figure 4.8). The figure also plots a regression line, and the equation of this line can be used to convert time-budgets into class-budgets.

4.6.6 Threats to Validity

There are two main threats to validity. The first is that out of the application, third party libraries and standard library, the standard library forms the largest part. Even though different programs interact with different parts of the standard library, it still means that the benchmarks are not perfectly independent for a static analysis. However, this issue occurs with any static-analysis benchmark-set where the programs access the standard library.

The second is that analysis time measurements for all the programs were performed using a single run, even though execution times can vary across runs. However, since the speedups are large (an order of magnitude) and the benchmarks are numerous, these variations matter less. Further, since the total experiment-time is already ten days, performing multiple runs is infeasible.

4.7 Related Work

The three research directions that focus on speeding up static analysis by avoiding the analysis of the entire program are library-summary based analysis, demand-driven analysis, and the analysis of program fragments. We discuss each of these in turn.

Library-summary based analysis The main idea behind the research in this area is to create an analysis summary for the library and use this library summary instead of the actual library code to analyze the application.

Averroes [AL13] heavily compresses the library into a small summary. This summary consists of a single summary-pointer to represent all library pointers, stubs for methods called directly from the application, and a single summary-method to perform all the object initializations and application call-backs. Since this summary is quite small compared to the library, using it in place of the library results in a massive speedup. However, the small size of the summary has two downsides:querymax- precision drops because information is merged in the single summary-pointer, and some kinds of information (like field initialization information for the null-pointer analysis) get left out out of the summary. *QueryMax*, in contrast, leaves out no information in the partial library that it chooses, and more importantly, preserves the precision.

The *component-level analysis* by Rountev et. al [RKM06, RSX08] differs from Averroes in that its library summary contains all the information necessary to get the same result as a whole program analysis. The first time an analysis is run, the library is separately analyzed and summarized, and the summary is integrated with the application analysis. This saves no time in the initial run (the overhead causes a slowdown). However, it saves time in subsequent runs when the same library summary is reused across different programs or future versions of the program. *QueryMax* on the other hand never uses the whole library and it speeds up the analysis of each program independently. Further, unlike the component-level analysis which needs a separate design for each type of analysis, *QueryMax* can be used off-the-shelf with any analysis.

Demand-driven analysis Demand-driven analyses [SGS05a, HT01, SB06b, SDA16] are well-accepted as the most efficient option for single analysis queries, and work best for resource-constrained environments like IDEs and JIT compilers. They also perform well when the number of queries is small [SGS05a]. However, when analyzing entire applications in which the number of queries is large, the demand-driven analysis could end up analyzing large parts of the program and cause a slowdown because of their overhead [HT01]. We

also see this observation in our benchmarks, where some programs get huge speedups over a whole-program analysis, but some experience slowdowns.

Unlike the demand-driven approach, *QueryMax* avoids expensive queries by assigning them a low priority, like in the example from Section 4.2. It also avoids the demand-driven overhead since it still runs a batch analysis, thereby performing better when there are many queries to be answered. Further, since *QueryMax* is only a preprocessor to an existing whole-program analysis, it can be used with an existing analysis, without requiring a design of a demand-driven version of it.

Analysis of Program Fragments There has been past research on analyzing program fragments in isolation. In our use-case, the program fragment is the application-code. Cousot and Cousot [CC02] describe four techniques for this general approach. The first is a simplification-based separate analysis, which analyzes the various fragments of a program separately and then combines their information. This idea is similar to the library-summary based analysis by [RKM06], and has the drawbacks as discussed above. The second technique is a worst-case analysis, which means running an application-only analysis, but using the top element of the abstract domain for library pointers. This introduces additional false-positives. Our experiments on this technique show that it gets a precision (averaged over both analyses) of 22% which is far below our 100% target precision. The third technique is to ask a user to provide stubs for the library (i.e. information about the library interface) and then perform an application-only analysis that incorporates these stubs instead of the library. This can give high recall, precision and speedup, but it requires a static-analysis expert to manually write and update the stubs for each library. The fourth technique uses a relational abstract domain and analyzes a program fragment by giving symbolic names to external pointers and lazily evaluating the values they pass. To the best of our knowledge, there are no recent implementations or experimental results to compare the effectiveness of this technique in practice.

Rountev et. al [RRL99] introduce a technique to improve the performance of a whole-program flow-sensitive analysis. They perform a flow-sensitive analysis for the application code and then use a whole-program flow-insensitive analysis to overapproximate the effect of the library pointers. The two limitations of this technique are that it drops precision as compared to the original flow-sensitive analysis, and it cannot be used to speed up a flow-insensitive analysis. *QueryMax* on the other hand maintains the same precision as the original analysis tool and works with any level of context-, flow- or field-sensitivity.

CHAPTER 5

RLFixer: Automated Repairs for Resource Leak Warnings

In this chapter, we introduce the third tool, *RLFixer*, which tackles the third biggest pain point for users: a lack of suggested fixes in existing static analysis tools. More specifically, *RLFixer* focuses on repairs for resource-leak warnings from static analysis tools. When applied to five popular Java resource-leak detectors, *RLFixer* generates repairs for 66% of their warnings with a 14 second repair time and a fix-correctness of 95%.

5.1 Overview

Motivation. Most programs use resources such as files, sockets and database connections. Resource leaks are a common bug introduced unintentionally by programmers, which can result in security vulnerabilities [CWE22] and severe failures [GCS20]. Resource leaks are elusive because they only cause crashes when many resources leak and the OS runs out of that resource-type; this typically does not happen during testing. An effective approach for identifying these resource leaks during development is static analysis [KSS21b]. Today, developers can choose from several open-source static-analysis tools that perform resource-leak detection [CDD15, KSS21b, Spo17, Cod20, PMD02], many of which provide accurate warnings.

While static analyzers can detect resource-leaks, users also need tool-support to fix these errors. For example, Christakis and Bird's empirical study [CB16c] shows that a lack of

suggested fixes is one of the top pain points reported by static analysis users. Other developer studies [JSM13b, SAE18a, DWA22] also report very similar findings. Hence, what we need is a tool to fix resource-leaks.

Existing repair tools. Since there are currently no specialized tools for resource-leak fixing, one could try using *general-purpose repair tools* [COZ21, LCL17a, LCL17b, AMS21, MYR16, NQR13, JXZ18, KNS13, LR15, LR16, WNL09, LWN20, JLT21, XZ22, LWN22, YMM22], which work on a wide variety of errors. These tools generate candidate patches using a variety of techniques, but they all validate a patch by checking if it passes the previously failing test case. Resource leaks, however, do not show up during tests, and hence cannot be fixed by such tools. Footpatch [TG18], one of the only general-purpose tools that does not rely on tests, is the current best tool for fixing resource leaks. However, it suffers from low-quality fixes for Java resource-leaks; it suggests fixes for only 15% of the leaks, out of which only 50% are correct.

Achieving a perfect *fixable-rate* (percentage of warnings for which a fix was suggested) and *fix-correctness* (percentage of correct fixes out of the suggested fixes) for Java resource-leaks is a lofty goal. The problem is at least as hard as compile-time object deallocation [GMF06, CR07] (i.e. replacing Java's runtime garbage collector with static deallocation), a known hard problem for compilers. Furthermore, in this repair problem, some corner cases involving loops or aliasing also reduce to undecidable problems. Hence, there will always be some resource-leaks that are infeasible to fix for a compile time-tool. However, we show that by separating the leaks that are infeasible to fix from those that are feasible to fix, it is possible to have better repairability than Footpatch in both fixable-rate and fix-correctness.

Our Approach. In this paper, we introduce *RLFixer*, a specialized repair tool for resource leaks that generates high-quality fixes. Fig. 5.1 gives an overview of its workflow. The warnings computed by an existing black-box resource-leak detector are first parsed to extract

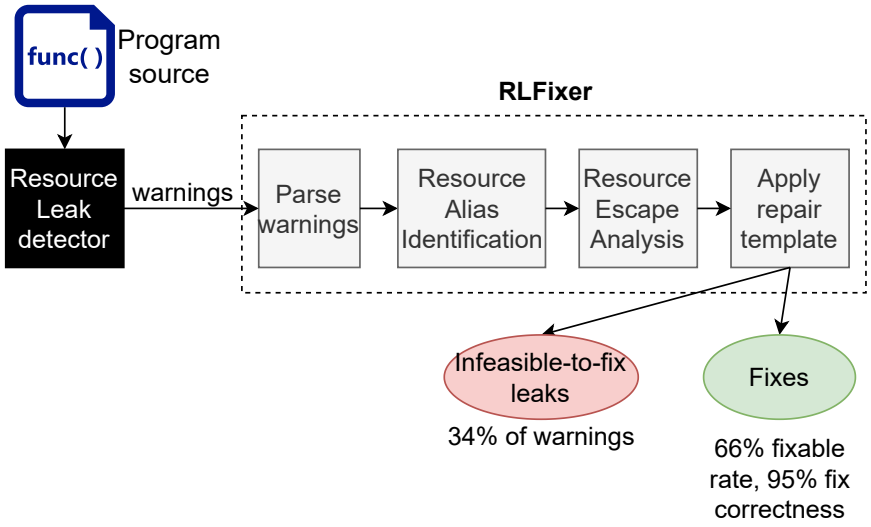


Figure 5.1: Overview of the RLFixer workflow

the location where the resource was created. Next, the *resource alias identification* step identifies pairs of resource objects that use the same underlying system resource. The third step tracks the data-flow of the resource object using a new demand-driven static analysis called a *resource escape analysis*. This analysis serves two purposes: it identifies leaks that are infeasible to fix, and it helps pick the correct repair-template for the feasible-to-fix ones. Finally, using this repair template, the last stage generates the correct fix.

In addition to generating correct fixes, we designed *RLFixer* to be fast. Repair time is important because suggested fixes typically accompany static analysis warnings in IDEs, which are time sensitive environments. *RLFixer's* demand-driven design enables it to analyze only those statements relevant to the resource leak and hence takes, on average, only 1 seconds per program, excluding the 13 seconds for setting up the call-graph, class-hierarchy, etc. This is significantly faster than most other repair tools, which take minutes or hours.

We evaluated *RLFixer* by applying it to five popular Java resource-leak detectors: Infer [CDD15], PMD [PMD02], Checker-Framework [KSS21b] , Codeguru [Cod20], and Spotbugs [Spo17], each of which is run on programs from the NJR dataset [UKL20].

Our Contributions. We begin with an example of RLFixer fixing a resource leak (Section 5.2), and then we detail our contributions:

- We introduce a new static analysis, *resource escape analysis*, which helps identify leaks that are infeasible to fix, as well as pick the repair template for feasible ones (Section 5.3).
- We design and implement *RLFixer*, a specialized repair tool for resource-leaks that is based on the *resource escape analysis*, and can repair leaks from multiple leak detectors (Section 5.4).
- We show, experimentally, that *RLFixer* generates high-quality fixes with low overhead for five popular Java resource-leak detectors. On average, it generates fixes for 66% of the warnings, out of which an estimated 95% are correct (Section 5.6).

We end with a discussion of related work (Section 5.7).

5.2 Examples

This section shows two simplified examples of how the five resource leak detectors report leaks, and how *RLFixer* goes about repairing them. It highlights the need for suggesting fixes for resource-leaks, as well as some of the challenges in generating a correct fix.

Fig. 5.2 shows a simplified Java code snippet from one of the NJR benchmarks. It has two methods, each with one resource object. First, let us look at the method `foo`. `foo` creates a `FileReader` resource, which gets passed in to the `bar` method. Note that `foo` continues using the `FileReader` after the `bar` function returns. The `foo` method also declares that it potentially throws an exception. This declaration is required by the Java-compiler's type and effect system when a resource's potential exception is not handled in a *try-catch* block. The lines highlighted in green constitute the fix suggested by *RLFixer*; they have not been added to the code yet.

```
1  ..
2  void foo(File a) throws IOException{
3    FileReader fr = null;
4  + try{
5    fr = new FileReader(a);
6    bar(fr);
7    int data = fr.read();
8  + }finally {
9  +   try{
10 +     fr.close();
11 +   }catch(Exception e){
12 +     e.printStackTrace();
13 +   }
14 + }
15 }
16
17 void bar(FileReader f) {
18   BufferedReader r = null;
19   try {
20     r = new BufferedReader(f);
21     System.out.println(r.readline());
22   } catch (IOException e) {}
23 }
```

Figure 5.2: Example of a resource leak fixed by RLFixer

Next, let's focus on the method `bar`. The `bar` method creates a `BufferedReader` resource object with the `FileReader` parameter `f` as an argument. Here, the `BufferedReader` is a wrapper resource that provides buffering functionality for the `FileReader f`. Hence we say that the resource variables `f` and `r` are *resource aliases*. This means that even though they point to different resource objects (`f` points to a `FileReader` and `r` points to a `BufferedReader`), the underlying system resource pointed to by those objects is the same. This implies that closing one resource object closes all its resource aliases. In this case, neither the `BufferedReader` nor the `FileReader` have been closed, and hence we get a resource leak.

We now run five resource-leak detectors (Infer, PMD, Checker-Framework, Codeguru, and Spotbugs) on this code, and Fig. 5.3 shows the output given by each. PMD and Infer identify a resource leak for the `BufferedReader`, whereas Checker-Framework identifies a

Tool	Output
Infer	Resource of type BufferedReader at line 20 is not released after line 21
PMD	Ensure that resources like this BufferedReader object are closed after use (line 18)
Checker-Framework	@MustCall method close may not have been invoked on 'fr' or any of its aliases (line 5)
Codeguru	N/A (Resource leak missed)
Spotbugs	N/A (Resource leak missed)

Figure 5.3: Outputs for the five resource-leak detectors, when given the code snippet from Fig. 5.2

resource-leak for the `FileReader`. Codeguru and Spotbugs do not report any resource leak. Even after getting one of these warning messages, a developer is still several steps away from a correct fix.

We also run the baseline repair tool, Footpatch, on this file-handle leak. Footpatch is tightly integrated with Infer, and relies on Infer's warning output for identifying fix locations. Footpatch first generates candidate patches by searching the code-base for program fragments that close a file, and then validates the patches by confirming that they fix the error. In this case, Footpatch is unable to generate any patch candidates for the warning. Furthermore, even if Footpatch did hypothetically find a patch, it would apply the patch at the location in Infer's warning (after line 21). Closing the `BufferedReader` after line 21, or anywhere in function *bar*, will mean that the file pointed to by its resource-alias *FileReader* will be closed before it is read on line 7. This fix is dangerous since it introduces a new use-after-close error.

Finally, let us examine how *RLFixer* deals with the resource leak, assuming the warning came from Infer (i.e. for line 20). *RLFixer* starts off by performing a *resource alias identification* for the new `BufferedReader` object. This analysis reveals that `f` is a resource-alias. Next, *RLFixer* performs a *resource escape analysis*, a static analysis that computes how the `BufferedReader` and any of its aliases escapes the method. The two ways the resource

```
1  class A{
2    FileWriter fw;
3    void create(File b) throws IOException{
4      FileWriter f = new FileWriter(b);
5      store(f);
6    }
7
8    void store(FileWriter a) throws IOException{
9      fw = a; /* Resource escapes to a field */
10   }
11 }
```

Figure 5.4: Example of a resource leak that is infeasible to fix

escapes the `bar` method are via the `readline` method call and via the parameter `f`. When a resource escapes via a parameter, we cannot close the resource in the current method, since the resource is still accessible after the method returns. Instead, we examine the caller, which is in `foo`. Carrying out the *resource escape analysis* for *fr* in `foo` shows that it only escapes via method calls, and hence can be closed in the method `foo` itself. *RLFixer* then picks the correct repair-template, and it suggests the fix highlighted in green in Fig. 5.2. The repair-code correctly fixes the leak without introducing new errors or modifying the semantics of the original program. *RLFixer* computes the same fix for the warnings given by PMD and Checker-Framework.

A Resource Leak that is Infeasible to fix

Figure. 5.4 shows an example of a resource leak that may be infeasible to fix at compile time. *RLFixer*, during its *resource escape analysis*, tracks the FileWriter resource through the call to the method `store`, and identifies that it is assigned the field `fw`. Since this field is accessible as long as its parent A object is alive, we can only safely close this resource when the A object is getting deallocated. This reduces the problem to compile-time object deallocation [GMF06, CR07], which has been a known hard problem for compilers. Prior research has only managed to statically deallocate some objects in the program [GMF06],

and the hardness of this problem is the reason why Java uses a runtime garbage collector. This is just one of the infeasible cases for resource-leak repair; we discuss the full list of cases in Section 5.3.

There will always be resource-leaks that are too hard to fix statically. *RLFixer* aims to identify and separate out the hard-to-fix leaks like the one in Fig. 5.4, while correctly fixing the rest of the resource-leaks, like the one in Fig. 5.2.

5.3 Approach

This section gives an overview of *RLFixer's* approach to fixing resource-leaks. Fig. 5.1 shows the four main components of *RLFixer*: the warning parser, the resource alias identification, the *resource escape analysis*, and the application of repair templates; we now discuss each of these in detail.

5.3.1 Warning Parser

The first component parses the resource-leak detector's warning and extracts the source file and line number where the leaked resource was created. Each resource-leak detector needs a separate parser because each tool uses a different output format, but this component is simple and small. On average, it takes only 15 lines of Python code per new tool.

5.3.2 Resource Alias Identification

The second step for *RLFixer* is identifying resource aliases for the leaked resource objects. Since a majority of resource usage in Java involves resource aliasing, this step is essential before proceeding with any kind of static analysis over resources. We have already seen an example of resource aliasing in Fig. 5.2, where the *FileReader* and *BufferedReader* objects pointed to the same OS resource. Prior research [KSS21b, TC10] has studied resource-

```
1  class WrapperType{
2    private ResourceType out;
3    public WrapperType(ResourceType w){
4      out = w;
5    }
6    public close(){
7      out.close();
8    }
9  }
10 ..
11   {
12     /* Resource-leak reported here*/
13     x = new ResourceType("a.txt");
14     y = new WrapperType(x);
15   }
```

Figure 5.5: Resource Alias Identification: checking if the WrapperType object is a wrapper for the ResourceType object

aliasing from the point of view of pruning false-positive resource-leak warnings. Here, we study resource-aliasing from the point of view of generating correct repairs. Below is the resource-aliasing definition that *RLFixer* uses.

1. Variables x and y are resource-aliases if x is a wrapper for y, or y is a wrapper for x.

2. Resource W constructed with the constructor C_W is a wrapper for resource R if:

   (a) R is passed as a parameter to C_W, and

   (b) R is a must-alias of a field of W at the end of C_W, and

   (c) The must-alias field always gets closed in the `close()` function of W

3. All pointer aliases are treated as resource-aliases.

This definition also serves as a specification for a static analysis, which *RLFixer* implements to identify resource alias pairs. Let us use the example from Fig. 5.5 to check if the variables x and y are resource-aliases. The resource leak warning is reported for the `ResourceType` object on line 13. The `ResourceType` cannot be a wrapper for any other

object because its constructor only takes a string input, and this will never satisfy condition (2a). So let us check the 3 conditions for the `WrapperType` to be a wrapper for the `ResourceType`. We first perform a def-use analysis [PLR94] of x, which identifies all uses of x. Since x is used as a parameter in the constructor for the `WrapperType`, condition (2a) is satisfied. Next, for condition (2b), we check the `WrapperType` constructor and its callees for an assignment of its parameter w (or one of its aliases) to a field of the `WrapperType`. In this case we have such a field, *out*; the condition is satisfied. Analyzing the `close` function reveals that the resource from the field *out* gets closed in it and this satisfies condition(2c). Thus, all three conditions are satisfied; the `WrapperType` and `ResourceType` are resource-aliases. The final part of the definition says that all pointer-aliases are resource-aliases. Computing pointer-aliases is a solved problem, and can be done with a typical demand-driven pointer analysis [SGS05b].

We now know how to identify pairs of resource-aliases, but we also need to consider resource-objects that are linked by multiple layers of resource-wrapping. This is quite common in Java programs; a resource can be wrapped in up to four or five layers of resource wrappers. We identify this linking by computing a transitive closure over the resource-aliasing relationship.

5.3.3 Resource Escape Analysis

The third component, the *resource escape analysis*, takes the warning parser's resource-leak location and list of resource-alias pairs as input, and computes all the types of program constructs that the resource can escape to. This analysis is used by *RLFixer* for two purposes: it helps separate out the infeasible-to-fix leaks, and it helps compute repairs for the feasible-to-fix leaks.

The *resource escape analysis* is carried out on the WALA IR [WAL15] [1] , since it is

[1]WALA is the static analysis framework used by RLFixer, and the WALA IR is very close to Java bytecode.

```
Program :: (C, C, ... C)
C :: <cname> Ext ImpDS {fields:{f;..f;} methods:{M,..,M}}
M :: <mname>(V_1 ,..,V_n) {instructions:{I;..I;}}
I :: ArrayStore | FieldWrite | Assgn | PhiStmt | ReturnStmt | Invoke |
    FieldRead | ConditionalBranch | NewStmt | ArrayLoad
ArrayStore :: V_arr[V_idx] = V_rhs
FieldWrite :: V_lhs.f = V_rhs
Assgn :: V_lhs = V_rhs
PhiStmt :: V_lhs = phi(V_1 ,..,V_n)
ReturnStmt :: return V
Invoke :: V_lhs = V_rhs.<mname>(V_1 ,..,V_n)
FieldRead :: V_lhs = V_rhs.f
ConditionalBranch :: if V_bool goto I
NewStmt :: V_lhs = new <cname>
ArrayLoad :: V_lhs = V_arr[V_idx]
V :: <var-name>
Ext :: extends <cname> | ''
ImpDS :: implements <data-structure-interface> | ''
```

Figure 5.6: Simplified grammar for the WALA IR

easier to write a data-flow analysis for WALA IR than Java source code. Fig. 5.6 gives a simplified grammar of the WALA IR. Since most of the grammar has terms that are common in most intermediate representations, we only explain a few terms. The `PhiStmt` is a special instruction in all SSA-based IRs to merge values from a source-variable that appears on two different control-flow paths. It is primarily introduced for the purpose of data-flow analysis, and can be thought of as an assignment statement with an OR between multiple variables on the right hand side. The `ImplDS` non-terminal specifies if the class implements any of the data-structure interfaces (such as Map, Collections, etc.) in Java.

Enumerating through all the instructions in the grammar shows that there are exactly five program constructs to which a resource can escape from its method: a field, a data-structure (an array or a data-structure class), a return variable, a parameter, and an invoke (i.e. method-call). We define five escape types corresponding to these five constructs: *Field-Escape, Data-Structure-Escape, Return-Escape, Parameter-Escape, Invoke-Escape*. Fig. 5.7 describes what each escape type means. A resource can have multiple escape types.

From an abstract-interpretation [SCS21] view of the analysis, the bottom element of the

Escape type	A resource object has this escape type if it:
Field-Escape	aliases with either an instance field or a static field
Data-Structure-Escape	aliases with an element in an array or data-structure.
Return-Escape	gets returned by the method it is created in.
Parameter-Escape	aliases with a formal parameter of the method it is created in
Invoke-Escape	is passed as an argument to an invoke statement (i.e. method call)

Figure 5.7: The five escape mechanisms for a resource object

lattice is the empty set and the top element is a set with all five escape types. The analysis is designed to be demand-driven, and its output is the set of possible escape types for the resource.

Fig. 5.8 gives pseudo-code for the analysis. The analysis procedure, whose name is shortened to `rea`, takes two arguments: the resource variable whose escape mechanisms we wish to analyze, and the method it is declared in. *escTypes*, the set of escape types, is initialized to the empty set in the beginning. The bulk of the method is a for loop over the uses of the resource variable.

For each use, we perform a case analysis based on the type of the use instruction. The possible use instructions come from the grammar in Fig. 5.6. If the use-instruction is an *ArrayStore*, it follows that the resource object aliases with an array element, and according to Fig. 5.7, this implies a *Data-Structure Escape*; we add this to *escTypes*. Similarly a *FieldWrite* implies a *Field Escape*. An *Assgn* or *PhiStmt* requires us to recursively track the assigned variable; hence we call `rea` on it. Being used in a *Return* statement in the warning's original method implies that *Return* should be in *escTypes*. Additionally, since we need to track the returned variable in all callers, we add the escape types of the callers' call-sites to *escTypes*. If the use-instruction is an *Invoke* (i.e. method call), we split it into two sub-cases. If the method belongs to a data-structure class, we add *Data-Structure* to *escTypes*. If not, we add *Invoke* to *escTypes*, and track the escape types in the method call by recursively calling `rea` on the matching argument in the invoke targets. We do not need to do anything

```
1: procedure REA(resourceVariable, method)
2:     escTypes = ∅
3:     for I in getUses(resourceVariable) do
4:         switch I.instructionType do
5:             case ArrayStore:
6:                 escTypes ∋ Data-Structure
7:             case FieldWrite:
8:                 escTypes ∋ Field
9:             case Assgn:
10:                escTypes ⊇ rea(Assgn.V_lhs, method)
11:            case PhiStmt:
12:                escTypes ⊇ rea(PhiStmt.V_lhs, method)
13:            case ReturnStmt:
14:                if method == originalWarningMethod then
15:                    escTypes ∋ Return
16:                end if
17:                for caller in callers(method) do
18:                    escTypes ⊇ rea(caller.V_lhs, caller.method)
19:                end for
20:            case Invoke:
21:                for M in invoke.targets do
22:                    if M.isDataStructureMethod() then
23:                        escTypes ∋ Data-Structure
24:                    else
25:                        escTypes ∋ Invoke
26:                        p =M.matchingParam(resourceVariable)
27:                        escTypes ⊇ rea(p, M)
28:                    end if
29:                end for
30:            case FieldRead: do nothing
31:            case ConditionalBranch: do nothing
32:            case NewStmt: do nothing
33:            case ArrayLoad: do nothing
34:    end for
35:    if isParameter(resourceVariable) then
36:        if method == originalWarningMethod then
37:            escTypes ∋ Parameter
38:        end if
39:        for caller in callers(method) do
40:            p = caller.matchingArgument(resourceVariable)
41:            escTypes ⊇ rea(p, caller.method)
42:        end for
43:    end if
44:    return escTypes
45: end procedure
```

Figure 5.8: Resource Escape Analysis (Name shortened to *rea*)

for the last four instruction types. A *FieldRead* does not propagate any escape information from the resource variable because only the field is read. *ConditionalBranch*, *NewStmt*, and *ArrayLoad* do not even support the use of a resource variable.

In addition to checking for the uses of the resource variable, we also need to check if it escapes to a parameter (see line 35). If so, we add *Parameter* to $escTypes$. Additionally, we recursively track the escape types in the caller methods, by calling `rea` on the resource variable's matching argument in the caller methods.

Finally, the analysis returns $escTypes$, the aggregate set of escape types for $resourceVariable$. Since resource aliases point to the same underlying resource, an escape type for one alias applies to all other aliases. Hence, the resource escape analysis must be repeated for all resource-aliases of $resourceVariable$, and their escape types added to $escTypes$.

5.3.4 Applying Repair Templates

The final step for *RLFixer* is generating repair code. The repair code has the following specification: it should close the leak after the last use of the resource, without introducing new errors (such as a new leak, a use-after-close error, or a null pointer exception) or modifying the semantics of the original program.

RLFixer uses the decision tree from Fig. 5.9 to pick the correct repair strategy. If a resource escapes to a field or data-structure, *RLFixer* marks it as infeasible to fix. If the resource does not escape to a field or data-structure, but does escape to a return or parameter, *RLFixer* creates dummy leak warnings at the caller methods, and closes the leak there. If it does not have any of these four escape types, we can close the resource in the same method as it was created. Based on how the resource is used, we then apply one of three repair templates. Since the decision tree covers all the five escape types, it exhaustively covers all the cases in which a single resource can leak. Let us now examine each of these cases in more detail.

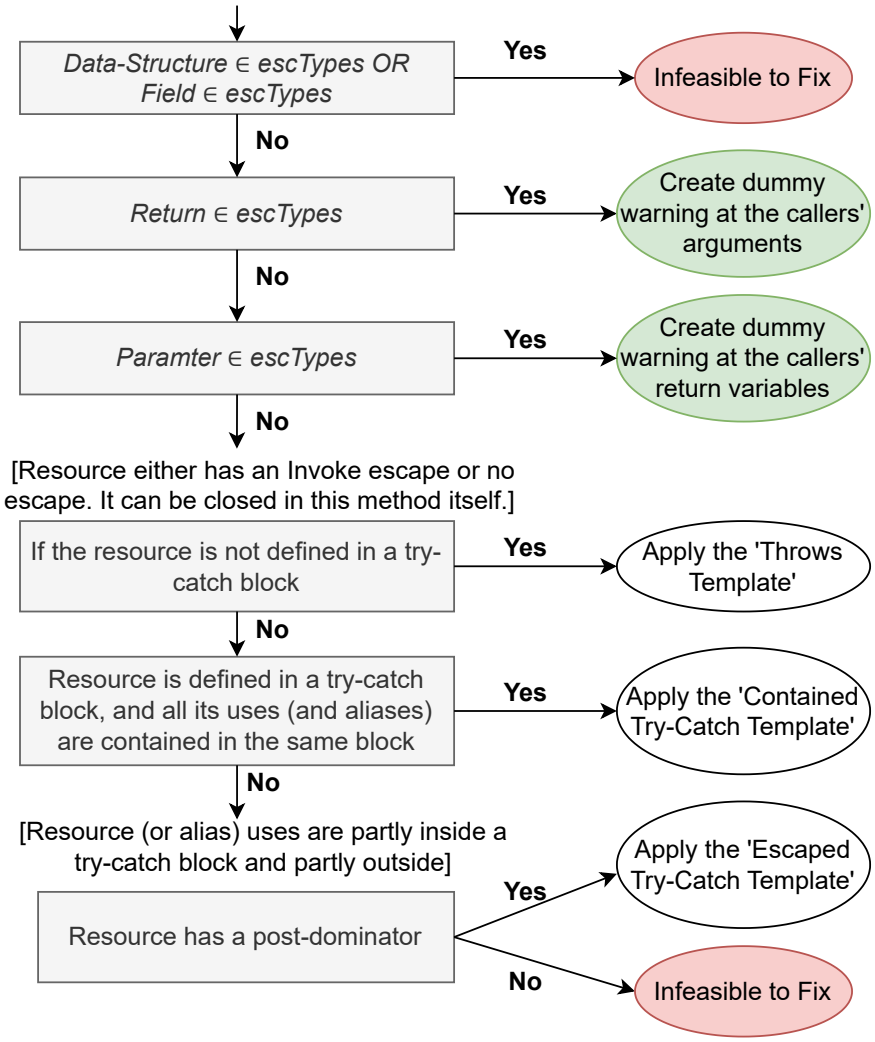


Figure 5.9: Decision-tree depicting how RLFixer decides which leaks are infeasible to fix, and picks the correct repair template to apply.

Field-Escape. Resources with a *Field Escape* are infeasible to fix. We have already seen why a resource that escapes to an instance field, like in the example from Fig. 5.4, is infeasible to fix. The problem reduces to compile-time object deallocation. A resource that escapes to a static field (i.e. a field associated with the class definition rather than any object) also cannot be closed, but for a different reason. Static fields are alive throughout the program, and hence their associated resources will not be safe to close before the end of the program. An additional roadblock to generating fixes for *Field-Escapes* is that we may not have access to all the code at compile time. For example, if *RLFixer* is given library code to fix, it does not have access to all the applications using this library code; such application code can also create library objects or access its fields.

Data-Structure-Escape. Fixing *Data-Structure-Escapes* is hard because it is well-known that unbounded data structures such as arrays are hard to accurately model using static analysis [GRS05b]. Hence, static analysis tools model arrays using *over-approximation.* An over-approximation captures the effects of all possible executions of the program, but it may additionally include effects from executions that are not possible. In the case of an array, the over-approximation is to assume that a read or write to the array could affect any possible index. Such an over-approximation is safe for resource-leak detection because it will never miss out on a leak that occurs in a possible execution. However, this over-approximation is unsafe for our repair problem because to close a resource from an array requires us to know the exact index that the resource was written to. A similar argument applies to other kinds of data-structures. Hence, *RLFixer* does not generate any repairs for these cases.

Return Escape. At the $Return \in escTypes$ node of the decision tree, we already know that the resource does not escape to a field or data-structure. If the resource does escape via a *Return*, we create one dummy warning for each caller at the returned variable. For example, in the snippet (`Resource r = getRes()`), a resource object gets returned by the `getRes` method and hence is still alive after the `getRes` method returns. Consequently, we

cannot close the resource inside the `getRes` method. Instead, we create dummy leak warning at the returned variable (`r` in this case). We then recursively apply *RLFixer* to the dummy warning(s), and suggest their repairs as a fix for the original warning.

Parameter Escape. The *Parameter Escape* case is similar to the *Return Escape* case, where we create a dummy warning at the caller methods. The only difference is that the dummy warning is created at the corresponding argument of the paramter in the caller. We already saw this strategy being applied to the example in Fig. 5.2. The resource leak was reported at the new Buffered Reader in the `bar` method of Fig. 5.2. The Buffered Reader resource escapes to the parameter via its resource alias (*FileReader f*). Hence, we create a dummy warning for the argument `fr` at the method call to `bar` in the caller method `foo`. We then recursively apply *RLFixer* on the dummy warning(s), and suggest their repairs as a fix for the original warning.

Invoke Escape and Non-escape. At the decision tree node where we have neither a *Data-Structure*, *Field*, *Return* or *Parameter* escape, we are left with resource-leaks that either have an *Invoke Escape* (i.e. escape via method call) or no escape types. In these two cases, the resource is not used after the method completes, and hence should closed in the same method. Based on whether the resource is defined inside or try-catch block or not, we define three repair templates: the *Throws Template*, the *Contained Try-Catch Template*, and the *Escaped Try-Catch Template*. Fig. 5.10 illustrates these templates. The lines highlighted in green give the fix suggested by *RLFixer*.

The *Throws Template* (see Fig. 5.10a) applies when the resource is not created or used within a *try-block*. The repaired code places all the resource (and alias) uses within a *try-finally* block. The *try* block starts at the first line where the resource is used. The *finally* block starts after the last line where the resource is used, but with adjustments to match the scope of the newly added try-block. Note that the new close statement is placed within

```
1  void foo() throws Exception{
2    Resource r = null;
3  + try {
4      r = new Resource(..);
5      r.useResource();
6  + } finally {
7  +     try{ r.close(); }
8  +     catch(Exception e) {
9  +       e.printStackTrace();
10 +     }
11 + }
12 }
```

(a) Throws Template

```
1    Resource r = null;
2    try {
3      r = new Resource(..);
4      r.useResource();
5    } catch (IOException e) {
6      e.printStackTrace();
7    }
8  + finally{
9  +   try{ r.close(); }
10 +   catch(Exception e) {
11 +     e.printStackTrace();
12 +   }
13 + }
14   /* No Resource use after this */
```

(b) Contained Try-catch Template

```
1  void bar() throws Exception{
2    Resource r = null;
3    try {
4      r = new Resource(..);
5      r.useResource();
6    } catch (Exception e) {}
7    if (b){
8  +   try{
9        r.useResource();
10 +   } catch (Exception e) {
11 +     try{ r.close(); }
12 +     catch (Exception f) {}
13 +     throw e;
14 +   }
15   }
16 + try{
17 +   r.close();
18 + } catch (Exception E) {
19 +   e.printStackTrace();
20 + }
21 }
```

(c) Escaped Try-Catch Template

Figure 5.10: *RLFixer's* Repair Templates

its own try-catch block to handle any exception (either related to resource access, or a null-pointer exception) it may throw. Without the enclosing try-catch block, an exception in the close statement will modify the control flow of the original program. We will see the same pattern with the next two templates. Similarly, we do not add a corresponding catch block to the outer try-finally block in the *Throws Template* because it will modify the control-flow of the program if an exception is thrown. Modifying the control flow of the original program modifies the semantics, and this goes against our repair specification.

The *Contained Try-Catch Template* (see Fig. 5.10b) is applied when the resource creation and all its uses (and resource aliases) are contained within a *try* block. In this case, the correct repair is to attach a corresponding *finally* block that closes the resource. The *finally* block in Java always executes after the *try-catch* block, even if the *try* block has a return statement or an exception. Hence, with this fix, the resource is closed on all program paths, including ones involving an exception. [2]

The *Escaped Try-Catch Template* (see Fig. 5.10c) applies when only some of the resource creation and use statements (and resource aliases) are inside a *try* block, whereas the rest are not. Here, we first place all statements that are outside a *try* block (like line 9 from Fig. 5.10c) in a new *try-catch* block; this prevents control from escaping the method *bar* before we get a chance to close the resource. Note that we re-throw the exception on line 13 to avoid modifying the semantics of the program when an exception is thrown. Without this re-throw, any exception handler in the caller method of *bar* will not execute, thereby modifying the program semantics in this case. Finally, the resource needs to be closed at its earliest *post-dominator*. A *post-dominator* for a resource is an instruction in the method that appears on every control-flow path from a resource use to the end of the method. In Fig. 5.10c, the earliest post-dominator for the resource is on line 16, which is where we close the resource. Choosing the earliest post-dominator for closing the resource is always

[2]The *Contained Try-Catch Template* has a corner case when a *finally* block is already present. In this case, we add the close statement to the existing finally block.

safe, but in corner cases with a method having multiple exit points, a resource may have no post-dominator; *RLFixer* does not suggest any fix in this corner case. [3] [4]

Loops and Existing Close Statements

There are two more details we need to handle in the repair code: loops and existing close statements.

Dealing with Loops. Our discussion so far has assumed that the resource was never created inside a loop, but we need to deal with this case. We divide the resource-leaks in loops into two sub-cases. The first, more common sub-case, occurs when the resource is created during a loop iteration and is never used after the end of that iteration. We deal with this sub-case by extracting the loop-body and computing the fix on this loop-body as we would for any loop-free resource leak. For the very rare sub-case where a resource created in a loop stays alive beyond the end of a loop iteration, *RLFixer* does not suggest a fix; this sub-case gives an undecidable problem. [5]

Deleting Existing Close Statements. In addition to adding repair code, we also need to delete unnecessary `close` statements added by the programmer, so that we can avoid a double close. For example, in Fig. 5.10b, if the programmer had inserted a close statement inside the `try` block, we would need to delete it. Such a close statement inside the `try` block does not prevent a resource-leak on the exception path, and once *RLFixer* adds the code highlighted in green, an additional close inside the try-block will lead to a double-close of the resource. We design each of our repair-templates to require a single close statement; hence,

[3] The *Escaped Try-Catch Template* has a corner case when the earliest post-dominator is inside a try-catch block. Here, the close statement goes inside a finally block.

[4] We avoid using Java's try-with-resources statement for the fix-templates because it only applies to resources that implement the java.lang.AutoCloseable interface. Furthermore, it supresses exceptions in the try-with statement in some cases, thereby modifying the semantics of the program. Also, it cannot handle the resource usage pattern from the Escaped Try-Catch Template.

[5] Fixing the leak in this rare sub-case requires us to identify the last loop iteration to close the leak. This is at least as hard as proving loop termination, a known hard problem for compilers.

RLFixer deletes any existing close statements that were added by the programmer.

5.4 Implementation

This section discusses the implementation details for *RLFixer*. *RLFixer* is primarily implemented in the WALA static analysis framework for Java bytecode [WAL15]. We choose to write *RLFixer's* analyses in the WALA IR instead of the Java source AST because the IR has simpler control flow, fewer instruction types, and is already in SSA form. Furthermore, WALA automatically sets up the core information needed by any static analysis, such as computing the class-hierarchy, call-graph (using the 0-CFA algorithm) and basic-blocks. The Repair-Template stage of *RLFixer* additionally uses JavaParser [Jav19] to scan the Java source ASTs for scoping and line number information. The front-end of the tool is implemented in Python because of Python's scripting capabilities, and its parsing libraries for XML and JSON, the most common output formats for resource-leak detectors.

The resource-escape analysis, call-graph, and resource-alias analysis all use a context-insensitive analysis. Context-sensitivity is not needed because we know of no way to represent context in repair code. Our analyses automatically get partial flow-sensitivity because of the WALA IR's SSA form. Field-sensitivity is redundant because all resources aliasing fields become *Field Escapes* and do not get fixed. Reflection support can trivially be added by turning on WALA's reflection analysis, but we skip this option; it only benefits a tiny fraction of repairs, while increasing call-graph computation time by many fold. In our experiments, out of the 150 resource-leaks that were manually examined, none were affected by reflection.

The output format of the tool is much like that in Fig. 5.2, and can easily be incorporated into an IDE or existing static analysis tool. Note that *RLFixer* does not automatically adjust variable scopes in its output; it is up to the programmer to correct this.

5.5 Dataset

We use the NJR-1 dataset (available here [UKL20]), as our benchmark-set. It has 293 Java-8 programs from GitHub, which run off-the-shelf with Java analysis tools that *RLFixer* uses, like WALA and JavaParser, as well as several existing resource-leak detectors. The programs also have several resource leaks. Furthermore, these benchmarks ship with pre-compiled dependencies and can be compiled with a simple javac command. We leave out 6 of the 293 benchmarks; Footpatch runs out of memory for three benchmarks, two are missing a library class, and one gets incorrect line numbers for WALA. This leaves us with 287 benchmarks for our experiments.

We run five popular Java resource-leak detectors on this dataset: Infer [CDD15], PMD [PMD02], Checker-Framework [KSS21b] (shortened to CF), Codeguru [Cod20], and Spotbugs [Spo17]. The warnings given by these tools is then fed to *RLFixer*. We ran all the tools with their default options, and after post-processing the warnings to filter out duplicate warnings, etc., we got a total of 2205 unique resource-leaks. During the duplicate filtering, if there are two warnings for a resource-alias pair (i.e. same root cause), one gets removed. Fig. 5.11 lists the number of warnings given by each tool. Checker-Framework gets the most resource-leaks, probably because of its commitment to soundness (i.e. catching as many possible leaks). Spotbugs gets the fewest resource-leaks, probably because soundness was traded off for speed. Similarly, the other tools differ in their set of reported leaks because of different design decisions.

Fig. 5.12 reports some statistics about the frequency of resource leaks in the NJR dataset. 207 out of 287 programs, or 72% of the benchmark programs, have at least one leak. By taking a union of the resource leaks by the five tools, we get an estimated total of 2205 unique resource leak warnings. This averages to 8 resource leaks per program. Given that the average number of lines of application code in the NJR dataset is almost 10k, we can expect one resource leak in every 1300 lines of code. Thus, resource leaks are prevalent in

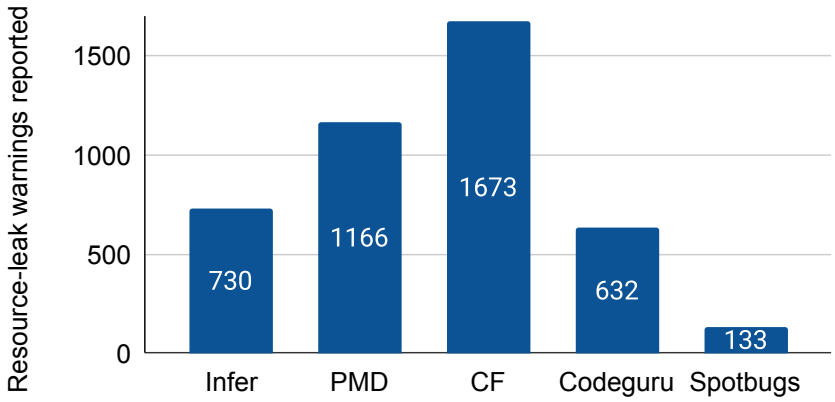


Figure 5.11: Warnings reported by the five resource-leak analyzers when applied to the NJR dataset.

Tool	Output
Total number of programs:	287
Programs with at least one resource leak:	207
Estimated number of unique resource leaks (across the five tools)	2205
Lines of application code per benchmark	9911

Figure 5.12: Statistics about the frequency of resource leaks in the NJR dataset

the dataset; developers need better tool-support for fixing these leaks.

5.6 Experimental Results

In this section, we discuss our experimental results which validate the following claims (all numbers are averages across the five tools).

1. RLFixer suggests fixes for 66% of the resource-leak warnings from the five tools.
2. 95% of the fixes suggested by RLFixer are correct.
3. RLfixer produces higher quality fixes than Footpatch.

4. RLFixer takes, on average, 1 seconds per program, excluding the 13-second WALA setup time.

The first four sub-sections below validate these four claims, and the fifth sub-section outlines the threats to validity. The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 GB RAM. For the JVM, the default heap size of 32GB, and default stack size of 1MB, was used.

5.6.1 Fixable Rate

Fixable Rate is the percentage of warnings for which a fix was suggested. It is defined as:

$$\text{Fixable Rate} = \frac{\#\text{ warnings for which a fix was suggested}}{\text{Total }\#\text{ warnings}}$$

Fig. 5.13 gives a split up of the fixable and unfixable resource-leaks for each of the five tools. On average, *RLFixer* gets a 66% fixable rate, with PMD getting the highest fixable-rate (75%). The unfixable resource-leaks are further split based on the reason they are not fixed: From the graph, we see that the main reason for unfixed leaks are *Field Escapes* (20%). Checker-Framework (shortened to CF), gets a lower fixable-rate than the other tools because of a large percentage of its leaks being *Field Escapes*. A smaller contributor to unfixed leaks are *Data-Structure Escapes* (9%). Some 1% of resource leaks escape to both, a data-structure and a field. We report these as data-structure escapes to simplify the graph. The last 5% of leaks are not fixed (in red color) because, as discussed in Section 5.3.4, there are corner cases for some templates that result in undecidable problems.

5.6.2 Fix Correctness

Another important metric is *Fix Correctness*, the percentage of correct fixes out of the suggested fixes. It is defined as:

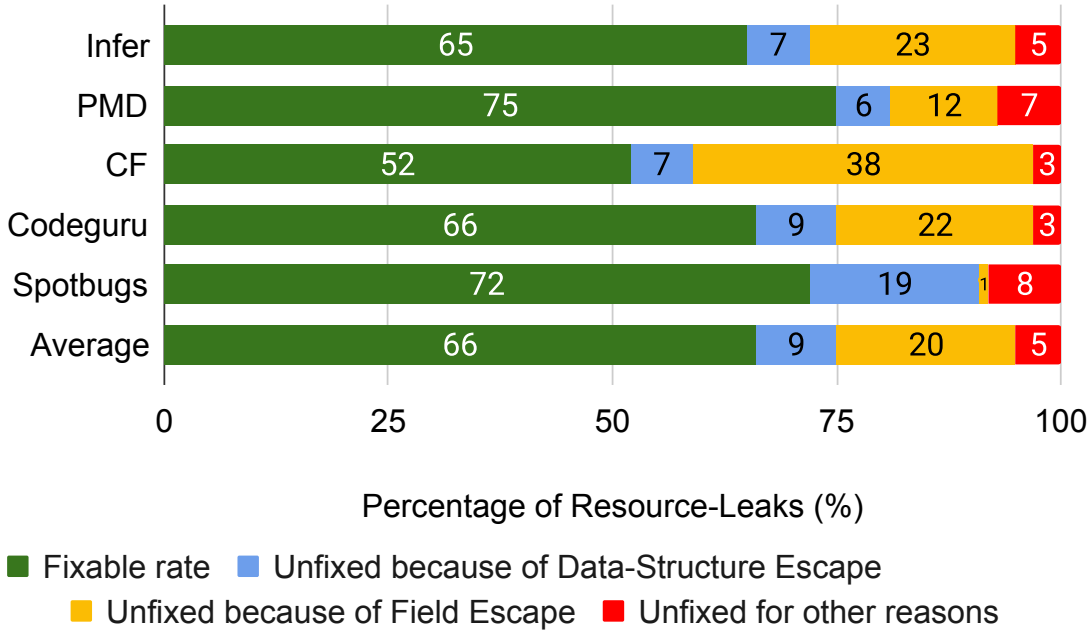


Figure 5.13: Fixable-rate for RLFixer for each resource-leak detector, along with reasons for the unfixed leaks.

$$\text{Fix Correctness} = \frac{\#\text{ warnings with a correct fix suggestion}}{\#\text{ warnings with a fix suggestion}}$$

We picked a sample of 150 fixes (30 per resource-leak detector) suggested by *RLFixer* to estimate the fix-correctness. We re-ran the resource-leak detector on the fixed code to ensure that the old leaks disappeared. For 2 fixes the old leaks remained, and these were marked as incorrect. For the remaining fixes, we had 5 volunteer programmers classify the fixes as correct or incorrect. The volunteers, none of whom are authors, are computer-science graduate students who are familiar with Java and resource leaks. The volunteers classify different subsets of the fixes, but each fix is classified by at least 3 volunteers. Each volunteer uses the following criteria to evaluate correctness, and a fix is considered incorrect even if one of these criteria is not met.

1. The fix repairs the leak.

2. The fix does not introduce a null-pointer error.

3. The fix does not introduce a use-after-close error (e.g. file written after being closed).

4. The fix does not introduce a double close.

5. The fix does not modify the behavior of the program.

Finally, we computed the fix-correctness by taking an average over the scores of the volunteers. The scores for each tool are shown in Fig 5.14. On average, *RLFixer's* fix-correctness is 95%, with Infer and Codeguru getting near perfect fixes. Given that less than one in twenty fixes by *RLFixer* are incorrect, we can put high confidence in its generated repairs.

Examining the small fraction of incorrect fixes shows that there are two major roadblocks to *RLFixer* reaching perfect fix-correctness. The first is that we use an under-approximation for the *resource-alias analysis*; this may lead to missed resource-aliases, which could in turn give incorrect fixes. We discuss in Section 5.3.2, why this is unlikely to ever be perfected. The second roadblock is that *RLFixer's* templates are designed to fix individual resource-leaks, and hence do not work perfectly when multiple resource-leaks occur in the same code block.

Another correctness issue that most repair-tools need to deal with is false-positive warnings, and whether one suggests repairs for these false warnings; this, however, does not seem to be an issue in practice for *RLFixer*. To get a measure of false-positive warnings, we asked the volunteers to also examine the same 150 repairs and decide whether the original leak-detector warning was a false positive. All five resource-leak detectors gave zero false-positive warnings for the leaks fixed by *RLFixer*. [6] This low false-positive rate is expected, since these are all mature tools that have been heavily engineered to weed out false-positive warnings.

[6]There could still be false-positive warnings among the infeasible-to-fix leaks.

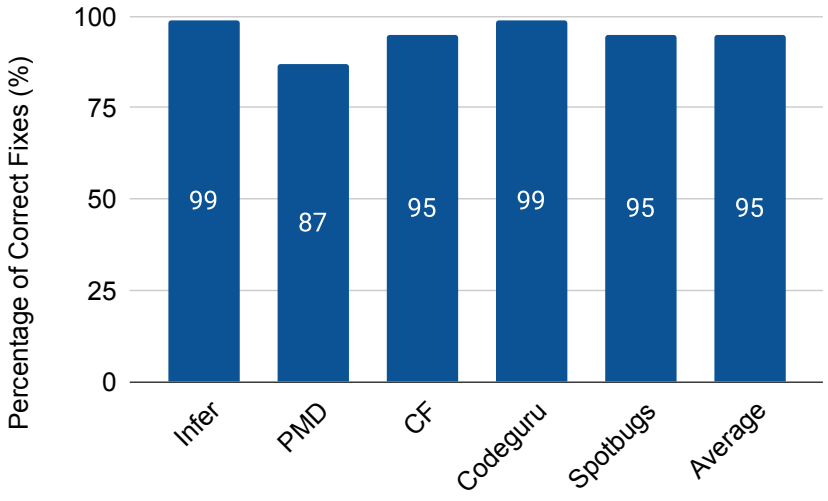


Figure 5.14: Percentage of correct fixes by *RLFixer* (i.e. fix-correctness) for the five resource-leak detectors

5.6.3 Comparison with Footpatch

Fig. 5.15 summarizes the comparison between Footpatch and *RLFixer*. We only use the Infer warnings for the comparison because Footpatch is tightly integrated with Infer; it cannot be used with other resource-leak detectors. We split the results into two parts: the first part (columns 2 and 3) shows the results on the warnings from the NJR benchmarks, and the second part (columns 4 and 5) gives the results on the *apktool* benchmark from the Footpatch paper [TG18].

For the NJR benchmarks, out of the 30 Infer warnings examined by users, Footpatch generated fixes for only 4 of them, out of which 2 were correct; this gives a fixable-rate of 15% and fix-correctness of 50%. *RLFixer*, on the other hand, gets a fixable-rate of 65% and fix-correctness of 99%.

For *apktool*, the only benchmark from the Footpatch paper [TG18] with Java resource leaks, Infer gives out 19 warnings. Out of these 6 are duplicates and we remove them. For the remaining 13 warnings, Footpatch attempts a fix for 1 warning (fixable-rate 8%), and RLFixer attempts a fix for 12 warnings (fixable-rate of 92%). Both tools produce only correct

Tool	NJR benchmarks		apktool	
	Fixable Rate	Fix Correctness	Fixable Rate	Fix Correctness
Footpatch	15%	50%	8%	100%
RLFixer	65%	99%	92%	100%

Figure 5.15: Comparing the repair quality of RLFixer and the Footpatch baseline when fixing the Infer warnings. We show the results separately for the NJR benchmarks and *apktool*.

Tool	Leak detector time (s)	Overall fix time	
		WALA setup time (s)	RLFixer time (s)
Infer	42	13	1
PMD	6	12	1
Spotbugs	13	13	1
CF	63	12	1
Average	31	13	1

Figure 5.16: Split up of the time taken per program by RLFixer and the resource-leak detectors

fixes for this benchmark (fix-correctness 100%).

The large gap in fix-quality between Footpatch and *RLFixer* is expected; Footpatch is a more general purpose tool that works with multiple kinds of errors, as well as on both C and Java. *RLFixer*, on the other hand, is specialized for resource-leaks in Java, and hence is able to vastly outperform Footpatch on this task.

5.6.4 Time overhead

Fig. 5.16 shows the time taken per program by *RLFixer* and four of the resource-leak detectors. We do not report the time for Codeguru because it is only accessible via a web service; this makes it hard to separate out the time taken for the network, web-service handler, etc. On average, resource-leak detection takes 31 seconds per program, but this varies widely across the four detectors. PMD and Spotbugs are very fast, whereas Infer and CF take much longer. The overall repair time is 14 seconds per program. A majority of this time

(13 seconds) is taken by the WALA setup, whereas *RLFixer*, as described in Section 5.3, takes just 1 second per program because of its demand-driven design. Its fix time per leak is even lesser. The WALA setup includes tasks like constructing the class-hierarchy, call-graph, basic-blocks, etc., but a majority of the time is taken for call-graph construction.

Call-graph construction is unavoidable for any inter-procedural analysis, but we could eliminate this component by integrating *RLFixer* with a WALA based resource-leak detector. Since *RLFixer* is implemented in the WALA framework, such a design allows *RLFixer* to reuse the WALA setup information, including the call-graph, computed during the resource-leak detector phase. This strategy will bring the total fix-time down to 1 second per program, but it will not work for tools built with other frameworks.

A direct comparison of repair times with Footpatch is not meaningful because Footpatch suggests very few fixes, and it is hard to factor out the time taken by unsuccessful fix attempts and fixes for other kinds of bugs. However, results from the Footpatch paper show that it takes several minutes per Java program, which is at least an order of magnitude larger than *RLFixer*. There are two reasons why *RLFixer* is quicker than Footpatch: it is demand-driven, and unlike other Footpatch, *RLFixer* does not need to perform a search over possible repair codes; it constructs fixes from repair templates.

5.6.5 Threats to Validity

The first threat to validity is that the human volunteers we used for the experiment in Section 5.6.2, could make mistakes in their evaluation of the fixes. We mitigate this threat by averaging scores over multiple volunteers and and a large number of leaks (150 in total) from different tools. Furthermore, we also re-ran the resource-leak detectors on the fixed code to confirm that the resource-leak warning disappeared.

The second threat is that our evaluation was carried out on Java-8 programs from the NJR-1 dataset. The assumption is that our results will generalize to other Java benchmarks.

The third threat is the applicability of RLFixer's approach to other langauges and platforms, since RLFixer's design and our experiments only focus on Java code.

5.7 Related Work

The research direction closest to this work is automated program repair, and one can split this category into general-purpose, special-purpose, and linter-based repair tools. Two other directions that seem deceptively close to our work because of their similar names but are not really close, are escape analysis and repairing Android resource leaks. We discuss each of these in turn.

General-purpose repair tools. General-purpose repair tools aim to fix a wide-variety of program errors. Most of these tools are test-based techniques, and can be split into three paradigms. The first paradigm, generate-and-validate [JXZ18, KNS13, LR15, LR16, WNL09], generates candidate patches by searching through existing patches and code. The second is the deep-learning based paradigm [LWN20, JLT21, XZ22, LWN22, YMM22, FTL22] that uses deep-learning to find patches, often by applying Neural Machine Translation models from NLP. The third semantics-based paradigm [LCL17a, LCL17b, MYR16, NQR13, AMS21] generates patches by casting the repair problem as a constraint satisfaction problem. Ultimately, all three paradigms validate each patch by checking if it passes the previously failing test case. These paradigms cannot be applied to resource-leaks because resource-leaks do not cause test-failures. Unlike these general-purpose tools, *RLFixer* is specialized to fix resource leaks.

Footpatch [TG18] is the only general-purpose tool which can be applied to resource-leak because it uses a slight modification to these test-based paradigms, where it relies on a static analysis tool instead of tests to verify the fix. Footpatch searches for patches in the same code-base and verifies its fixes using the Infer static analysis tool. It is semi-specialized to

heap errors including null-pointer errors, resource leaks, etc. The limitation of Footpatch's approach is that such static analysis verification does not ensure that the semantics of the rest of the program are unmodified and no new errors are introduced. Nevertheless, we performed a quantitative comparison of the fixable-rate and fix-correctness with Footpatch [TG18]. As seen in Section 5.6.3, Footpatch generates low quality fixes for resource leaks; it pays for its generality with a lower fixable-rate and lower fix-correctness.

First and foremost, *RLFixer* has an order-of-magnitude better fixable-rate and fix-correctness, as seen in Section 5.6.3. Second, *RLFixer's* repair templates, by design, do not modify the semantics of the program or introduce any new errors. Thirdly, *RLFixer* is much faster. it takes seconds instead of minutes or hours.

Special-purpose repair tools. Special-purpose repair tools, as opposed to general-purpose tools, focus on repairing a single kind of error; this enables them to produce much higher quality fixes. Most of these tools report fixable-rates of 40-70% and a high fix-correctness, which is very similar to what we see with *RLFixer*, and this is usually significantly higher than what general-purpose tools can achieve. The kind of errors tackled by special-purpose tools include null-pointer errors [LHO22, XSY19], integer overflows [CZS17, MMS21], buffer overflows [SDH14], concurrency errors [AMK17, JSZ11, LCL16], performance bugs [SP15], and memory leaks [HLL20, GXM15, LHO18].

Among the existing special-purpose repair tools, memory-leak fixing [HLL20, GXM15, LHO18] is the closest to resource-leak fixing because it has a similar specification: repair the leak without modifying the semantics of the program. However, these tools focus on C programs, and memory-leaks present different challenges than resource-leaks. For example, features such as Java's exception mechanism, its reliance on try-catch blocks for resource management, resource aliasing, and the presence of class fields are a few of the challenges in Java resource-leak fixing that do not appear when dealing with memory leaks in C.

Unlike all these specialized repair tools, *RLFixer* focuses on resource-leaks, a problem

that has not been tackled by any specialized tool before. Additionally, *RLFixer's* demand-driven design makes it significantly faster than most special-purpose tools. Most other tools use time-budgets of a few minutes or more per program, whereas *RLFixer* finishes in 1 second, excluding the 13 seconds it takes for WALA to setup the call-graph, etc. At this speed, *RLFixer* is one of the few repair tools that are fast-enough to match the timing requirements of IDEs.

Linter-based repair tools. Linters scan code for style or coding-convention violations using pattern matching on the AST (Abstract Syntax Tree). Linter-based repair tools [MFB20, EHL22, LMM22, BYP19] use a similar AST pattern-matching approach to apply repairs for a linter's warnings. These tools vary in the coding-conventions they target, and in their method of learning repair patterns. Phoenix [BYP19] mines patches from a corpus of Github programs, and uses it to learn generalized repair strategies that are represented as executable programs in a domain specific language (DSL). Styler [LMM22] similarly learns fix patterns for code-formatting violations from a corpus, but it learns using an LSTM neural network. Getafix [BSP19] applies a hierarchical clustering algorithm to effectively summarize fix patterns, and then uses a novel ranking technique based on past human fixes to pick the most plausible fix. TFix [BHR21] formulates the linter-repair problem as a text-to-text prediction task and then uses a pre-trained text-to-text Transformer model to generate fixes. SpongeBugs [MFB20] and Sorald [EHL22] create manually defined fix templates for a handful of linter violations.

The errors targetted by linter-based repair tools are often local and can be represented using AST patterns. Hence, unlike *RLFixer*, their techniques will not work for a more complex bug such as resource-leaks which requires data-flow tracking and an inter-procedural analysis.

Escape analysis. Escape analysis [CGS99, WR99] is a research direction which sounds similar to our *resource escape analysis* from Section 5.3.3, but these two have very different designs and goals. Escape analysis characterizes how objects allocated in one region of the program escape to code outside this region. It cares less about the kind of program construct (such as an array or field) it escapes to. On the other hand, our *resource escape analysis* tries to establish the kinds of program constructs (such as a field or parameter) that a resource aliases with, and has no concept of regions. Hence, the two analyses end up having different abstract domains, constraints, and design decisions.

Repairing Android Resource Leaks. Another related direction is repairing *Android Resource Leaks*. *Android Resource Leaks* are leaks involving event-driven control flow from Android events, and are different from the Java resource leaks discussed in this paper, which involve sequential control flow. Let us take a closer look at how these two kinds of leaks are different to understand why they need different kinds of tools.

An Android application is an event-driven system with event-handlers responding to a sequence of events such as user-interaction or the application life-cycle events. For example, Android defines the event handlers `onPause` and `onDestroy` for when the user pauses and closes an application, respectively. *Android Resource Leak* detectors [WLX16, LXC16] model these event sequences and find ones that can leak some Android resource. For example, if a resource is not closed in the `onPause` or `onDestroy` event handlers, we may get an *Android Resource Leak*. Liu et. al [LWW19] prepare a database of such *Android Resource Leaks*. *Android Resource Leak* repair tools such as [BCB18, BF22, LWY16] then suggest the correct event-handler to close the resource in. Hence, all these leak detection and repair tools for *Android Resource Leaks* focus exclusively on Android's event-driven control flow.

On the other hand, tools such as *RLFixer* and Footpatch [TG18] focus on Java resource leaks resulting from the control-flow in sequential Java code. Thus, they solve a completely different problem than *Android Resource Leak* repair tools, and their design is consequently

different.

CHAPTER 6

Conclusion

Most static analysis tools have prioritized soundness (or recall), and consequently their designs has sacrificed some precision (i.e. true-positive rate), analysis time or repairability. However, several user studies have shown that the average user cares most deeply about exactly these three criteria: precision, analysis time and repairability. In fact, they have very high expectations on these three criteria, and since most static analysis tools are optional to use in practice, tools that don't meet these user expectations stop getting used. Soundness, on the other hand, is a low-priority criteria for the user.

To enable existing soundness-focused static analysis tools to meet user-expectations, we designed three new tools, CGPruner, QueryMax, and RLFixer to improve the three criteria that users care about the most. These tools act as pre-processors or post-processors to an existing static analysis, and trade off a small amount of soundness, for fewer false positives, faster analysis time, and automated repair suggestions. We conducted experiments that showed how our three tools can get several existing static analysis tools much closer to meeting user expectations.

An interesting future direction is to combine all three new tools in a single static analysis pipeline. This idea has the potential to improve a soundness-focused static analysis on all three user criteria simultaneously. The main challenge, however, is that all our three tools trade off a little recall; the cumulative loss of recall may be unacceptable. Hence, for a combined pipeline to be effective, it would have to jointly manage the overall recall traded off across all three tools.

REFERENCES

[AB14] Steven Arzt and Eric Bodden. "Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes." In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, p. 288–298, New York, NY, USA, 2014. Association for Computing Machinery.

[AKG07] Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. "Combined Static and Dynamic Mutability Analysis." In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, p. 104–113, New York, NY, USA, 2007. Association for Computing Machinery.

[AL12] Karim Ali and Ondřej Lhoták. "Application-Only Call Graph Construction." In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pp. 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[AL13] Karim Ali and Ondřej Lhoták. "Averroes: Whole-Program Analysis without the Whole Program." In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, p. 378–400, Berlin, Heidelberg, 2013. Springer-Verlag.

[ALS06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[AMK17] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. "Repairing Event Race Errors by Controlling Nondeterminism." In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, p. 289–299. IEEE Press, 2017.

[AMS21] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. "SOSRepair: Expressive Semantic Search for Real-World Program Repair." *IEEE Transactions on Software Engineering*, **47**(10):2162–2181, 2021.

[BB12] James Bergstra and Yoshua Bengio. "Random Search for Hyper-parameter Optimization." *J. Mach. Learn. Res.*, **13**(1):281–305, February 2012.

[BBC10a] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." *Commun. ACM*, **53**(2):66–75, February 2010.

[BBC10b] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." *Commun. ACM*, **53**(2):66–75, February 2010.

[BCB18] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. "EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps." *IEEE Transactions on Software Engineering*, **44**(5):470–490, 2018.

[BCC02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*, p. 85–108. Springer-Verlag, Berlin, Heidelberg, 2002.

[BCS13] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. "Thresher: Precise Refutations for Heap Reachability." In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, p. 275–286, New York, NY, USA, 2013. Association for Computing Machinery.

[BCS15] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. "Selective Control-Flow Abstraction via Jumping." In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, p. 163–182, New York, NY, USA, 2015. Association for Computing Machinery.

[BCS19] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. "NullAway: Practical Type-Based Null Safety for Java." In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, p. 740–750, New York, NY, USA, 2019. Association for Computing Machinery.

[BF22] Bhargav Nagaraja Bhatt and Carlo A. Furia. "Automated Repair of Resource Leaks in Android Applications." *J. Syst. Softw.*, **192**(C), oct 2022.

[BGO18] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. "RacerD: Compositional Static Race Detection." **2**(OOPSLA), oct 2018.

[BHR21] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. "TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer." In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 780–791. PMLR, 18–24 Jul 2021.

[BKL20a] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges, Eric Bodden, and Andreas Zeller. "Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 1061–1072, New York, NY, USA, 2020. Association for Computing Machinery.

[BKL20b] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges, Eric Bodden, and Andreas Zeller. "Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 1061–1072, New York, NY, USA, 2020. Association for Computing Machinery.

[Bre96] Leo Breiman. "Bagging predictors." *Machine Learning*, **24**(2):123–140, Aug 1996.

[BS09] 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*, OOPSLA '09, pp. 243–262, New York, NY, USA, 2009. ACM.

[BSP19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. "Getafix: Learning to Fix Bugs Automatically." *Proc. ACM Program. Lang.*, **3**(OOPSLA), oct 2019.

[BSS11] 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*, ICSE '11, p. 241–250, New York, NY, USA, 2011. Association for Computing Machinery.

[BW09] Raymond P. L. Buse and Westley Weimer. "The Road Not Taken: Estimating Path Execution Frequency Statically." In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, p. 144–154, USA, 2009. IEEE Computer Society.

[BYP19] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. "Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations." In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, p. 613–624, New York, NY, USA, 2019. Association for Computing Machinery.

[CB16a] Maria Christakis and Christian Bird. "What Developers Want and Need from Program Analysis: An Empirical Study." In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, p. 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[CB16b] Maria Christakis and Christian Bird. "What Developers Want and Need from Program Analysis: An Empirical Study." In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, p. 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[CB16c] Maria Christakis and Christian Bird. "What Developers Want and Need from Program Analysis: An Empirical Study." In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, p. 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[CC02] Patrick Cousot and Radhia Cousot. "Modular Static Program Analysis." In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, p. 159–178, Berlin, Heidelberg, 2002. Springer-Verlag.

[CDD15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification." In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pp. 3–11, Cham, 2015. Springer International Publishing.

[Cen05] NASA Ames Research Center. "Java PathFinder." `https://github.com/javapathfinder/jpf-core`, 2005.

[CGS99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. "Escape Analysis for Java." In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, p. 1–19, New York, NY, USA, 1999. Association for Computing Machinery.

[CHR21] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. "Boosting Static Analysis Accuracy with Instrumented Test Executions." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, p. 1154–1165, New York, NY, USA, 2021. Association for Computing Machinery.

[CJO18] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. "A scalable learning algorithm for data-driven program analysis." *Information and Software Technology*, **104**:1–13, 2018.

[Cod20] "Amazon Codeguru Reviewer." `https://aws.amazon.com/codeguru/`, 2020.

[COZ21] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. "Fast and Precise On-the-Fly Patch Validation for All." In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1123–1134, 2021.

[CR07] Sigmund Cherem and Radu Rugina. "Uniqueness Inference for Compile-Time Object Deallocation." In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, p. 117–128, New York, NY, USA, 2007. Association for Computing Machinery.

[CWE22] "Common Weakness Enumeration (CWE-400)." `https://cwe.mitre.org/data/definitions/400.html`, 2022.

[CZS17] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. "IntPTI: Automatic Integer Error Repair with Proper-Type Inference." In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, p. 996–1001. IEEE Press, 2017.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, p. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DD12] Arnab De and Deepak D'Souza. "Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates." In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, p. 665–687, Berlin, Heidelberg, 2012. Springer-Verlag.

[DDA08] Isil Dillig, Thomas Dillig, and Alex Aiken. "Sound, Complete and Scalable Path-Sensitive Analysis." In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, p. 270–280, New York, NY, USA, 2008. Association for Computing Machinery.

[DFL19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. "Scaling Static Analyses at Facebook." *Commun. ACM*, **62**(8):62–70, jul 2019.

[DWA22] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. "Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations." *IEEE Transactions on Software Engineering*, **48**(3):835–847, 2022.

[EHL22] Khashayar Etemadi Someoliayi, Nicolas Yves Maurice Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations." *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2022.

[FSS18] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. "Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification

Models." In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, SQUADE '18, p. 13–20, New York, NY, USA, 2018. Association for Computing Machinery.

[FTL22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. "VulRepair: A T5-Based Automated Software Vulnerability Repair." In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, p. 935–947, New York, NY, USA, 2022. Association for Computing Machinery.

[FWS19] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. "Smoke: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code." In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 72–82. IEEE Press, 2019.

[FYD08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. "Effective Typestate Verification in the Presence of Aliasing." *ACM Trans. Softw. Eng. Methodol.*, **17**(2), may 2008.

[GCS20] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. "Memory and Resource Leak Defects and Their Repairs in Java Projects." *Empirical Softw. Engg.*, **25**(1):678–718, jan 2020.

[GFF18a] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. "Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots." In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, p. 198–208, New York, NY, USA, 2018. Association for Computing Machinery.

[GFF18b] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. "Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots." In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, p. 198–208, New York, NY, USA, 2018. Association for Computing Machinery.

[GMF06] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. "Free-Me: A Static Analysis for Automatic Individual Object Reclamation." In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, p. 364–375, New York, NY, USA, 2006. Association for Computing Machinery.

[GRS05a] Denis Gopan, Thomas Reps, and Mooly Sagiv. "A Framework for Numeric Analysis of Array Operations." In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, p. 338–350, New York, NY, USA, 2005. Association for Computing Machinery.

[GRS05b] Denis Gopan, Thomas Reps, and Mooly Sagiv. "A Framework for Numeric Analysis of Array Operations." In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, p. 338–350, New York, NY, USA, 2005. Association for Computing Machinery.

[GXM15] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. "Safe Memory-Leak Fixing for C Programs." In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, p. 459–470. IEEE Press, 2015.

[HJP08] Laurent Hubert, Thomas Jensen, and David Pichardie. "Semantic Foundations and Inference of Non-null Annotations." In Gilles Barthe and Frank S. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, pp. 132–149, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[HLL20] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. "SAVER: Scalable, Precise, and Safe Memory-Error Repair." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 271–283, New York, NY, USA, 2020. Association for Computing Machinery.

[Ho95] Tin Kam Ho. "Random Decision Forests." In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, p. 278, USA, 1995. IEEE Computer Society.

[HP04] David Hovemeyer and William Pugh. "Finding Bugs is Easy." *SIGPLAN Not.*, **39**(12):92–106, December 2004.

[HT01] Nevin Heintze and Olivier Tardieu. "Demand-Driven Pointer Analysis." In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, p. 24–34, New York, NY, USA, 2001. Association for Computing Machinery.

[HW09] Sarah Heckman and Laurie Williams. "A Model Building Process for Identifying Actionable Static Analysis Alerts." In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, p. 161–170, USA, 2009. IEEE Computer Society.

[IKY00] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler." In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, p. 294–310, New York, NY, USA, 2000. Association for Computing Machinery.

[Jav19] "JavaParser." `https://javaparser.org`, 2019.

[JJC17] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. "Data-Driven Context-Sensitivity for Points-to Analysis." *Proc. ACM Program. Lang.*, **1**(OOPSLA), oct 2017.

[JKS05] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. "Taming False Alarms from a Domain-Unaware c Analyzer by a Bayesian Statistical Post Analysis." In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, p. 203–217, Berlin, Heidelberg, 2005. Springer-Verlag.

[JLO20] Minseok Jeon, Myungho Lee, and Hakjoo Oh. "Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features." *Proc. ACM Program. Lang.*, **4**(OOPSLA), November 2020.

[JLT21] Nan Jiang, Thibaud Lutellier, and Lin Tan. "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair." In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, p. 1161–1173. IEEE Press, 2021.

[JSM13a] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 672–681. IEEE Press, 2013.

[JSM13b] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 672–681. IEEE Press, 2013.

[JSM13c] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 672–681. IEEE Press, 2013.

[JSZ11] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. "Automated Atomicity-Violation Fixing." In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, p. 389–400, New York, NY, USA, 2011. Association for Computing Machinery.

[JXZ18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. "Shaping Program Repair Space with Existing Patches and Similar Code." In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, p. 298–309, New York, NY, USA, 2018. Association for Computing Machinery.

[KE03] Ted Kremenek and Dawson Engler. "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations." In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, p. 295–315, Berlin, Heidelberg, 2003. Springer-Verlag.

[KNS13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. "Automatic Patch Generation Learned from Human-Written Patches." In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 802–811. IEEE Press, 2013.

[Kot07] S. B. Kotsiantis. "Supervised Machine Learning: A Review of Classification Techniques." In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering*, p. 3–24, NLD, 2007. IOS Press.

[KP18] Christian Gram Kalhauge and Jens Palsberg. "Sound Deadlock Prediction." *Proc. ACM Program. Lang.*, **2**(OOPSLA), October 2018.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing." In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, p. 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[KRS13] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. "The Empirical Commit Frequency Distribution of Open Source Projects." In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13, New York, NY, USA, 2013. Association for Computing Machinery.

[KSS21a] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. "Lightweight and Modular Resource Leak Verification." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, p. 181–192, New York, NY, USA, 2021. Association for Computing Machinery.

[KSS21b] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. "Lightweight and Modular Resource Leak Verification." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, p. 181–192, New York, NY, USA, 2021. Association for Computing Machinery.

[LCL16] Haopeng Liu, Yuxi Chen, and Shan Lu. "Understanding and Generating High Quality Patches for Concurrency Bugs." In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, p. 715–726, New York, NY, USA, 2016. Association for Computing Machinery.

[LCL17a] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. "JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder." In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, p. 376–379, New York, NY, USA, 2017. Association for Computing Machinery.

[LCL17b] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. "S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples." In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, p. 593–604, New York, NY, USA, 2017. Association for Computing Machinery.

[LH03a] Ondřej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using SPARK." CC'03, p. 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[LH03b] Ondřej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using SPARK." In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, p. 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[LH03c] Ondřej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using SPARK." In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, p. 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[LH22] Bozhen Liu and Jeff Huang. "SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java." *Proc. ACM Program. Lang.*, **6**(OOPSLA1), apr 2022.

[Lho07] Ondrej Lhoták. "Comparing Call Graphs." In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, p. 37–42, New York, NY, USA, 2007. Association for Computing Machinery.

[LHO18] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. "MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C." In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, p. 95–106, New York, NY, USA, 2018. Association for Computing Machinery.

[LHO22] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. "NPEX: Repairing Java Null Pointer Exceptions without Tests." In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 1532–1544, New York, NY, USA, 2022. Association for Computing Machinery.

[Lin23] Linux. "Linux Github Repository." `https://github.com/torvalds/linux`, 2023.

[LL05] V. Benjamin Livshits and Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, p. 18, USA, 2005. USENIX Association.

[LLK17] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. "Sound Non-Statistical Clustering of Static Analysis Alarms." *ACM Trans. Program. Lang. Syst.*, **39**(4), aug 2017.

[LMM22] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. "Styler: Learning Formatting Conventions to Repair Checkstyle Violations." *Empirical Softw. Engg.*, **27**(6), nov 2022.

[LR15] Fan Long and Martin Rinard. "Staged Program Repair with Condition Synthesis." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 166–178, New York, NY, USA, 2015. Association for Computing Machinery.

[LR16] Fan Long and Martin Rinard. "Automatic Patch Generation by Learning Correct Code." *SIGPLAN Not.*, **51**(1):298–312, jan 2016.

[LS10] Wei Le and Mary Lou Soffa. "Path-Based Fault Correlations." In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, p. 307–316, New York, NY, USA, 2010. Association for Computing Machinery.

[LSS15a] 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." *Commun. ACM*, **58**(2):44–46, January 2015.

[LSS15b] 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." *Commun. ACM*, **58**(2):44–46, jan 2015.

[LTM18a] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. "Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity." In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, p. 129–140, New York, NY, USA, 2018. Association for Computing Machinery.

[LTM18b] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. "Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity." In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, p. 129–140, New York, NY, USA, 2018. Association for Computing Machinery.

[LTM20] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. "A Principled Approach to Selective Context Sensitivity for Pointer Analysis." *ACM Trans. Program. Lang. Syst.*, **42**(2), may 2020.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. "Reflection Analysis for Java." In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, p. 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[LWN20] Yi Li, Shaohua Wang, and Tien N. Nguyen. "DLFix: Context-Based Code Transformation Learning for Automated Program Repair." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 602–614, New York, NY, USA, 2020. Association for Computing Machinery.

[LWN22] Yi Li, Shaohua Wang, and Tien N. Nguyen. "DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair." In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 511–523, New York, NY, USA, 2022. Association for Computing Machinery.

[LWR20] Yi Lu, Daniel Wainwright, and Michael Reif. "Probabilistic call-graph construction." US Patent No. 10,719,314 B2, Jul 2020.

[LWW19] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. "DroidLeaks: a comprehensive database of resource leaks in Android apps." *Empirical Software Engineering*, **24**:1–49, 12 2019.

[LWY16] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. "Fixing Resource Leaks in Android Apps with Light-Weight Static Analysis and Low-Overhead Instrumentation." In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 342–352, 2016.

[LXC16] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. "Understanding and Detecting Wake Lock Misuses for Android Applications." In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, p. 396–409, New York, NY, USA, 2016. Association for Computing Machinery.

[MFB20] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings." *Journal of Systems and Software*, **168**:110671, 2020.

[MMS21] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. "IntRepair: Informed Repairing of Integer Overflows." *IEEE Transactions on Software Engineering*, **47**(10):2225–2241, 2021.

[MMZ01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver." In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, p. 530–535, New York, NY, USA, 2001. Association for Computing Machinery.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. "Parameterized Object Sensitivity for Points-to Analysis for Java." *ACM Trans. Softw. Eng. Methodol.*, **14**(1):1–41, jan 2005.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis." In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, p. 691–701, New York, NY, USA, 2016. Association for Computing Machinery.

[MZN15a] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. "A User-Guided Approach to Program Analysis." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 462–473, New York, NY, USA, 2015. Association for Computing Machinery.

[MZN15b] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. "A User-Guided Approach to Program Analysis." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 462–473, New York, NY, USA, 2015. Association for Computing Machinery.

[NQR13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. "SemFix: Program repair via semantic analysis." In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, 2013.

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, p. 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[PL16] Rachel Potvin and Josh Levenberg. "Why Google Stores Billions of Lines of Code in a Single Repository." *Commun. ACM*, **59**(7):78–87, jun 2016.

[PL18] Jens Palsberg and Cristina V. Lopes. "NJR: A Normalized Java Resource." In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, p. 100–106, New York, NY, USA, 2018. Association for Computing Machinery.

[PLR94] H.D. Pande, W.A. Landi, and B.G. Ryder. "Interprocedural def-use associations for C systems with single level pointers." *IEEE Transactions on Software Engineering*, **20**(5):385–403, 1994.

[PLS19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Semantic Fuzzing with Zest." In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, p. 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[PMD02] "PMD Source Code Analyzer." `https://pmd.github.io`, 2002.

[PR10] Corina S. Păsăreanu and Neha Rungta. "Symbolic PathFinder: Symbolic Execution of Java Bytecode." In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, p. 179–180, New York, NY, USA, 2010. Association for Computing Machinery.

[PVG11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python ." *Journal of Machine Learning Research*, **12**:2825–2830, 2011.

[Ric53] H. Gordon Rice. "Classes of recursively enumerable sets and their decision problems." *Transactions of the American Mathematical Society*, **74**:358–366, 1953.

[RKG04] Atanas Rountev, Scott Kagan, and Michael Gibas. "Static and Dynamic Analysis of Call Chains in Java." In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, p. 1–11, New York, NY, USA, 2004. Association for Computing Machinery.

[RKH18a] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. "User-Guided Program Reasoning Using Bayesian Inference." In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, p. 722–735, New York, NY, USA, 2018. Association for Computing Machinery.

[RKH18b] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. "User-Guided Program Reasoning Using Bayesian Inference." In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, p. 722–735, New York, NY, USA, 2018. Association for Computing Machinery.

[RKM06] Atanas Rountev, Scott Kagan, and Thomas Marlowe. "Interprocedural Dataflow Analysis in the Presence of Large Libraries." In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, p. 2–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[RPM08] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. "Predicting Accurate and Actionable Static Analysis Warnings:

An Experimental Approach." In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, p. 341–350, New York, NY, USA, 2008. Association for Computing Machinery.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. "Data-Flow Analysis of Program Fragments." In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, p. 235–252, Berlin, Heidelberg, 1999. Springer-Verlag.

[RSX08] Atanas Rountev, Mariana Sharp, and Guoqing Xu. "IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries." In Laurie J. Hendren, editor, *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4959 of *Lecture Notes in Computer Science*, pp. 53–68. Springer, 2008.

[SAE18a] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from Building Static Analysis Tools at Google." *Commun. ACM*, **61**(4):58–66, mar 2018.

[SAE18b] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from Building Static Analysis Tools at Google." *Commun. ACM*, **61**(4):58–66, March 2018.

[SB06a] Manu Sridharan and Rastislav Bodík. "Refinement-Based Context-Sensitive Points-to Analysis for Java." In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, p. 387–400, New York, NY, USA, 2006. Association for Computing Machinery.

[SB06b] Manu Sridharan and Rastislav Bodík. "Refinement-Based Context-Sensitive Points-to Analysis for Java." In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, p. 387–400, New York, NY, USA, 2006. Association for Computing Machinery.

[Sci] Scikit-learn. "Feature importances with a forest of trees." `https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html`.

[SCS21] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. "Demanded Abstract Interpretation." In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, p. 282–295, New York, NY, USA, 2021. Association for Computing Machinery.

[SDA16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java." In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Keywords: Demand-Driven; Static Analysis; IFDS; Aliasing; Points-to Analysis.]

[SDH14] Alex Shaw, Dusten Doggett, and Munawar Hafiz. "Automatically Fixing C Buffer Overflows Using Program Transformations." In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, p. 124–135, USA, 2014. IEEE Computer Society.

[SDT20] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. "On the Recall of Static Call Graph Construction in Practice." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 1049–1060, New York, NY, USA, 2020. Association for Computing Machinery.

[SGJ15] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. "Tricorder: Building a Program Analysis Ecosystem." In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, p. 598–608. IEEE Press, 2015.

[SGS05a] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. "Demand-Driven Points-to Analysis for Java." In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, p. 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[SGS05b] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. "Demand-Driven Points-to Analysis for Java." In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, p. 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[Shi91] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages of Taming Lambda*. PhD thesis, USA, 1991. UMI Order No. GAX91-26964.

[SHR00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. "Practical Virtual Method Call Resolution for Java." In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, p. 264–280, New York, NY, USA, 2000. Association for Computing Machinery.

[SP15] Marija Selakovic and Michael Pradel. "Poster: Automatically Fixing Real-World JavaScript Performance Bugs." In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 811–812, 2015.

[Spo11] Fausto Spoto. "Precise Null-Pointer Analysis." *Softw. Syst. Model.*, **10**(2):219–252, may 2011.

[Spo17] "SpotBugs Static Analysis Tool." `https://spotbugs.github.io`, 2017.

[SR11] J. Sawin and A. Rountev. "Assumption Hierarchy for a CHA Call Graph Construction Algorithm." In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pp. 35–44, 2011.

[SSC18] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. "Modern Code Review: A Case Study at Google." In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, p. 181–190, New York, NY, USA, 2018. Association for Computing Machinery.

[SWF20] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. "Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 812–823, New York, NY, USA, 2020. Association for Computing Machinery.

[Syn17] Synopsys. "2017 Coverity Scan Report." `https://www.synopsys.com/blogs/software-security/2017-coverity-scan-report-open-source-security/`, 2017.

[SZ20] Qingkai Shi and Charles Zhang. "Pipelining Bottom-up Data Flow Analysis." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 835–847, New York, NY, USA, 2020. Association for Computing Machinery.

[TC10] Emina Torlak and Satish Chandra. "Effective Interprocedural Resource Leak Detection." In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, p. 535–544, New York, NY, USA, 2010. Association for Computing Machinery.

[TG18] Rijnard van Tonder and Claire Le Goues. "Static Automated Program Repair for Heap Properties." In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, p. 151–162, New York, NY, USA, 2018. Association for Computing Machinery.

[TGP14] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. "ALETHEIA: Improving the Usability of Static Security Analysis." In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, p. 762–774, New York, NY, USA, 2014. Association for Computing Machinery.

[TPF09a] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. "TAJ: Effective Taint Analysis of Web Applications." In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, p. 87–97, New York, NY, USA, 2009. Association for Computing Machinery.

[TPF09b] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. "TAJ: Effective Taint Analysis of Web Applications." In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, p. 87–97, New York, NY, USA, 2009. Association for Computing Machinery.

[UKL20] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. "NJR-1 Dataset." `https://doi.org/10.5281/zenodo.4839913`, June 2020.

[UKL21] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. "Artifact for ICSE-22 submission "Striking a Balance: Pruning False-Positives from Static Call Graphs"." `https://doi.org/10.5281/zenodo.5177161`, August 2021.

[ULK22] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. "Striking a Balance: Pruning False-Positives from Static Call Graphs." In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 2043–2055, New York, NY, USA, 2022. Association for Computing Machinery.

[UP21] Akshay Utture and Jens Palsberg. "Artifact for ICSE-22 submission "Fast and Precise Application Code Analysis using a Partial Library"." `https://doi.org/10.5281/zenodo.5551128`, August 2021.

[UP22] Akshay Utture and Jens Palsberg. "Fast and Precise Application Code Analysis Using a Partial Library." In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 934–945, New York, NY, USA, 2022. Association for Computing Machinery.

[WAL15] WALA. "IBM, "T.J. Watson Libraries for Analysis (WALA),"." `http://wala.sourceforge.net`, 2015.

[web01] Checkstyle website. "Checkstyle." `https://checkstyle.org`, 2001.

[WLX16] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. "Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps." *IEEE Transactions on Software Engineering*, **42**(11):1054–1076, 2016.

[WNL09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. "Automatically finding patches using genetic programming." In *2009 IEEE 31st International Conference on Software Engineering*, pp. 364–374, 2009.

[WR99] John Whaley and Martin Rinard. "Compositional Pointer and Escape Analysis for Java Programs." In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, p. 187–206, New York, NY, USA, 1999. Association for Computing Machinery.

[XSY19] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. "VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences." In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 512–523. IEEE Press, 2019.

[XZ22] Chunqiu Steven Xia and Lingming Zhang. "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning." In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, p. 959–971, New York, NY, USA, 2022. Association for Computing Machinery.

[YMM22] He Ye, Matias Martinez, and Martin Monperrus. "Neural Program Repair with Execution-Based Backpropagation." In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 1506–1518, New York, NY, USA, 2022. Association for Computing Machinery.

[YS13] U. Yüksel and H. Sözer. "Automated Classification of Static Code Analysis Alerts: A Case Study." In *2013 IEEE International Conference on Software Maintenance*, pp. 532–535, 2013.

[ZR07a] Weilei Zhang and Barbara G. Ryder. "Automatic Construction of Accurate Application Call Graph with Library Call Abstraction for Java: Research Articles." *J. Softw. Maint. Evol.*, **19**(4):231–252, July 2007.

[ZR07b] Weilei Zhang and Barbara G. Ryder. "Automatic Construction of Accurate Application Call Graph with Library Call Abstraction for Java: Research Articles." *J. Softw. Maint. Evol.*, **19**(4):231–252, July 2007.