Program Analyses for Understanding the Behavior and Performance of Traditional and Mobile Object-Oriented Software

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Dacong Yan
Graduate Program in Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

Atanas Rountev, Advisor

Feng Qin

Michael D. Bond

ABSTRACT

The computing industry has experienced fast and sustained growth in the complexity of software functionality, structure, and behavior. Increased complexity has led to new challenges in program analyses to understand software behavior, and in particular to uncover performance inefficiencies. Performance inefficiencies can have significant impact on software quality. When an application spends a substantial amount of time performing redundant work, software performance and user experience can deteriorate. Some inefficiencies can use up certain types of resources and lead to program crashes. In general, performance inefficiency is an important and challenging problem for modern software systems. It is also a shared problem for traditional and mobile object-oriented software. Static and dynamic analyses need to keep up with this trend, and this often requires novel technical approaches.

One important symptom of performance inefficiencies is *run-time bloat*: excessive memory usage and work to accomplish simple tasks. Bloat significantly affects scalability and performance, and exposing it requires good diagnostic tools. As the first contribution of this dissertation, we present a novel analysis that profiles the run-time execution to help programmers uncover potential performance problems. The key idea of the proposed approach is to track object references, starting from object creation statements, through assignment statements, and eventually ending at statements that perform useful operations. An abstract view of reference propagation is provided with path information specific to reference producers and their run-time contexts. Several client analyses demonstrate the use of this abstract view to uncover run-time inefficiencies.

Memory leaks, both for traditional and for mobile object-oriented software, present a significant problem for software quality. Static memory leak detection is challenging because it is extremely difficult to statically compute precise object liveness for large-scale applications. We bypass this difficulty by leveraging a common leak pattern. In many cases, severe leaks occur in loops where, in each iteration, some objects created by the iteration are unnecessarily referenced by objects external to the loop. These unnecessary references are never used in later loop iterations. Based on this insight, we shift our focus from computing liveness, which is very difficult to achieve precisely and efficiently for large programs, to the easier goal of identifying objects that flow out of a loop but never flow back in. We formalize this analysis using a type and effect system and present its key properties. This technique was applied on eight real-world programs, such as Eclipse, Derby, and log4j. It not only identified known leaks, but also discovered new ones whose causes were unknown beforehand, while exhibiting a false positive rate suitable for practical use.

In addition to static analysis, performance testing is an effective approach to discover memory leaks. For example, sustained growth in memory usage during test execution can indicate potential memory leaks. However, performance testing to expose leaks for arbitrary software is very difficult, because, similar to other dynamic approaches, it also requires specific leak-triggering program inputs. As the third contribution of this dissertation, we introduce LeakDroid, a novel and comprehensive approach for systematic testing of resource leaks in Android applications. At the core of the proposed testing approach is model-based test generation that focuses specifically on coverage criteria aimed at resource leak defects. These criteria are based on the novel notion of *neutral cycles*: sequences of GUI events that should have a "neutral" effect and should not lead to increases in resource usage. Several important categories of neutral cycles are considered in the proposed test coverage criteria. As demonstrated by experimental evaluation and case studies on eight Android applications, the proposed approach is very effective in exposing resource leaks.

Model-based test generation such as LeakDroid depends critically on GUI models, which describe accessible GUI objects and corresponding user actions. GUI models ultimately determine the possible flow of control and data in GUI-driven applications. The ability to understand Android GUIs is critical for the reasoning of the semantics of an Android application. We introduce the first static analysis to model GUI-related Android objects, their flow through the application, and their interactions with each other via the abstractions defined by the Android platform. We first develop a formal semantics for the relevant Android constructs to provide a solid foundation for this and other analyses. Based on the semantics, we define a constraint-based reference analysis. The analysis employs a constraint graph to model the flow of GUI objects, the hierarchical structure of these objects, and the effects of relevant Android operations. Experimental evaluation on real-world Android applications strongly suggests that the analysis achieves high precision with low cost. The analysis enables static modeling of control/data flow that is foundational for compiler analyses, instrumentation for event/interaction profiling, static error checking, security analysis, test generation, and automated debugging. It provides a key component to be used by static analysis researchers in the growing area of Android software.

GUI applications are usually organized as a series of GUI windows containing structures of GUI widgets. User interaction with these windows (e.g., navigating from one to another and then going back) drives the control flow of the application. In Android, an *activity* plays the role of a GUI window, and transitions between activities are managed with the help of an *activity stack*. To understand this additional aspect of Android semantics, we introduce the first static analysis to model the Android activity stack, the changes in this stack, and the related interactions between activities. The analysis is an important step toward fully modeling the control/data flow of an Android application. It can be leveraged by other researchers to prune infeasible control flow paths in static analysis for Android, or to discover more paths that would be missing without modeling of the activity stack.

In conclusion, this dissertation presents several dynamic and static program analysis techniques to understand the behavior of object-oriented software systems, to uncover potential performance inefficiencies in them, and to locate the root causes of these problems. The programs studied by these techniques are all written in Java, but we believe the proposed techniques are general enough to also be applied to systems written in other object-oriented languages. With these techniques, we advocate the insight that a carefully-selected subset of high-level behavioral patterns and program semantics must be leveraged in order to perform practical program analyses for modern software.

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor Nasko Rountev for his support, patience and guidance throughout the duration of my Ph.D. study. He has always been ready to help, and has devoted enormous amount of time and effort in training me to become a computer science researcher. I would also like to thank members of the PRESTO group for all the discussions and collaborations. I especially want to thank Harry Xu, who has given me a lot of useful advice and insightful comments. I thank Prof. Feng Qin and Prof. Michael D. Bond for serving on my dissertation committee. I am grateful to Wei Huang and Yi Zhao for their mentoring during my internship at Google, which helped me become a better programmer. Finally, I would like to thank my parents for their unconditional support.

The material presented in this dissertation is based upon work supported by the U.S. National Science Foundation under CAREER grant CCF-0546040 and grants CCF-1017204 and CCF-1319695, by an IBM Software Quality Innovation Faculty Award, and by a Google Faculty Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

September 2009 – August 2014	Graduate Teaching/Research Associate, The Ohio State University
May 2013	M.S. Computer Science, The Ohio State University
June 2009	B.Eng. Software Engineering, Shanghai Jiao Tong University

PUBLICATIONS

Research Publications

Atanas Rountev and Dacong Yan. Static Reference Analysis for GUI Objects in Android Software. In *International Symposium on Code Generation and Optimization (CGO'14)*, pages 143-153, February 2014.

Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *International Symposium on Code Generation and Optimization (CGO'14)*, pages 87-97, February 2014.

Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic Testing for Resource Leaks in Android Applications. In *IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*, pages 411-420, November 2013.

Shengqian Yang, Dacong Yan, and Atanas Rountev. Testing for Poor Responsiveness in Android Applications. In *International Workshop on the Engineering of Mobile-Enabled Systems (MOBS'13)*, pages 1-6, May 2013.

Shengqian Yang, Dacong Yan, Guoqing Xu, and Atanas Rountev. Dynamic Analysis of Inefficiently-Used Containers. In *International Workshop on Dynamic Analysis (WODA'12)*, pages 30-35, July 2012.

Dacong Yan, Guoqing Xu, and Atanas Rountev. Rethinking Soot for Summary-Based Whole-Program Analysis. In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP'12)*, pages 9-14, June 2012.

Guoqing Xu, Dacong Yan, and Atanas Rountev. Static Detection of Loop-Invariant Data Structures. In *European Conference on Object-Oriented Programming (ECOOP'12)*, pages 738-763, June 2012.

Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering Performance Problems in Java Applications with Reference Propagation Profiling. In *International Conference on Software Engineering (ICSE'12)*, pages 134-144, June 2012.

Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-Driven Context-Sensitive Alias Analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 155-165, July 2011.

Atanas Rountev, Kevin Van Valkenburgh, Dacong Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential Java programs. In *IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1-9, September 2010.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
High-End Systems	Prof. P. Sadayappan
Software Systems	Prof. Srinivasan Parthasarathy

TABLE OF CONTENTS

Page

Chapters:

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1: Introduction

Performance inefficiencies can have significant impact on software quality. When an application spends a substantial amount of time performing redundant work, software performance and user experience can deteriorate. Some inefficiencies can use up certain types of resources and lead to program crashes. In general, performance inefficiency is an important problem for modern software systems, and it is also a difficult challenge due to the increasing complexity of software functionality, structure, and behavior.

Performance inefficiency is an even more important problem for mobile computing platforms since they typically have much more limited resources. Due to performance problems, user experience could degrade significantly, and the mobile application could even be uninstalled by unsatisfied users. Android applications, as one important example of applications for mobile devices, have a mixture of both the complexity of object-oriented systems, and the constraint of limited resources. Therefore, software engineering techniques and tools targeting their performance problems are greatly needed.

1.1 Challenges

Challenges for compiler optimizations. While the symptoms of performance inefficiencies could be easily observed through measurements of running time and memory usage, the underlying causes for these problems could be very difficult to track down. It is not unusual to see performance problems resulting from inappropriate design and implementation choices. Often, there does not exist a single hot spot in the program for compiler optimizations to work on. In such cases, the removal of performance inefficiencies requires insights of the programmer's intention, which usually cannot be fully captured by compilers. Thus, fully automatic optimization is often not a feasible solution to these problems.

Challenges for application developers. Since compiler optimizations cannot detect and eliminate all performance problems, developers often rely on manual tuning techniques to incorporate human insight into the optimization process. However, as software systems are increasingly relying on reusable libraries and frameworks, it becomes much more difficult for application developers to fully understand the run-time behavior of the whole system. When reusable components are not used properly, or, worse yet, when they have bugs within themselves, performance problems are extremely hard to detect without good tool support.

Challenges for performance testing. Performance testing is an approach that evaluates the performance of software systems. It is widely applied for server applications. Metrics such as throughput and response time are commonly accepted for evaluation of server performance. However, GUI-based applications present new challenges to performance testing. Most end-user applications are based on graphical interfaces; examples include desktop applications, web applications, and mobile applications. To be able to perform systematic testing for these applications, a well-defined model of the GUI is needed. Next, test cases that could expose performance problems should be derived from this model. While many techniques have been proposed to generate test cases to detect functional problems, there is little work on test case generation for detecting performance inefficiencies in GUI-based applications. At present, it is difficult to cover a sufficient number of interesting code paths and observe potential performance inefficiencies for GUI-based applications, one of the most important types of software.

Challenges for static analysis. Static analysis is a key technique for modeling the possible behaviors of a program, and can be used to both (1) drive the exploration of run-time behaviors that aim to expose performance problems, and (2) statically detect potential performance-related defects. For Android software, foundational static analyses such as reference analysis and control-flow analysis are still not developed. For example, a new reference analysis is needed for identifying GUI objects and their corresponding event handlers, which in turn is essential for program understanding and test generation. Similarly, control-flow analysis needs to account for new abstractions introduced in Android (e.g., activities and the activity stack), in order to allow effective exploration of dynamic behaviors, as well as precise static checking of correctness/performance properties. The comprehensive and complicated libraries provided by Android present new challenges to make static analysis efficient and precise. Analysis that is agnostic of the high-level platform semantics would inevitably produce results that are too conservative to be useful for client analyses. Another unique challenge for static analysis of Android applications is their component-based nature. It is essential to model the interactions between Android components (e.g., activities) in order to perform general control-flow and data-flow analysis. These static analysis challenges have not been addressed by existing work.

1.2 Understanding the Behavior and Performance of Traditional and Mobile Object-Oriented Software

The goal of this dissertation is to develop several dynamic and static program analysis techniques to understand the behavior of object-oriented software systems, to uncover the (potential) performance inefficiencies in them, and to locate the root causes of these problems.

Understanding object behaviors. One of the most difficult tasks during manual performance tuning is to understand object behaviors—where and why objects are created, how they flow through the heap, and where they are being used. Chapter 2 of this dissertation introduces a dynamic analysis technique, called *reference propagation profiling*, that profiles the run-time execution to track the propagation of object references. The tracking starts from object creation statements, through assignment statements, and eventually ends at statements where object references are used to perform useful operations. This propagation is abstracted by a representation we refer to as a *reference propagation graph*. This graph provides path information specific to reference producers and their run-time contexts. Several client analyses demonstrate the use of reference propagation profiling to uncover run-time inefficiencies. We also present a study of the properties of reference propagation graphs produced by profiling 36 Java programs. Several case studies discuss the inefficiencies identified in some of the analyzed programs, as well as the significant improvements obtained after code optimizations.

Static detection of memory leaks. Memory leaks, both for traditional and for mobile object-oriented software, present a significant problem for software quality. Dynamic analyses can be used to detect leaks, but they require specific leak-triggering program inputs, which are especially difficult to find during development and in-house testing. To overcome this limitation and detect memory leaks before software release, we develop a novel (and the first practical) static memory leak detection technique for Java programs. This approach, called LeakChecker, is described in Chapter 3. A long-standing issue that prevents the design of such a technique is that it can be extremely difficult to statically compute precise object liveness for large-scale applications. LeakChecker bypasses this difficulty by leveraging a common leak pattern. In many cases, severe leaks occur in loops where, in each iteration, some objects created by the iteration are unnecessarily referenced by objects external to the loop. These unnecessary references are never used in later loop iterations. Based on this insight, we shift our focus from computing liveness, which is very difficult to achieve precisely and efficiently for large programs, to the easier goal of identifying objects that flow out of a loop but never flow back in. We formalize this analysis using a type and effect system and present its key properties. LeakChecker was used to detect leaks in eight real-world programs, such as Eclipse, Derby, and log4j. It not only identified known leaks, but also discovered new ones whose causes were unknown beforehand, while exhibiting a false positive rate suitable for practical use.

Exposing leaking behaviors. In addition to static analysis, performance testing is an effective approach to discover memory leaks. For example, sustained growth in memory usage during test execution can indicate potential memory leaks. However, performance testing to expose leaks for arbitrary software is very difficult, because, similar to other dynamic approaches, it also requires specific leak-triggering program inputs. As the third contribution of this dissertation, we introduce LeakDroid, a novel and comprehensive approach for systematic testing of resource leaks in Android applications (Chapter 4). An Android application is driven by a graphical user interface (GUI), with GUI objects responding to user actions. A GUI model describes for each moment of time the GUI objects users can interact with, and correspondingly the possible user actions. At the core of the proposed testing approach is model-based test generation that focuses specifically on coverage criteria aimed at resource leak defects. These criteria are based on the novel notion of *neutral cycles*: sequences of GUI events that should have a "neutral" effect and should not lead to increases in resource usage. Several important categories of neutral cycles are considered in the proposed test coverage criteria. Experimental evaluation and case studies were performed on eight Android applications. The approach exposed 18 resource leak defects, 12 of which were previously unknown. These results provide motivation for future work on analysis, testing, and prevention of resource leaks in Android software. The implemented testing tool and the experimental subjects used in this work have been made available online in the LeakDroid release page.[1]

Understanding GUI-driven Android applications. The key components of a GUI-driven application are GUI objects and the associated event handlers. They ultimately determine the possible flow of control and data in these applications. As discussed earlier, Android applications are GUI-driven. The ability to understand Android GUIs is critical for the reasoning of the semantics of an Android application. Furthermore, automated test generation (e.g., for the coverage criteria defined in LeakDroid) depends critically on GUI models. In Chapter 5, we introduce the first static analysis to model GUI-related Android objects, their flow through the

[1] `http://www.cse.ohio-state.edu/presto/software`

application, and their interactions with each other via the abstractions defined by the Android platform. The proposed analysis falls into the category of a static object reference analysis, which models the flow of object references. Existing reference analyses for traditional Java applications cannot be applied directly because Android applications are component-based and event-driven. We first develop a formal semantics for the relevant Android constructs to provide a solid foundation for this and other analyses. Based on the semantics, we define a constraint-based reference analysis. The analysis employs a constraint graph to model the flow of GUI objects, the hierarchical structure of these objects, and the effects of relevant Android operations. Experimental evaluation on real-world Android applications strongly suggests that the analysis achieves high precision with low cost. The analysis enables static modeling of control/data flow that is foundational for compiler analyses, instrumentation for event/interaction profiling, static error checking, security analysis, test generation, and automated debugging. It provides a key component to be used by static analysis researchers in the growing area of Android software.

Understanding control flow in Android applications. GUI applications are usually organized as a series of GUI windows containing structures of GUI widgets. User interaction with these windows (e.g., navigating from one to another and then going back) drives the control flow of the application. In Android, an *activity* plays the role of a GUI window, and transitions between activities are managed by an *activity stack*. In Chapter 6, we introduce the first static analysis to model the Android activity stack, the changes in this stack, and the related interactions between activities. We extend the formal semantics developed in Chapter 5 to include abstractions to represent the state and changes of the activity stack. Based on the semantics, we encode relevant Android constructs in an *activity transition graph* and perform traversal on this graph to compute the set of possible activity stack states. The output of the analysis is encoded in a *stack transition graph*, whose nodes represent stacks and edges represent abstract operations to trigger the transition between two stacks. The analysis is an important step toward fully modeling the control/data flow of an Android application. It can be leveraged by other researchers to prune infeasible control flow paths in static analysis for Android, or to discover more paths that would be missing without modeling of the activity stack.

1.3 Outline

The rest of this dissertation is organized as follows. Chapters 2–6 present the novel program analysis techniques contributed by this dissertation. Related work is described in Chapter 7. Chapter 8 summarizes this dissertation's contributions.

CHAPTER 2: Reference Propagation Profiling to Uncover Performance Inefficiencies

Many applications suffer from *chronic run-time bloat*—excessive memory usage and run-time work to accomplish simple tasks—that significantly affects scalability and performance. This is a serious problem for software systems, yet their complexity and performance are not well understood by application developers and software researchers. Some run-time inefficiencies are even a result of adapting common practices (such as creating APIs for general use, and favoring method reuse without specialization). The conclusion from detailed analysis of dozens of real-world applications is that great amounts of work and memory resources are often needed to accomplish very simple tasks [73]. A few redundant objects, calls, and assignments may seem insignificant, but the inefficiencies easily get magnified when the inefficient code is frequently reused, causing significant system slowdowns and soaking up excessive memory resources.

While a modern compiler (such as the just-in-time compiler in a virtual machine) offers sophisticated optimizations, they often are of very limited help in removing bloat. This is because dataflow analyses in a compiler often have small scopes (i.e., they are intraprocedural), which makes it impossible to tackle problems that can cross dozens of calls and even multiple frameworks. In addition, compiler analyses are generally unaware of the domain semantics of the program, while bottlenecks often result from inappropriate design/implementation choices. Finding and fixing these performance problems requires human insight, and thus it is highly desirable to develop diagnostic tools that can expose performance bottlenecks to the developers.

In this chapter, we present a novel tool that profiles the execution to help programmers uncover potential performance problems. The key idea of the proposed approach is to track *object references*, starting from their producers (object creation statements), through assignment statements that propagate the references, eventually reaching statements that use the corresponding objects to perform useful operations. This run-time propagation is abstracted by a representation we refer to as a *reference propagation graph*. This graph contains nodes that represent statements, and edges that correspond to the flow of references between them. The edges are annotated with run-time frequencies. We have designed several client analyses that identify common patterns of bloat by analyzing various graph properties. An initial description of these contributions appeared in [120].

2.1 Motivation

The motivation for the proposed reference profiling analysis is threefold. First, the creation and manipulation of objects is at the core of modern object-oriented applications. In cases where the object behavior exhibits suspicious symptoms (e.g., many objects are created by a statement, but only few of them are ever used), it is natural to investigate such symptoms. Second, the specific abstraction of run-time behavior—the reference propagation graph—provides enough information to relate the profiling information back to the relevant source code entities; this makes it easier for a tool user to understand the problematic behavior. Furthermore, the representation maintains separate propagation paths for different sources of object references, and for different contexts of the producers of these references, which allows precise identification of problematic paths. Finally, it is important not only to identify potential performance issues, but also to provide guidance on how to focus the efforts to fix them. Our approach characterizes the complexity of interprocedural propagation, as well as of interactions with heap data structures, in order to identify the problems that are likely to be easier to explain and eliminate.

Performance inefficiency often comes from extraneous work performed to accomplish a task. One symptom of such inefficiency is the imbalance between the cost of constructing and propagating an object, and the benefit the object can bring to the progress of the application. For example, an object may be propagated to many parts of the code, but only a subset of this affected code actually benefits from having access to the object.

To characterize such imbalance, and to use it to detect potential performance problems, we track three types of run-time events: *object allocation*, *reference assignment*, and *object usage*. In Java, an object is always accessed through references. Such references can be propagated through either stack locations or heap locations. As a form of stack propagation, references can also cross method boundaries via parameter passing or method returns. By writing such references to fields of other objects, they can become accessible to large portions of the application's code.

Such propagation greatly increases the difficulty of manually tracking and understanding the behavior of the object of interest. An automatic *reference propagation profiling tool* can provide significant value and insights needed for performance tuning, especially for complex Java applications. The rest of this section demonstrates through an example how reference propagation profiling can be useful in uncovering

```
1 class Vector {
2    double x, y;
3    Vector sub(Vector v) {
4       Vector res = new Vector(x - v.x, y - v.y);
5       return res;
6    }
7    void sub_rev(Vector v, Vector res) {
8       res.x = this.x - v.x;
9       res.y = this.y - v.y;
10   }
11   Vector copy() {
12      return new Vector(x, y);
13   }
14 }
15 Vector[][] a = ...;  // input data
16 Vector[][] d = ...;  // intermediate result
17 Vector temp = new Vector();
18 // m, n are typically large numbers
19 int m = readInput();
20 int n = readInput();
21 void compute() {
22    for (i = 1; i < m; i++) {
23      for (j = 1; j < n; j++) {
24        if (cond1) {
25          temp = a[i+2][j].sub(a[i-1][j]);
26        } else {
27          temp = new Vector(...);
28        }
29        ... // read/write fields of temp
30        d[i][j] = temp;
31        if (cond2) {
32          temp = a[i+1][j].sub(a[i-2][j]);
33        } else {
34          temp = new Vector(...);
35        }
36        d[i][j].x += temp.x;
37        d[i][j].y += temp.y;
38      }}
39 }
40 static void main(String[] args) {
41    compute();
42    ... // access the fields of d[i][j]
43 }
```

Figure 2.1: Running example.

performance inefficiencies. The next section describes the formulation and implementation of this dynamic analysis.

Motivating Example. Figure 2.1 shows a code example simplified and adapted from the euler program of the JavaGrande benchmark suite [54]. Class Vector represents coordinates in a 2D space. Its sub method subtracts one Vector from another, and returns the result in a newly-created Vector (line 4). The method would be invoked many times during a typical execution. A very large number of objects of type Vector would be allocated, since the loops in lines 22–38 would be executed many times. The cost of calls to sub (lines 25 and 32) and the object allocations inside sub (line 4) is very high. However, not all of this work is necessary. Note that the object is created solely for the purpose of storing the result of the subtraction. Once the result is retrieved from the object, that object becomes useless and would be deallocated by the garbage collector.

We can reuse a single Vector object across multiple calls to sub. A variant of sub called sub_rev is shown at lines 7–10. The new method has an extra parameter to store the result of the subtraction, and the caller of this method is responsible for allocating the object. In this way, the caller would have the flexibility to reuse the object across multiple invocations of sub_rev. Specifically, the object returned at the call to sub at line 32 is immediately read (lines 36–37) and discarded. A call to sub_rev at line 32, with reuse of a single temporary object allocated before the i loop, will eliminate the cost of frequent allocation and garbage collection for these short-lived objects.

In cases when the resulting Vector is assigned to the heap (line 30) and becomes part of a global data structure, we need to investigate how this heap data structure (d[i][j] in this example) is being used. This is necessary to determine whether it is safe to perform the code transformation. We need to track how the object propagates in the memory space through references. For example, the object created at line 4 is propagated through the references res, temp, and then d[i][j]. After the object is assigned to d[i][j], which is a heap location, we need to know whether it is ever read back from the heap. If it is not, we can safely reuse the object; if it is, meaning that there exists an assignment such as v=d[i][j], we have to continue tracking how the local variable v is used. In this example, the object is indeed read back from the heap (line 42). Thus, the call at line 25 cannot be replaced with a call to sub_rev.

As described later, the reference propagation profiling can provide insights into the behavior of the objects created at line 4. In the actual euler benchmark, we observed that a large number of objects created at this allocation site are propagated through the call at line 32, but not any further. In the analysis results, this propagation path is clearly distinguished from the path through the call at line 25, for which there do not exist easy performance optimizations. With the code transformation outlined above, we observed a reduction of 13.3% in running time and 73.3% in number of allocated objects for this benchmark.

Similarly to other dynamic analysis techniques, the conclusions drawn from the propagation graph depend on the quality of the run-time information. In our experiments we run well-defined benchmarks on representative inputs that come with them. In practical use, such representative inputs are necessary for this (or any other) profiling analysis.

2.2 Reference Propagation Profiling

Reference Propagation Graph. The propagation of (references to) an object during its lifetime is encoded as a *reference propagation graph.* For illustration, the graph for the example from the previous section is shown in Figure 2.2.

There are three types of nodes in the graph. A *producer node* represents object allocations. Each producer node has (1) an allocation site ID which encodes the static location of the allocation expression in the source code, and (2) context information obtained when this allocation occurs at run time. The degree of context sensitivity can be tuned as a parameter of the analysis. A *reference assignment node* represents the assignment statements that propagate the objects through references. We distinguish stack-only propagation (at object allocations, between local variables, or due to parameter passing and return values), and propagation between heap and stack (caused by reading or writing instance fields, static fields, or array elements). The nodes are uniquely determined by their static location in the code, and the producer node that reaches them—that is, a single statement in the code can be represented by multiple graph nodes, one per producer of object references. A single *consumer node* represents the usage of objects. If a producer node reaches this node through a certain path, some objects propagated through that path are used. An object is used when (1) it is the receiver of a method call, (2) a field of the object is read or written, (3) it is used as a parameter in a call to a native method, or (4) it is an operand of `instanceof`, `==`, `!=`, or casting.

There are three types of edges in the graph. An *alloc-assign edge*, between a producer node and a reference assignment node, corresponds to object allocations $ref = new\ X$. A *def-use edge*, connecting two reference assignment nodes, represents the def-use relationship between two reference assignment statements such as $ref = \ldots$ and $\ldots = ref$. A *usage edge*, from a reference assignment node to the consumer node, indicates a def-use relationship between an assignment $ref = \ldots$ and another statement in which the value of *ref* is used (as described above).

Example. The subgraph related to the allocation at line 4 for the example in Figure 2.1 is shown in Figure 2.2. In this example, a context-insensitive scheme is used to model run-time objects. (Context sensitivity will be discussed later in this section.) Thus, the objects created at line 4 are abstracted solely with the line number, and a node Producer(4) is added to the graph. Immediately after the allocation, the object is assigned to local variable `res`, so there is a node RefAssign(4,4) and an alloc-assign edge to it. This node is then connected, via a def-use edge, to RefAssign(4,25), which represents the return value of the call at line 25. Here the first label on the node is the ID of the producer node (4) that created the propagated object, and the second label

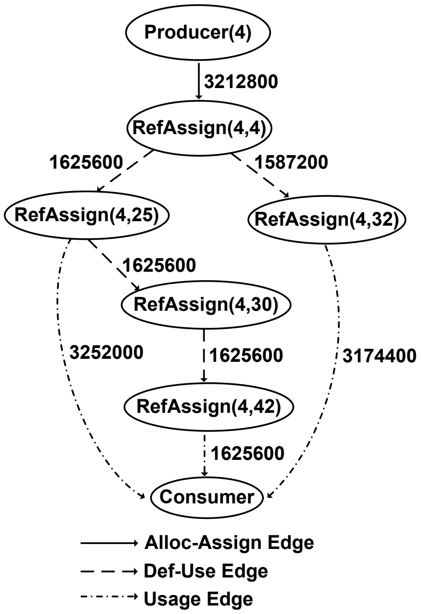


Figure 2.2: Reference propagation graph for the running example.

is the line number (25) of the actual statement that does the propagation. Similarly, RefAssign(4,32) and an edge to it are created due to the call at line 32.

In subsequent statements, fields of the object are accessed (lines 29 and 36–37); thus, the two reference assignment nodes are connected to the consumer node. The objects that are propagated along the path through line 25 are later assigned to the heap at line 30 and retrieved back at line 42, so the path is extended accordingly. Such an extension is not performed for the path via 32, since there is no further propagation along that path. The graph is annotated with run-time frequency information for graph edges, similar to the frequencies observed in the actual `euler` benchmark.

This graph provides the foundation for *reference propagation profiling.* Each edge in the graph is associated with a counter. Whenever a statement is executed at run time, the counter of the corresponding edge is incremented. Both the *structure* of the graph as well as the *edge weights* can be used to identify execution inefficiencies. For example, with this graph, it becomes significantly easier to understand the behavior of run-time objects created at line 4 of Figure 2.1. First, all paths starting from the producer node contain nodes that go to the consumer node, so it is not possible to simply remove the allocation. In other words, we have to explicitly create the object (or, perhaps, use some form of object inlining [31]). Second, the path through line 32 is very short, does not contain writes to the heap (i.e., the object does not become part of larger heap data structures), and represents a significant volume of reference propagation. Thus, it presents an interesting target for performance analysis and optimization. Third, the path through line 25 is longer, spans three methods, involves propagation through the heap, and therefore is likely to be harder to understand and optimize.

Producer-Specific and Context-Specific Propagation. Each reference assignment node is specific to a particular producer node. For example, the statement at line 30 in the example is represented by RefAssign(4,30), corresponding to the flow of references produced at line 4. This same statement can also propagate the references produced at line 27. A separate node RefAssign(27,30) would represent this propagation. Similarly, line 42 would correspond to two separate nodes, one for each producer. Such per-producer representation allows better precision when characterizing the flow of references. For example, consider the flow from line 30 to line 42. If this flow is *not* distinguished based on the producer, a single frequency would be associated with this pair of statements, making it impossible to attribute the behavior to individual sources of run-time objects.

A producer node is an abstraction of a set of run-time objects, and the choice of this abstraction is an important parameter of the analysis. The simplest abstraction is to use the ID of the allocation site that created the object. However, it is well known that this abstraction can be refined by considering the *context* of the allocation. There are various definitions of context, and they can be easily incorporated in our analysis. For the current implementation, we employ the so called *object-sensitive* abstraction. In this approach, a producer node corresponds to a pair (s_1, s_2) of allocation site IDs. The first ID s_1 is for the site that creates the object. That site is in some method, and the receiver object of that method (i.e., the object to which `this` refers to when s_1 is executed) is the context of the allocation. Thus, s_2 is the ID of the allocation site that created this receiver. This technique is appropriate for modeling of object-oriented data structures [72] and is currently used in our implementation. The generalization in which a node is a tuple $(s_1, \ldots, s_{k+1})$ (i.e., k-object-sensitivity [72]) can also be easily applied.

Intended Uses. The graph described above can provide useful information for efficient manual investigation of application code. The patterns of reference propagation, across method calls/returns and heap reads/writes, are easy to discern from the structure of the graph. Direct connection with relevant source code locations can be visualized inside a code browsing tool. The frequency information provides insights into the amount of work related to reference propagation, and helps identify hotspots in this propagation.

The graph can also serve as the foundation for a number of client analyses (Section 2.4). The key feature of these approaches is that they automatically identify "suspicious" allocation sites, based on properties of the propagation graph. Furthermore, the graph can provide a characterization of the complexity of propagation patterns and the required program transformations. As a result, programmers or performance tuning experts can focus on parts of the code that not only exhibit run-time inefficiencies, but are also likely to be relatively easy to understand and transform.

Certain aspects of the proposed analysis are similar to information flow analysis (e.g., [16,17,22,23,46,65,78,78,80,88,119]). However, we record and report not only the source of the transitive run-time dependence, but also the intermediate statements along the dependence chain, as well as (an abstraction of) the actual reference value being propagated. Furthermore, the execution frequencies are collected per-producer-node, which allows unrelated flows through the same statement to be separated.

2.3 Analysis Implementation

The analysis is implemented in the Jikes RVM (Research Virtual Machine) version 3.1.1 [56]. The instrumentation is implemented in the optimizing compiler in Jikes. During execution, only this compiler is used, and every method is compiled with it before being executed for the first time.

Shadow Locations. Each memory location containing reference values is associated with a shadow location [79]. Local variables in the compiler IR are represented as symbolic registers. To create shadows for locals, we assign an ID to each symbolic register at "compile time" (actually, at run time when the optimizing compiler is compiling the method), and associate that ID with graph nodes created as the program executes. Shadows of static fields are stored in a global table, and indices into the table are determined by the class loader. Shadows of instance fields are stored in place with originally declared fields, and accessed by offsets from the base objects. The offsets are also determined during class loading. Array elements are shadowed similarly to static fields, except that per-array tables are used.

In cases when an object is moved by a copying garbage collector, its corresponding shadows should also be moved. This can be done by modifying the garbage collector, but we choose to use a non-moving GC for ease of implementation. This decision does not affect the results of the analysis.

Abstractions for Run-time Entities. The reference propagation graph construction has two components: (1) “compile-time” instrumentation, which happens in the optimizing compiler at run time, and (2) run-time profiling, which builds the graph as the program executes. The instrumentation tags each object with its allocation site information. Specifically, we write an allocation site ID to the header of each object, and the ID can be used to look up the source code location of the allocation site. For a context-sensitive setting, the context information is also recorded in the header. For example, when we use the object-sensitivity representation, the allocation site ID of the receiver object is written to the object header as well. To introduce approximations and tune the overhead, we map the allocation site IDs id of receiver objects into c equivalence classes using a simple mapping function $f(id, c) = id \ \% \ c$, where c is a pre-defined value. To achieve full precision (i.e., no approximations), c can be set to the number of allocation site IDs, in which case every equivalence class is a singleton.

Besides allocation site information, we also reserve one extra word in the object header for uses specific to client analyses. For example, such analyses can use one bit to mark whether an interesting event occurs on the object (e.g., whether the object is ever assigned to the heap). Section 2.4 discusses how this can be useful for implementing client analyses. The source information of executed reference assignment statements is maintained in a similar way.

Run-time Event Tracking. Each run-time object has a producer node associated with it. To enable fast lookups, producer nodes are stored in a table *prods*, and can be accessed with an index i, a combination of the allocation site IDs of the object, and the receiver object of the surrounding method (a default value 0 is used for a context-insensitive setting). Suppose the two IDs are *allocId* and *recvId*, and c equivalence classes are used in the object-sensitivity encoding. The index i is computed as $i = \mathit{allocId} \times c + \mathit{recvId}\%c$. Thus, each pair $(\mathit{allocId}, \mathit{recvId})$ is mapped to an index ranging from 0 to the number of allocation sites multiplied by c. When an object is created, we first look up the table to see whether there is already a producer node at $\mathit{prods}[i]$. If there is one, we increase the frequency of the existing node; otherwise, we create a new producer node, remember it in $\mathit{prods}[i]$, and write the IDs to the header of the newly-created object. In addition, we create a reference assignment node to be the shadow of the variable getting the new object, and connect the producer node with it. If the producer node already exists, the frequency of the edge is incremented.

For a reference assignment `lhs = rhs`, we (1) create a new reference assignment node, (2) remember the node in the shadow of `lhs`, and (3) connect the node stored in the shadow of `rhs` to it. When the edge between the two nodes already exists, its frequency is incremented instead. Parameter passing and method returns are treated as special forms of reference assignments. To pass the shadow information into and retrieve it back from callees, we maintain a per-thread scratch space to temporarily store shadows of parameters and return variables.

As described in Section 2.2, an object can be used at certain statements. For example, when a heap access $v.\mathit{fld} = \ldots$ or $\ldots = v.\mathit{fld}$ is executed, we create a usage edge between the node stored in the shadow of v, and the consumer node. If such an edge already exists, its frequency is incremented.

2.4 Client Analyses

This section describes several client analyses built on top of the reference propagation profiling described earlier. These analyses examine the reference propagation graph and report to programmers a ranked list of suspicious producer nodes that should be examined for performance tuning. The criterion as to what producers are suspicious is defined by individual client analyses. The reported producer nodes are ranked based on the number of times they are instantiated.

In addition, for each reported node, several metrics are computed and provided in the analysis output. The role of these metrics is to estimate the ease with which the propagation starting from this producer can be understood and optimized. Specifically, all reference assignment nodes reachable from a reported suspicious producer node are examined. The number of such reachable nodes that correspond to *calls and returns* is an indication of how widely the references are propagated throughout the calling structure. The higher this number, the more complex the interprocedural propagation, which means that code transformations are likely to be difficult (or impossible). Another metric is the number of reachable nodes that represent *heap reads and writes*. A large number of such nodes indicates that the objects created by the producer node interact in complex ways with heap data structures, which makes their understanding and transformation more challenging.

Not-Assigned-To-Heap (NATH) and Mostly-NATH Analysis. The NATH client analysis detects allocation sites that create many objects, but none of these objects are stored into the heap (i.e., no instance field, static field, or array element ever contains a reference to them). These sites are promising for tuning because the objects created at these sites may be roots of temporary data structures that are expensive to construct. In addition, these objects are typically short-lived, potentially leading to frequent garbage collection. The escape analysis performed by a JIT compiler usually cannot identify such redundancies, because many such objects do escape the methods that created them. Using the propagation graph, this analysis finds and reports all producer nodes that never reach reference assignment nodes corresponding to assignments from the stack to the heap.

If most of the objects created by a site are NATH, that site is still a good candidate for tuning. We refer to such sites as "mostly-NATH". For example, Line 4 in Figure 2.1 is a mostly-NATH site, and refactoring it brings significant performance improvements for the `euler` benchmark. Implementing this analysis requires a small extension to tag each object with an assigned-to-heap bit, and store a counter of assigned-to-heap objects in the producer node. The analysis reports any producer node for which the percent of NATH objects exceeded a given threshold. When such sites are reported, the propagation graph can be used to determine the specific paths in the code along which these objects are assigned to the heap (e.g., the path through line 25 in Figure 2.2). This information provides insights into the run-time object propagation, and eases the task of refactoring the NATH paths (i.e., the paths through which objects are not assigned to the heap).

Analysis of Cost-Benefit Imbalance. In cases when run-time cost is significantly higher than benefits, there could be some redundancies; in terms of objects, there may be excessive allocation or propagation. In general, it is inefficient to allocate a lot of objects but seldom use them. Also, it is suspicious to write an object to the heap significantly more times than it is being read back. This client analysis is a framework to detect such imbalances between cost and benefit, and can be instantiated with different definitions of cost and benefit. For example, we can consider writing an object to the heap as *cost* (because the object had to be created and propagated), and reading it back as *benefit* (since the object was needed by some method). If the ratio between these two is very high (*write-read-imbalance*), it is possible that we do not need that many objects, or the way the program organizes data is problematic. To implement this analysis, we can analyze the reference propagation graph. For a producer node, the cost is the sum of node frequencies for the reachable *stack-to-heap* reference assignment nodes, and the benefit is defined similarly for the *heap-to-stack* ones. The analysis reports all producer nodes for which this ratio is greater than a certain threshold value.

Analysis of Never-Used and Rarely-Used Allocations. One can identify *never-used object allocations* by finding the producer nodes that cannot reach the consumer node; the next section provides several examples of this situation. Or, similarly to the mostly-NATH analysis, one can develop an analysis of *rarely-used allocations*: allocation sites that instantiate many objects, but only a small percentage of these objects are used. As discussed later, our experimental results indicate that never-used objects and never-used allocation sites occur surprisingly often.

Other Potential Uses. There are other performance analyses that can make use of reference propagation profiling. For example, such profiling can be used to study *container-related inefficiencies.* The write-read-imbalance objects, those that are written to the heap significantly more times than they are read back, are often written to a heap location which is part of a container data structure. We can locate low-utility containers (many elements are added but only a few are retrieved) by tracking the heap locations to which those imbalanced objects are written. This can be done through inspection of the source code, aided by the path information in the reference propagation graph.

2.5 Case Studies

To evaluate the effectiveness of reference propagation profiling, we performed several case studies on Java applications from prior work [93,116,118], and found several interesting examples of performance inefficiencies. All problems uncovered in these case studies are completely new and have never been reported before. It took us about two days to locate and fix these problems. All programs were new to us. Most of the time was spent on producing a correct fix rather than locating problematic data structures. Such manual tuning is commonly used in practice [73], and without tool support it can be very labor-intensive.

mst This program, from a Java version [64] of the Olden benchmarks, solves the minimum spanning tree (MST) problem [27]. The tool report shows that for an input graph with 1024 nodes, 1047552 objects of type `Integer` are created; the same number of instances is also reported for type `HashEntry`. All of these objects are assigned to the heap, but only half of the `Integer` objects are read back. The large volume of object allocation and the significant cost-benefit imbalance (recall Section 2.4) are highly suspicious. We inspected the code and found that the program uses an adjacency list representation. For each node in the graph, it uses a hash table to store the distances to its adjacent nodes. The distance is represented by an `Integer` object. Thus, for each distance value, it has to create a new `Integer` object. For a graph with 1024 nodes, it creates 1024 hash tables (the tool shows that 1024 arrays of `HashEntry` are created, which corresponds to the 1024 hash tables), and each table has 1023 entries, storing the distances to the other 1023 nodes. So, the program needs $1047552 = 1024 \times 1023$ objects of type `Integer`, and similarly for type `HashEntry`. In addition, the input graphs used by the benchmark are all complete graphs (i.e., each node is connected to each other node).

In general, an adjacency matrix is the preferred representation for dense graphs. Also, for undirected graphs, the distance from node n_1 to n_2 is the same as that from n_2 to n_1, so the way this program stores distances has unnecessary space overheads, which is exactly why only half of the `Integer` objects are read back from the heap, rendering the other half redundant. To confirm our understating on the tool report without too much refactoring effort, we kept the adjacency list representation, and only slightly changed the code to store and look up distances in an undirected manner. Specifically, for nodes n_1 and n_2, we do not add n_1 to the adjacent list of n_2 anymore, and when we need the distance between them, we look up the adjacency list of n_1, the one with a smaller node ID. This simple change alone reduced running time by 62.5%, and object creation by 39.6% (measured with input graphs of 1024 nodes, and large enough heap sizes). For a fixed heap size of 128MB, the original version can only finish its execution with graphs of at most 1731 nodes, while the modified version can handle 2418 nodes, an input size 39.7% larger. If we refactor the code more aggressively and use an adjacency matrix representation instead, the performance improvement could potentially be even higher.

euler This program is from the Java Grande benchmark suite [54]. The tool shows that the `svect` method of the `Statevector` class creates a large number of `Statevector` objects, while only a small percentage of them are assigned to the heap. After inspecting the code, we found that the program creates temporary objects to serve as the return value of the `svect` method. Once the method finishes its execution, the caller would retrieve the computation result. Afterward, some of the returned objects are stored in an array to be used later, but most of them are not (recall the running example from Figure 2.1). Method `svect` is invoked inside nested loops that iterate many times, so it is very likely that it will degrade the performance significantly. To solve this problem, we modify the code to make `svect` share one common `Statevector` object to store the result, and make a copy of the objects only when they are to be assigned to the heap. By changing this site alone, we achieved performance improvement of 13.3% in running time and 73.3% in the number of allocated objects.

jflex In the report generated from running JFlex, we found that a large number of `String` and `StringBuffer` objects are created in the `toString` method of a variety of classes. Most of the `String` objects created at these sites are ultimately used to construct the parameter of the static method `Out.debug` which prints out debugging messages when certain debugging flag is turned on. The debugging message is constructed even when the debugging flag is turned off, making the `String` objects redundant. This is confirmed by our report that the `String` objects created at call sites of the Out.debug method are never used. To eliminate such redundancies, we change the code to manually inline the calls to Out.debug so that no debugging messages would be constructed when the debugging flag is turned off. This modification reduced the running time by 2.9% and the number of created objects by 26.9%.

bloat The analysis of this DaCapo benchmark [28] shows that there is excessive object allocation in method entrySet of class NodeMap. The program uses NodeMap, an inner class of Graph, to ensure there are no duplicate nodes in the graph, and the NodeMap uses a HashMap for the underlying storage. To implement entrySet, one can simply return the entry set of the underlying HashMap. However, the program instead returns a newly-created instance of a specialized AbstractSet implementation which incorporates sanity checks whenever element removal is to be performed. Specifically, it adds sanity checks to the remove and removeAll methods of the set object. In addition, in the set implementation, it has a specialized Iterator implementation which has similar checks in its remove method. These objects are not assigned to the heap, and present an opportunity for optimizations.

The specializations introduced by these objects are useful for debugging purposes. They are needed during the development phase, but redundant after the correctness of the program has been established. To eliminate the redundancy, we removed the checks and used the entry set of the underlying HashMap as the return value instead. After the refactoring, we achieved reduction of 10.4% in running time and 11.3% in the number of allocated objects.

chart As shown in the next section, 67.2% of the allocation sites in the chart DaCapo benchmark are never-used, meaning that all objects created at such sites are never used. When we examined these sites, we found that the most significant source of never-used objects was a site that creates a large number of `SeriesChangeEvent` objects, but none of them are used. The program creates these objects to notify the listeners that the data series has been changed, and they only contain one single field to represent the source of the event. Since there is no concurrent access to the listener-notification method, we can share one common `SeriesChangeEvent` and update its event source field whenever it is about to be passed to listeners. After this code transformation, we achieved a reduction of 7.7% in running time and 7.8% in the number of allocated objects.

2.6 Properties of Propagation Graphs

This section presents measurements that provide insights into the properties of reference propagation graphs. The measurements are based on a set of 36 programs used in prior work [93, 116, 118], including benchmarks from SPEC JVM98 [101], Java Grande v2.0 (Section 3) [54], a Java version [64] of the Olden benchmarks, and DaCapo 2006-MR2 [28]. The experimental results were obtained on a machine with a 3.4GHz Quad Core Intel i7-2600 processor.

As with similar work on dynamic analysis, a threat to external validity comes from the choice of analyzed programs and their test inputs. We have tried to ameliorate this problem by using a large number of programs from diverse sources, and the representative inputs included with them.

The running time overhead of the analysis is typically around 30–50×. Such overheads are common for similar performance analyses from existing work, and are also acceptable for performance tuning and debugging tasks (rather than for production runs). In our case, the overhead is high because we have to track all instructions

Program	Classes	Methods	Alloc	NATH		Never-Used		WRI Sites		Call/	Write/
			Sites	Sites	Objs	Sites	Objs	$t = 2$	$t = \infty$	Ret	Read
compress	18	67	22	9	109	0	0	1	1	5.14	3.41
db	9	52	31	16	122	1	30236	1	1	6.87	4.48
jack	53	294	264	107	457449	12	34104	6	5	8.16	7.09
javac	146	779	409	88	1141931	24	254718	41	18	26.69	29.98
jess	140	445	206	36	3359830	6	2087	53	53	8.79	5.93
mpegaudio	49	225	104	7	7	5	212	16	16	5.54	5.13
mtrt	34	196	137	52	4577717	23	465747	4	1	17.6	6.26
search	6	25	3	3	3	0	0	0	0	9.67	0
euler	5	25	19	11	4789005	2	19630	1	1	3.53	18.89
moldyn	5	22	6	2	2	0	0	0	0	5.33	17
montecarlo	14	96	23	15	365202	0	0	0	0	9.48	2.04
JGraytracer	13	55	44	18	51238212	11	4753813	5	5	4.23	3.73
bh	7	49	12	5	126422990	0	0	0	0	15.17	11.33
bisort	3	14	2	1	2	0	0	0	0	14	8.5
em3d	6	16	8	2	2	0	0	0	0	5.88	4.38
health	6	18	17	8	2571333	1	21895	1	1	4.41	2.35
mst	7	31	10	4	1026	0	675444	1	0	6.8	4.8
perimeter	11	42	10	3	3	0	0	0	0	13.8	5.5
power	7	31	16	4	21	0	0	0	0	5.31	3.75
treeadd	3	5	5	3	3	0	0	0	0	4.4	1.2
tsp	3	14	3	1	1	0	4	0	0	9	34.33
voronoi	7	43	12	6	196609	0	0	0	0	19.75	5.75
antlr	109	1256	1151	796	482074	115	152949	89	66	9.05	5.03
bloat	230	1639	969	508	9439879	46	113786	40	25	28.84	10.63
chart	285	1418	1926	732	1174156	1295	578854	243	237	2.96	2.1
eclipse	1210	9558	3694	1485	1802921	688	1678586	363	285	13.08	10.04
fop	663	2661	1246	484	243275	535	42110	109	95	6.34	3.32
hsqldb	112	1012	461	243	67745	60	29735	26	20	7.02	3.66
jython	622	2775	3328	457	5579384	269	807139	1725	1368	11.36	4.82
luindex	96	529	258	141	2167954	46	17888	8	5	6.78	4
lusearch	100	508	228	89	2280855	43	2119431	16	12	7.93	4.09
pmd	377	2175	669	232	11382153	150	2187057	87	65	17.24	7.25
xalan	343	2133	778	149	367534	141	233745	151	134	14.29	7.49
JFlex	35	264	286	74	990370	70	20325	65	65	7.06	4.56
jbb2000	56	476	512	385	7693562	70	10070051	16	12	7.21	3.47
jbb2005	73	601	566	378	916103	69	1642515	22	17	6.95	2.31

Table 2.1: Properties of the context-insensitive reference propagation graphs.

involving reference values. The typical memory usage overhead is around 2–3×. Still, we were able to use the tool to study real-world programs, including large applications such as `eclipse`, and to uncover interesting performance inefficiencies in them. An intriguing possibility for future work is to consider how to reduce the overhead. For example, static analysis can rule out certain uninteresting sites. It may also be possible to apply sampling to track the propagation for only some of the objects created at an allocation site.

Table 2.1 shows measurements of the reference propagation graphs obtained in a context-insensitive setting. The first two columns contain the number of loaded non-library classes and the number of executed methods in those classes. The third column shows the number of allocation sites (in these methods) that were executed at least once. These measurements characterize the sizes of the analyzed programs.

The NATH columns show the number of NATH allocation sites and NATH run-time objects. NATH objects are those that are never assigned to the heap. A NATH allocation site creates only NATH objects, but some NATH objects may be created by non-NATH sites. These measurements indicate the existence of objects that do not interact with the rest of the heap. An interesting observations is that the percentages of NATH allocation sites (the ratios between columns 4 and 3) are typically large for almost all of the programs. This result indicates that Java programs often employ relatively temporary and localized data structures, which presents opportunities for optimizations.

The next two columns report the number of never-used allocation sites and never-used run-time objects. An allocation site is said to be never-used when all of the objects it allocates are never used. These measurements characterize how efficiently the allocated objects are used. If a program creates a large number of objects, but never or seldom uses them, it is certainly inefficient, and improvements may be achievable after code transformations. High percentages of never-used sites (i.e., ratios between columns 6 and 3) provide a symptom of potential bloat, and could lead a programmer or a performance tuning expert to uncover performance problems.

Columns "WRI Sites" show the number of *write-read-imbalance* sites under two different threshold values t. Recall from Section 2.4 that for a producer node, a cost-benefit ratio is taken between the sum of node frequencies for the reachable *stack-to-heap* reference assignment nodes (heap writes), and that of *heap-to-stack* ones (heap reads). An allocation site is counted when the cost-benefit ratio of its corresponding producer node is greater than the threshold. The sites without any heap writes (i.e., NATH sites) are already identified by the NATH analysis, and are not considered for the WRI analysis. Threshold $t = 2$ selects sites whose allocated objects are written to the heap at least twice as many times as they are read back from the heap. The special threshold value $t = \infty$ covers the cases when the objects are only written to the heap but never read back. Larger numbers of WRI sites indicate higher degrees of wasted heap propagation, which could potentially be eliminated by code transformations.

The last two columns show the average numbers of (1) method invocation nodes (calls and returns), and (2) heap propagation nodes (heap writes and reads) reachable from a producer node. They characterize the complexity of the reference propagation, from the perspective of inter-procedural control-flow and heap data structure interactions. If the number of method invocation nodes is high, objects are propagated through large portions of the call structure, and the propagation is likely to be more difficult to understand and refactor. The same is true for the average number of heap propagation nodes, which indicate points of interaction with other heap objects. By presenting to the programmer these two metrics for a suspicious allocation site, our analysis can help to distinguish objects that are relatively easy to understand from objects whose behavior may be too complex to be worth further investigation.

Table 2.2 shows the size of the reference propagation graph (number of nodes and number of edges) under different context-sensitivity abstractions. The first two columns show the measurements for the context-insensitive setting, followed by object-sensitive settings with different numbers c of equivalence classes in the context

Program	ctx-insen		c=4		c=8		c=16		c=#AllocSites	
	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges
compress	227	375	227	375	227	375	227	375	227	375
db	405	692	457	768	489	808	511	835	511	835
jack	4383	7793	4699	8241	4874	8468	5117	8775	6459	10332
javac	23307	26126	25632	50256	26366	51575	26706	52097	27407	53205
jess	3340	5507	3602	5933	3611	5947	3735	6148	3840	6289
mpegaudio	1232	2063	1740	2903	1813	2982	1813	2982	1813	2982
mtrt	3727	7913	8872	19447	13079	27990	13508	28654	14347	30484
search	37	73	37	73	37	73	37	73	37	73
euler	451	950	451	950	451	950	451	950	451	950
moldyn	156	277	156	277	156	277	156	277	156	277
montecarlo	312	536	312	536	330	570	333	570	333	570
JGraytracer	425	575	456	610	462	616	462	616	462	616
bh	331	654	343	671	343	671	343	671	343	671
bisort	55	137	55	137	55	137	55	137	55	137
em3d	92	151	92	151	92	151	92	151	92	151
health	146	216	146	216	146	216	146	216	146	216
mst	131	232	131	232	131	232	131	232	131	232
perimeter	214	415	214	415	214	415	214	415	214	415
power	195	287	269	387	269	387	269	387	269	387
treeadd	39	56	51	72	57	76	57	76	57	76
tsp	121	419	121	419	121	419	121	419	121	419
voronoi	327	758	327	758	327	758	327	758	327	758
antlr	18199	36137	18815	37249	19175	37696	19385	38245	19561	38409
bloat	39505	85509	47569	102695	51734	111081	53674	114273	61618	126784
chart	12646	16876	14346	19486	15314	20853	16950	23431	17664	23955
eclipse	92206	179670	106219	201934	107015	206926	109244	209218	110346	212998
fop	14305	22741	15153	23811	15787	24708	16043	24973	16528	25524
hsqldb	5873	10601	6649	11656	7104	12464	7280	12686	7973	13819
jython	58835	96797	60495	99011	61774	100825	63221	103286	67127	107407
luindex	3226	5727	3441	6024	3480	6084	3511	6096	3558	6158
lusearch	3102	5158	3469	5660	3601	5888	3607	5956	3730	6103
pmd	17469	32074	18046	32984	18126	32996	18409	33431	19288	34749
xalan	18056	32584	19645	35293	20458	36887	20437	36805	21486	38397
JFlex	3860	5887	4204	6510	4304	6668	4313	6676	4421	6861
jbb2000	6451	11596	7457	13480	7957	14491	8707	16042	8729	16287
jbb2005	6198	10577	6870	11627	7037	11830	7245	12082	7559	12526

Table 2.2: Comparison of graph sizes for context-insensitive and four object-sensitive settings.

encoding ($c = 4, 8, 16$). The last column shows the measurements under a full object-sensitivity setting, where each receiver object ID belongs to a separate equivalence class (Section 2.3).

As the degree of context-sensitivity increases, graph size typically remains about the same or grows slightly. With more precise context information, we can better distinguish the run-time allocations, and more producer nodes can be created. Such a graph presents a more precise and detailed picture: instead of describing the "per producer" propagation, it provides insights into the "per producer, per context" behavior of objects. Although in principle the cost of collecting this more precise information can be high (in terms of running time and memory consumption), in reality this does not appear to be the case: context-sensitive information can be collected with little additional overhead. For the programs we studied, the average running time overhead when using the fully context-sensitive encoding is 1.4% (compared to using the context-insensitive one). For the memory usage overhead, the increase is 3.5%. This observation indicates that future work could investigate even more precise context-sensitivity abstractions.

2.7 Summary

This chapter presents a novel reference propagation profiling tool used to uncover performance problems in Java applications. It tracks the propagation of object references and encodes the results in a reference propagation graph. The information stored in the graph is specific to producers of object references (and the run-time contexts of these producers). Several client analyses are developed to analyze these graphs, and to report to developers a ranked list of suspicious allocation sites, annotated with information about the likely ease of performing transformations for them. Interesting performance inefficiency patterns are discovered by these clients. The properties of the reference propagation graphs are studied on 36 Java programs. The experimental results show that the degree of context-sensitive precision can be increased without significant additional costs. The running time reduction achieved by optimizing suspicious allocation sites can be significant, as demonstrated in several case studies. These findings suggest that our approach is a good foundation for implementing various client analyses to uncover reference-propagation performance problems, and to explain these problems to the developers.

CHAPTER 3: LeakChecker: Practical Static Memory Leak Detection for Managed Languages

In managed languages such as Java and C#, developers do not need to worry about memory correctness issues such as dangling pointers and double free errors. However, it remains challenging to avoid leaks. A memory leak in a managed language is caused by keeping unnecessary references to objects that are no longer used. These objects cannot be reclaimed by the garbage collector (GC), often leading to severe performance degradation and even program crashes.

Problems and Motivation Static analysis techniques [19, 49, 50, 59, 83, 104, 114] have been widely used to detect memory leaks for unmanaged languages such as C and C++. The explicit memory management in such languages allows the formulation of leak detection as a reachability problem—a control-flow path that creates an object but does not free it may reveal a leak. This formulation cannot be adopted for managed languages, because object deallocation is done automatically by GC. To the best of our knowledge, [96] and [30] are the only two techniques that can statically detect Java memory leaks. At the core of [96] is an algorithm to detect live regions of arrays, while [30] uses shape analysis to identify the objects that are reachable but no longer used. However, there is no evidence that precisely computing array live regions [96] or performing bi-abduction [30] can scale to large-scale applications such as Eclipse, and any attempt to trade off precision for efficiency can lead to reports that are of little value due to great numbers of false warnings. In addition, no evaluation is provided in [96] and [30], and thus, their effectiveness is unclear.

Dynamic analysis [14, 38, 48, 57, 74, 90, 117, 118] is typically used to find memory leaks in managed languages. Existing dynamic analyses are debugging techniques that require appropriate test inputs and can detect leaks only if they are triggered in a test execution. It can be very difficult to find such leak-triggering test inputs, especially during development and in-house testing, when it may be complicated to set up appropriate inputs and execution environments. This is particularly the case with component-based software (such as the development of Eclipse plugins and smartphone apps): components are developed separately and tested only in simulated environments; problems may be seen only after they are shipped and start communicating with other components in production settings. In this regard, a static analysis is highly desirable because it can detect leaks without running a program, thereby leading to improved software quality.

Challenges The major challenge for developing a static memory leak detector for managed languages is the difficulty of precisely computing *object liveness* properties. Even with highly precise heap modeling and data-flow analysis, it is still expensive to determine precisely whether an object would be used after a certain point in the program execution. The second challenge is that detecting and reporting unnecessary references at a low level (e.g., at heap reads and writes) can be of very limited help because such reads and writes can be far away from the root cause of the leak. For example, reporting the statement that writes an object into a `HashMap.Entry` without any context does not provide any suggestion as to how the leak can be fixed. A useful static analysis should be more informative and should provide high-level information more closely related to the application semantics and the cause of the leak. In addition, such precise static analysis should have practical cost when analyzing real-world large-scale applications.

Insight We present LeakChecker, the first practical static leak detector for Java that overcomes all these challenges by exploiting developer insight, and by identifying and reporting unnecessary references at a higher level of abstraction. An important observation that motivates the design of LeakChecker is that a severe leak is often related to frequently occurring program events. If each such event does not appropriately clean up a small number of references, unnecessary references can quickly accumulate and cause the memory footprint to grow. These events include, for example, database transactions, processing of user requests in web servers, iterative refinements of certain program properties in a static analysis, etc. For example, in Eclipse 3.2, a number of objects are unnecessarily kept alive every time a diff between two zip files is performed [14, 57, 117]. Comparing a few large zip files can quickly make Eclipse run out of memory. In general, an object created by one event instance may escape this instance and be used by future instances of the event. However, if such an escaping object is never used by future event instances, it is very likely to be a leaking object.

Events are often generated by loops. LeakChecker *analyzes each important loop in a program and detects the objects that escape one iteration of the loop and never flow back into any later iteration from the memory locations to which they escape.* A large-scale application may have a large number of loops, and precise analysis of each one can be prohibitively expensive. The developer usually has a clear understanding of which loop is the "main" event loop and which loop contains performance-critical computations; these loops are given as input to LeakChecker.

To further improve tool usefulness, the developer can also specify a repeatedly-executed code region (not a loop) for checking. This is particularly suitable for component-based software where the developer of a component does not have access to the event loop. For example, the developer of an Eclipse [37] plugin could specify the method that achieves the core plugin functionality as a checkable region—this method may be invoked in an invisible loop located in another plugin or in the framework.

The benefit of using a developer's specification is two-fold: (1) memory leak detection can be performed within a relatively small scope, leading to improved practicality and scalability; and (2) the reported leaks are easy to understand and fix, because their root causes are very likely to be the operations that store the leaking objects into objects created outside of the region. Once the important loops and code regions are specified by the tool user, the rest of the approach is fully automated. Because any repeatedly-executed code region can be thought of as the body of an (artificial) loop, this work discusses only leaks in a loop.

Analysis Technique LeakChecker attempts to identify a path p_{out}—a sequence of statements that write to heap objects—through which an object escapes a loop iteration, as well as a path p_{in} of heap read statements through which an object flows back into a loop iteration. Identification of these two paths considers inter-procedural control flow with properly matched method calls and returns. Objects that only flow out through p_{out} but do not flow back in through an appropriate p_{in} are immediately considered leaking. For objects with a proper p_{in}, we further consider two conditions. First, p_{out} and p_{in} should be related to the same outside object. This condition holds when, for the last heap write statement $a.fld = b$ in p_{out} and the first heap read statement $c = d.fld$ in p_{in}, variables a and d may point to the same outside object. Second, we use an *extended recency abstraction* to check whether the loop iteration associated with p_{out} occurs earlier than the one associated with p_{in}. To achieve this, the analysis distinguishes objects by the loop iteration in which they are created. If both conditions hold, the object is considered to be properly shared between iterations; otherwise, it is reported as leaking. The analysis is formalized as a type and effect system described in Section 3.2. The analysis implementation (Section 3.3), employs a demand-driven context-free-language (CFL)-reachability formulation to explore p_{out} and p_{in} individually for each object created inside the loop, without requiring an initial whole-program analysis. This on-demand nature is particularly suitable for analyzing partial programs and components.

Key to the success of LeakChecker is the shift of focus from computing object liveness, which is very difficult to achieve precisely and efficiently for large programs, to the easier goal of identifying objects that flow out of a loop but never flow back in. This leak pattern is inspired by experience from dynamic leak detectors (e.g., [57, 74, 117]), where repeatedly-executed code regions and objects escaping from them are often shown to be the culprits. LeakChecker was implemented using the Soot analysis framework [109] and evaluated on eight large Java applications that have memory leaks. The tool found both known and unknown leaks in all applications, and reported comprehensive context information that can help to quickly identify their root causes. These promising initial findings demonstrate that the proposed static checking technique can be used successfully to find potential leaks during real-world software development. The contributions described in this chapter first appeared in [121].

3.1 Overview

This section presents an overview of the static analysis used in LeakChecker. Figure 3.1 shows a simple example adapted from the SPECjbb2000 benchmark. `Order` objects are created (line 5) and processed by a `Transaction` (line 6). Each transaction contains `Customers` (lines 12–15), and order processing saves the `Order` in field `curr` of the transaction (line 19) and adds it into one of the `Customer`s (line 22). Before an `Order` is processed, the transaction first displays its current order in `this.curr` (lines 26–30), which is set in the previous iteration. At the end of `display`, the current order is removed from `curr` (line 29). While the developer thinks this removal will make the `Order` object unreachable, he/she forgets to clean up references from the `Customer` object. These unnecessary references can lead to a severe memory leak.

Extended recency abstraction We first define a new abstraction for heap objects, called *extended recency abstraction* (ERA), which will be computed by the type and effect system described in Section 3.2. ERA extends the traditional notion of recency abstraction [12, 77] by distinguishing the objects that are carried over from one iteration to another from those that escape the loop but never flow back in. The terms "object" and "allocation site" will be used to refer to a static abstraction of heap objects (the *new* expression that created the object), while "instance" will refer to a run-time instance of the abstraction. The ERA for an object can have one of four abstract values: $\bar{\mathbf{o}}$ (outside), $\bar{\mathbf{c}}$ (current), $\bar{\mathbf{f}}$ (future), and $\top$ (unknown). For a particular loop l, an object whose ERA is $\bar{\mathbf{o}}$ with respect to l must be created outside

l; otherwise, the object is created inside the loop. If the object's ERA is $\bar{\mathbf{c}}$, the object must be *iteration-local.* To illustrate, consider a run-time iteration i of loop l where an instance of an allocation site a is created. If a's ERA is $\bar{\mathbf{c}}$, this guarantees that the instance must die before iteration i finishes. If a's ERA is $\bar{\mathbf{f}}$, this instance may escape iteration i, and if it does escape, it may be used in the loop (i.e., it may flow back) in a later iteration. Finally, a's ERA value of $\top$ implies that this instance may escape iteration i, and if it does escape, it will *not* be used in a later iteration.

Example We use a_k to denote the allocation site at line k of Figure 3.1. Consider ERAs with respect to the loop at line 3. The ERAs for `Transaction` a_2, array a_{10}, `Customer` a_{13}, and `Order` array a_{34} are all $\bar{\mathbf{o}}$, because they are created before the loop starts. The ERA for a_5 is $\bar{\mathbf{f}}$ because every instance of `Order` escapes the iteration in which it is created, and is used in the next iteration (at line 26). We are particularly interested in objects whose ERA is $\bar{\mathbf{f}}$ or $\top$, because iteration-local objects can never be leaks for the loop.

Transitive flows-out relationship To understand the reference flow, we compute a transitive flows-out relationship $d \succeq_g^* b$ between an object d whose ERA is $\bar{\mathbf{f}}$ or $\top$ and an object b whose ERA is $\bar{\mathbf{o}}$. This relationship indicates that a run-time instance of d may still be live after its creating iteration finishes, because it is inside a data structure that is saved in field g of an instance of the outside object b. In other words, it is the reference $b.g$ that prevents this instance of d from being garbage collected. Note that b must be the closest outside object in this reference path—there does not exist any other outside object c such that $d \succeq_{g'}^* c$ and c can be reached from b. In our example, there are two flows-out relationships for loop l: $a_5 \succeq_{curr}^* a_2$ and $a_5 \succeq_{elem}^* a_{34}$. Here, as typically done in prior work, *elem* denotes an artificial field

```
1        static void main(String[] args) {
2          Transaction t = new Transaction();
3          for (int i = 0; i < N; i++) {
4            t.display();
5            Order order = new Order(...);
6            t.process(order);
7          }
8        }
9        class Transaction {
10         Customer[] customers = new Customer[...];
11         Transaction() {
12           for (int i = 0; i < numCusts; i++) {
13             Customer newCust = new Customer(...);
14             customers[i] = newCust;
15           }
16         }
17         Order curr;
18         void process(Order p) {
19           this.curr = p;
20           Customer[] custs = this.customers;
21           Customer c = custs[p.custId];
22           c.addOrder(p);
23           ...// process order
24         }
25         void display() {
26           Order o = this.curr;
27           if(o != null) {
28               ... // display o
29               this.curr = null; //remove o
30           }
31         }
32       }
33       class Customer {
34         Order[] orders = new Order[...];
35         void addOrder(Order y) {
36           Order[] arr = this.orders;
37           arr[...] = y;
38         }
39       }
```

Figure 3.1: An example adapted from SPECjbb2000.

of the array object representing all array elements. For loop 12, only one flows-out relationship exists: $a_{13} \succeq^*_{elem} a_{10}$.

Transitive flows-in relationship We are also interested in how an object flows into the loop from a field of an outside object. Each flows-in relationship is of the form $d \preceq^*_g b$, indicating that the data structure containing an instance of d created from a previous iteration of the loop is carried over to the current iteration by field g of an instance of the outside object b. The ERA for b must be $\bar{\mathbf{o}}$, and we are interested only in d whose ERA is either $\bar{\mathbf{f}}$ or $\top$. Similarly, b must be the closest outside object in this reference path—there does not exist any other outside object c such that $d \preceq_{g'}^{*} c$ and c can be reached from b. In the example shown in Figure 3.1, one flows-in relationship ($a_5 \preceq_{curr}^{*} a_2$) exists for the loop at line 3, implying that an `Order` instance created in one iteration of the loop may be carried over to a later iteration through field `curr` of the `Transaction` instance.

Leak detection First, any object whose ERA is $\top$ is considered by LeakChecker as a potential leak because it may not flow back into the loop. Second, for each flows-out relationship $d \succeq_{g}^{*} b$ such that d's ERA is $\bar{\mathbf{f}}$ and b's ERA is $\bar{\mathbf{o}}$, if there does not exist a corresponding flows-in relationship $d \preceq_{g}^{*} b$, LeakChecker considers d as a potential leak because it is saved in a field from which it is never retrieved and used. This field unnecessarily maintains a reference that may keep instances of d from being garbage collected. In our example, the `Order` object a_5 will be recognized and reported as a leaking object, because it has two flows-out relationships but only one flows-in relationship. The reference edge from a_{34} to a_5 is a redundant edge because a_5 is never retrieved from this edge. While in this example the allocation sites are used directly to represent objects, LeakChecker is a context-sensitive analysis that uses a CFL-reachability formulation [100] to distinguish objects created by the same allocation site under different calling contexts. The analysis reports each leaking object (e.g., a_5), the redundant reference edge (e.g., $a_{34}.elem$), and the calling context under which the leaking object is saved through the edge (a by-product of the context-sensitive CFL-reachability computation).

LeakChecker soundness The first phase of the analysis computes ERA for each object and the two kinds of flow relationships, and the second phase matches these relationships to find leaks. The first phase is sound: any object that flows out of/into a loop through an outside object is correctly identified and the relationships are appropriately classified. The second phase is unsound, due to the matching of flows-in and flows-out relationships. For example, even if a flows-in relationship ($d \preceq_g^* b$) matches a flows-out relationship ($d \succeq_g^* b$), which indicates that d is not a leak, the two run-time instances of b (that an instance of d is added into and retrieved from, respectively) may not be the same. In addition, if g represents an array element (i.e., *elem*), the two heap locations may be different. While in the first case a must-alias analysis could be used to verify whether the two instances of b are the same, for the second case a practical static analysis often cannot precisely handle array indices. Although it may be possible to perform more precise analysis of array indices (e.g., [103]), these techniques are generally expensive and cannot scale to large applications.

Despite these sources of unsoundness, LeakChecker has not missed any known leaks in our studies on eight large applications. This is because in order to have severe effects on program performance, a leak has to exhibit *sustained* behaviors: an allocation site keeps creating instances that escape the loop and are no longer used. Very often these instances escape to an outside container, and this container is never read by later loop iterations. We have not seen any case where only a fixed set of elements are retrieved from the container (but a growing number of elements is untouched), which would cause LeakChecker to miss a sustained leak.

LeakChecker precision For each analyzed loop, the approach can precisely identify the objects that escape the loop through the references that are never read again in the loop. However, these references may be used later after the loop terminates, leading to an imprecise leak report. Hence, the precision of the analysis relies on the appropriate selection of loops to be checked. In the real-world (e.g., enterprise) applications that often suffer from leaks, this task is relatively easy; the case studies from Section 3.4 illustrate this point. For example, it is natural to select loops that create transactions in a database system, or that process events in an event-based system. To find leaks in an Eclipse plugin, we can use an artificial loop containing the body of a plugin interface method. Of course, it can be difficult to specify such a loop in certain applications, such as program analysis tools, because they often save all objects created in one phase and carry them over to another phase for further processing. However, these applications often do not run repeated tasks, and leaks may not have as significant of an impact in them as in business applications that exhibit repeated behavior.

Another source of imprecision is the lack of precise handling of destructive updates. For example, suppose an inside object flows to a field of an outside object and later this field is assigned `null` without being read in between. If the analysis cannot perform a strong update at the `null` assignment, the flows-out without a matching flows-in will be considered a symptom of a leak and a false warning will be reported. In practice, however, cases in which a reference is removed without being read are quite uncommon. Finally, an unused reference does not necessarily imply that the referenced object is no longer used. The object may be loaded from other (necessary) references and used in later iterations of the loop. In such a case, although the reported object is a false leak, this information is still useful because the redundant reference is worth inspecting and fixing.

$$
\begin{array}{llll}
\textsc{Variable} & b,\, c \in \mathbb{V} & & \\
\textsc{Alloc Site} & a \in \mathbb{A} & & \\
\textsc{Instance Field} & g \in \mathbb{F} & & \\
\textsc{Loop Label} & l \in \mathbb{L} & & \\
\textsc{Statement} & s \in \mathbb{S} & & \\
\multicolumn{4}{l}{\quad s ::= \quad b = c \mid b = \mathit{new}\ a \mid b = c.g \mid b.g = c \mid b = \mathsf{null} \mid} \\
\multicolumn{4}{l}{\qquad\qquad s\ ;\ s \mid \text{if } (*) \text{ then } s \text{ else } s \mid \text{while}^{l}\ (*) \text{ do } s} \\
\textsc{Iteration Count} & j & ::= 0 \mid 1 \mid 2 \mid \ldots & \in \mathbb{N} \\
\textsc{Iteration Map} & \nu & \in \mathbb{L} \rightarrow \mathbb{N} & \\
\textsc{Loop State} & \pi & ::= \langle l\ ,\ j \rangle & l\ \in \mathbb{L} \cup \{0\} \\
\textsc{Labeled Object} & \hat{o} & ::= o^{\pi} & \in \mathbb{P} \\
\textsc{Heap} & \sigma & \in \mathbb{P} \times \mathbb{F} \rightarrow \mathbb{P} \cup \{\bot\} & \\
\textsc{Environment} & \rho & \in \mathbb{V} \rightarrow \mathbb{P} \cup \{\bot\} & \\
\textsc{Heap Store Effect} & \Psi & ::= \emptyset \mid \Psi \cup \{\hat{o}_1 \succ_g^j \hat{o}_2\} & \\
\textsc{Heap Load Effect} & \Omega & ::= \emptyset \mid \Omega \cup \{\hat{o}_1 \prec_g^j \hat{o}_2\} &
\end{array}
$$

Figure 3.2: A `while` language: syntax and semantic domains.

3.2 Memory Leak Detection

This section formalizes the notion of a memory leak and formally defines the core analysis to find such leaks. First, we define a simple Java-like `while` language, its abstract syntax, and its operational semantics. Using this semantics, we formally define what we mean by a loop-related memory leak in an object-oriented program. Second, we present a type and effect system that abstracts the concrete objects and the flows-in/flows-out relationships. Finally, the memory leak detection algorithm is presented based on the abstract effects computed by the type and effect system.

3.2.1 A Type and Effect System

Language The abstract syntax of the `while` language and its semantic domains are shown in Figure 3.2. This language has all important features of an object-oriented language except function calls. They are eluded in this section to ease the formal development. In our implementation, call semantics and calling context sensitivity are modeled by the CFL-reachability formulation that treats the entry and the exit of the same method as a pair of balanced parentheses; this technical issue is elaborated later.

Each loop has a label l and an iteration count j that is incremented per iteration. Map ν maps each loop to its current iteration count. Each run-time object $\hat{o}$ is a regular object annotated with a loop state $\langle l, j \rangle$, indicating that the object is created in the j-th iteration of loop l. If $\hat{o}$ is created in the loop, its j is a positive number; otherwise, its j is always 0. Environment ρ maps a variable b to the heap object $\hat{o}$ pointed-to by b. Heap σ records when a instance field g of one heap object $\hat{o}_2$ points to another heap object $\hat{o}_1$. Both ρ and σ are augmented with $\perp$, representing a `null` value.

A concrete heap store effect captures a reference relationship $\hat{o}_1 \succ_g^j \hat{o}_2$, representing that (a reference to) $\hat{o}_1$ is saved in instance field g of object $\hat{o}_2$ in the j-th iteration of the loop. A concrete heap load effect captures a retrieval action $\hat{o}_1 \prec_g^j \hat{o}_2$ where $\hat{o}_1$ is obtained from field g of $\hat{o}_2$ in the j-th iteration. These two kinds of effects will be employed to compute the transitive flows-out and flows-in relationships.

Concrete Semantics Figure 3.3 shows the semantics of the language. A judgment $s, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \mathit{\Psi}, \mathit{\Omega}$ starts with a statement s, followed by loop iteration map ν, heap σ, and environment ρ. The execution of s terminates with an iteration map ν', heap σ', environment ρ', heap store effect set Ψ, and heap load effect set Ω.

Rules Assign, Comp, If-Else1, If-Else2 (not shown), and While are standard. In rule new, the loop state pair associated with each run-time object o is $\langle l, \nu(l) \rangle$, where l is the loop in which the object is allocated and $\nu(l)$ is the current iteration count of l. If the allocation site is not in any user-specified loop, $l = 0$, indicating that any object created here is an outside object for any loop. At each

$$\frac{\sigma' = \sigma[\lambda g.(\hat{o}.g \mapsto \bot)] \qquad \hat{o} = (\textbf{if } a \text{ is outside loop } l \textbf{ then } o^{\langle 0,0\rangle} \textbf{ else } o^{\langle l,\nu(l)\rangle})}{b = \text{new } a, \nu, \sigma, \rho \Downarrow \nu, \sigma', \rho[b \mapsto \hat{o}], \emptyset, \emptyset} \quad (\textsc{New})$$

$$b = c, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[b \mapsto \rho(c)], \emptyset, \emptyset \quad (\textsc{Assign})$$

$$b = \text{null}, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[a \mapsto \bot], \emptyset, \emptyset \quad (\textsc{Assign-Null})$$

$$\frac{\begin{array}{c}\hat{o}_c = \rho(c) \qquad \hat{o}_b = \sigma(\hat{o}_c.g)\\ \Omega = (\textbf{if } \hat{o}_b = \bot \textbf{ then } \emptyset \textbf{ else } \{\hat{o}_b \prec_g^{\nu(l)} \hat{o}_c\})\end{array}}{b = c.g, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[b \mapsto \hat{o}_b], \emptyset, \Omega} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c}\rho(c) = \hat{o}_c \qquad \rho(b) = \hat{o}_b\\ \Psi = (\textbf{if } \hat{o}_b = \bot \textbf{ then } \emptyset \textbf{ else } \{\hat{o}_b \succ_g^{\nu(l)} \hat{o}_c\})\end{array}}{c.g = b, \nu, \sigma, \rho \Downarrow \nu, \sigma[\hat{o}_c.g \mapsto \hat{o}_b], \rho, \mu, \Psi, \emptyset} \quad (\textsc{Store})$$

$$\frac{s_1, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega \qquad s_2, \nu', \sigma', \rho' \Downarrow \nu'', \sigma'', \rho'', \Psi', \Omega'}{s_1; s_2, \nu, \sigma, \rho \Downarrow \nu'', \sigma'', \rho'', \Psi \cup \Psi', \Omega \cup \Omega'} \quad (\textsc{Comp})$$

$$\frac{s_1, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega}{\text{if } (*) \text{ then } s_1 \text{ else } s_2, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega} \quad (\textsc{If-Else-1})$$

$$\frac{s, \nu[l \mapsto \nu(l)+1], \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega \qquad \text{while}^l\ (*) \text{ do } s, \nu', \sigma', \rho' \Downarrow \nu'', \sigma'', \rho'', \Psi', \Omega'}{\text{while}^l\ (*) \text{ do } e, \nu, \sigma, \rho \Downarrow \nu'', \sigma'', \rho'', \Psi \cup \Psi', \Omega \cup \Omega'} \quad (\textsc{While})$$

Figure 3.3: Concrete operational semantics.

store $c.g = b$ into the heap, an effect $\hat{o}_b \succ_g^j \hat{o}_c$ is recorded in Ψ, while at each load $b = c.g$ from the heap, an effect $\hat{o}_b \prec_g^j \hat{o}_c$ is recorded in Ω. Note that if the loop iteration count k of the retrieved object $\hat{o}_b$ is $< j$, the load retrieves an object created in a previous iteration. The operational definition of a memory leak is as follows:

Definition 3.2.1 (Leaking Object) *A run-time object $o^{\langle l,j\rangle}$ is the root of an escaping data structure during the execution of loop l if there exists a heap store effect $o^{\langle l,j\rangle} \succ_g^k q^{\langle 0,0\rangle} \in \Psi$. An object $r^{\langle l,j'\rangle}$ is a leaking object if*

$$r^{\langle l,j'\rangle} \succ^* o^{\langle l,j\rangle} \wedge \begin{pmatrix} {}^{(1)}\nexists(m > k) : o^{\langle l,j\rangle} \prec_g^m q^{\langle 0,0\rangle} \in \Omega\ \vee \\ {}^{(2)}\nexists(n > j' \wedge \hat{w} \in \mathbb{P}) : r^{\langle l,j'\rangle} \prec_{g'}^n \hat{w} \in \Omega \end{pmatrix}$$

where $\succ^*$ is the transitive closure of relation $\succ$.

Ext. Recency Abst.	$\tilde{j}$	$::=$	$\bar{\mathbf{o}} \mid \bar{\mathbf{c}} \mid \bar{\mathbf{f}} \mid \top$		$\in \mathbb{N}_{\mathbb{A}}$
Loop State Abst.	$\tilde{\pi}$	$::=$	$\langle l, \tilde{j} \rangle$	l	$\in \mathbb{L} \cup \{0\}$
Type	$\tilde{\tau}$	$::=$	$o^{\tilde{\pi}}$		$\in \mathbb{T}$
Type Environment	Γ	$\in$	$\mathbb{V} \rightarrow \mathbb{T} \cup \{\bot, \top\}$		
Type Heap	H	$\in$	$\mathbb{T} \times \mathbb{F} \rightarrow \mathbb{T} \cup \{\bot, \top\}$		
Abst. Store Effect	$\tilde{\Psi}$	$::=$	$\emptyset \mid \tilde{\Psi} \cup \{\tilde{\tau}_1 \succeq_g \tilde{\tau}_2\}$		
Abst. Load Effect	$\tilde{\Omega}$	$::=$	$\emptyset \mid \tilde{\Omega} \cup \{\tilde{\tau}_1 \preceq_g \tilde{\tau}_2\}$		

(a)

$$(1)\ \tilde{\tau} \sqsubset \hat{o} \quad \Leftrightarrow \quad \tilde{\tau} = \top \vee (\tilde{\tau} = \bot \wedge \hat{o} = \bot) \wedge(\hat{o}.o = \tilde{\tau}.o \wedge \tilde{\tau}.\tilde{\pi} \sqsubset \hat{o}.\pi)$$

$$(2)\ \tilde{\pi} \sqsubset \pi \quad \Leftrightarrow \quad \tilde{\pi}.l = \pi.l \wedge (\pi.l = 0 \vee \tilde{\pi}.\tilde{j} \sqsubset \pi.j)$$

$$(3)\ \tilde{j} \sqsubset j \quad \Leftrightarrow \quad (j = 0 \wedge \tilde{j} = \bar{\mathbf{o}}) \vee (j > 0 \wedge \tilde{j} \neq \bar{\mathbf{o}})$$

$$(4)\ \tilde{\Psi} \sqsubset \Psi \quad \Leftrightarrow \quad \forall(\hat{p_1} \succ_g^j \hat{p_2}) \in \Psi : \exists(\tilde{\tau}_1 \succeq_g \tilde{\tau}_2) \in \tilde{\Psi} : (\tilde{\tau}_1 \sqsubset \hat{p_1}) \wedge(\tilde{\tau}_2 \sqsubset \hat{p_2}) \wedge (\hat{p_1}.\pi.j \neq j \Rightarrow (\tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \vee \tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \top)))$$

$$(5)\ \tilde{\Omega} \sqsubset \Omega \quad \Leftrightarrow \quad \forall(\hat{p_1} \prec_g^j \hat{p_2}) \in \Omega : \exists(\tilde{\tau}_1 \preceq_g \tilde{\tau}_2) \in \tilde{\Omega} : (\tilde{\tau}_1 \sqsubset \hat{p_1}) \wedge(\tilde{\tau}_2 \sqsubset \hat{p_2}) \wedge (\hat{p_1}.\pi.j \neq j \Rightarrow (\tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \vee \tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \top)))$$

$$(6)\ \Gamma \sqsubset \rho \quad \Leftrightarrow \quad (\forall v \in \textsc{Dom}(\rho) : \Gamma(v) \sqsubset \rho(v))$$

$$(7)\ \mathsf{H} \sqsubset \sigma \quad \Leftrightarrow \quad (\forall \hat{o}.g \in \textsc{Dom}(\sigma) : \exists \tilde{\tau}.g \in \textsc{Dom}(\mathsf{H}) : \tilde{\tau} \sqsubset \hat{o} \wedge \mathsf{H}(\tilde{\tau}.g) \sqsubset \sigma(\hat{o}.g))$$

(b)

Figure 3.4: Abstract semantic domains: (a) types and abstract effects; (b) abstraction details.

This definition formally describes the leaking objects targeted by our approach. If an inside object o is assigned to a field g of an outside object q in iteration k, o is considered to be the root of an escaping data structure. Any inside object r that is transitively reachable from o (i.e., $r^{\langle l,j' \rangle} \succ^* o^{\langle l,j \rangle}$) is thus considered escaping. The escaping inside object r is a leaking object if (1) the root o of the data structure is leaking, that is, o is never loaded back in any later iteration through $q.g$, or (2) r itself never flows back to the loop in a later iteration. The second condition represents a scenario where a subset of this escaping data structure may flow back into the loop, but this subset does not include r. The formulation does not consider nested loops; although object flow across iterations of nested loops can be easily modeled, we have not found it useful in detecting real-world leaks.

Abstract Semantics We develop a type and effect system that uses an abstract semantics to conservatively approximate the two heap effects. Figure 3.4(a) shows the types and effects used to abstract the concrete semantics. Iteration counts j are abstracted by ERA $\tilde{j}$ (which can have four abstract values $\bar{\mathbf{o}}$, $\bar{\mathbf{c}}$, $\bar{\mathbf{f}}$, and $\top$), objects $\hat{o}$ are abstracted by types $\tilde{\tau}$, environment ρ is abstracted by type environment Γ, heap σ is abstracted by type heap H, and the two concrete effects in Ψ and Ω are abstracted by the abstract effects in $\tilde{\Psi}$ and $\tilde{\Omega}$. Type environment Γ and type heap H are augmented with $\bot$ and $\top$, which represent, respectively, no type and any type.

Details of how the concrete semantic domains are abstracted can be found in Figure 3.4(b), where $\sqsubset$ denotes the abstraction relation. In particular, rules (4) and (5) show how the two heap effects are abstracted. An abstract store effect $\tilde{\tau}_1 \preceq_g \tilde{\tau}_2 \in \tilde{\Omega}$ appropriately abstracts concrete effect $\hat{p_1} \prec_g^j \hat{p_2} \in \Omega$ if types $\tilde{\tau}_1$ and $\tilde{\tau}_2$ appropriately abstract $\hat{p_1}$ and $\hat{p_2}$, respectively. In addition, if this store happens in an iteration different from the one where $\hat{p_1}$ is created ($\hat{p_1}.\pi.j \neq j$), the ERA of $\tilde{\tau}_1$ must be either $\bar{\mathbf{f}}$ or $\top$.

The abstract semantics of our analysis is shown in Figure 3.5 and Figure 3.6. We discuss only a few important rules. When an allocation site is executed in an iteration, the ERA of the type is set to $\bar{\mathbf{c}}$, indicating that this object is created in the current iteration of the loop (rule TNew). In the beginning of each iteration, the abstract loop state $\tilde{\pi}$ of each type in environment Γ is incremented by rule TWhile using operator $\oplus$ (whose definition is shown in rule (6) of Figure 3.6). This sets the ERA of each existing loop object (created in previous iterations) to $\top$. If an existing object is iteration-local and cannot escape to the current iteration, its ERA will be updated back to $\bar{\mathbf{c}}$ when its allocation site is encountered again. If the object escapes

$$\frac{\Gamma' = \Gamma[b \mapsto \tilde{\tau}] \qquad \mathsf{H}' = \mathsf{H}[\lambda g.(\tilde{\tau}.g \mapsto \bot)] \qquad \tilde{\tau}.o = a \qquad \tilde{\tau}.\tilde{\pi} = (\textbf{if } a \text{ is outside loop } l \textbf{ then } \langle 0\ ,\ \bar{\mathbf{o}} \rangle \textbf{ else } \langle l\ ,\ \bar{\mathbf{c}} \rangle)}{\Gamma, \mathsf{H} \vdash b = \text{new } a : \Gamma', \mathsf{H}', \emptyset, \emptyset}$$

$$(\text{TNew})$$

$$\Gamma, \mathsf{H} \vdash b = c : \Gamma[b \mapsto \Gamma(c)], \mathsf{H}, \emptyset, \emptyset \quad (\text{TAssign}) \; \Gamma, \mathsf{H} \vdash b = \text{null} : \Gamma'[b \mapsto \bot], \mathsf{H}, \emptyset, \emptyset \quad (\text{TAssign-Null})$$

$$\frac{\begin{array}{c}\tilde{\tau_c} = \Gamma(c) \qquad \tilde{\tau_b} = \mathsf{H}(\tilde{\tau_c}.g) \\ \tilde{\tau_b}' = (\textbf{if } \tilde{\tau_b}.\tilde{\pi}.\tilde{j} = \top \textbf{ then } \tilde{\tau_b}.o^{\langle \tilde{\tau_b}.\tilde{\pi}.l, \bar{\mathbf{f}} \rangle} \textbf{ else } \tilde{\tau_b}) \qquad \tilde{\Omega} = (\textbf{if } \tilde{\tau_b}' \neq \bot \wedge \tilde{\tau_c} \neq \bot \textbf{ then } \{\tilde{\tau_b}' \preceq_g \tilde{\tau_c}\} \textbf{ else } \emptyset)\end{array}}{\Gamma, \mathsf{H} \vdash b = c.g : \Gamma[b \mapsto \tilde{\tau}_b'], \mathsf{H}, \emptyset, \tilde{\Omega}} \; (\text{TLoad})$$

$$\frac{\tilde{\tau_c} = \Gamma(c) \qquad \tilde{\tau_b} = \Gamma(b) \qquad \tilde{\Psi} = (\textbf{if } \tilde{\tau_b} \neq \bot \wedge \tilde{\tau_c} \neq \bot \textbf{ then } \{\tilde{\tau_b} \succeq_g \tilde{\tau_c}\} \textbf{ else } \emptyset)}{\Gamma, \mathsf{H} \vdash c.g = b : \Gamma, \mathsf{H}[\tilde{\tau_c}.g \mapsto \tilde{\tau_b}], \tilde{\Psi}, \emptyset} \; (\text{TStore})$$

$$\frac{\Gamma, \mathsf{H} \vdash s_1 : \Gamma', \mathsf{H}', \tilde{\Psi}, \tilde{\Omega} \qquad \Gamma', \mathsf{H}' \vdash s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi}', \tilde{\Omega}'}{\Gamma, \mathsf{H} \vdash s_1; s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi} \cup \tilde{\Psi}', \tilde{\Omega} \cup \tilde{\Omega}'} \quad (\text{TComp})$$

$$\frac{\Gamma, \mathsf{H} \vdash s_1 : \Gamma', \mathsf{H}', \tilde{\Psi}, \tilde{\Omega} \qquad \Gamma, \mathsf{H} \vdash s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi}', \tilde{\Omega}'}{\Gamma, \mathsf{H} \vdash \text{if } (*) \text{ then } s_1 \text{ else } s_2 : \Gamma' \uplus \Gamma'', \mathsf{H}' \uplus \mathsf{H}'', \tilde{\Psi} \cup \tilde{\Psi}', \tilde{\Omega} \cup \tilde{\Omega}'} \; (\text{TIf-Else})$$

$$\frac{\Gamma[\lambda v.(v \mapsto \Gamma(v).o^{\Gamma(v).\tilde{\pi} \oplus 1})], \mathsf{H} \vdash e : \Gamma, \mathsf{H}, \tilde{\Psi}, \tilde{\Omega}}{\Gamma, \mathsf{H} \vdash \text{while}^{j}\ (*) \text{ do } e : \Gamma, \mathsf{H}, \tilde{\Psi}, \tilde{\Omega}} \quad (\text{TWhile})$$

Figure 3.5: Type rules.

the loop and flows into the current iteration via a load, its ERA is then updated to $\bar{\mathbf{f}}$ by rule TLoad, indicating that the object is used in an iteration different from the one where it is created. If the object escapes the loop and never flows back in, its ERA will remain $\top$.

At each control flow merge point, type joins are performed (rule TIf-Else). The definition of the join operator $\uplus$ can be found in rules (1)–(5) in Figure 3.6. A finite-height type lattice can be defined based on the join operations, with $\top$ and $\bot$ as the maximum and minimum types in the lattice. Types with different allocation sites are not comparable. Because joining any type with $\top$ results in $\top$, LeakChecker reports a potential leak as long as there exists a control flow path in which an object escapes the loop but does not flow back. Abstract effects are recorded by rules TLoad and TStore. Rule TWhile describes a fixed-point computation—the analysis of the loop does not terminate until the type of each object does not change any more.

[Type Join]

$$(1)\ \tilde{\tau_1} \uplus \tilde{\tau_2} = \begin{cases} \tilde{\tau_1} & \text{if } \tilde{\tau_2} = \bot \\ \tilde{\tau_2} & \text{if } \tilde{\tau_1} = \bot \\ (\tilde{\tau_1}.o)^{\tilde{\tau_1}.\tilde{\pi} \uplus \tilde{\tau_2}.\tilde{\pi}} & \text{if } \tilde{\tau_1}.o = \tilde{\tau_2}.o \\ \top & \text{otherwise} \end{cases}$$

$$(2)\ \tilde{\pi_1} \uplus \tilde{\pi_2} = \begin{cases} \langle \tilde{\pi_1}.l\ ,\ \tilde{\pi_1}.\tilde{j} \uplus \tilde{\pi_2}.\tilde{j} \rangle & \text{if } \tilde{\pi_1}.l = \tilde{\pi_2}.l \\ \langle 0, 0 \rangle & \text{otherwise} \end{cases}$$

$$(3)\ \tilde{j_1} \uplus \tilde{j_2} = \begin{cases} \tilde{j_1} & \text{if } \tilde{j_1} = \tilde{j_2} \\ \top & \text{otherwise} \end{cases}$$

$$(4)\ \Gamma_1 \uplus \Gamma_2 = \Gamma_3,\ \text{where } \forall v \in \textsc{Dom}(\Gamma_3),\ \Gamma_3(v) = \begin{cases} \Gamma_1(v) & \text{if } v \in \textsc{Dom}(\Gamma_1) \text{ and } v \notin \textsc{Dom}(\Gamma_2) \\ \Gamma_2(v) & \text{if } v \in \textsc{Dom}(\Gamma_2) \text{ and } v \notin \textsc{Dom}(\Gamma_1) \\ \Gamma_1(v) \uplus \Gamma_2(v) & \text{if } v \in \textsc{Dom}(\Gamma_1) \cap \textsc{Dom}(\Gamma_2) \end{cases}$$

$$(5)\ \mathsf{H}_1 \uplus \mathsf{H}_2 = \mathsf{H}_3,\ \text{where } \forall \tilde{\tau}.g \in \textsc{Dom}(\mathsf{H}_3),$$

$$\mathsf{H}_3(\tilde{\tau}.g) = \begin{cases} \mathsf{H}_1(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \textsc{Dom}(\mathsf{H}_1) \text{ and } \tilde{\tau}.g \notin \textsc{Dom}(\mathsf{H}_2) \\ \mathsf{H}_2(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \textsc{Dom}(\mathsf{H}_2) \text{ and } \tilde{\tau}.g \notin \textsc{Dom}(\mathsf{H}_1) \\ \mathsf{H}_1(\tilde{\tau}.g) \uplus \mathsf{H}_2(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \textsc{Dom}(\mathsf{H}_1) \cap \textsc{Dom}(\mathsf{H}_2) \end{cases}$$

[Operator $\oplus$]

$$(6)\ \tilde{\pi} \oplus 1 = \begin{cases} \tilde{\pi} & \text{if } \tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \\ \langle \tilde{\pi}.l\ ,\ \top \rangle & \text{otherwise} \end{cases}$$

Figure 3.6: Join operations on types and domains.

Example Consider the following simple example:

$b = \texttt{new}\ o_1;\ \texttt{while}^l\ (...)\ \texttt{do}\ \{$

$c = \texttt{new}\ o_2;\ d = \texttt{new}\ o_3;\ e = \texttt{new}\ o_4;$

$m = b.g;\ \texttt{if}(...)\ n = m.h;$

$\texttt{if}(...)\ \{b.g = d;\ d.h = e;\}\ \}$

When our analysis terminates, the abstract semantic domains contain the following values:

$$\begin{aligned} \Gamma &= [b \mapsto o_1^{\langle 0, \bar{\mathbf{o}} \rangle},\ c \mapsto o_2^{\langle l, \bar{\mathbf{c}} \rangle},\ d \mapsto o_3^{\langle l, \bar{\mathbf{f}} \rangle}, \\ &\quad e \mapsto o_4^{\langle l, \top \rangle},\ m \mapsto o_3^{\langle l, \bar{\mathbf{f}} \rangle},\ n \mapsto o_4^{\langle l, \top \rangle}], \\ \mathsf{H} &= [o_1^{\langle 0, \bar{\mathbf{o}} \rangle}.g \mapsto o_3^{\langle l, \bar{\mathbf{f}} \rangle},\ o_3^{\langle l, \bar{\mathbf{f}} \rangle}.h \mapsto o_4^{\langle l, \top \rangle}], \\ \tilde{\Psi} &= \{o_3^{\langle l, \bar{\mathbf{f}} \rangle} \succeq_g o_1^{\langle 0, \bar{\mathbf{o}} \rangle},\ o_4^{\langle l, \top \rangle} \succeq_h o_3^{\langle l, \bar{\mathbf{f}} \rangle}\}, \\ \tilde{\Omega} &= \{o_3^{\langle l, \bar{\mathbf{f}} \rangle} \preceq_g o_1^{\langle 0, \bar{\mathbf{o}} \rangle},\ o_4^{\langle l, \top \rangle} \preceq_h o_3^{\langle l, \bar{\mathbf{f}} \rangle}\}. \end{aligned}$$

The ERAs for o_1, o_2, o_3, and o_4 are $\bar{\mathbf{o}}$, $\bar{\mathbf{c}}$, $\bar{\mathbf{f}}$, and $\top$, respectively. Here o_1 is an outside object and o_2 is an iteration-local object. Both o_3 and o_4 may escape the loop, and thus their ERA is changed to $\top$ by rule TWhile. If o_3 escapes, it must be used in a later iteration (via $m = b.g$), and thus, its ERA is updated to $\bar{\mathbf{f}}$. While o_4's ERA is also updated to $\bar{\mathbf{f}}$ at load $n = m.h$, it is changed back to $\top$ by the environment join at the end of the first `if` statement (because there exists a CFG path in which it does not flow back into the loop).

3.2.2 Leak Detection

Based on the abstract load and store effects computed by the type and effect system, leaks can be detected as follows.

Definition 3.2.2 (Flows-out Relation $\succeq_g^*$ and Flows-in Relation $\preceq_g^*$) *Suppose* $\succeq^*$ *and* $\preceq^*$ *are the transitive closures of relations* $\succeq$ *and* $\preceq$*, respectively. A pair* $(\tilde{\tau}_1, \tilde{\tau}_2) \in$ *flows-out relation* $\succeq_g^*$ *if*

$\tilde{\tau}_1.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau}_2.\tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \wedge \exists \tilde{\tau}_3 : \tilde{\tau}_3.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau}_1 \succeq^* \tilde{\tau}_3 \wedge \tilde{\tau}_3 \succeq_g \tilde{\tau}_2.$

Similarly, a pair $(\tilde{\tau}_1, \tilde{\tau}_2) \in$ *flows-in relation* $\preceq_g^*$ *if*

$\tilde{\tau}_1.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau}_2.\tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \wedge \exists \tilde{\tau}_3 : \tilde{\tau}_3.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau}_1 \preceq^* \tilde{\tau}_3 \wedge \tilde{\tau}_3 \preceq_g \tilde{\tau}_2.$

As discussed in Section 3.1, $\tilde{\tau}_1 \succeq_g^* \tilde{\tau}_2$ if $\tilde{\tau}_1$ represents an inside object, $\tilde{\tau}_2$ represents an outside object, and there exists a sequence of store effects that connects them. Field g is the field of $\tilde{\tau}_2$ through which the leaking data structure is saved. Based on the definitions of $\succeq_g^*$ and $\preceq_g^*$, we give the following definition of a memory leak.

Definition 3.2.3 (Memory Leak) *An object* o *is a leaking object, if it has a type* $\tilde{\tau}$ *such that*

$\tilde{\tau}.\tilde{\pi}.\tilde{j} = \top \vee (\tilde{\tau}.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \wedge \exists (\tilde{\tau}, \tilde{\tau}') \in\succeq_g^*\colon (\tilde{\tau}, \tilde{\tau}') \notin\preceq_g^*)$

In the above example, object o_4 is a leaking object, because its ERA is $\top$.

3.3 Implementation

Calls and Calling Context LeakChecker is implemented based on the Soot Java program analysis framework [109]. A demand-driven CFL-reachability formulation of points-to analysis [100] is used to identify leaking objects interprocedurally and with modeling of calling context. In such a formulation, program semantics is encoded as a flow graph in which nodes represent variables and edges represent propagation of object references. Points-to relationships are determined by traversing the graph, and flows-in/out information is derived from them. For example, at a heap store statement $c.g = b$, the points-to sets of c and b are computed on demand to identify flows-out pairs of objects. The analysis is calling-context-sensitive in that edge labels representing interprocedural control flow—i.e., method calls and returns—along a traversed graph path are required to satisfy a matching parentheses property (defined by a context-free language, thus the term CFL-reachability). With this addition, objects are distinguished not only by their allocation sites and their ERA, but also by their calling contexts. When a leaking object is detected, its allocation site, the field through which it escapes, and the calling context under which it escapes are all reported.

Flow into Library Methods Many popular Java data structures, such as `HashMap` and `ArrayList`, use arrays to store objects. These arrays are read in certain operations that are not meant to retrieve objects. For example, in method `HashMap.put`, entries that match the hashcode of the given key are read from the array to determine whether the key already exists. If a loop calls `put` and these reads are treated as regular object retrievals, LeakChecker may miss some leaks. To avoid this, we distinguish application code and library code, and use a stronger condition

Case	*Mtds*	*Stmts*	*Time* (s)	*LO*	*LS*	*FP*	*FPR*
SPECjbb2000	4717	97387	82	95	21	8	38.1%
Eclipse Diff	26300	400647	1638	309	7	3	42.9%
Eclipse CP	30487	466307	1000	123	7	4	57.1%
Mckoi	20373	342041	1985	450	18	17	94.4%
MySQL Connector/J	11868	210053	1059	181	15	9	60%
log4j	3385	62568	35	10	4	0	0%
FindBugs	3817	70177	82	72	9	5	55.6%
Derby	8661	147899	700	165	8	4	50%

Table 3.1: Analysis results.

to identify leaking objects: if an object is read from the heap by a library class, a flows-in relationship (as defined earlier) exists only when the object is returned to the application code, accounting for calling context. Hence, even if the array is read in `HashMap.put`, LeakChecker does not generate a flows-in relationship because the loaded object is not returned by the method. This treatment is used not only for array objects, but also for objects that are not of an array type. This stronger condition for leak identification is applied by our approach to all library methods in the standard Java libraries.

Pivot Mode For any two leaking objects o_1 and o_2 such that $o_1 \succeq^* o_2$, object o_2 is more likely to be the root of a leaking data structure, and object o_1 can not be garbage collected as long as o_2 is unnecessarily kept alive. In such a case, the leak can be understood and fixed by just inspecting o_2 and removing its unnecessary reference(s). LeakChecker provides a *pivot mode* under which leaking objects such as o_1 are omitted from the leak report. The experiments described in Section 3.4 use this mode.

3.4 Empirical Evaluation

We evaluated LeakChecker on eight large programs. Some programs (*log4j*, *FindBugs*, and *Derby*) have never been studied before, while leaks in others were discussed in existing work [15, 106, 117]. These programs cover a variety of domains, including enterprise trading, software development, database management, logging, and static program analysis.

3.4.1 Summary of Results

All experiments were performed on a machine with a 3.4 GHz Quad Core Intel i7-2600 processor, and the analysis was run with a maximum Java heap size of 4 GB. Characteristics of the studied programs and a summary of leak detection results are shown in Table 3.1. The table shows the number of reachable methods in the call graph (Mtds), the number of Soot's Jimple statements in these methods (Stmts), LeakChecker's analysis time (Time) in seconds, the number of context-sensitive allocation sites in analyzed loops (LO), the number of reported context-sensitive leaking allocation sites (LS), the number of false positives (FP), and the false positive rate (FPR = FP / LS). For each studied program, one suspicious loop was specified for checking.

Due to the client-driven nature of the analysis (checking user-specified loops), LeakChecker is able to quickly detect leaks for all the applications, including large programs such as Eclipse. The approach does not generate many leak warnings, so we verified each warning manually. With detailed leak reports, we pinpointed the root cause of the leak and fixed the underlying defect in less than 2 hours for each report. LeakChecker's average false positive rate is 49.8%, which indicates that is may be suitable for practical use.

3.4.2 Case Studies

We performed case studies on all eight programs, but due to space limitations only six of them are discussed below.

SPECjbb2000 SPECjbb2000 is a transaction-based system. In this program, there is a `TransactionManager` class that runs different types of transactions, and the transaction-creating method is only a few calls away from the `main` method. It contains a loop that, in each iteration, retrieves a command from an input map, and then creates and runs a transaction whose type corresponds to the command received. Thus it is natural to apply LeakChecker on this loop. The tool reported 5 allocation sites (corresponding to 21 context-sensitive allocation sites), among which 4 (under 6 different calling contexts) can be immediately excluded because the outside heap locations they flow to are overwritten in each iteration of the loop. The remaining site allocates `longBTreeNode` objects. These objects are created to hold element objects, when the elements are added to a `longBTree` container data structure. We focused our efforts on these `longBTreeNode` objects.

We found that the calling contexts are particularly useful in understanding the root cause of this problem. In the report, `longBTreeNode` objects are shown to be created under 15 different calling contexts. We first examined the top call sites in these calling contexts. There are only 3 distinct top call sites, and they are all in the method enclosing the specified loop. These call sites correspond to the processing of 3 types of commands: `new_order`, `multiple_orders`, and `payment`. The last one, the only one that is irrelevant to the leak, indicates that History objects, a representation of payment history, are saved in the long-lived Warehouse objects. However, we found that every time a new History object is added, the oldest one will be removed. Through this particular leaking context, these History objects cannot cause constant increase in memory footprint.

Excluding payment commands, we were left with 13 calling contexts, all related to processing of new orders. They indicate that Order objects are kept alive unnecessarily by longBTreeNode objects. One such relevant context-sensitive allocation site is shown below:

```
* Leaking Object (longBTreeNode; createlongBTreeNode(...), ln 102)
  Context - at longBTreeNode.Insert(long,Object), ln 760
            at longBTree.put(long,Object), ln 1521
            at District.addOrder(Order), ln 264
            at NewOrderTransaction.process(), ln 293
            at TransactionManager.go(), ln 296
* Outside Object (longBTree; createLongBTree(...), ln 790)
  Context - at District.initDistrict(short,byte), ln 184
            at District.createDistrict(...), ln 100
            at Warehouse.setUsingRandom(short), ln 396
            at Company.loadWarehouseTable(), ln 761
* Heap Write (r0.<longBTree: longBTreeNode root> = r5)
  Context - at longBTree.put(long,Object), ln 1521
            at District.addOrder(Order), ln 264
            at NewOrderTransaction.process(), ln 293
            at TransactionManager.go(), ln 296
```

Order object is stored in the newly created longBTreeNode object, which is inserted into the longBTree and later itself becomes the root of the longBTree:

```
btree.root = btree.root.Insert(key, order)
```

The longBTree object is stored in a field of a long-lived outside District object to represent orders processed through this district. Thus, Order objects are kept alive and leaking.

Eclipse Diff Eclipse is an IDE that allows plugins to be added into a unified platform. Plugins are usually developed separately, but they can interact with each other at run time. It is often unclear to developers how one plugin could be affected by others. For example, the leak in this case manifests only after the structures of two large JAR files are compared multiple times by plugin `org.eclipse.compare`. Files selected for comparison are represented by `ISelection` objects, which are passed into a `runCompare` method, the entry method of this plugin. We created an artificial loop in which `runCompare` is called, and applied the analysis on it.

LeakChecker reported 7 context-sensitive leaking allocation sites. Three of them are for temporary GUI objects (e.g., a temporarily shown dialog to indicate progress of computation) and can be immediately discarded. The rest of them all point to one allocation site that creates `HistoryEntry` objects. The associated contexts indicate that these objects are created when `History.addEntry` is called. `History` records the history of opened editors in a list of `HistoryEntry` objects, and the editors are used to show the comparison results. Calling `runCompare` multiple times would lead to the creation of multiple history entry objects. These objects are added to the list, but not properly cleared. Note that `History` is a class in the platform, and thus it is very difficult for developers to find and fix the bug (in fact, the root cause of this bug was found almost one year after it was reported). LeakChecker started from a code stub that uses the compare plugin, and quickly reported the root cause. To detect this leak using a dynamic analysis, a full-fledged executable program has to be developed to automatically select items in the GUI and trigger the comparison action. This task could be quite challenging for programmers without Eclipse GUI programming/testing experience.

Mckoi Mckoi [66] is an open-source database system. It has a memory leak when used as an embedded application. It is leaking because `DatabaseSystem` objects are kept alive by running threads. We created a simple client that repeatedly establishes a database connection and closes it. When we first ran LeakChecker on the program, there was only one leaking object reported. The reported `LocalBootable` object is a singleton object created only at the first time a connection is established, and the only outside object to which it escapes is the (outside) JDBC driver object. This is a false warning, because at run time it is guaranteed to be one instance of `LocalBootable` created per connection, which cannot be understood by LeakChecker.

LeakChecker fails to detect the leak because threads are not explicitly modeled. To solve the problem, threads that never terminate should be treated as outside objects. However, this is non-trivial as it is generally undecidable to determine whether a thread would terminate. As a workaround, we tag an object as an outside object if (1) it is a thread object (an instance of `java.lang.Thread`) regardless of whether or not it may terminate, and (2) method `start` has been called on this object. After this new modeling was employed, 18 context-sensitive allocation sites were reported. To verify whether they are true leaks, we manually examined the `run` method of each (outside) thread object and found that (1) most of the reported sites are false positives because they escape to threads that must terminate; and (2) the allocation site of `DatabaseSystem` leads to the root cause of the leak, related to non-terminating thread `DatabaseDispatcher`. Due to the lack of a thread termination analysis, we saw a high false positive rate for this program.

log4j log4j [6] is a logging library for Java. When a client application uses `JDBCAppender` to write log messages to a remote database, the memory usage increases significantly. We created a simple program that mimics such a client by sending multiple log requests. Four context-sensitive allocation sites were reported as leaks, all of them related to a list called `removes` in `JDBCAppender`. We inspected the code and found that (1) log requests are first added to a buffer list; (2) they are retrieved (but not removed) one by one from the list for processing, and added to removes list afterwards; and (3) at the end of one bulk processing, all the request objects in removes are removed from buffer. However, the removes list itself is never cleared, leading to the leak.

FindBugs FindBugs [41] is a static analysis tool in which bug detectors are organized as plugins, while the base framework provides common functionality (e.g., parsing of class files). A leak is exposed when FindBugs2.execute is called many times to analyze a large number of JAR files. We created a loop that iterates over a list of JAR files and parses the class files contained in each JAR.

LeakChecker reported 9 leaking allocation sites, 5 of which were obviously irrelevant to the leak. Objects created at these sites are stored into HashMap objects reachable from a global DescriptorFactory object. Because the HashMaps are cleared at the end of the analysis of each JAR file, no objects can be leaking through them. These (false) warnings were reported due to the lack of precise handling of destructive updates. The remaining 4 sites all point to a long-lived IdentityHashMap object, to which a number of MethodInfo objects are added. However, these MethodInfo objects are never used or removed. After inspecting these 4 sites, one can easily fix the leak by appropriately clearing the IdentityHashMap.

Derby In Apache Derby 10.2.1.6 [5], a leak can be seen if a Statement or a ResultSet is not closed after being used in client/server mode. We created a simple loop that executes one SQL query per iteration but does not call close on Statement or ResultSet. Eight leaking allocation sites were reported. Half of them are related to a Hashtable in SectionManager that saves ResultSet objects—these objects are never retrieved, causing the memory leak. All other reported allocation sites are related to saving `Section` objects in a `Stack`. These are false warnings because at the reported sites only one object instance can be created and escape the loop, due to use of the singleton pattern.

3.4.3 Experience Summary

Our case studies demonstrate that deep implementation knowledge is not required for effective use of LeakChecker. In the wide variety of programs we studied, loops relevant to leaking behavior can be easily identified/created, even for users unfamiliar with the program. The specified loop can serve as a *client* that interacts with a complicated system. To pinpoint bugs in database systems (e.g., Derby), we only need to create a loop that performs database queries. Similarly, for a plugin-based system such as Eclipse, we can perform checks on plugins, and leak detection can be done regardless of whether the bug is in the plugin or in the base system. This is very useful because it allows testers or performance experts to quickly create the necessary setup to check the code, without the need to dig into the details of a large system, or create leak-triggering test cases. Of course, there may be scenarios where the selection of the loop to be checked is not as straightforward, and additional considerations may be needed: e.g., identifying loops that are likely to frequently invoke important subcomponents of the analyzed component, or using application-specific knowledge to focus on loops whose frequent execution is expected under realistic usage scenarios. In cases where actual run-time frequency information is available, the checking effort could be targeted toward the most frequently executed loops.

A leak report generated by LeakChecker contains both leaking allocation sites and the specific loops they escape from. Understanding why a reported object never flows back into a loop is often sufficient to locate the cause of a defect. Relevant code in the program usually can be easily identified as LeakChecker also reports the calling contexts and escaping store statements for each leaking object. Our experience indicates that, given a detailed LeakChecker report, the developer effort to identify the root cause of a leak is typically small.

In the experiments, most of the false positives were due to internal constraints used by developers to prevent multiple instances of a loop object from escaping the loop. In future work, it is worth investigating how LeakChecker can be extended to detect such code patterns.

3.5 Summary

This chapter presents LeakChecker, the first practical static memory leak detector for Java. Leak detection is based on the observation that an event loop is often the place where severe leaks occur, and these leaks are commonly caused by objects outside the loop keeping unnecessary references to objects created inside the loop. Such a loop often iterates a large number of times, causing these references to accumulate and degrade program performance. LeakChecker uses a novel static analysis to identify such unnecessary references and reports leaks with sufficient information that can quickly help the developer find the root causes and come up with the necessary fixes. We have implemented the analysis and evaluated it on eight large programs with leaks. The experimental results show that LeakChecker successfully finds leaks in each of them and the false positive rate is reasonably low. These promising initial results strongly suggest that the proposed technique can be used in practice to help programmers find and fix memory leaks during development. Future work can investigate algorithmic refinements to achieve higher precision (e.g., through modeling of destructive updates). Approaches to identify suspicious loops to be checked—for example, using structural information extracted from the code, or frequency information from run-time execution—are also of significant interest.

CHAPTER 4: LeakDroid: Systematic Testing for Resource Leaks in Android Applications

Android devices currently lead the smartphone marketplace in the United States [25] and similar trends can be seen in other countries. Android also has significant presence in one of the fastest-growing segments of the computing landscape: tablets (e.g., Google Nexus 7/10, Samsung Galaxy Tab/Note) and media-delivery devices (e.g., Amazon Kindle Fire, Barnes & Noble Nook HD). The widespread use of these mobile devices poses great demands on software quality. However, meeting these demands is very challenging. Both the software platforms and the accumulated developer expertise are immature compared to older areas of computing (e.g., desktop applications and server software). The available research expertise and automated tool support are also very limited. It is critical for software engineering researchers to contribute both foundational approaches and practical tools toward higher-quality software for mobile devices.

The features of Android devices and the complexity of their software continue to grow rapidly. This growth presents significant challenges for software correctness and performance. In addition to traditional defects, a key consideration are defects related to the *limited resources* available on these devices. One such resource is the memory. In Android's Dalvik Java virtual machine (VM) the available heap memory typically ranges from 16 MB to 64 MB. In contrast, in a desktop/laptop VM there are many hundreds of MB available in the heap. Examples of other limited resources include threads, binders (used for Android's inter-process communication), file handles, and bitmaps. An application that consumes too many resources can lead to slowdowns, crashes, and negative user experience.

Resource management is challenging and developers are made aware of this problem in basic Android training materials [102] and through best-practice guidelines (e.g., [33]), with the goal of avoiding common pitfalls related to resource usage. A typical example of such a problem is a *resource leak*, where the application does not release some resource appropriately.

Examples. We studied a version of `ConnectBot` [26], an SSH client with more than a million installs according to the Google app store. The code contains a leak: when the application repeatedly connects with a server and subsequently disconnects from it, bitmaps are leaked, which eventually leads to a crash. As another example, we studied a version of the `APV` PDF viewer [7] (which also has more than a million installs) and discovered a leak, occurring when a PDF file is opened and then later the BACK button is pressed to close the file. In our experience, leak defects are related to diverse categories of events such as screen rotation, switching between applications, pressing the BACK button, opening and closing of files, and database accesses. If application users observe crashes and slowdowns due to such leaks, they may uninstall the application and submit a negative review/rating in the application marketplace.

Challenges. Even though resource leaks can significantly affect software reliability and user experience, *there does not exist a comprehensive and principled approach for testing for such leaks.* The large body of work on dynamic analysis of memory leaks (e.g., [14, 24, 29, 38, 58, 74, 117, 118]) has the following purposes: (1) observe run-time symptoms that indicate a potential leak, and (2) provide information to diagnose the root cause of the defect (e.g., by identifying fast-growing object subgraphs on the heap). However, all these approaches fail to address one crucial question: how can we generate the test data that triggers the leaking behavior? Answering this question for arbitrary applications is difficult, because leaks may be related to a wide variety of program functionality. However, as discussed later, a key insight of our approach is that leaks in Android applications *often follow a small number of behavioral patterns*, which makes it possible to perform systematic, targeted, and effective generation of test cases to expose such leaks.

Each Android application is centered around a graphical user interface (GUI), defined and managed through standard mechanisms provided by the Android platform. Some leak patterns are directly related to aspects of these mechanisms—for example, the management of the lifetime for an *activity* [63], which is an application component that interacts with the user. Such leaks cannot be exposed through unit testing because of the complex execution context managed by the platform (e.g., lifetime and internal state of GUI widgets, persistent state, etc.), as well as the complicated interactions due to callbacks from the platform to the application. It is essential to develop a system-level GUI-centric approach for testing for Android leaks, with sequences of GUI events being triggered to exhibit the leak symptoms. At present, no such approach exists.

Our proposal. We propose a *novel and comprehensive approach for testing for resource leaks in Android software.* This leak testing is similar to traditional GUI-model-based testing. Finite state machines and other related GUI models have been used in a number of testing techniques (e.g., [44,67,69,70,112,113]), including recent work on testing for Android software [1, 2, 11, 105, 124]. Given a GUI model, test cases can be generated based on various coverage criteria (e.g., [69]). As with these existing approaches, we consider GUI-model-based testing, but focused specifically on coverage criteria aimed at resource leaks. We define the approach based on a GUI model in which nodes represent Android activities and edges correspond to user-generated and framework-generated events. The same approach can be used with other GUI models for Android (e.g., event-flow graphs [4, 67]) in which paths in the model correspond to event sequences.

The proposed coverage criteria are based on the notion of *neutral cycles*. A neutral cycle is a sequence of GUI events that should have a "neutral" effect—that is, it should not lead to increases in resource usage. Such sequences correspond to certain cycles in the GUI model. Through multiple traversals of a neutral cycle (e.g., rotating the screen multiple times; repeated switching between apps; repeatedly opening and closing a file), a test case aims to expose leaks. This approach directly targets several common leak patterns in Android applications, and successfully uncovers 18 resource leak defects in a set of eight open-source Android applications used in our studies.

Contributions. The contributions of this work are:

- *Test coverage criteria:* We define several test coverage criteria based on different categories of neutral cycles in the GUI model. This approach is informed by knowledge of typical causes of resource leaks in Android software.

- *Test generation and execution:* We describe LeakDroid, a tool that generates test cases to trigger repeated execution of neutral cycles. When the test cases are executed, resource usage is monitored for suspicious behaviors.

- *Evaluation:* We evaluate the approach on several Android applications. The evaluation demonstrates that the proposed test generation effectively uncovers a variety of resource leaks.

- *Case studies:* We present case studies of leak defects exposed by the approach. This provides insights into the root causes of these leaks, which may be useful for future work on testing and debugging of Android software.

These contributions are in the emerging and important area of software testing for mobile devices. The proposed testing approach adds to a growing body of research on improving the reliability and performance of Android applications. The experimental evaluation and case studies contribute to better understanding of certain classes of defects in such applications, and highlight open problems for future investigations. The work described in this chapter first appeared in [122].

4.1 Background

4.1.1 Android Activities

An Android *activity* is an application component that manages a hierarchy of GUI widgets and uses them to interact with the user. An activity has a well-defined lifecycle, and developers can define callback methods to handle different stages of this lifecycle (Figure 4.1). When an activity is started, `onCreate` is called on it by the Android runtime. The activity becomes ready to terminate after `onDestroy` is called on it. The loop defined by `onStart` and `onStop` is the *visible* lifetime. Between calls to these two callback methods, the activity is visible to users. Finally, the innermost loop `onResume`/`onPause` defines the *foreground* lifetime, in which the activity is on the foreground and can interact with the user. A resource leak can be introduced

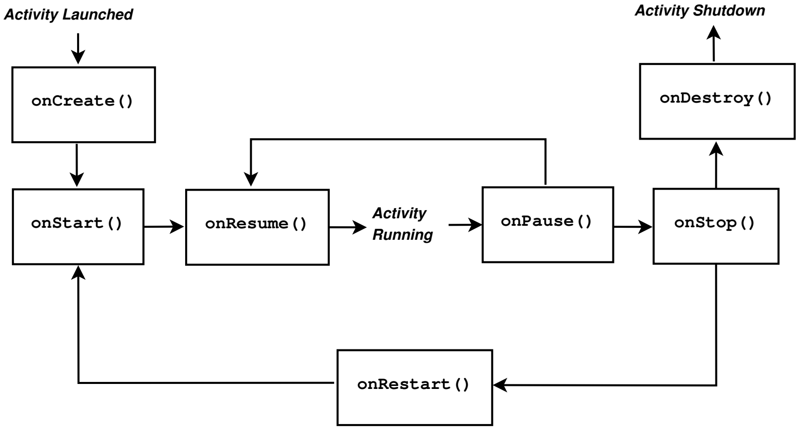
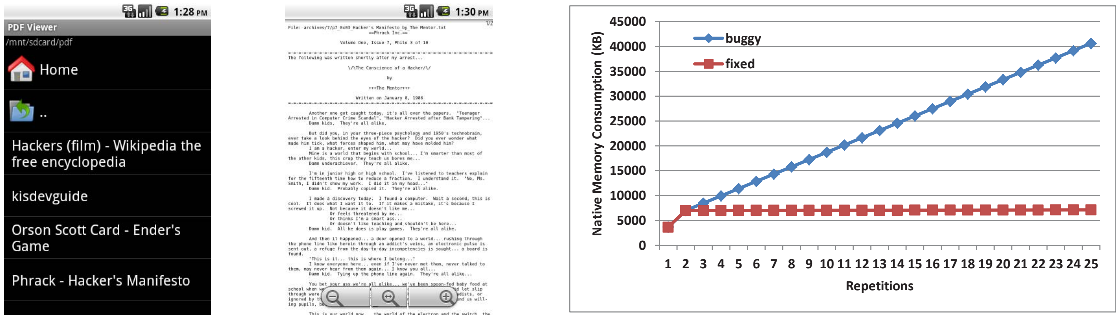


Figure 4.1: Activity lifecycle.

if a certain resource is allocated at the beginning of a lifetime (e.g., in `onCreate`) but not reclaimed at the end (e.g., in `onDestroy`). Thus, one desirable property of a test generation strategy is to cover these three pairs of lifecycle callback methods, especially because prior studies of Android applications [51] indicate that defects are often caused by incorrect handling of the activity lifecycle. An application usually has several activities, and transitions between them are triggered through GUI events. When an application is launched, a start activity is first displayed.

Example. Figure 4.2(a) shows `ChooseFileActivity` in the `APV` PDF viewer application [7], displayed when the application is launched. The activity shows a list of files and folders. A PDF file can be selected by tapping on the corresponding list item, and the file is displayed in `OpenFileActivity` as shown in Figure 4.2(b). These two activities correspond to two different states of the application; each has its own visible GUI elements and allowed GUI events. The reverse transition occurs through the hardware BACK button. This transition closes the file and returns to the previous



```
File: archives/7/p7_0x03_Hacker's Manifesto_by_The Mentor.txt
                        ==Phrack Inc.==

                 Volume One, Issue 7, Phile 3 of 10

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
The following was written shortly after my arrest...

                    \/\The Conscience of a Hacker/\/

                                   by

                             +++The Mentor+++

                       Written on January 8, 1986
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

        Another one got caught today, it's all over the papers.  "Teenager
Arrested in Computer Crime Scandal", "Hacker Arrested after Bank Tampering"...
        Damn kids.  They're all alike.

        But did you, in your three-piece psychology and 1950's technobrain,
ever take a look behind the eyes of the hacker?  Did you ever wonder what
made him tick, what forces shaped him, what may have molded him?
        I am a hacker, enter my world...
        Mine is a world that begins with school... I'm smarter than most of
the other kids, this crap they teach us bores me...
        Damn underachiever.  They're all alike.

        I'm in junior high or high school.  I've listened to teachers explain
for the fifteenth time how to reduce a fraction.  I understand it.  "No, Ms.
Smith, I didn't show my work.  I did it in my head..."
        Damn kid.  Probably copied it.  They're all alike.

        I made a discovery today.  I found a computer.  Wait a second, this is
cool.  It does what I want it to.  If it makes a mistake, it's because I
screwed it up.  Not because it doesn't like me...
                Or feels threatened by me...
                Or thinks I'm a smart ass...
                Or doesn't like teaching and shouldn't be here...
        Damn kid.  All he does is play games.  They're all alike.

        And then it happened... a door opened to a world... rushing through
the phone line like heroin through an addict's veins, an electronic pulse is
sent out, a refuge from the day-to-day incompetencies is sought... a board is
found.
        "This is it... this is where I belong..."
        I know everyone here... even if I've never met them, never talked to
them, may never hear from them again... I know you all...
        Damn kid.  Tying up the phone line again.  They're all alike...

        You bet your ass we're all alike... we've been spoon-fed baby food at
school when we                                                     id let slip
through were                                                       dists, or
ignored by th                                                      and us will-
ing pupils, bu

        This is our world now... the world of the electron and the switch, the
```

(a) (b) (c)

Figure 4.2: `APV` application: (a) `ChooseFileActivity` lists files and folders. (b) `OpenFileActivity` displays the selected PDF file. (c) Native memory usage before and after fixing the leak.

screen. The sequence of operations that opens a file and then closes it is expected to have a "neutral" effect on resource usage, and is an example of a neutral cycle. Repeated execution of this cycle normally should not lead to a sustained pattern of resource usage growth.

When executing an automated test case that repeatedly exercises these two transitions (selecting a file and then pressing the BACK button), we observed that the native memory usage increases significantly and ultimately leads to a crash. After examining the application code, we determined that certain amount of native memory is allocated during the initialization of `OpenFileActivity` and freed when the PDF file is closed, via a call to a native method `freeMemory`. However, `freeMemory` does not free all allocated memory, which results in a memory leak. In fact, in a later version of the application, the developers checked in a fix for this issue. The native memory consumption before and after this fix are shown in Figure 4.2(c); the x-axis shows the number of repetitions of the neutral cycle.

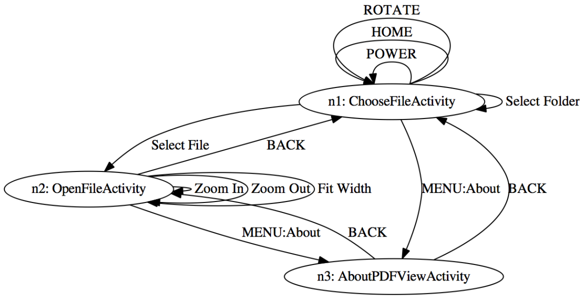


Figure 4.3: A subset of the GUI model for APV.

4.1.2 Leak Testing with a GUI Model

Following a large body of work on GUI-model-based testing [1, 2, 11, 44, 67, 69, 70, 105, 112, 113, 124], the starting point of our approach is a model of the Android application's GUI. To focus the presentation, we discuss one particular kind of model. However, the notion of neutral cycles and the coverage criteria based on them should be easily applicable to other GUI models (e.g., [4, 67]), where there is a natural correspondence between paths in the model and sequences of events. A partial GUI model for APV is shown in Figure 4.3. The figure shows only a subset of GUI states and transitions, as needed for explanation purposes.

The models we discuss are directed graphs, with one node per activity, and with edges representing transitions triggered by GUI events. The set of nodes is defined by the set of application classes that subclass (directly or transitively) class `android.app.Activity`: each such class is a node in the model. In addition to traditional events, the model should capture Android-specific events. For example, a user can press the hardware MENU button and then select a menu item from a list specific to the current activity. In Figure 4.3, edges labeled with MENU: represent such events; for example, MENU:About corresponds to choosing the "About" menu item. As another example, the hardware BACK button can be used to destroy the current activity and to transition to another one. (Although the programmer can choose to override this BACK button behavior with application-specific logic.) In addition to such application-specific events, several important GUI events are defined by the platform and not by the application:

ROTATE events. When the user rotates the screen, the current activity is recreated with a different orientation. In the model this event is represented by a self-transition labeled with ROTATE. A rotation event is important for testing because it covers the `onCreate`/`onDestroy` pair in the activity lifecycle from Figure 4.1. It is well known that repeated execution of this pair of methods can leak activity objects (instances of `android.app.Activity`), GUI widget objects (instances of `android.view.View`), visual resources (instances of `android.graphics.drawable.Drawable`) such as bitmaps, and other categories of resources [33,102]. To simplify Figure 4.3, only the ROTATE edge for n_1 is shown; both n_2 and n_3 have similar edges.

HOME events. When the user presses the hardware HOME button, the application is hidden. The launcher, a special application to allow the user to launch any application, is then brought to the foreground. For testing purposes, we are interested in the scenario where the original application is immediately selected to be reactivated. Edge HOME in Figure 4.3 represents pressing HOME and then going back to the same application. (A similar self-edge exists for each other node in the model.) Another situation with behavior equivalent to a HOME event is when the user receives a phone call while the application is active; once the phone call is completed, the application is reactivated. A HOME transition corresponds to the `onStart`/`onStop` loop in Figure 4.1 and could be considered for coverage during testing.

POWER events. The hardware POWER button puts the device in a low-power state. In this case, `onPause` is called on the current activity. When the button is pressed again and the screen is unlocked, the activity becomes active and its `onResume` method is called. Edge POWER in Figure 4.3 represents this sequence of operations. The same behavior and callbacks are observed in other scenarios unrelated to power usage—e.g., when an activity is partially blocked by a popup dialog. A testing strategy could consider coverage of POWER transitions.

Sensor events. The platform can generate other events due to user actions. For example, an accelerometer can trigger events because of shaking or tilting motions. More generally, acceleration forces and rotational forces can be sensed by accelerometers, gravity sensors, gyroscopes, and rotational vector sensors [84]. These sensor events are GUI events triggered by the user, and they can activate interesting behaviors. Our current approach does not include these events, but can be easily extended to consider them as well.

4.1.3 Obtaining GUI Models

Various reverse-engineering techniques (e.g., [11, 44, 68, 70, 124]) can be used to automatically construct GUI models. One example is AndroidRipper [1, 2, 107], a tool to perform GUI reverse engineering for Android applications. Its implementation uses the Robotium testing framework [92] to systematically explore the GUI. At each GUI state, the tool examines the run-time GUI widgets and the events that can be fired upon them. The models produced by the tool are very detailed. For example, a

MENU transition is represented by two edges, one for pressing the hardware MENU button and another for choosing a menu item (e.g., "About"). As another illustration of this level of detail, the same activity may be represented by many states in the model. For example, there are many possible lists of files/folders that can be displayed by activity `ChooseFileActivity` shown in Figure 4.2(a), by following the "parent folder" list item (labeled with ".." in the figure), or another list item representing a sub-folder. Each such file/folder list would be represented by a different state, resulting in a very large model.

To reduce model size and the number of generated test cases, we chose to use an abstracted model with one-to-one correspondence between activities and model states. For our experiments these models were created manually after examining the output of AndroidRipper and the source code of the application. We also added HOME and POWER transitions, which were not captured by AndroidRipper. It was an intentional decision *not* to focus on fully automating the model construction in this work, but instead focus on evaluating the model-based coverage criteria and showing that they are indeed useful for exposing leak defects. The next chapter describes a static analysis that provides essential building blocks for automated model construction.

4.2 Generation and Execution of Test Cases

The testing approach is based on a set of *test coverage criteria.* Each criterion is aimed at a particular category of neutral cycles in the model of the application's GUI. Note that we expect this kind of leak testing to be performed after—and be complementary to—traditional functional testing during which high block/branch coverage is achieved. Thus, we focus specifically on coverage of repeated behavior that may be related to leaks.

4.2.1 Test Coverage Criteria

To illustrate a coverage criterion, consider the ROTATE transition shown in Figure 4.3. In general, for each state n_i in the model, there is a self-transition representing a ROTATE event. We can define the following coverage criterion: for each state n_i, execute at least one test case that corresponds to a path $(s, \ldots, n_i, n_i, \ldots, n_i)$. Here s is the start state, prefix $(s, \ldots, n_i)$ represents a cycle-free path, and suffix $(n_i, n_i, \ldots, n_i)$ contains only ROTATE transitions. This suffix corresponds to k repetitions of the neutral cycle $n_i \rightarrow n_i$. The motivation for this coverage is clear: resource usage should not increase when the screen is rotated repeatedly [33], even for large k. Executions of this cycle will trigger repeated `onCreate`/`onDestroy` lifecycle callbacks (recall Figure 4.1). As mentioned earlier, resource leaks often occur because of defects related to lifecycle management. We have seen a number of examples of this pattern in our studies.

Application-independent cycles. One category of cycles to be covered are those defined by ROTATE, HOME, and POWER events—i.e., events defined by the platform, not by the application. An example of a ROTATE-based coverage was given above. Similar coverage can be defined for HOME cycles (to trigger repeated `onStart`/ `onStop`) and POWER cycles (for repeated `onPause`/`onResume`). Note that even though repeated ROTATE events also result in repeated start/stop and pause/resume, they do not necessarily expose leaks related to stopping or pausing an activity: because

ROTATE destroys the activity, it may release resources that are leaked by `onStop` or `onPause`. We have observed this situation in our studies.

Cycles with BACK transitions. The coverage criteria described above target only the activity that is currently interacting with the user. Cycles involving the hardware BACK button involve multiple activities, and present another target for coverage. For each BACK transition $n_i \rightarrow n_j$, we can execute a path $(s, \ldots, (n_j, \ldots, n_i)^k, n_j)$. Here the k transitions from n_i to n_j are done with the BACK button, and the shortest path from n_j to n_i is taken each time to reach the BACK edge. In our experience, cycle $(n_j, \ldots, n_i, n_j)$ is invariably a neutral cycle: resource usage growth over multiple repetitions is unexpected and suspicious. Coverage of cycles involving BACK edges may expose leaks that depend on the interplay among several activities. For example, we have observed cases where coverage of single-activity cycles (e.g., ROTATE cycles) does not expose a leak, but coverage of cycles with BACK transitions triggers the leaking behavior.

Application-specific neutral operations. We also consider cycles involving pairs of operations that "neutralize" each other. For example, node n_2 in Figure 4.3 has two self-transitions "zoom in" and "zoom out", triggered by two of the buttons shown at the bottom of Figure 4.2(b). The zooming-in operation, followed by the zooming-out one, should have a neutral effect, and a neutral cycle can be defined with these two operations. Other examples include connecting to/disconnecting from a server, opening/closing a file, adding an email account and then deleting it, etc. In addition, a single operation that only refreshes the GUI state of an activity (e.g., refreshing a list of email messages) should have neutral effect on resource usage.

Test case context. For a neutral cycle $(n_i, \ldots, n_i)$, any executable test case must contain a prefix path $(s, \ldots, n_i)$ where s is the start state. How should this prefix be chosen? In our current approach, we choose the shortest path from s to n_i. However, *context-sensitive* variations of the coverage could also be defined, where different execution contexts for the neutral cycle (i.e., different prefix paths leading to n_i) need to be covered. Making such choices is very similar to defining different calling contexts for functions in code analysis and testing, and presents interesting opportunities for future work.

4.2.2 Test Generation and Execution

Given a GUI model and a coverage goal, test generation can be achieved by traversing paths in the model. We have developed LeakDroid, a tool that implements this approach. In the generated test cases GUI events are triggered with the help of the Robotium testing framework [92]. A test case is shown in Figure 4.4. It corresponds to a path $(s = n_1, (n_2, n_3)^k, n_2)$ in the GUI model from Figure 4.3, and covers the BACK edge from n_3 to n_2. The start state is n_1. Line 4 makes an API call to select the third list item, assuming that the item represents a PDF file, and makes the transition to state n_2. The loop at lines 6–9 executes k repetitions of a neutral cycle that involves the BACK edge $n_3 \rightarrow n_2$. The call at line 7 selects a menu item, and the call at line 8 presses the BACK button. The API calls for GUI events are generated automatically by LeakDroid based on the given model and the coverage goal. The tool input also includes information about application-specific pairs of operations with neutral effects (e.g., open/close) and single neutral operations (e.g., refresh). Data-specific elements (e.g., choosing the third list item at line 4) are

```
// @PreCondition
//   A PDF file at position 3 of list
void test_n3_BACK_n2() {
  robotium.clickInList(3);  // n1 -> n2
  // Cycle: n2 -> n3 -> n2 -> ...
  for (int i = 0; i < k; i++) {
    robotium.clickOnMenuItem("About");
    robotium.goBack();
  }
}
```

Figure 4.4: An example of a generated test case.

subsequently provided by the tester. We found that the manual effort for this is trivial—once the Robotium calls are generated automatically, test setup (e.g., setting up an SSH host name at a specific position in the host list, or a file name at a certain position in the file list) is very easy.

During test case execution, various resources can be monitored. Currently we collect the following measurements.

Java heap memory. This is the memory space used to store Java objects. Existing memory leak detection techniques for Java typically focus on leaks in this memory space. The space is automatically managed by the garbage collector, so there can be leaks only when unused objects are unnecessarily referenced. Note that some resource leaks (e.g., leaking of database `Cursor` objects) also exhibit usage growth in this memory space.

Native memory. This memory space is used by native code, and is made accessible to Java code via JNI (Java Native Interface) calls. It requires explicit memory management by the developers as in programs written in non-garbage-collected languages such as C/C++, and thus could suffer from all well-known memory-related defects in those languages (e.g., dangling pointers, double-free errors). For example, the native `recycle` method of the `Bitmap` class has to be explicitly called to prevent leaking of native bitmap objects. This memory space is particularly important to monitor as many Android applications make heavy use of native code and thus native memory.

Binders. Binders provide an efficient inter-process communication mechanism in Android. In essence, a binder is the core component of a high-performance remote procedure call (RPC) mechanism directly supported by the underlying Linux kernel in the Android operating system. Usage of binders requires creation of global JNI references, and these references are made visible to the garbage collector. Unnecessarily keeping these references could lead to leaking of other potentially large Java objects. The global JNI references are deleted in native methods called by the finalizer of `android.os.Binder`, so the number of `Binder` instances is a good indicator of whether unnecessary JNI references are kept. There is likely to be an underlying software defect if this number grows significantly, and we collect measurements of it to identify binder leaks. Such leaks are distinguished from memory leaks because they are related to an Android-specific feature and behavior, which allows more precise diagnosis of the root problem.

Threads. Threads are usually created to perform time-consuming operations in a GUI application to maintain good responsiveness. For example, the e-book reader `VuDroid` [110] creates new threads to compute rendering data for requested files. A buggy implementation could hang thread execution, while new threads are being created. A sustained growth in the number of active threads in an application is an indication of software defects, and thus the proposed testing approach collects measurements of the number of active threads.

All of the discussed measurements can be easily collected via system services provided by the Android platform, and does not require any code changes or system modifications. To reduce the running time for test execution, we stop a test case early if it does not exhibit a pattern of growth. Various techniques can be used to decide whether a test case should be stopped. Currently we use a technique which monitors resource usage for 500 repetitions of the neutral cycle, performs linear regression on the measurements, and stops the test case if the rate of growth is below a certain threshold (e.g., less than 5% memory growth per hour). Although simple, this technique stops early the majority of test cases (76% in our experiments), allowing testing resources to be focused on a smaller set of test cases with non-trivial growth in resource usage. Each such "suspicious" test case is executed until it fails or until a predefined limit on the number of neutral cycle repetitions is reached. An interesting observation is that some non-failing test cases exhibit slow-leak behavior: there is a pattern of slow growth that may indicate an underlying defect. Our current reporting and evaluation focus only on failing test cases, in which a defect is clearly manifested; slow leaks will be investigated in future work.

4.2.3 Diagnosis of Failing Test Cases

When a test case fails, various techniques can be used to diagnose the root cause. For example, heap snapshots and object reference graphs derived from them are available in a number of tools (e.g., [38]). Information derived from such graphs is often analyzed manually to understand memory usage and diagnose memory leaks in Android applications [71]. Various automated analyses of heap graphs have also been proposed (e.g., [58,74]). Such analyses can potentially be extended to reflect the structure of the generated test cases. For example, a crashing test case that exhibits memory growth can be re-executed with a small number n of repetitions of the neutral cycle. As the test case is running, a heap snapshot is taken after each cycle repetition. After re-execution, n heap snapshots $H_1, H_2, \ldots, H_n$ are available, and $n-1$ heap differences $\Delta_i = H_{i+1} - H_i$ can be computed and analyzed. Our initial experience with manually applying this approach was very promising, and helped to identify the causes of all memory-growth test cases we observed. The diagnosis was performed with the help of the MAT memory analysis tool [38] (which is commonly used by Android developers [33]), followed by code inspection. An interesting question for future work is how to apply this approach to automated heap-differencing techniques (e.g., [58, 74]) and how to generalize it for analysis of native memory and resources other than memory.

4.3 Evaluation

We evaluated the proposed testing approach on eight open-source Android applications. The test cases were generated with our LeakDroid tool. We debugged all failing test cases and identified the underlying defects. All experiments were performed in the standard Android emulator from the Android SDK. The experimental subjects, their GUI models, the test cases, the description of identified defects, and the source code of LeakDroid are all publicly available at `http://www.cse.ohio-state.edu/presto/software`.

4.3.1 Study Subjects

We used search engines to establish a set of potential study subjects. The subjects were restricted to open-source Android applications; however, the proposed approach

Application	Version	Activities/ States	Transitions	Classes	Test Cases	Memory Leaks	Thread Leaks	Binder Leaks	Unique Defects
APV	r131	4	16	56	22	1	0	0	1
astrid	cb66457	11	27	481	40	3	0	0	1
ConnectBot	e63ffdd	9	27	301	32	3	0	10	3
FBReader	a53ed81	22	31	757	30	6	0	0	2
KeePassDroid	085f2d8	7	30	126	33	4	0	1	4
K9	v0.114	15	45	418	57	4	0	16	4
VLC	dd3d61f	8	22	176	32	4	0	0	2
VuDroid	r51	3	11	67	17	0	2	0	1

Table 4.1: Characteristics of study subjects, and experimental results.

can be easily applied to applications without publicly accessible source code. Applications that were less popular (e.g., with only a few installs) or not well-maintained (e.g., applications without a bug database, with only a few commits) were excluded from consideration. For an initial set of candidate applications, we searched their bug databases and code commit log messages. Search terms such as "leaks" and "out of memory error" were used to identify application versions that may contain leak defects. During or after this process, we did *not* examine carefully the bug reports and code commits, in order to ensure that the test cases generated by our approach were not biased toward any particular existing faults.

Characteristics of the study subjects are shown in the first five columns of Table 4.1. The number of application classes that subclass `android.app.Activity` is shown in column "Activities/States". Even for applications with only a few activities (e.g., `APV`), there could be several dozen other application classes to provide supporting functionality for the activities, which can lead to complicated run-time behavior. Each activity shown in "Activities/States" corresponds to a state in the GUI model. Column "Transitions" shows the number of edges in the model. This number does not include implicit application-independent self-transitions (that is, ROTATE, HOME, and POWER transitions). The applications represent a variety of domains: e-book/PDF readers (`APV`, `FBReader`, `VuDroid`), to-do list management tool (`astrid`), email client (`K9`), SSH client (`ConnectBot`), password manager (`KeePassDroid`), and multimedia player (`VLC`).

4.3.2 Experimental Methodology

For each application, test cases were generated (as described in Section 4.2.2) to achieve complete coverage with respect to the test coverage criteria defined earlier. Since ROTATE/HOME/POWER transitions exist for each state in the GUI model, the test cases are guaranteed to cover each activity in the application. Next, all generated test cases were executed as described in Section 4.2.2. During execution, usage measurements for various resources were collected for detection of growth patterns. When a test case fails, these measurements provide some initial clues as to what type of resource is leaking and what could be the underlying defect. For each failing test case, we investigated the application (using code inspection and a memory analysis tool) to determine the root cause of the failure and whether this cause was indeed related to resource leaks. Details of this investigation are presented in Section 4.4.

On average the execution time for a failing test case is less than two hours, with the majority of test cases failing in less than an hour. These times are artificially inflated due to a particular deficiency of our initial implementation. When firing an event through a Robotium call (recall Figure 4.4), it is necessary to wait until the effects of the event are processed by the application and shown in the GUI, so that the next event can be fired on the updated GUI. In our current prototype implementation we automatically introduce a large delay after each event in the test case. It is an interesting open problem how to automatically fine-tune the durations of these delays in order to reduce test execution time; we plan to investigate this problem in the near future.

4.3.3 Detection of Leak Defects

The rest of Table 4.1 shows measurements to demonstrate the effectiveness of the coverage criteria in detecting leak-related defects. Shown in order are the number of generated test cases, the number of failing test cases due to memory leaks, the number of failing test cases due to thread leaks, the number of failing test cases due to binder leaks, and the number of unique leak defects exposed by these failing tests (and confirmed by us through investigation of the source code).

Column "Test Cases" shows how many test cases were generated based on the coverage criteria described earlier. The test cases tend to be relatively small: on average, the number of Robotium calls per test case (i.e., the number of events fired) was 4.04. As discussed in Section 4.2.2, some of these test cases were filtered out early in their execution: we used a filtering approach to detect growth patterns and stop test cases that are not likely to cause sustained growth in resource usage. This approach was quite effective, and only 24% of the generated test cases needed to be executed further after the filtering step. Test cases that cover the POWER events are not included in these measurements, because we did not observe any resource usage growth related to these events.

A test case is included in column "Memory Leaks" when a crash is caused by a memory leak that leads to an out-of-memory error. It can be a memory leak in either the managed heap or the native heap. These memory leaks could have various underlying reasons, and some of them are related to inappropriate management of resources—for example, leaking of bitmap objects, database cursors, and event listeners. Some of these resources have memory budgets (e.g., bitmaps). Exceeding the budget limit will immediately lead to an out-of-memory error, although there could still be sufficient memory space left in the heap. Thus, it is important to force immediate reclamation of such resources when they are no longer needed.

"Thread Leaks" refers to the test cases that have a large number (100 in our experiments) of simultaneously active threads. It is our experience that an application exhibiting such behavior is very likely to have a thread leak problem, and it is unnecessary to wait until it exceeds the system-wide limit on the number of threads, which is usually even larger (4096 for Android). In fact, such test cases could crash very quickly when the amount of memory reachable from each newly-spawned thread is substantial.

In the Android emulator, there is a system-wide limit on the maximum number of global JNI references. When an application exceeds this limit, the emulator crashes. Although this limit by default is not enabled on real Android devices, exceeding it is still an indication of software defects. As discussed earlier (Section 4.2.2), usage of binders requires creation of global JNI references. So, there will be a crash when an application keeps creating new binder objects and maintains references to them. Column "Binder Leaks" counts the number of failing test cases that fall into this category.

The last column in the table shows the number of defects that are responsible for the failing test cases. Details on some of these problems are presented shortly. Among the 18 defects discovered, only 6 could be connected to existing bug reports and code commit logs; the remaining 12 were previously unknown. For the 6 defects related to existing reports/commits, this relationship was established (by examining bug reports and commit logs) only *after* the defects were uncovered by the generated test cases and confirmed by code examination. The 18 discovered defects were exposed using only a systematic model-based test generation approach, without any prior knowledge of their symptoms and root causes.

Note that it is impossible to ascertain how many leak defects actually exist in the studied applications. The vast majority of bug reports are vague and do not provide enough information to construct an actual test case to reproduce the failure, or to identify its root cause. Similarly, code commits typically have short and uninformative commit messages, again without details on how to reproduce the incorrect behavior being fixed. Only after generating our model-based test cases and debugging the failing ones, we have been able to determine that some existing bug reports and code commits are referring to the same defects.

4.3.4 Defect Detection for Coverage Criteria

We have defined and used several coverage criteria to target various types of neutral cycles. To understand the leak detection capabilities of each kind of neutral cycle, we categorized failing test cases based on the type of cycles they exercise. Figure 4.5 provides a summary of this study. The chart shows the numbers of failing test cases that exercise a neutral cycle to cover ROTATE transitions, HOME transitions, BACK transitions, and application-specific operations (a single neutral operation or a pair of neutralizing operations). The last two categories of neutral cycles exhibit the best ability to uncover leak defects. These cycles often involve both resource

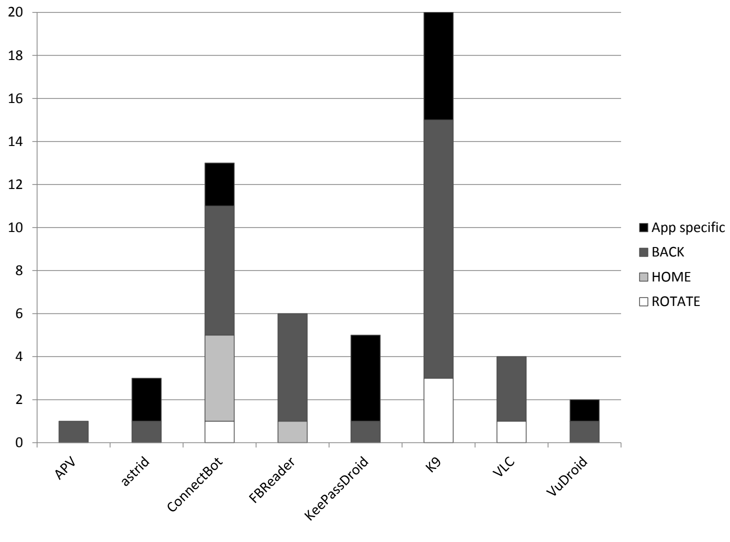


Figure 4.5: Failing test cases for each category of neutral cycles.

allocation and reclamation (reclamation could be missing if the application has a leak defect). Application-independent cycles (i.e., cycles defined by ROTATE, HOME, and POWER transitions) have weaker leak detection capabilities. As mentioned earlier, test cases that exercise POWER transitions were excluded from the presented measurements because they do not exhibit leaking behavior in any of the applications we studied.

4.4 Case Studies

This section presents several case studies to demonstrate the resource leak problems we found in the studied applications. The leak problems were confirmed by code fixes written either by the application developers or by the authors. The detailed description of these defects can provide insights into possible new approaches for detection, diagnosis, and prevention of leaks in Android applications.

APV. As discussed earlier, there is a leak in native memory for APV. The application crashes with a large native memory footprint due to incorrect implementation in native memory reclamation. This crash is triggered by the neutral cycle $n_1 \rightarrow n_2 \rightarrow n_1$ in Figure 4.3. This defect cannot be easily discovered: examining the Java source code alone, the resource seems to be properly managed. It is also not easy to fix this problem due to lack of heap profiling tools for the native memory. In fact, after we discovered this defect during testing, we examined the code repository and observed that it took the developers several revisions to fix this problem. As native code and native memory are more heavily used in Android applications, compared to traditional server and desktop Java applications, new testing and diagnostic tools/techniques specifically targeting the usage of native memory are greatly needed for the Android platform.

ConnectBot. SSH client `ConnectBot` has a defect related to leaking of event listener objects. `TerminalView` represents the graphical interface of an SSH session, and it is a listener for font size changes. When `onStart` of `ConsoleActivity` is called, a new `TerminalView` is created and added to a container of listeners. However, it is never removed from the container. When `onStart` is called multiple times on the same `ConsoleActivity` object, `TerminalView` is leaked. One way to trigger this behavior is to start `ConsoleActivity` first, and then repeatedly go to the HOME screen and go back. Note that this leak cannot be triggered by rotation events, because a new `ConsoleActivity` is created whenever rotation occurs. This is an example showing why both ROTATE and HOME events should be considered for test coverage.

KeePassDroid. This password management tool saves user-provided login credentials in a password-protected database file, so that users can access them with one master password. Multiple database files can be maintained. When the application is first launched, it displays a list of database files in `FileSelectActivity` for the user to choose. When a database file is selected, a query is launched to retrieve the information in the file, and the result can be accessed through a `Cursor` object. The `Cursor` is remembered in a container so that it can be synchronized with the activity (i.e., it has the same lifecycle as the activity). The `Cursor` object is automatically cleaned up when its managing activity is destroyed. However, when we keep the same instance of `FileSelectActivity` alive, and come back to the selection list to select database files repeatedly, multiple `Cursor` objects would be saved in `FileSelectActivity`. Several crashing test cases are caused by this problem. A whole hierarchy of objects representing the query results are reachable from `Cursor` objects, leading to fast growth in memory consumption. A similar problem was also found in the `astrid` application. This is an important pattern to consider, because Android applications often interact with the built-in SQLite database, and `Cursor` objects are used to access the results of SQL queries. Testing the interactions between the application and the database is an important consideration for Android software development.

K9. In `K9`, a popular email client, a leak was discovered when rotating the screen after an email message is selected for display. Since it crashes after only a few repetitions of the ROTATE neutral cycle, this is an example of a leak that can be easily observed and thus cause negative user perception of the application. Heap snapshots suggest that a large number of objects are kept alive through a few `Thread` objects. The only code that creates threads is in `MessageView`, an activity that displays individual email messages. The threads are executed with the help of a thread pool executor, and because of this they are not explicitly started by calling `start` on them. In the standard library used by Android, a thread whose `start` method is never called is guaranteed to be leaking due to complex interplay between threads and thread groups. Since a leaking thread keep references to `MessageView`, a whole hierarchy of GUI objects are kept alive, leading to a quick crash. A simple fix is to create a `Runnable` object rather than a `Thread` object. Our understanding of the behavior of `Thread` and `ThreadGroup` was also confirmed by Android platform developers.[2] This is an example where a seemingly-innocent mistake (using `Thread` instead of `Runnable`), together with the unexpected behavior of the platform code, lead to problematic behavior. In fact, we have seen other leaks in the platform's management of resources (e.g., binders), in which case the application code does not have any defects, but still crashes. These observations highlight the need to repeatedly exercise resource-management code during testing, in order to expose unexpected interactions with the Android platform.

VLC. `VLC` is a popular cross-platform multimedia player. A leak was exposed when screen rotation is performed multiple times on its `AboutActivity`. This activity is implemented as a `FragmentActivity`, a new feature introduced in Android 4.0 and ported back to earlier versions. A fragment activity can manage `Fragments`, more flexible containers of GUI components. `Fragment` objects are created in `AboutActivity` and registered with the platform. Heap snapshots indicate that many fragments are kept alive. By default, the state of a `FragmentActivity` is (silently) saved and restored by the platform. In particular, all registered fragments are saved in memory before the activity is destroyed, and then restored from memory before it is recreated. Because of this behavior, `Fragment` objects created inside `AboutActivity` can never

[2] `http://groups.google.com/d/topic/android-platform/y3G7v_U-hvA/discussion`

be garbage collected. For the developers this is an unexpected change to the management of an activity's lifetime, caused by this new Android feature. Subsequently we discovered that a later version of `VLC` disabled this default save/restore (by overriding relevant callback methods), which fixed the leak. This defect illustrates how new features introduced in the still-evolving Android platform can lead to defects due to poor understanding, which in turn motivates the need for regression testing and comprehensive strategies for test generation.

Application-specific neutral operations. We observed several examples where a leak is triggered by a neutral cycle with an application-specific functionality. For example, in the `astrid` task management application, database cursors are leaked when the user changes the sort order of tasks, and then reverts back to the original order. Similarly, adding and then removing a task causes a cursor leak. The application-specific neutral cycles we consider for coverage are rather simple: they either involve a pair of neutralizing operations (e.g., add/remove, open/close) or a single operation that updates the displayed content (e.g., to refresh the current, unchanged list of email messages in `K9`). Currently, these cycles are provided as input to LeakDroid. Automatic identification of such cycles, and perhaps even static analysis of their correctness with respect to leaks, are interesting problems for future work.

4.4.1 Discussion

Among the exposed defects, a diversity of resources are involved. Examples include not only traditional leaks such as memory leaks and thread leaks, but also Android-specific leaks such as binder leaks. Even for defects that exhibit the same “out-of-memory error” symptom, the underlying relevant resources could be different. Bitmaps, database cursors, and event listeners are some examples that fall into this category. Leaks caused by defects inside the Android platform and the standard library were also uncovered. This experience suggests that the proposed testing approach can effectively expose diverse resource leaks across a variety of applications.

Our experiments and case studies indicate that systematic model-based testing for resource leaks in Android software can be done effectively. They also point to interesting directions for future work. First, leak patterns based on neutral cycles can be leveraged to develop automated leak diagnosis tools. Based on our experience, we believe that a substantial part of the diagnosis process can be automated. It is particularly useful to identify strong correlations between the number of executed repetitions and the growth in the number of instances of certain classes. Allocation of and references to instances of classes that exhibit such correlations are usually related to the leak defects. Second, it is beneficial for understanding of resource usage/leaks to have analysis techniques that can automatically identify methods related to resource manipulation (e.g., allocation and reclamation). To achieve this goal, both static and dynamic analyses techniques may have to be developed. Finally, it is important to consider new mechanisms for prevention of resource leaks, with the help of better software abstractions and patterns for resource management.

4.5 Summary

This chapter proposes a systematic and effective technique for testing of resource leaks in Android applications. Neutral cycles—sequences of GUI events that should not lead to increases in resource usage—are used to define test coverage criteria.

Evaluation of this approach indicates that complicated and diverse resource leaks can be exposed by the generated test cases. These promising initial results suggest that such a testing technique is feasible and effective for detection of resource leak defects. Our investigation also points to several important directions for future work, including additional coverage criteria; better diagnosis techniques (e.g., by correlating repeated behavior with heap growth); increased focus on analysis of native memory as well as analysis of specific resources (e.g., database cursors, bitmaps); automated static or dynamic discovery/analysis of code that allocates and reclaims important resources; improved resource management through new software abstractions and patterns.

CHAPTER 5: Static Reference Analysis for GUI Objects in Android Software

Reference analysis for object-oriented languages [95] (also referred to as pointer analysis) is a static analysis technique that models the flow of object references. It has been studied extensively due to its essential role as a prerequisite for many other static analyses. For example, interprocedural control-flow analysis for object-oriented software requires information about the object references that can be observed at polymorphic calls. Another typical example is data dependence analysis, which also requires object reference information.

The previous chapter introduces a systematic testing strategy to uncover resource leaks in Android applications. An important input to the approach is a GUI model for the Android application under test. Constructing a model for an application GUI is a reverse engineering problem. Reverse engineering of GUI models has been studied by others (e.g., [44, 68, 70]) and has been applied to Android applications (e.g., [1, 2, 4, 11, 107, 124, 130]). However, all of these existing approaches rely fully or partially on a dynamic program analysis that cannot cover all possible program executions and thus the constructed model is inevitably incomplete. Therefore, a fully static approach for GUI model construction is greatly desired. One critical building block for such a static approach is a reference analysis for GUI objects in an Android application.

Existing reference analyses cannot be applied directly to Android applications. Although the underlying language is Java, Android software is built using a *component-based* approach where the platform manages the lifetime, behavior, and data of application components. Furthermore, the software is *event-driven*: an Android application is driven by a graphical user interface (GUI), with GUI-related objects responding to user actions (e.g., pressing a button). The set of GUI objects and the event handlers associated with them ultimately determine the possible flow of control and data in the application. To the best of our knowledge, at present there does not exist any work on static analyses to model the details of this GUI-driven control/data flow.

Proposed analysis We propose *the first static analysis* to model the set of GUI-related Android objects, their flow through the application, and their interactions with each other via the abstractions defined by the Android platform. A number of features make this analysis different from a traditional reference analysis. First, the creation of GUI objects is often implicit, based on external declarative information. The correct modeling of this creation is essential for the proposed analysis. Second, there is a hierarchical structure of GUI objects; this structure affects the run-time behavior and must be modeled statically. In addition, GUI objects are typically accessed via object ids, which requires tracking of such ids and modeling of the effects of their use. Finally, it is critical to track the association between a GUI object and the event handlers that respond to user actions on this object.

Motivation The creation, propagation, and interactions of GUI-related objects directly affect the application's run-time behavior. They define the control flow, and in particular the possible GUI events that drive the application, and the invocations of handlers for these events. They also determine aspects of the data flow: for example, text entered by the user (e.g., a password) is obtained with the help of a particular GUI object and flows from it, via the event handler, to the rest of the application. Static analysis to model this control/data flow is foundational for compiler analyses, instrumentation for event/interaction profiling, static error checking, security analysis, test generation, and automated debugging. A body of existing work can directly benefit from our analysis, including static error checking (e.g., [86, 87, 129]), run-time exploration for dynamic analyses for profiling, energy analysis, security analysis, responsiveness analysis, and systematic testing (e.g., [2,11,40,55,85,111,122,123]), static security analysis (e.g., [10, 18, 20, 42, 43, 82]), and reverse engineering (e.g., [124]).

Contributions The contributions of this work are

- A *formal semantics* for GUI-related Android constructs. This semantics provides a solid foundation for the development of this and other analysis algorithms.

- A *constraint-based static analysis*. The analysis employs a constraint graph to model the flow of GUI-related objects, the hierarchical structure of these objects, and the effects of relevant Android operations.

- An *experimental evaluation* on real-world Android applications. The results strongly suggest that the analysis achieves high precision with low cost.

These contributions provide a key component for an analysis infrastructure to be used by compile-time analysis researchers in the increasingly important area of Android software. An earlier version of this work appeared in [94].

5.1 Background and Example

Figure 5.1 shows an example derived from `ConnectBot` [26], an SSH client with more than one million installations according to the Google Play Store statistics. `ConsoleActivity` defines an *activity*. An activity class is an application class that is a direct or transitive subclass of `android.app.Activity`. Activity objects (referred to as "activities") are the core application components, and they are managed by the platform through various callbacks. When an activity is started (by another activity or by an external application), the platform creates the activity object and invokes the `onCreate` callback method defined at lines 8–16.

An activity presents to the user a window with GUI elements, defined with the help of *views*. A view class v is a direct or transitive subclass of `android.view.View`. Instances of such classes represent GUI widgets that can be observed and manipulated by the user (e.g., buttons), as well as logical groups of such objects. A developer may (but does not have to) introduce application-specific view classes. The example uses standard view classes `ViewFlipper`, `ImageView`, and `RelativeLayout`, as well as an application-defined view class `TerminalView`.

`ImageView` displays an image; in this example, it is used to show the icon of an ESC button for an SSH terminal. `RelativeLayout` is a container for a set of children views, and it itself is not directly visible to the user. `ViewFlipper` is a container that can animate between several children views, by flipping between children (e.g., flipping could happen when the user swipes across the touchscreen). `TerminalView` is an application class providing the GUI for an SSH terminal window; the source code of this class is not shown in the figure. The two XML files `act_console` and

```
class ConsoleActivity extends Activity {
  ViewFlipper flip;

  View findCurrentView(int a) {
    ViewFlipper b = this.flip;
    View c = b.getCurrentView(); // FindView
    View d = c.findViewById(a); // FindView
    return d; }

  void onCreate() {
    this.setContentView(R.layout.act_console); // Inflate
    View e = this.findViewById(R.id.console_flip); // FindView
    ViewFlipper f = (ViewFlipper) e;
    this.flip = f;
    View g = this.findViewById(R.id.button_esc); // FindView
    ImageView h = (ImageView) g;
    EscapeButtonListener j = new EscapeButtonListener(this);
    h.setOnClickListener(j); // SetListener }

  void addNewTerminalView(TerminalBridge bridge) {
    LayoutInflater inflater = ... // helper object
    View k = inflater.inflate(R.layout.item_terminal); // Inflate
    RelativeLayout m = (RelativeLayout) k;
    TerminalView n = new TerminalView(bridge);
    n.setId(R.id.console_flip); // SetId
    m.addView(n); // AddView
    ViewFlipper p = this.flip;
    p.addView(m); // AddView } }

class EscapeButtonListener implements OnClickListener {
  ConsoleActivity cact;

  EscapeButtonListener(ConsoleActivity q) {
    this.cact = q; }

  void onClick(View r) {
    ConsoleActivity s = this.cact;
    View t = s.findCurrentView(R.id.console_flip);
    TerminalView v = (TerminalView) t;
    // send ESC key to terminal associated with v } }

act_console.xml:
   <RelativeLayout ... >
      <ViewFlipper android:id="@+id/console_flip" ... />
      <RelativeLayout android:id="@+id/keyboard_group" ... >
         <ImageView android:id="@+id/button_esc" ... />
         ...
      </RelativeLayout>
   ...
   </RelativeLayout>

item_terminal.xml
   <RelativeLayout ... >
      <TextView android:id="@+id/terminal_overlay" ... />
   </RelativeLayout>
```

Figure 5.1: Example based on `ConnectBot` [26].

item_terminal shown at the bottom of the figure define hierarchies of GUI widgets based on this set of classes.

Layouts Line 9 reads a *GUI layout definition* from XML file act_console and instantiates a hierarchy of views. Android best practices suggest that the definition of the visual layout be separated from the code. A layout definition describes a hierarchical structure of views. Using standard tools, these definitions are compiled to Java code. For each layout, there is a unique integer id defined by a final static field in an automatically-generated class R.layout (e.g., R.layout.act_console and R.layout.item_terminal). The values of these ids are used as parameters to several *layout inflaters*. A layout inflater is a method that, given a layout id, "inflates" the definition to a view hierarchy. In general, the parameter of an inflater call can be an integer variable that is (transitively) assigned a layout id. In the example, the inflater call at line 9 associates the new hierarchy with the activity, while the inflater call at line 19 just returns the root view.

A layout definition describes a tree. An inner node represents a container view (e.g., RelativeLayout), which is a wrapper around children views. The leaf nodes represent basic GUI components (e.g., ImageView and TextView). Some nodes may have string ids. For each such *view id* in the definition there is a corresponding integer field in class R.id: for example, fields R.id.console_flip, R.id.keyboard_group, and R.id.button_esc correspond to the view ids defined in act_console.

Operations on views In onCreate, lines 10 and 13 contain *find-view* calls. Such calls use a view id to search for a view in a given hierarchy—in this case, the hierarchy associated with the activity. In addition, line 16 contains a *set-listener* call, which associates a listener (i.e., handler) of click events with the ImageView representing an

ESC button. The handling of a click event (method onClick at lines 30–34) will be discussed shortly. Both find-view operations and set-listener operations are modeled by our analysis, in order to represent statically the propagation of views and the control/data flow due to event handlers for these views.

Method addNewTerminalView (lines 17–25) updates the GUI when a new SSH terminal is opened. Calls to this method occur in the rest of the code of ConsoleActivity; for brevity, this code is not shown in the example. The call at line 19 inflates a layout defined by XML file item_terminal and returns the root RelativeLayout. Line 21 shows a programmatically-created instance of TerminalView. This class (a subclass of View) is not defined or instantiated by the Android platform but rather by the application. At line 22, the id is explicitly set through a *set-id* operation, and at line 23 the new terminal view becomes a child of the inflated RelativeLayout through an *add-child* operation. At line 25 the RelativeLayout becomes a child of the ViewFlipper due to another add-child operation. As several SSH terminals are opened, each call to addNewTerminalView extends the hierarchy rooted at the ViewFlipper with a new subtree of widgets. The flipper allows the user to flip through the multiple terminals, using swiping motions.

This method illustrates several challenges for the proposed static analysis. First, the program can freely mix inflated views and programmatically-created views. Second, the parent-child relationships between views can be established either through inflation or through explicit add-child operations (lines 23 and 25). Similarly, view ids can be set during inflation or via set-id calls (line 22). Changes to children and ids can in turn affect find-view operations. Consider the call at line 32 in event handler onClick, which handles a click event on the ESC button defined by view id `button_esc`. This handler calls helper method `findCurrentView(int)`, which queries the flipper about which of its children is currently visible; this is done by the find-view call at line 5. These children are the `RelativeLayout`s created at line 19 and added as children at line 25; the effects of these two operations need to be modeled properly in order to handle correctly the call at line 5. Furthermore, another find-view call at line 6 searches the hierarchy rooted at the `RelativeLayout` for a view with the given id. This behavior is affected by the set-id operation at line 22, the add-child operation at line 23, and the interprocedural propagation of the view id to parameter `a`.

Event handlers GUI objects in the application can be associated with event handlers. Consider the ESC button, defined by view id `button_esc` and represented at run time by an instance of `ImageView`. This instance is created by the inflater call at line 9 and retrieved by the find-view call at line 13. Click events on this GUI widget are handled by the *listener object* created at line 15. This listener is registered with the GUI object through the set-listener call at line 16. The event handling happens in method `onClick` (lines 30–34). This method's signature is defined in interface `android.view.View.OnClickListener` and is used for the callback methods invoked by the Android platform when a click event occurs. A parameter of the callback is the view `r` on which the event occurred—in this case, the `ImageView` for the ESC button. This example illustrates that a static analysis of GUI objects should account for (1) the association between a GUI object and its listener objects, and (2) the implicit flow of the GUI object as a parameter of callbacks to event handler methods.

Additional Android constructs There are additional Android constructs that are not demonstrated in the example shown in Figure 5.1. A *menu* displays items of choices when user presses the MENU button, or long clicks on a view. A *list* defines a sequence of data items to be shown. A *dialog* is used to display short messages or ask the user a brief question. All of these are frequently used GUI features used in real-world applications. Their semantics, together with propagation of related GUI object references, should be modeled as well.

5.2 Semantics of Relevant Android Constructs

The creation and propagation of views, together with their interactions with activities and listeners, define a critical component of the run-time behavior of Android applications. These interactions affect the control flow—for example, they define the set of possible GUI events at each moment of the execution (based on the available views), the event-handling code for them, and the effects of this handling (e.g., starting new activities, sending data over the network, etc.). The data flow is also directly affected: for example, text entered by the user is associated with a particular view and flows from that view to the corresponding listener, and from there to other components of the application.

Static analysis to model this run-time behavior is highly desirable as foundation for compiler analyses, instrumentation for profiling of GUI-driven events and interactions, static error checking, security analysis, test generation, and automated debugging. To the best of our knowledge, at present there does not exist such a static analysis. Our goal is to develop semantic foundations and analysis algorithms for solving this problem. We propose a principled solution, starting with a definition of the semantics of relevant Android constructs (this section) and using it to define a constraint-based analysis for it (next section).

5.2.1 Syntax and Semantics of JLite

This section describes JLite, a subset of Java that contains all essential language features needed to present the proposed reference analysis for Android software.

Syntactic entities A program contains a set of Java classes. ("Class" will be used to refer to both classes and interfaces.) The following syntactic categories are considered: classes $c \in \mathit{Class}$, methods and constructors $m \in \mathit{Method}$, fields $f \in \mathit{Field}$, statements $s \in \mathit{Stmt}$, and locals/formals $x, y, z, p, r \in \mathit{Var}$. Each method m has a name, formal parameters $\mathtt{this}_m$ and p_m, and an artificial return variable r_m. Each method's body is a statement s, defined by $s ::= s_1; s_2 \mid x := \mathtt{new}\ c \mid x := y \mid x := y.f \mid x.f := y \mid z := x.m(y)$. In a minor abuse of notation, here c and m denote the name of class c and method m. Since we are interested in an analysis that abstracts away the intraprocedural control flow—as typically done in reference analysis for Java—conditional statements and loops are omitted. Each variable is of reference type. An assignment to return variable r_m represents a possible return value of method m.

Operational semantics Figure 5.2 shows the domains and functions used to define the semantics of JLite. A heap location ξ_c is labeled with the class c it instantiates. An environment ρ defines values for locals, formals, and return variables; the values are heap locations. For simplicity of presentation, we formulate the semantics in the absence of recursion (i.e., the elements of Var provide unique names for stack locations), and we also assume that all stack and heap locations are properly initialized before being read. A heap η represents the values of fields of heap objects.

Semantic function $\mathcal{E} : \mathit{Expr} \rightarrow \mathit{Env} \times \mathit{Heap} \rightarrow \mathit{Loc}$ provides the meaning of an expression $e \in \mathit{Expr}$. In the rule for $\mathcal{E}[\![\mathtt{new}\ c]\!]$, ξ_c is a new location that does not

$$
\begin{array}{rcll}
\xi_c & \in & \mathit{Loc} & \text{heap locations} \\
\rho & \in & \mathit{Env} = \mathit{Var} \to \mathit{Loc} & \text{environments} \\
\eta & \in & \mathit{Heap} = \mathit{Loc} \times \mathit{Field} \to \mathit{Loc} & \text{heaps}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![x]\!](\rho, \eta) & = & \rho(x) \\
\mathcal{E}[\![x.f]\!](\rho, \eta) & = & \eta(\rho(x), f) \\
\mathcal{E}[\![\texttt{new}\ c]\!](\rho, \eta) & = & \xi_c \\
\langle x := e, \rho, \eta \rangle & \to & \langle \rho[x \mapsto \mathcal{E}[\![e]\!](\rho, \eta)], \eta \rangle \\
\langle x.f := y, \rho, \eta \rangle & \to & \langle \rho, \eta[(\rho(x), f) \mapsto \rho(y)] \rangle \\
\langle z := x.m(y), \rho, \eta \rangle & \to & \langle (s_m; z := r_m), \\
 & & \rho[\texttt{this}_m \mapsto \rho(x), p_m \mapsto \rho(y)], \eta \rangle
\end{array}
$$

Figure 5.2: Semantic domains and functions.

occur in ρ or η. The standard rules for sequencing ($s ::= s_1; s_2$) are not shown. For a method call, formals $\texttt{this}_m$ and p_m of m obtains their values from the corresponding actuals. The body s_m of m, followed by propagation of m's return variable r_m, are executed in the updated environment. For brevity, the presentation assumes that each call is statically resolved to a unique target method.

5.2.2 Syntax and Semantics of ALite

Next we describe ALite, an extension of JLite that introduces the relevant Android constructs. An input program contains a set *Class* of Java classes with the syntactic structure described earlier. Some of these classes are application classes, while others are provided by the Android platform. The analysis aims to model explicitly the complex high-level semantics of Android, rather than analyzing the low-level semantics of platform code; thus, the bodies of methods in platform classes are not included in the input program. The following categories are of particular interest: $a \in \mathit{ActivityClass}$, $v \in \mathit{ViewClass}$, and $h \in \mathit{ListenerClass}$. As discussed in Section 5.1, an activity class a is an application class that is a subclass of `Activity`, and a view class v is a subclass of `View`. A listener class h implements event handlers associated with views, as discussed later.

Layout Definitions

To represent the effects of layout definitions, the syntax of statements can be extended with $s ::= \ldots \mid x := \texttt{R.layout}.f \mid x := \texttt{R.id}.f$ to reflect the occurrences of layout ids and view ids in the code. XML layout information can be abstracted as follows. A set of $\mathit{id} \in \mathbb{Z}$ defines layout ids and view ids. A layout/view id is the integer value of a constant field from class `R.layout/id` (e.g., `0x7f030000`). A node in a layout definition is (v, id) where v is a view class. There could be several nodes that are instances of the same v (e.g., several buttons in a layout). A layout edge shows a parent-child relationship between views. For example, for `act_console` in Figure 5.1, one of the layout edges is from parent $(\texttt{RelativeLayout}, \texttt{keyboard\_group})$ to child $(\texttt{ImageView}, \texttt{button\_esc})$. A layout definition is a set of layout edges that form a rooted tree.

Semantics To express the effects of layout inflation, we first generalize the environment and the heap:

$$\begin{array}{rcl} \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Loc} \cup \mathbb{Z} \\ \mathit{Heap} & = & \mathit{Loc} \times \mathit{Field} \rightarrow \mathit{Loc} \cup \\ & & \mathit{View} \times \{\texttt{vid}\} \rightarrow \mathbb{Z} \cup \\ & & \mathit{View} \times \{\texttt{children}\} \rightarrow \mathcal{P}(\mathit{View}) \end{array}$$

The value of a stack location $x \in \mathit{Var}$ can now be a layout/view id. An assignment $x := \texttt{R.layout}.f$ updates $\rho(x)$ with the appropriate layout id (and similarly for view ids).

Set $\mathit{View} \subset \mathit{Loc}$ denotes all instances of all view classes in the heap. An artificial field `vid` for a view refers to the corresponding view id. Another artificial field `children` refers to the set of children views. The effects of inflating a layout can be captured by inflater semantic functions:

$$\mathcal{I}_N : \mathbb{Z} \rightarrow \mathit{Env} \times \mathit{Heap} \rightarrow \mathcal{P}(\mathit{View} \times \mathbb{Z})$$
$$\mathcal{I}_E : \mathbb{Z} \rightarrow \mathit{Env} \times \mathit{Heap} \rightarrow \mathcal{P}(\mathit{View} \times \mathit{View})$$

For layout id id_l, $\mathcal{I}_N[\![id_l]\!](\rho, \eta)$ is a set of pairs (ξ_v, id), one for each layout node (v, id). Here ξ_v is a new heap location representing an instance of view class v, and id is the corresponding view id. For the inflation of layout edges, $\mathcal{I}_E[\![id_l]\!](\rho, \eta)$ defines pairs (ξ, ξ') that correspond to layout edges. The semantics of an inflater call is

$$[\textsc{Inflate}_1] \quad \langle z := x.m(y), \rho, \eta \rangle \rightarrow \langle \rho[z \mapsto \xi^{root}], \eta' \rangle$$

where ξ^{root} is the view at the root of the hierarchy, since the return value of the inflater call is that root. In the updated heap η', for each newly-created view ξ, fields `vid` and `children` are initialized based on $\mathcal{I}_N[\![\rho(y)]\!]$ and $\mathcal{I}_E[\![\rho(y)]\!]$. The object referred to by x is a helper object provided by the platform to implement the inflation. For example, at line 19 in Figure 5.1, variable `inflater` refers to this helper. For this call, $\mathcal{I}_N[\![\texttt{item\_terminal}]\!]$ produces $(\xi^1, \texttt{no\_id})$ and $(\xi^2, \texttt{terminal\_overlay})$, where ξ^1 is a new instance of `RelativeLayout` and ξ^2 is a new instance of `TextView`. (Here `no_id` is a special value used to denote the absence of a view id.) Using rule $\textsc{Inflate}_1$, heap η' has $\xi^1.\texttt{children} = \{\xi^2\}$, $\xi^1.\texttt{vid} = \texttt{no\_id}$, and $\xi^2.\texttt{vid} = \texttt{terminal\_overlay}$; in addition, $\rho(\texttt{k}) = \xi^1$.

Operations on Views

Views created through inflation or through explicit instantiation (`new` v) can be subjected to several operations defined by the Android platform. Correct modeling of the semantic effects of these operations is essential for our analysis.

Associations with activities A view can be associated with an activity. When the activity is active, this view and the view hierarchy rooted at it define the GUI content displayed to the user. As discussed shortly, this association allows hierarchy elements to be accessed programmatically through the activity. The relevant operations are as follows. First, inflater method `Activity.setContentView(int)` can be invoked on an activity, with the parameter being the layout id. As a result, the root of the inflated view hierarchy becomes associated with the activity, rather than being returned from the call. The generalization of the semantics is $\mathit{Heap} = \ldots \cup \mathit{Activity} \times \{\texttt{root}\} \rightarrow \mathit{View}$ where `root` is an artificial field for the activity. The semantic rules are extended as expected

$$[\textsc{Inflate}_2]\langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[\ldots][(\rho(x), \texttt{root}) \mapsto \xi^{root}]\rangle$$

where $\rho(x) \in \mathit{Activity}$ and $[\ldots]$ represents the heap updates due to inflation, similarly to rule $\textsc{Inflate}_1$. For example, at line 9 in Figure 5.1, $\rho(\texttt{this})$ is an activity object ξ^1, and $\xi^1.\texttt{root} = \xi^2$ where ξ^2 is the `RelativeLayout` at the root of the new `act_console` layout instance. Note that similar inflation operations exist for objects other than activities (e.g., for dialogs) and can be modeled in the same manner.

A call to `Activity.setContentView(View)` can be used to create an association between an activity and an existing view. The parameter is a view that could come from several sources—for example, it could be programmatically created, or it could be looked up from an inflated view hierarchy. The semantic effects are

$$[\textsc{AddView}_1]\langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{root}) \mapsto \rho(y)]\rangle$$

The same approach applies to similar operations on non-activity objects (e.g., dialogs).

Associations with other views The parent-child relationship between two views can be established during layout inflation, as discussed earlier. Another mechanism is to explicitly invoke an add-child operation. Several methods with the name `addView` can be used for this purpose; lines 23 and 25 in Figure 5.1 show two examples. Abstracting such calls as $x.m(y)$ where x refers to the parent and y refers to the child,

$$\begin{aligned}&[\textsc{AddView}_2]\ \langle x.m(y), \rho, \eta\rangle \rightarrow \\ &\langle \rho, \eta[(\rho(x), \texttt{children}) \mapsto \{\rho(y)\} \cup \eta(\rho(x), \texttt{children})]\rangle\end{aligned}$$

where $\rho(x), \rho(y) \in \mathit{View}$. The platform ensures that the new hierarchy is well-formed—specifically, that the parent-child relation corresponds to a tree and not to a more general graph. For brevity, we do not express these constraints.

Associations with ids A view id is an integer identifier associated with a view during inflation. A similar effect can be achieved by using a set-id operation: a call to method `setId(int)`, as shown at line 22 in Figure 5.1. We use the following rule for such a call:

$$[\textsc{SetId}]\quad \langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{vid}) \mapsto \rho(y)]\rangle$$

Associations with event handlers A view can be associated with several listeners, which are instances of classes $h \in \mathit{ListenerClass}$. Each such h implements one or more listener interfaces. For example, `EscapeButtonListener` in Figure 5.1 implements interface `View.OnClickListener` and defines a handler `onClick` for click events. The listeners attached to a view determine which events will be handled by the view, which in turn defines the possible flow of control in response to user actions. To capture these associations, the semantics is extended with $\mathit{Heap} = \ldots \cup \mathit{View} \times \{\texttt{listeners}\} \rightarrow \mathcal{P}(\mathit{Listener})$ where `listeners` is an artificial field and $\mathit{Listener} \subset \mathit{Loc}$ denotes all instances of all listener classes in the heap. Set-listener operations are calls $x.m(y)$ where x is the view and y is the listener; one example is shown at line 16 in Figure 5.1. The semantic rule is

$$[\textsc{SetListener}]\ \langle x.m(y), \rho, \eta \rangle \rightarrow$$
$$\langle \rho, \eta[(\rho(x), \texttt{listeners}) \mapsto \{\rho(y)\} \cup \eta(\rho(x), \texttt{listeners})] \rangle$$

Retrieval of views The view ids play an important role in find-view operations. `View.findViewById(int)` searches the hierarchy rooted at the view and returns the descendant view with the given id. A similar operation can also be applied to an activity, in which case the activity's entire view hierarchy is searched. The semantics is captured by

$$[\textsc{FindView}_1] \quad \langle z := x.m(y), \rho, \eta \rangle \rightarrow$$
$$\langle \rho[z \mapsto \mathit{find}(\rho(x), \rho(y))], \eta \rangle$$

where $\mathit{find}(\xi^0, \mathit{id}) = \xi^n$ if there exists a sequence $\xi^0, \ldots, \xi^n$ such that $\eta(\xi^n, \texttt{vid}) = \mathit{id}$ and $\xi^{k+1} \in \eta(\xi^k, \texttt{children})$ for all k. When `findViewById` is invoked on an activity,

$$[\textsc{FindView}_2] \quad \langle z := x.m(y), \rho, \eta \rangle \rightarrow$$
$$\langle \rho[z \mapsto \mathit{find}(\eta(\rho(x), \texttt{root}), \rho(y))], \eta \rangle$$

There are also operations that, when invoked on a view, retrieve some descendant view with a particular run-time property. A typical example is method `findFocus()`, which returns the descendant view that currently has focus. Similarly, the call to `getCurrentView()` at line 5 of Figure 5.1 returns the child view that is currently visible. We represent such operations by

$$[\textsc{FindView}_3]\ \langle z := x.m(), \rho, \eta\rangle \rightarrow \langle \rho[z \mapsto \mathit{find}_m(\rho(x))], \eta\rangle$$

where function find_m abstracts the specifics of m's run-time behavior when invoked on view $\rho(x)$.

Effects of callbacks The Android platform interacts with the application classes through various callback methods. One typical example is method `onCreate` (lines 8–16 in Figure 5.1), which is invoked on a `ConsoleActivity` object by the platform code that manages the activity lifecycle. Another example is callback method `onClick` (lines 30–34), which is invoked to handle a click event. The general problem of handling such callbacks in static analysis for Android is challenging. While some techniques have been considered in prior work (e.g., [10, 42]), at present there does not exist a fully comprehensive and precise solution. In our work we do not attempt to model all callbacks or their possible orderings: instead, we focus on two important categories that directly affect GUI-related behavior. First, for an activity class a, the implicit creation of an instance of a can be modeled by $t :=$ `new` a. Any Android-defined callback to an application method m on an instance of a can be modeled as a call $t.m()$. For the example, we conceptually extend the program with `t := new ConsoleActivity` and `t.onCreate()`, which is similar to the approach from [10]. In addition to this modeling of activities, we also model the effects of callbacks to handler methods for GUI events. This modeling is conceptually equivalent to creating additional statements, one per set-listener call. Recall that for a set-listener call $x.m(y)$, x refers to a view and y refers to a listener. The declared type of variable y and the signature of m determine the type of GUI event being handled.

Let n be the Android-defined signature of handlers for this event. The callback to the handler can be modeled as $y.n(x)$. For the running example, set-listener call `h.setOnClickListener(j)` at line 16 corresponds to an additional statement `j.onClick(h)`.

5.2.3 Modeling of Menus

Menus are common elements of user interfaces in many GUI-driven applications. A menu allows users to select from a set of different actions represented by menu items. The relevant classes defined by the Android platform are `Menu` and `MenuItem`. Class `Menu` is not a subclass of `View`, but instead an abstract description of the view objects to be displayed on the screen to represent a menu window. Similarly, `MenuItem` is not a subclass of `View`. Instances of this class describe view objects that represent individual selectable items under a menu. For simplicity, we treat `Menu` and `MenuItem` as view classes in both the formal semantics and the analysis. Two types of menus are most frequently used in Android applications: options menu and context menu.

Options menu An options menu is displayed when the user presses the MENU button. Typically, one activity can be associated with one options menu. For an options menu to exist, certain callback methods[3] should be defined. A `Menu` object is created by the Android framework, and passed into these callback methods. In these methods, developers can associate `MenuItem` objects with the `Menu`. The implicit creation of `Menu` objects can be modeled in a way similar to the implicit creation of activity objects. Furthermore, at any given moment of time, at most one options menu can be associated with an activity. So, a generalization of the semantics is

[3] For example, `Activity.onCreateOptionsMenu` and `Activity.onPrepareOptionsMenu`.

$\mathit{Heap} = \ldots \cup \mathit{Activity} \times \{\texttt{options}\} \rightarrow \mathit{View}$ where `options` is an artificial field for the activity and the view referenced by the field represents the `Menu`. Consider an activity class `a` which defines a required callback method `m`. Conceptually, we can model the effects of the callback to `m` as "$t_1 := \ldots;\ t_2 := \texttt{new}\ \mathit{Menu};\ t_1.\texttt{options} := t_2;\ t_1.m(t_2);$", where t_1 contains a reference to an instance of an activity class.

In the callback methods, `Menu.add` can be called on the `Menu` object to create a new menu item and associate it with the `Menu`. First, we need to model the implicit creation of a menu item object $y := \texttt{new}\ \mathit{MenuItem}$. Then, we can apply the rule $\textsc{AddView}_2$ on the existing `Menu` object (as the parent) and the newly created `MenuItem` object referenced by y (as the child). Alternatively, we can represent the semantics effects with the following rule

$$
\begin{array}{ll}
[\textsc{MenuAdd}] & \langle z := x.m(\ldots), \rho, \eta \rangle \rightarrow \\
& \langle \rho[z \mapsto \xi_{\mathit{MenuItem}}], \\
& \eta[(\rho(x), \texttt{children}) \mapsto \{\xi_{\mathit{MenuItem}}\} \cup \eta(\rho(x), \texttt{children})] \rangle
\end{array}
$$

where $\rho(x) \in \mathit{View}$ represents a `Menu` and ξ_{MenuItem} represents the newly-created `MenuItem` object.

Another way to construct the menu items for a menu is to call the API method `MenuInflater.inflate`. This call takes an integer-typed parameter corresponding to an XML file that defines the menu structure. The XML file defines the menu items to be created and associated with the menu. This is similar to the layout definition XML files discussed in Section 5.2.2. The difference is that the menu object, which is the root of the XML-defined menu structure, has been created before the inflater call. In the inflater call, the value of the integer parameter to specify the XML file comes from a constant defined in the `R.menu` class, which we refer to as menu ids. We extend the syntax of statements with $s ::= \ldots \mid x := \texttt{R.menu}.f$ to reflect the occurrences of menu ids. The effects of inflating a menu can be captured by the same inflater semantic functions $\mathcal{I}_N$ and $\mathcal{I}_E$ defined earlier in Section 5.2.2. For a menu id id_m, $\mathcal{I}_N[\![id_m]\!](\rho, \eta)$ is a set of pairs $(\xi_{MenuItem}, id)$, one for each layout node $(MenuItem, id)$. Here $\xi_{MenuItem}$ is a new heap location representing an instance of `MenuItem`, and id is the corresponding view id. Also similar to before, $\mathcal{I}_E[\![id_m]\!](\rho, \eta)$ defines pairs (ξ, ξ') that correspond to layout edges, except that the parent view node is now a `Menu` that has been created before the inflater call. Therefore, the semantics of an inflater call is

$$[\textsc{MenuInflate}] \quad \langle x.m(y, z), \rho, \eta\rangle \rightarrow \langle \rho, \eta' \rangle$$

where x is a helper object provided by the framework to implement the inflation, y contains a reference to the previously-created `Menu` object ξ_{Menu}, and z is the integer parameter specifying the menu id. In the updated heap η', $\xi_{MenuItem}.\texttt{vid} = id$, and $\xi_{Menu}.\texttt{children}$ contains all inflated $\xi_{MenuItem}$ objects.

Context menu A context menu is displayed when the user long clicks on a view, and the context menu is considered associated with the view. Similar to an options menu, a `ContextMenu` (subclass of `Menu`) object is created implicitly by the Android framework when the triggering event happens, and is passed to certain pre-defined callback methods `m` (e.g., `Activity.onCreateContextMenu`). The view object is also passed as a parameter of the callback. To model this behavior, the generalization of the semantics is $Heap = \ldots \cup View \times \{\texttt{context}\} \rightarrow View$ where `context` is an artificial field for the view. The effect of calling the callback method `m` can be modeled conceptually as "$t_1 := \ldots;\ t_2 := \texttt{new}\ ContextMenu;\ t_3 := \ldots;\ t_3.\texttt{context} := t_2;\ t_1.m(t_2, t_3);$", where t_1 contains a reference to an instance of the class that defines `m` and t_3 contains a reference to the view object to be associated with the `ContextMenu`.

In the callback method, `ContextMenu.add` can be called and the semantics is the same as `Menu.add`. Also in the callback method, a `MenuInflater.inflate` call could be made to construct the menu items for a `ContextMenu`; the modeling of this case is the same as discussed earlier.

Event listeners for menus and menu items Event listeners can be associated with menus and menu items to define the effect of selecting a menu item. Calls for listener association can be identified as set-listener operations and modeled using the SetListener semantic rule.

5.2.4 Modeling of Lists

`ListView` is a frequently used Android GUI construct to display a list of similar entities. For example, a list of files and directories can be displayed in a `ListView`. Each entity in the list is a list item. A list item can be as simple as a sequence of characters, or it could have its own structure and contain a set of other GUI constructs. Considering the same example, a file could be displayed with its file name, and a directory could be displayed with its name plus a folder-like icon. The Android platform provides specific APIs for construction and management of `ListView` objects. The semantics of these APIs need to be considered in our formal development of ALite.

Modeling of `ListActivity` and `ListActivity.getListView` `ListActivity` is a subclass of `Activity` that is guaranteed to contain a `ListView` in its view hierarchy. The contained `ListView` must be associated with the platform-defined view id `list`. Given a `ListActivity`, the corresponding `ListView` can be located by calling the `getListView` method, which is equivalent to calling `findViewById` with id list. However, if no view hierarchy is currently associated with the ListActivity, a platform-defined layout list_layout will be inflated on-demand when getListView is called. (Thus, the developer has the option to not explicitly call setContentView on a ListActivity.) The modeling of a call to getListView is as follows:

$$[\textsc{FindListView}] \quad \langle z := x.m(), \rho, \eta\rangle \rightarrow \langle \rho'', \eta''\rangle$$

if $\langle z := x.\mathit{findView}_2(\texttt{list}), \rho', \eta'\rangle \rightarrow \langle \rho'', \eta''\rangle$. Here $\langle \rho', \eta'\rangle$ is defined either by $\langle \rho, \eta\rangle$ if $\eta[(\rho(x), \texttt{root})]$ exists, or by $\langle x.\mathit{Inflate}_2(\texttt{list\_layout}), \rho, \eta\rangle \rightarrow \langle \rho', \eta'\rangle$ otherwise.

Modeling of ListView and ListAdapter As discussed earlier, ListView is used to display a list of similar entities. For each ListView, a ListAdapter manages the underlying data items and displays them in the rows of the list view. The association between ListView and ListAdapter is established through the setAdapter API call. To model this association, we generalize the semantics with $\mathit{Heap} = \ldots \cup \mathit{ListView} \times \{\texttt{adapter}\} \rightarrow \mathit{Adapter}$ where adapter is an artificial field for ListView referencing adapter objects represented by set *Adapter*. The effects of setAdapter can be represented by

$$[\textsc{SetAdapter}] \quad \langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{adapter}) \mapsto \rho(y)]\rangle$$

where $\rho(x)$ and $\rho(y)$ represent a ListView instance and a ListAdapter instance, respectively. The getView method of the associated ListAdapter is responsible for defining View objects for list items in the ListView. When a ListView needs to be displayed, getView is called by the Android platform to construct list item objects, which are added, also by the platform, as child views into the ListView. This interaction among ListView, its associated ListAdapter, and the underlying platform can be modeled by "$t_1 := x.\texttt{adapter};\ t_2 := t_1.\texttt{getView}();\ x.\texttt{addView}(t_2);$", where x contains a reference to a ListView object.

5.2.5 Modeling of Dialogs

In GUI applications, a dialog is typically used to display short messages or ask the user a brief question. In the Android platform, a dialog is very similar to an activity in that (1) a dialog is associated with a hierarchy of views; (2) views in a dialog can be associated with listeners; (3) the operations on views discussed earlier (Section 5.2.2) also apply to views associated with dialogs; and (4) dialogs have lifecycle callbacks such as `onCreate`. To model dialogs, we first generalize the semantics with $\mathit{Heap} = \ldots \cup \mathit{Dialog} \times \{\texttt{root}\} \rightarrow \mathit{View}$ where `root` is an artificial field for the dialog. Then, the semantic rules for activities are extended to handle dialogs as well. For example, the change needed for rule $\textsc{Inflate}_2$ is to let $\rho(x) \in \mathit{Activity} \cup \mathit{Dialog}$ for an inflater operation $x.m(y)$. The rules that require such changes are $\textsc{Inflate}_2$, $\textsc{AddView}_1$, and $\textsc{FindView}_2$.

5.3 Static Reference Analysis

Given the abstracted language ALite, we aim to develop a static analysis of the creation and propagation of views, as well as their interactions with activities, dialogs[4], listeners, and other views.[5] Specifically, the analysis

- defines static abstractions of run-time objects: views, activities, and listeners
- models the flow of (references to) such objects to stack variables and object fields

[4] Since handling of dialogs and activities are very similar, for brevity we will only discuss activities.

[5] As described in Section 5.2.3, menus and menu items are treated as views, so we discuss only “ordinary” views.

- determines the relevant structural relationships, including (1) associations of views with activities and listeners, and (2) parent-child relationships between views

A similar problem for the plain-Java language JLite can be solved using standard existing techniques. We consider one such solution, based on the construction and analysis of a *constraint graph*. A graph node corresponds to $x \in \mathit{Var}$ (a variable node), $f \in \mathit{Field}$ (a field node), or an expression $\texttt{new}\ c$ (an allocation node; the set of these expressions will be denoted by Alloc). Edges represent constraints on the flow of values. For example, an assignment $x := y$ is mapped to an edge $y \rightarrow x$, to encode the constraint that any value that flows to y also flows to x. Similarly, $x := \texttt{new}\ c$ is mapped to $\texttt{new}\ c \rightarrow x$ to represent the constraint that $\texttt{new}\ c$ is among the values that flow to x. Reachability from an allocation node determines all locations to which references to the corresponding run-time objects can flow. Such an analysis is usually referred to as a control-flow/calling-context-insensitive, field-based reference analysis [60, 95], and is the starting point for our analysis for Android. Various refinements of this technique have been investigated (e.g., [60, 99, 100]); our analysis developments for Android are orthogonal to these refinements and can be combined with them.

5.3.1 Constraint Graph

Figure 5.3 shows several constraint graph nodes and edges for the running example. Some of the nodes have subscripts referring to the line numbers from Figure 5.1 where the corresponding element occurs for the first time. Additional nodes and edges are shown in Figure 5.4; gray nodes represent views.

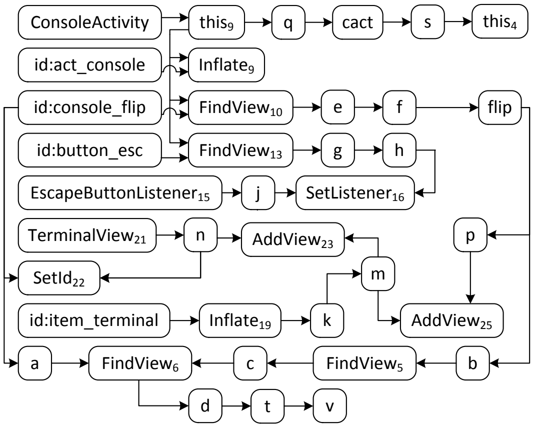


Figure 5.3: Partial constraint graph for the running example.

Nodes For every integer value in `R.layout`, there is a layout id node $id_l \in \mathit{LayoutId}$. Similarly, a view id node $id_v \in \mathit{ViewId}$ corresponds to each value from `R.id`. Next, an activity node $act \in \mathit{Activity}$ is created for each activity class, to represent instances of this class created implicitly by the Android platform (such instances are never created by `new` in the application code). As a minor abuse of notations, we use *Activity* to represent both the semantic domain of heap locations of activity objects in Section 5.2, as well as the set of constraint graph nodes for these objects (i.e., their static abstractions) in this section. The same convention is followed for other semantic domains and corresponding constraint graph nodes. Four id nodes, as well as the activity node for `ConsoleActivity`, are shown in Figure 5.3.

A view inflation node $view_{infl} \in \mathit{ViewInfl}$ is introduced for each layout node from XML layouts. This node represents the view created during inflation—that is, the heap object ξ_v created for a layout node (v, id), as defined by rules $\textsc{Inflate}_{1,2}$. If the same layout is inflated in several places in the application, a "fresh" set of graph nodes is introduced at each inflation site. Six view inflation nodes are illustrated in

Figure 5.4; a subscript $x.y$ refers to the y-th object inflated at line x from Figure 5.1. We also distinguish the subset of allocation nodes $\mathit{ViewAlloc} \subset \mathit{Alloc}$ that instantiate view classes, and use $\mathit{view}_{\mathit{alloc}}$ to denote such nodes. Similarly, let $\mathit{Listener} \subset \mathit{Alloc}$ be the subset of allocation nodes that instantiate listener classes; elements of this set are denoted by *lst*. In general, any object could be a listener, including activities and views. To simplify the presentation we assume that activities and views are not listeners, but our implementation handles the general case.

The flow of nodes $\mathit{view} \in \mathit{View} = \mathit{ViewInfl} \cup \mathit{ViewAlloc}$ and the associations of such nodes with *act* and *lst* nodes are the core concern of the analysis. This requires modeling of the operations described earlier. For each call $z := x.m(y)$ corresponding to one of the semantic rules, an operation node $op \in \mathit{Op}$ is added to the graph, and the nodes for variables x, y, and z are connected to it. For example, for the find-view operation `d=c.findViewById(a)` at line 6, the graph contains a *FindView* node with incoming edges from `c` and `a`, and an outgoing edge to `d` (shown in Figure 5.3).

Edges In addition to the JLite-based edges described earlier, the constraint graph contains edges for Android features. An assignment $x :=$ `R.layout.`f results in an edge $id_l \rightarrow x$ from the corresponding layout id node to the variable node x. Similar edges are added for view id nodes id_v. For an activity node *act*, an edge is added from it to all $\texttt{this}_m$ variable nodes, where m is a callback method that could be invoked by the framework with this activity as the receiver object. For example, in Figure 5.3 there is an edge from the activity node to parameter $\texttt{this}_9$ of `onCreate`.

All edges described so far model the flow of values. We also use edges $n \Rightarrow n'$ to represent constraints on other relevant relationships. For example, an edge $\mathit{view}_1 \Rightarrow \mathit{view}_2$ between two view nodes shows a parent-child relationship—that is,

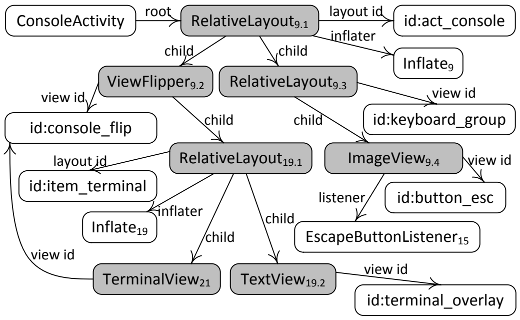


Figure 5.4: Additional graph nodes and edges.

the constraint $view_2 \in view_1.\texttt{children}$. An edge $view \Rightarrow id_v$ indicates that the view is associated with this view id (by rules $\textsc{Inflate}_{1,2}$ and $\textsc{SetId}$). An edge $view_{\mathit{infl}} \Rightarrow id_l$ connects the root of an inflated hierarchy with the layout id of the layout that was inflated. Similarly, $view_{\mathit{infl}} \Rightarrow \mathit{Inflate}_{1,2}$ is introduced when the view is the root of the hierarchy inflated by this *Inflate* operation node. An edge $act \Rightarrow view$ indicates that the view is the root of the hierarchy associated with the activity (as set up by rules $\textsc{Inflate}_2$ and $\textsc{AddView}_1$). Finally, $view \Rightarrow lst$ shows that the view is associated with this listener because of rule $\textsc{SetListener}$. All these categories of edges are illustrated in Figure 5.4, with edge labels added for clarity. Although not covered by the example, similar edges related to options menus, context menu, and their menu items are also part of the constraint graph.

5.3.2 Constraint-Based Analysis

We define the analysis in terms of constraints over the nodes and edges of the graph, with the help of two binary relations. First, $\mathit{ancestorOf} \subseteq \mathit{View} \times View$ is the transitive closure of the parent-child relation: $view_1$ $\mathit{ancestorOf}$ $view_2$ if and only if there exists a path in the constraint graph starting at $view_1$, ending at $view_2$, and containing only view nodes and ⇒ edges labeled with *child*. The second relation is $\mathit{flowsTo} \subseteq (\mathit{View} \cup \mathit{LayoutId} \cup \mathit{ViewId} \cup \mathit{Activity} \cup \mathit{Listener}) \times (\mathit{Var} \cup \mathit{Field} \cup \mathit{Op})$. This relation shows that the value represented by the first node—a view, an id, an activity, or a listener—flows to the variable, field, or operation represented by the second node. Both relations can grow during the analysis. For example, when two views flow to an *AddView* operation node (corresponding to rule $\textsc{AddView}_2$), a new parent-child edge is added to the constraint graph, which in turn affects *ancestorOf*. The basic inference rules for these two relations are as follows:

$$\frac{n_1 \in \mathit{View} \cup \mathit{LayoutId} \cup \mathit{ViewId} \cup \mathit{Activity} \cup \mathit{Listener} \qquad n_2 \in \mathit{Var} \qquad n_1 \rightarrow n_2}{n_1 \;\; \mathit{flowsTo} \;\; n_2}$$

$$\frac{n_2 \in \mathit{Var} \cup \mathit{Field} \qquad n_3 \in \mathit{Var} \cup \mathit{Field} \cup \mathit{Op} \qquad n_2 \rightarrow n_3 \qquad n_1 \;\; \mathit{flowsTo} \;\; n_2}{n_1 \;\; \mathit{flowsTo} \;\; n_3}$$

$$\frac{\mathit{view} \in \mathit{View}}{\mathit{view} \;\; \mathit{ancestorOf} \;\; \mathit{view}}$$

$$\frac{\mathit{view}_1 \;\; \mathit{ancestorOf} \;\; \mathit{view}_2 \qquad \mathit{view}_2 \Rightarrow \mathit{view}_3}{\mathit{view}_1 \;\; \mathit{ancestorOf} \;\; \mathit{view}_3}$$

For example, in Figure 5.3, view id `console_flip` flows to operation node $\mathit{FindView}_6$ via variable node `a`, and view $\texttt{TerminalView}_{21}$ flows to SetId_{22} and $\mathit{AddView}_{23}$ via `n`. Considering the parent-child edges in Figure 5.4, the root node $\texttt{RelativeLayout}_{9.1}$ is an ancestor of seven nodes.

The inference rules for the semantic rules are described below. For example, for $\textsc{AddView}_1$ we have

$$\frac{\mathit{act} \;\; \mathit{flowsTo} \;\; \mathit{AddView}1 \qquad \mathit{view} \;\; \mathit{flowsTo} \;\; \mathit{AddView}1}{\mathit{act} \Rightarrow \mathit{view}}$$

where *act* is the activity node. Similarly,

$$\frac{\mathit{view}_1 \;\; \mathit{flowsTo} \;\; \mathit{AddView}2 \qquad \mathit{view}_2 \;\; \mathit{flowsTo} \;\; \mathit{AddView}2}{\mathit{view}_1 \Rightarrow \mathit{view}_2}$$

assuming that $view_1$ flows to the operation node $AddView_2$ in the role of the parent. For example, `TerminalView`$_{21}$ flows to $AddView_{23}$ in the role of the child (Figure 5.3). As described shortly, `RelativeLayout`$_{19.1}$ flows to this operation in the role of the parent, via `k` and `m`. As a result, a parent-child edge `RelativeLayout`$_{19.1}$ ⇒ `TerminalView`$_{21}$ is created by the analysis, as shown in Figure 5.4.

For semantic rules SetId and SetListener we have

$$\frac{view \;\; flowsTo \;\; SetId \qquad id_v \;\; flowsTo \;\; SetId}{view \Rightarrow id_v}$$

$$\frac{view \;\; flowsTo \;\; SetListener \qquad lst \;\; flowsTo \;\; SetListener}{view \Rightarrow lst}$$

In Figure 5.3, both `TerminalView`$_{21}$ and `console_flip` flow to $SetId_{22}$. This leads to the creation of `TerminalView`$_{21}$ ⇒ `console_flip` (shown in Figure 5.4), which in turn affects relation *ancestorOf* and the find-view operations.

For rules FindView$_{1,2,3}$ the constraints are

$$\frac{\begin{array}{c} view_1 \;\; flowsTo \;\; FindView1 \qquad id_v \;\; flowsTo \;\; FindView1 \\ FindView1 \rightarrow n \quad view_1 \; ancestorOf \; view_2 \quad view_2 \Rightarrow id_v \end{array}}{view_2 \;\; flowsTo \;\; n}$$

$$\frac{\begin{array}{c} act \;\; flowsTo \;\; FindView2 \qquad id_v \;\; flowsTo \;\; FindView2 \\ FindView2 \rightarrow n \qquad act \Rightarrow view_1 \\ view_1 \;\; ancestorOf \;\; view_2 \qquad view_2 \Rightarrow id_v \end{array}}{view_2 \;\; flowsTo \;\; n}$$

$$\frac{view_1 \;\; flowsTo \;\; FindView3 \qquad FindView3 \rightarrow n \qquad view_1 \; ancestorOf \; view_2}{view_2 \;\; flowsTo \;\; n}$$

For example, `ConsoleActivity` and id `button_esc` flow to $FindView_{13}$, and the outgoing edge is to variable `g`. Furthermore, `ConsoleActivity` ⇒ `RelativeLayout`$_{9.1}$ because this view is the root of the hierarchy inflated by $Inflate_9$ and associated with the activity. This root is an ancestor of `ImageView`$_{9.4}$, which has an edge to the same view id. Thus, the analysis can conclude that $\texttt{ImageView}_{9.4}$ *flowsTo* `g`. Later this is used to determine that the view flows to $\mathit{SetListener}_{16}$.

Recall that semantic rule $\textsc{FindView}_3$ retrieves some descendant view with a particular run-time property. The static approximation is to assume that any descendant view can be retrieved, as shown in the constraint rule for *FindView*3 operation nodes. Sometimes more restricted semantics applies: for example, for the call to `getCurrentView()` at line 5 in Figure 5.1, any child view can be retrieved, but not any deeper descendant. Such refinements are not discussed, but they are employed by our implementation.

For rules $\textsc{Inflate}_{1,2}$, suppose that a layout id id_l flows to an *Inflate* operation node. In that case, the corresponding layout is inflated and its root node is connected with the inflater node and with the layout id (to capture the origin of the inflated hierarchy). The rules are

$$\frac{\begin{array}{cc} id_l \ \ \mathit{flowsTo} \ \ \mathit{Inflate}1 & \mathit{Inflate}1 \rightarrow n \\ \mathit{view} \Rightarrow \mathit{Inflate}1 & \mathit{view} \Rightarrow id_l \end{array}}{\mathit{view} \ \ \mathit{flowsTo} \ \ n}$$

$$\frac{\begin{array}{cc} \mathit{act} \ \ \mathit{flowsTo} \ \ \mathit{Inflate}2 & id_l \ \ \mathit{flowsTo} \ \ \mathit{Inflate}2 \\ \mathit{view} \Rightarrow \mathit{Inflate}2 & \mathit{view} \Rightarrow id_l \end{array}}{\mathit{act} \Rightarrow \mathit{view}}$$

In the first case, the root is propagated to the left-hand side variable at the inflater call. For example, $\mathit{Inflate}_{19}$ has an outgoing edge to `k`, and the analysis determines that $\texttt{RelativeLayout}_{19.1}$ flows to `k` (and from there to several other nodes). In the second case, the call associates the activity with the root object: e.g., at $\mathit{Inflate}_9$ an edge $\texttt{ConsoleActivity} \Rightarrow \texttt{RelativeLayout}_{9.1}$ is created.

Additional semantic rules introduced in Sections 5.2.3–5.2.5 can all be represented by some composition of the rules introduced in this section; these details are omitted.

Algorithm 5.1: ReferenceAnalysis($\mathcal{A}$)

```
   Input: Source code and XML resources of an Android application $\mathcal{A}$
   Output: Constraint graph $G$
   Output: Sets $solutionReceiver$, $solutionParameter$, $solutionResult$
   // Build an initial constraint graph, similar to the one shown in Figure 5.3
1  $G \leftarrow$ ConstructInitialConstraintGraph($\mathcal{A}$)
2  Initialize($G$)
3  ProcessInflaterCalls($G$)
4  $changed \leftarrow$ true
5  while $changed$ do
6      $changed \leftarrow$ false
7      if ProcessAddView1($G$) then
8          $changed \leftarrow$ true
9      if ProcessAddView2($G$) then
10         $changed \leftarrow$ true
11     if ProcessSetId($G$) then
12         $changed \leftarrow$ true
13     if ProcessSetListener($G$) then
14         $changed \leftarrow$ true
15     if ProcessFindView1($G$) then
16         $changed \leftarrow$ true
17     if ProcessFindView2($G$) then
18         $changed \leftarrow$ true
19     if ProcessFindView3($G$) then
20         $changed \leftarrow$ true
```

5.3.3 Analysis Algorithm and Implementation

To find a solution to the system of constraints, we employ a fixed-point algorithm. The overall process is outlined in Algorithm 5.1. Lines 1–3 correspond to the initialization stage of the algorithm, and the fixed-point computation is shown at lines 4–20. The output is the constraint graph together with three sets *solution*.... As discussed later, this information can be used to answer various queries about the flow of views and about their associations with activities, listeners, and other views.

Initial constraint graph First, the analysis creates an initial constraint graph (line 1 in Algorithm 5.1). This graph contains all edges that can be directly inferred from program statements: for example, edges due to assignments and object allocation. All edges in Figure 5.3 fall in this category. Each method in the application code is considered executable and thus analyzed. Polymorphic calls are resolved using class hierarchy information. Calls to application methods result in constraint graph edges that represent parameter passing and return values.

The abstracted semantics refers to a small number of broad categories of relevant operations (e.g., AddView, SetListener, etc.) which in reality correspond to a wide variety of Android APIs. Some of these APIs have semantic variations that are not discussed here, but are handled by our implementation. Occurrences of these APIs in the application code are recognized and modeled appropriately in the constraint graph. The effects of callbacks from the Android platform are also modeled at this time, as outlined at the end of Section 5.2.2. However, instead of creating explicit statements, the analysis simply adds constraint graph nodes and edges to simulate the corresponding semantic effects.

Helper data structures The rest of the analysis uses several helper data structures to encode reference flow information based on the constraint graph. One example is *solutionReceiver*, a set containing pairs $(view, op)$ where operation node $op \in AddView_2 \cup SetId \cup SetListener \cup FindView_1 \cup FindView_3$ and node *view* flows to *op* as the receiver object of the operation. Sets *solutionParameter* and *solutionResult* also contain pairs $(view, op)$, where the view flows in *op* as a parameter, or flows out of *op* as a result.

Four sets *reaching*... are used to collect certain reachability information. These sets are computed by the call at line 2 in Algorithm 5.1. The invoked procedure Initialize uses graph reachability to compute relationships that do not depend on the effects of operation nodes. Examples of such relationships include *id flowsTo n* and *act flowsTo n*. For example, set $reachingLayoutIds \subseteq LayoutId \times Op$ encodes the

Algorithm 5.2: INITIALIZE(G)

```
   Input: Constraint graph $G$
   Output: Sets $solution\ldots$ and $reaching\ldots$
1  foreach $id_l \in LayoutId$ do
2      foreach $op \in (Inflate_1 \cup Inflate_2)$ reachable from $id_l$ do
3          $reachingLayoutIds \leftarrow reachingLayoutIds \cup \{(id_l, op)\}$
4  foreach $id_v \in ViewId$ do
5      foreach $op \in (FindView_1 \cup FindView_2 \cup SetId)$ reachable from $id_v$ do
6          $reachingViewIds \leftarrow reachingViewIds \cup \{(id_v, op)\}$
7  foreach $act \in Activity$ do
8      foreach $op \in (Inflate_2 \cup FindView_2 \cup AddView_1)$ reachable from $act$ do
9          $reachingActivities \leftarrow reachingActivities \cup \{(act, op)\}$
10 foreach $lst \in Listener$ do
11     foreach $op \in SetListener$ reachable from $lst$ do
12         $reachingListeners \leftarrow reachingListeners \cup \{(lst, op)\}$
13 foreach $view \in View$ do
14     foreach $op \in (AddView_2 \cup SetId \cup SetListener \cup FindView_1 \cup FindView_3)$ with receiver reachable from
       $view$ do
15         $solutionReceiver \leftarrow solutionReceiver \cup \{(view, op)\}$
16     foreach $op \in (AddView_1 \cup AddView_2)$ with parameter reachable from $view$ do
17         $solutionParameter \leftarrow solutionParameter \cup \{(view, op)\}$
18 COMPUTEPATHEDGES($G$)
```

layout ids that reach an inflate operation node op. Sets $reaching\ldots$ are computed as expected (lines 1–12 in Algorithm 5.2). Furthermore, views that flow directly to operation nodes can also be computed at this stage of the analysis, as shown at lines 13–17. For example, in Figure 5.3, due to path $TerminalView_{21} \rightarrow n \rightarrow AddView_{23}$, the pair $(TerminalView_{21}, AddView_{23})$ is added to set $solutionParameter$.

Path edges Of course, indirect flow of a view to an operation is also possible. For example, in Figure 5.3, path $Inflate_{19} \rightarrow k \rightarrow m \rightarrow AddView_{23}$ propagates the output of one operation node to the input of another. To represent such flow, we introduce *path edges* in the constraint graph. Each such edge $op_1 \rightarrow op_2$ starts with a source operation $op_1 \in (Inflate_1 \cup FindView_1 \cup FindView_2 \cup FindView_3)$. A receiver path edge has $op_2 \in (AddView_2 \cup SetId \cup SetListener \cup FindView_1 \cup FindView_3)$, while

Algorithm 5.3: ProcessInflaterCalls(G)

```
   Input: constraint graph $G = (N, E)$
1  foreach $(id_l, op) \in reachingLayoutIds$ do
2      $xml \leftarrow$ ParseXml($id_l$)
3      $rootNode \leftarrow$ null
4      foreach $(c \in ViewClasses, f \in \{$R.id.$*\})$ in xml in depth-first order do
5          $viewNode \leftarrow new\ view_{infl,c} \in ViewInfl$
6          $idNode \leftarrow unique\ id_v\ node\ for\ f$
7          $N \leftarrow N \cup \{viewNode, idNode\}$
8          $E \leftarrow E \cup \{viewNode \overset{\text{vid}}{\Longrightarrow} idNode\}$
9          if $rootNode =$ null then
10             $rootNode \leftarrow viewNode$
11         else
12             $E \leftarrow E \cup \{parentNode \overset{\text{child}}{\Longrightarrow} viewNode\}$
13     if $op \in Inflate_1$ then
14         $solutionResult \leftarrow solutionResult \cup \{(rootNode, op)\}$
15         PropagateAlongPathEdges($G, op, rootNode$)
16     else
           // $op \in Inflate_2$
17         foreach $(act, op) \in reachingActivities$ do
18             $E \leftarrow E \cup \{act \overset{\text{root}}{\Longrightarrow} rootNode\}$
```

a parameter path edge has $op_2 \in (AddView_1 \cup AddView_2)$. The subsequent fixed-point computation propagates information only along path edges, as discussed shortly. Line 18 in Algorithm 5.2 contains a call to a helper function ComputePathEdges to perform this computation. This function, which is not shown here, performs graph reachability from the left-hand-side variable at each $Inflate_1$ and $FindView_{1,2,3}$ node to identify all reachable receivers and parameters of operation nodes.

Inflation operations Given the reachability information, $Inflate_{1,2}$ nodes are processed (based on reaching layout ids) to create inflated view nodes and the parent-child edges for them. This processing is done at line 3 in Algorithm 5.1, through the call to Algorithm 5.3. Helper function ParseXml parses the corresponding layout definition file. In a depth-first traversal of the hierarchy, the parent node (line 12) is easy to obtain. Different variations of the inflater semantics are handled as necessary, and edges to represent relevant semantic effects (e.g., the association between an

Algorithm 5.4: PROCESSADDVIEW1(G)

```
Input: Constraint graph $G = (N, E)$
Output: Boolean indicating whether new edges were added
$changed \leftarrow$ false
foreach $AddView_1$ node $op$ do
    foreach $(act, op) \in reachingActivities$ do
        foreach $(view, op) \in solutionParameter$ do
            $candidate \leftarrow$ edge $act \overset{\mathtt{root}}{\Longrightarrow} view$
            if $candidate \notin E$ then
                $E \leftarrow E \cup \{candidate\}$
                $changed \leftarrow$ true
return $changed$
```

activity and a root GUI object at INFLATE$_2$) are introduced. In the case when the inflater call returns the root node, helper function PROPAGATEALONGPATHEDGES (called at line 15; the function's body is not shown) considers all path edges from *op* to other operation nodes, and adds *rootNode* to the corresponding sets *solutionReceiver* and *solutionParameter* at the target nodes.

Fixed-point computation In the final phase of the analysis, a fixed-point computation propagates views through the constraint graph based on the path edges (lines 4–20 in Algorithm 5.1). For each node that has a view as input or output, a set of reaching views is maintained and updated as necessary. This information is represented by the three sets *solution*... described earlier.

The processing of most operation nodes is straightforward. For example, the code for PROCESSADDVIEW1 is shown in Algorithm 5.4. Functions PROCESSADDVIEW2 and PROCESSSETID are defined similarly, and are not shown here. For a $FindView_{1,2,3}$ node op_1, when a new view is resolved as the output result of op_1, this view is propagated along path edges to each operation op_2 whose receiver or parameter depends on the output of op_1. This processing is illustrated by Algorithm 5.5.

Algorithm 5.5: PROCESSFINDVIEW1(G)

```
Input: Constraint graph $G = (N, E)$
Output: Boolean indicating whether new edges were added
$changed \leftarrow$ false
foreach $FindView_1$ node $op$ do
    foreach $(view_1, op) \in solutionReceiver$ do
        foreach $view_2$ reachable from $view_1$ along edges $\overset{child}{\Longrightarrow}$ do
            foreach edge $view_2 \overset{vid}{\Longrightarrow} id_v$ do
                if $(id_v, op) \in reachingViewIds$ then
                    $candidate \leftarrow (view_2, op)$
                    if $candidate \notin solutionResult$ then
                        $solutionResult \leftarrow solutionResult \cup \{candidate\}$
                        $changed \leftarrow$ true
                        PROPAGATEALONGPATHEDGES($G, op, view_2$)
return $changed$
```

For each receiver $view_1$ at a $FindView_1$ node, the hierarchy rooted at this view is examined for a descendant node $view_2$ whose view id matches an id that reaches the operation. Each such descendant is added to *solutionResult*, indicating that the return value of the operation could be a reference to this view. Furthermore, helper function PROPAGATEALONGPATHEDGES (discussed earlier) propagates $view_2$ from op to other operation nodes. The processing of $FindView_{2,3}$ nodes (lines 17 and 19 in Algorithm 5.1) is done in a similar manner.

The processing of *SetListener* nodes is described in Algorithm 5.6. In addition to associating a view with a listener object (line 5), new constraint graph edges may be created to represent the flow of views and listeners to the corresponding event handlers (lines 9–10). Given the listener object *lst* and the set-listener statement, RESOLVEHANDLER considers the event handler signatures in the corresponding listener interface, and performs virtual dispatch using the type of *lst* to find the corresponding event handler methods. For each such method, the listener object node

Algorithm 5.6: ProcessSetListener(G)

```
   Input: Constraint graph $G = (N, E)$
   Output: Boolean indicating whether new edges were added
1  $changed \leftarrow$ false
2  foreach $SetListener$ $node$ $op$ do
3      foreach $(view, op) \in solutionReceiver$ do
4          foreach $(lst, op) \in reachingListeners$ do
5              $candidate \leftarrow$ edge $view \xRightarrow{listener} lst$
6              if $candidate \notin E$ then
7                  $E \leftarrow E \cup \{candidate\}$
8                  $changed \leftarrow$ true
                   // ResolveHandler resolves the set of possible event handler methods
                   // corresponding to the given listener object $lst$ and $SetListener$ node $op$
9                  foreach $m \in$ ResolveHandler($lst$, $op$) do
                       // $this_m \in Var$ represents "this" variable of $m$; $par_m \in Var$ represents
                       // the formal parameter of $m$ getting the view object
10                     $E \leftarrow E \cup \{lst \rightarrow this_m, view \rightarrow par_m\}$
                       // Propagate $lst$ and $view$ to receivers and parameters of operation
                       // nodes, in a way similar to lines 11-12 and 14-17 in Algorithm 5.2
11                     PropagateToOpNodes($lst$, $view$)
12 return $changed$
```

is connected to the `this` variable node, and the view object (i.e., the receiver object for the set-listener node) is connected to the corresponding parameter node. For the running example in Figure 5.1, due to the set-listener call at line 16, edges $\texttt{EscapeButtonListener}_{15} \rightarrow \texttt{this}_{31}$ and $\texttt{ImageView}_{9,4} \rightarrow \texttt{r}$ are created, since the resolved event handler is `onClick` (defined at lines 30–34 in Figure 5.1).

Due to these additional edges, reachability in the constraint graph may change and the solution sets may need to be updated. Specifically, the relevant listener object and view object ($\texttt{EscapeButtonListener}_{15}$ and $\texttt{ImageView}_{9,4}$ in this example) are propagated to the receivers and parameters of reachable operation nodes. This is done in a way similar to lines 11–12 (for listeners) and 14–17 (for views) from Algorithm 5.2. The propagation is performed by calling helper function PropagateToOpNodes (line 11 in Algorithm 5.6).

Analysis implementation Our implementation is based on the Soot analysis framework [109]. Soot's intermediate representation can be constructed either from source code, or from the Dalvik bytecode specific to Android [13, 81]. Certain Android GUI features are not handled by the current implementation (e.g., fragments). Another limitation is that native code is not analyzed, although we have not observed native code that creates GUI objects or registers listeners.

5.3.4 Analysis Output

The information computed by the analysis can be used to answer two types of queries. First, as with traditional reference analysis, the basic query is: *Given a variable x, what are the objects that x may reference?* Such information is widely used by various other static analyses (e.g., to perform control-flow analysis and data dependence analysis). In this work we focus on variables that can reference view objects. Given a view-typed variable x, the query is answered by traversing the constraint graph backward from x. Any reached view node (allocated or inflated) is included in the answer. In addition, if the left-hand-side variable at an operation node is reached, set *solutionResult* for this node is also included in the answer.

The output of the proposed analysis can also provide essential information about the structure and behavior of the application's GUI. This information plays a key role in program understanding, test generation, and profiling. One key question that can be answered is the following: *For each user-visible window (i.e., activity, menu, or dialog), what are the GUI widgets and what is their hierarchical structure?* The constraint graph can be used to answer this question directly. Another important question is: *Which widgets in a window have application-defined behavior and what* *are the handler methods implementing this behavior?* This question can be answered by considering *SetListener* nodes and the corresponding pairs (*view*, *lst*) derived from sets *solutionReceiver* and *solutionParameter* at these nodes. Several case studies presented later indicate that the proposed analysis answers these questions with very high precision.

5.4 Experimental Evaluation

We apply the analysis on 20 open-source Android applications and seek to answer the following questions. First, how often do these applications use the Android features modeled by our analysis? This characterization is presented in Section 5.4.1. Second, how precisely does the analysis answer queries related to these features? Section 5.4.2 and Section 5.4.3 address this question.

5.4.1 Application Characteristics

Characteristics of the 20 applications are described in Table 5.1 and Table 5.2. Almost all programs have been used in prior work [87,122,124,128]. The tables show the number of application classes and methods, as well as the breakdown of constraint graph nodes. These measurements characterize the relevant application features and provide motivation for the proposed analysis of GUI-related behavior.

Columns “activities”, “menus”, and “dialogs” measure the number of independent on-screen GUI windows that can be presented to the user. The total number of windows (activities + menus + dialogs) in an analyzed application is typically larger than 20. It likely would be impractical to manually create and maintain models of these GUIs for the purposes of program understanding and test generation (e.g., as

App	Classes	Methods	activities	menus	dialogs	ids (L/V)	views (I/A)	listeners
APV	68	415	4	4	5	4/14	92/21	17
Astrid	1228	5782	41	3	48	106/237	1487/46	177
BarcodeScanner	126	594	9	4	6	14/34	181/0	12
Beem	284	1883	12	6	5	25/54	200/0	13
ConnectBot	371	2366	11	8	17	21/48	404/7	53
FBReader	954	5452	27	9	8	27/117	390/9	45
K9	815	5311	32	3	19	43/168	733/8	56
KeePassDroid	465	2784	20	11	9	29/84	349/12	32
Mileage	221	1223	50	15	9	36/64	529/0	30
MyTracks	485	2680	32	8	20	34/163	2218/4	29
NPR	249	1359	13	12	6	21/92	348/9	17
NotePad	89	394	8	3	10	10/17	225/4	9
OpenManager	60	252	8	2	9	10/54	331/0	21
OpenSudoku	140	726	10	6	18	17/36	446/6	16
SipDroid	331	2863	12	5	13	8/51	326/4	10
SuperGenPass	65	268	3	3	4	8/15	144/0	11
TippyTipper	57	241	6	3	0	7/42	147/22	27
VLC	242	1374	10	2	13	44/110	457/11	46
VuDroid	69	385	3	2	1	4/7	23/6	4
XBMC	568	3012	22	20	24	33/171	931/23	107

Table 5.1: Analyzed applications, and object and id nodes in the constraint graph.

needed in prior work [55, 105, 122, 123]). The output of our analysis can be useful for automated construction and evolution of structural and behavioral GUI models.

Column "ids" shows the number of layout ids (L) followed by the number of view ids (V). Based on the measurements of layout ids as well as inflater nodes (column "Inflate"), it is clear that XML layouts are widely used and their flow/use be modeled in a static analysis. Another observation is that the number of view ids is large, and their use must be accounted for in a static analysis, especially because the number of find-view operations where these ids are used (column "FindView") is also quite large.

Column "views" shows the number of inflated (I) and explicitly allocated (A) view nodes. The large number of views implies a complex GUI structure that requires careful modeling (e.g., in order to generate representative input events for profiling and for high test coverage). Most views are inflated, but explicitly allocated views

App	Inflate	FindView	AddView	SetId	SetListener
APV	6	8	43	0	19
Astrid	84	491	82	1	188
BarcodeScanner	12	37	1	0	11
Beem	24	55	1	0	29
ConnectBot	24	75	37	1	57
FBReader	25	140	28	0	45
K9	45	254	23	2	101
KeePassDroid	37	122	25	0	31
Mileage	37	106	29	6	31
MyTracks	26	156	2	0	25
NPR	21	150	22	0	40
NotePad	7	26	23	0	10
OpenManager	14	64	16	0	22
OpenSudoku	17	77	31	3	17
SipDroid	7	68	20	0	10
SuperGenPass	5	15	1	0	11
TippyTipper	8	43	20	0	30
VLC	51	159	13	1	69
VuDroid	5	8	7	0	4
XBMC	31	270	116	0	80

Table 5.2: Operation nodes in the constraint graph.

are also present in 15 out of the 20 applications. Explicit manipulation of the view hierarchy via add-child operations (column “AddView”) occurs in all 20 applications. Our analysis was specifically designed to handle all these features. Event handlers (column “listeners”) and the associated set-listener operations (column “SetListener”) are commonly used by the applications. Static control/data flow analysis for Android must account for the association between views and the event handlers that respond to them, since they play a critical role in the run-time control flow.

5.4.2 Analysis Cost and Precision

Table 5.3 shows the running time of the analysis and measurements of the computed solution. Even for the larger programs, the analysis time is very practical. Column “receivers” shows the average number of view objects that are receivers at operation nodes (e.g., *FindView* and *AddView*2). Smaller numbers imply higher precision, with 1 being the lower bound. For 16 out of the 20 programs, this average is

App	Time	Average number			
	(sec)	receivers	parameters	results	listeners
`APV`	0.70	1.00	1.00	1.44	1.00
`Astrid`	7.42	3.42	2.74	1.84	1.37
`BarcodeScanner`	0.76	1.20	1.00	1.10	1.07
`Beem`	0.80	1.04	1.00	1.08	1.00
`ConnectBot`	1.59	1.00	1.03	1.02	1.02
`FBReader`	3.57	1.49	1.11	1.72	1.37
`K9`	5.81	1.14	1.04	1.21	1.03
`KeePassDroid`	2.00	1.81	1.00	1.80	1.06
`Mileage`	0.63	2.38	1.17	2.12	1.82
`MyTracks`	1.32	1.31	1.00	2.49	1.41
`NPR`	0.71	1.86	1.00	1.61	4.94
`NotePad`	0.60	1.02	1.00	1.00	1.00
`OpenManager`	0.71	1.41	1.00	1.69	1.24
`OpenSudoku`	0.54	1.07	1.06	1.06	1.20
`SipDroid`	1.05	1.66	1.00	1.00	1.50
`SuperGenPass`	0.24	2.00	1.00	1.94	1.96
`TippyTipper`	0.66	1.15	1.00	1.00	1.34
`VLC`	0.93	1.11	1.08	1.12	1.00
`VuDroid`	0.34	1.21	1.00	1.00	1.00
`XBMC`	1.86	7.23	1.17	1.78	2.84

Table 5.3: Analysis running time (in seconds) and average number of objects in the solution for operation nodes.

less than 2. Similar observations can be made for column "parameters", which shows the average number of views reaching an *AddView* node as a parameter. Column "results" shows how many views, on average, are results (i.e., outputs) from operations such as *FindView*. Finally, column "listeners" shows how many listener objects, on average, are associated with a view object at a set-listener operation. As with the other measurements, this number is typically small, indicating good precision.

Case studies In order to further evaluate the precision of the analysis, detailed case studies were performed on six applications: `APV`, `BarcodeScanner`, `OpenManager`, `SuperGenPass`, `TippyTipper`, and `VuDroid`. These applications have the smallest number of windows (activities + menus + dialogs), as shown in Table 5.1, and thus were chosen to make it feasible to perform comprehensive manual examination of their behavior and code.

By manually reasoning about the application and all solution sets (whose sizes are presented in Table 5.3), we determined which solution elements are part of the "perfectly-precise" static solution (i.e., the solution capturing all and only possible run-time behaviors). For `APV`, `SuperGenPass`, `TippyTipper`, and `VuDroid`, our solution achieves perfect precision.

For `BarcodeScanner`, the perfectly-precise measurements would be 1.15 for "receivers" (instead of 1.20), 1.08 for "results" (instead of 1.10), and unchanged for the other two columns. The imprecision is due to the lack of integer range analysis. Specifically, there is a $FindView_3$ operation "`lhs = view.getChildAt(i)`" which retrieves the i-th child view of the input view. Since the possible values of i are not modeled by the analysis, all child views are considered as possible outputs for this operation, while in reality one of them is infeasible. Later, the output view referenced by *lhs* is used as a receiver for another operation. Therefore, the over-approximation for the output of this $FindView_3$ operation affects both the "receivers" and "results" measurements.

In `OpenManager`, there is a `WirelessManager` activity which cannot be triggered by any means. If we consider it as dead code and exclude it from the solution, the recomputed measurements would be 1.43 for "receivers", unchanged for "parameters", 1.63 for "results", and 1.25 for "listeners". Compared with this refined solution, the perfectly-precise measurements would be 1.21 for "receivers", unchanged for "parameters", 1.13 for "results", and still 1.25 for "listeners". After manual investigation, we found that three $Inflate_2$ operations are performed independently on the same dialog object in three different branches of a switch statement, with the same layout id. Right after each $Inflate_2$ operation, a $FindView_2$ operation is performed to retrieve

App	Windows	Views	Handlers
`APV`	92.31%	97.30%	97.56%
`BarcodeScanner`	100%	100%	100%
`OpenManager`	100%	100%	79.66%
`SuperGenPass`	100%	100%	96.77%
`TippyTipper`	100%	100%	100%
`VuDroid`	100%	100%	100%

Table 5.4: Run-time coverage for the 6 selected apps.

some view, followed by a *SetListener* operation to associate a listener with this result view. For each $FindView_2$ operation in one branch, due to the control-flow-insensitive nature of the analysis, views specific to the other two branches are also included in set *solutionResult*, corresponding to the “results” column. Then, the propagation of views from $FindView_2$ to *SetListener* further affects set *solutionReceiver*, corresponding to the “receivers” column. In short, the lack of flow-sensitive treatment renders the analysis imprecise for this particular case.

The conclusion from these case studies is that, for these six applications, the precision of the analysis is very close to (and in some cases the same as) the perfect solution. This indicates that static modeling of inputs/output of GUI-related operations can be performed with high precision.

5.4.3 Case Studies of GUI Structure and Behavior

As discussed earlier, in addition to traditional reference analysis queries about the possible values of view-typed variables, the analysis output can also provide structural and behavioral information about the GUIs themselves: specifically, (1) the GUI widget structure for each window, and (2) the application-specific event handlers associated with these widgets. Using the same six applications described earlier, we performed case studies to evaluate the precision of the analysis solution with respect to this information. These results have direct relevance to existing model-based testing approaches for Android [55, 105, 122, 123], where knowledge of GUI hierarchies and event handlers is needed to create GUI models.

In the case studies we examined the windows, views, and associated event handlers reported by the static analysis. We then tried to achieve run-time coverage of these entities, or to confirm that that such coverage cannot be achieved in any run-time execution. Specifically, for each window w (an activity, a menu, or a dialog reported by our analysis), the goal was to display window w on the screen. Next, for each pair (w, v), where the analysis reported view v as part of the GUI hierarchy of w, the goal for run-time coverage was to display the widget corresponding to v when window w was active. To make the case study effort feasible, we focused on views v for which the analysis reported at least one application-defined event handler h associated with v, and on windows w containing at least one such v. Finally, for each triple (w, v, h), where handler method h was associated with v in the analysis solution, the goal was to invoke h by triggering some event on the v, when v was displayed as part of w.

Table 5.4 shows the coverage measurements (as percentages) for the six applications. In most cases, 100% coverage can be achieved, indicating a low false positive rate of the analysis. Similar to the treatment discussed at the end of Section 5.4.2, dead code was excluded from these measurements.

Case studies For those elements that could not be covered at 100%, we investigated the source code to understand why coverage is infeasible and thus determine why the analysis is imprecise for these cases. The result of this investigation is discussed below.

In `APV`, there is a dialog to display an error message when PDF rendering cannot be done successfully. It requires a very specific corrupted PDF file input to craft a test execution that can cover the dialog, and further the associated views and event handlers. However, we were unable to obtain such a PDF file from the application developers[6] or to create one ourselves. If such a PDF file were available, 100% coverage would have been achieved.

In `OpenManager`, as discussed in Section 5.4.2, due to the flow-insensitive nature of the analysis, some views are falsely considered possible receivers for certain *SetListener* operations. As a result, resolved listener objects and the corresponding event handler methods would be falsely associated with these views. Given the high cost of a flow-sensitive reference analysis, addressing this imprecision efficiently remains an interesting open question.

In `SuperGenPass`, one event handler method `onNothingSelected` cannot be covered by any run-time execution. It is associated with a drop-down list view (more precisely, an `android.widget.Spinner`), and it is supposed to be invoked when an empty "default" item is defined in the application and selected at runtime. However, such a "default" item is not defined in this particular application, so, correspondingly, coverage of `onNothingSelected` is indeed infeasible.

Despite these examples of imprecision, the overall conclusion is that the GUI hierarchies and the related event handlers in the studied applications can be inferred precisely. This is a promising indication that precise reverse engineering of GUI models for Android through static analysis is feasible.

[6] http://goo.gl/IhtY05

5.4.4 Discussion

The evaluation strongly suggests that real-world applications commonly use the Android features we aim to model, and that our approach analyzes these features with high precision and low running time. This makes the proposed analysis a promising building block for a variety of other analyses for Android.

5.5 Summary

Building a foundation of static analyses for Android is essential for new compile-time and run-time techniques and tools in this increasingly important area of computing. We propose the first static analysis to focus on GUI-related Android objects. The analysis defines abstractions of views (including menus), activities, dialogs, and listeners. It models the flow of such objects, the effects of Android operations, and the relevant structural relationships, including associations of views with activities/dialogs and listeners, and parent-child view relationships. Our constraint-based algorithm exhibits high precision and low cost. This analysis is an important building block for existing and future compiler analyses, profiling techniques, static error checkers, security analyses, and testing approaches.

CHAPTER 6: Static Analysis of the Android Activity Stack

In the previous chapter we proposed a static reference analysis for GUI objects in Android software. The focus was the creation, propagation, and interaction of Android GUI objects. In this chapter, we consider other aspects of the Android runtime behavior—*activity stacks and activity lifecycles*—which are at the core of the Android application execution model.

As discussed earlier, an activity is an important application component that represents an on-screen window for users to interact with. Responding to certain user events and system events, one activity may start another to display new GUI hierarchies. For example, the `ConnectBot` app [26], a popular SSH client for Android, has an activity to display a list of SSH servers. When the user selects one such server in the list, a new activity is started to display the login screen and later the SSH session upon successful authentication. Furthermore, the same activity can be started by activities defined in other apps as well. For example, the user may click on a link to an SSH server displayed in a web browser app, and start the activity defined in `ConnectBot` for a connection to that server. When the user is done with the SSH connection, the activity in `ConnectBot` is dismissed and destroyed while the web browser app resumes. Although the activities are implemented by two different apps, what the user perceives is still one integrated app. The Android platform allows such seamless user experience by running the activities in the same *task*.

A task is a collection of activities relevant to the currently active user interactions. In most cases, these activities are organized in a stack, referred to as an *activity stack*. When one activity starts another, the new one is pushed on the top of the stack and takes focus. The previous activity is stopped but still kept in the stack. When the BACK button is pressed, the top activity on the stack is popped and the previous activity is resumed and brought back to the foreground. If the user keeps pressing the BACK button, the control of execution will ultimately go to the HOME screen, an activity of the special `Launcher` app running in another task. At this point, the user could either go into other apps or get back to the original app. This behavior is made possible by maintaining multiple tasks (each with its own activity stack) in the background.

Depending on the state of the activity stack, the accessible user events are different, leading to different feasible sequences of method calls. Modeling the semantics of the activity stack is a foundational requirement for modeling of control/data flow of an Android application, which in turn is a critical building block for many other software analyses and tools for Android. One can draw an analogy with the *call stack* of an ordinary Java program: static modeling of the possible states of the call stack is a key concern of interprocedural control-flow and data-flow analysis. However, the behavior of the activity stack is significantly more complicated, as described shortly. Furthermore, this behavior drives other critically-important behaviors, including the sequence of calls to manage the lifecycle of activities, and the flow of data between activities.

6.1 Background and Example

Recall from Chapter 4 that activities have well-defined lifecycles. Callback methods can be defined to handle different stages of the activity lifecycle. For example, an activity is visible and can respond to user events only in between calls to its `onResume` and `onPause` callback methods. These two lifecycle callback methods define the *foreground* lifetime of an activity. When the execution is outside the foreground lifetime of an activity, either before `onResume` or after `onPause`, it is impossible to trigger an event handler method as a response to a user event performed on a GUI object in this activity.

The pairing of `onResume` and `onPause` defines an ordering constraint with respect to one single activity. That is, for each activity object, `onPause` can only be called after `onResume`. There are also ordering constraints between methods that involve multiple activities. Informally, when an activity starts another, some methods must be called to (1) pause the execution of the current activity, and (2) create and start the new activity. On the other hand, when execution goes from a current activity back to a previous activity, some other methods must be called to (1) stop and destroy the current activity, and (2) resume the previous activity. Understanding such invocation sequences is the basis for modeling control/data flow of an Android application.

One key prerequisite for the analysis of method invocation sequences is to model the behavior of the *activity stack*. The effects and corresponding invoked methods of an activity launch or termination event are dependent on the state and changes in the activity stack. For example, the handling of certain start activity event needs to examine and even manipulate the activity stack. When BACK button is pressed, the state of activity stack determines what the previous activity is. In the rest of this

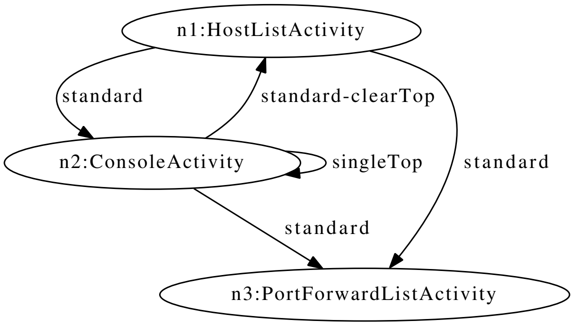


Figure 6.1: A partial activity transition graph for `ConnectBot`.

section, the complicated behavior of activity stacks and the invocation sequence of activity lifecycle callbacks is demonstrated through an example.

6.1.1 Example

`ConnectBot` [26] is an SSH client for Android. Three activities in this app are relevant to our discussion: `HostListActivity` displays a list of available SSH servers; when a server is selected, `ConsoleActivity` displays a terminal view to host the SSH session; `PortForwardListActivity` allows the user to edit the port forward list for a selected server.

The starting point of understanding the activity stack is the modeling of transitions between activities. As proposed by existing work (e.g., [11]), activities and the transitions between them can be represented in an *activity transition graph.* Figure 6.1 shows part of such a graph for `ConnectBot` [26]. In this graph, each node represents the top activity in the current activity stack. The edges represent the events that may change the activity stack, including particularly the top activity (thus the name "activity transition"). The transition events include two categories: launch of an activity and termination of an activity. For simplicity, edges for activity termination

($n2 \xrightarrow{\texttt{finish}} n1$, $n3 \xrightarrow{\texttt{finish}} n2$, and $n3 \xrightarrow{\texttt{finish}} n1$) are omitted from the figure. The shown edges all represent events of activity launch, and they are annotated with the associated event configurations.

Main activity Each Android application must define a main activity. The main activity is the first displayed window when a user launches the application. In this particular example, `HostListActivity` (represented by node $n1$) is the main activity. When `ConnectBot` is launched, this activity is created and shown to the user. Conceptually, the pseudocode for the executed statements is "`a = new HostListActivity; a.onCreate(...); a.onStart(); a.onResume()`". A `HostListActivity` instance is first created, and then the lifecycle callback methods (e.g., `onStart()`) are invoked in the specific sequence on this instance. At this point, the activity stack contains only one activity, and can be written as ($\xi_{\texttt{HostListActivity}}$) where $\xi_{\texttt{HostListActivity}}$ represents an object of type `HostListActivity`. If the BACK button is pressed now, the activity will be stopped and popped from the stack, and control will return to the HOME screen. The resulting behavior is equivalent to a sequence of statements "`a.onPause(); a.onStop(); a.onDestroy()`".

Launch with `standard` When a user selects a server shown by `HostListActivity`, the GUI event listener invokes API method `startActivity` in order to start an instance of `ConsoleActivity` to display the SSH session. This activity launch uses the `standard` configuration and corresponds to edge $n1 \xrightarrow{\texttt{standard}} n2$ in the graph. Here `standard` is the default and most commonly used configuration for an activity launch event. Assuming that variable `a` references the `HostListActivity` instance, the runtime behavior is equivalent to the sequence of statements "`a.onPause(); b = new ConsoleActivity; b.onCreate(...); b.onStart(); b.onResume(); a.onStop()`".

In addition, the newly created `ConsoleActivity` instance is pushed on to the activity stack, changing it to $(\xi_{\texttt{HostListActivity}}, \xi_{\texttt{ConsoleActivity}})$ where the rightmost element represents the top of the stack. Similar execution sequences can be observed when user selects a server to edit its port forward list (edge $n1 \xrightarrow{\texttt{standard}} n3$).

Termination When a user finishes the SSH connection with `ConsoleActivity` and presses BACK button with the intention to return back to `HostListActivity`, the behavior occurs: "`b.onPause(); a.onRestart(); a.onStart(); a.onResume(); b.onStop(); b.onDestroy()`", assuming `a` references the `HostListActivity` instance and `b` references the `ConsoleActivity` instance. The `ConsoleActivity` instance $\xi_{\texttt{ConsoleActivity}}$ is popped from the stack and destroyed, changing the stack back to $(\xi_{\texttt{HostListActivity}})$. Note that this transition corresponds to the omitted edge $n2 \xrightarrow{\texttt{finish}} n1$, which is different from $n2 \xrightarrow{\texttt{standard-clearTop}} n1$ (discussed shortly). Other omitted edges (e.g., $n3 \xrightarrow{\texttt{finish}} n1$) that represent activity termination have similar behavior.

Launch with `singleTop` When a user is in an SSH session with `ConsoleActivity`, one interesting action is to connect to another SSH server whose URL is shown on the screen. In such a case, a launch activity event is triggered to start a `ConsoleActivity` with the `singleTop` configuration, in order to host the new SSH session. The `singleTop` configuration requests the Android platform to reuse the activity instance on top of the stack if it is of the same type as the specified target activity. When this reuse condition is met, no new activity instance is created and the callback method `onNewIntent` is called on the existing top activity instance. For the edge

$n2 \xrightarrow{\texttt{singleTop}} n2$, this is indeed the case and the corresponding behavior can be modeled as "`b.onPause(); b.onNewIntent(...); b.onResume();`" (here variable `b` references the top of stack activity instance $\xi_{\texttt{ConsoleActivity}}$). In this process, the activity stack does not change.

Launch with `standard-clearTop` The `ConsoleActivity` also provides a short-cut button that allows a user to navigate to `HostListActivity`. The handler for the button starts `HostActivity` with configuration `standard-clearTop` (or simply `clearTop` since `standard` is the default implicit configuration). This corresponds to the edge $n2 \xrightarrow{\texttt{standard-clearTop}} n1$, and, as mentioned earlier, is different from the omitted edge $n2 \xrightarrow{\texttt{finish}} n1$. The `clearTop` configuration requests the Android platform to search for existing instance of the specified target activity. If such an instance is found in the activity stack, all the other instances on top of this matched instance will first be removed (thus the name `clearTop`). Next, when starting the target activity, the existing matched instance may or may not be reused, depending on whether `singleTop` or `standard` is specified. When the configuration contains `standard`, the existing matched instance will be removed as well, and a new instance will be created and pushed to the stack. For this example, a `HostListActivity` instance does already exist in the stack and the specified configuration is `standard-clearTop`, so the stack will change from $(\xi_{\texttt{HostListActivity}}, \xi_{\texttt{ConsoleActivity}})$ to $(\xi'_{\texttt{HostListActivity}})$, where $\xi'_{\texttt{HostListActivity}}$ is a new instance of `HostListActivity`. This behavior corresponds to the sequence "`b.onPause(); a.onDestroy(); a2 = new HostListActivity; a2.onCreate(...); a2.onStart(); a2.onResume(); b.onStop(); b.onDestroy();`".

Launch with `singleTop-clearTop` For the same transition between $n2$ and $n1$, the configuration `singleTop-clearTop` allows reuse of the matched `HostListActivity`.

If the launch activity event were configured with `singleTop-clearTop`, the stack after the event would be ($\xi_{\texttt{HostListActivity}}$) and the executed statements would be "`b.onPause; a.onNewIntent(...); a.onRestart(); a.onStart(); a.onResume(); b.onStop(); b.onDestroy();`" where variable `a` references $\xi_{\texttt{HostListActivity}}$ and variable `b` references $\xi_{\texttt{ConsoleActivity}}$ in the original stack before the event. Note that the existing `HostListActivity` is reused, and `onNewIntent` is called on the existing instance.

Launch with `reorderToFront` Another way to reuse a matched activity is to trigger the launch event with `reorderToFront`. The effect of this configuration is to pull the matched activity instance off the stack and place it back on the top. All other instances remain intact in the stack. If such a configuration were to be used in this example (instead of `singleTop-clearTop`), the sequence of executed statements "`b.onPause(); a.onNewIntent(...); a.onRestart(); a.onStart(); a.onResume(); b.onStop();`" would be slightly different, since `onDestroy` is not called on the `ConsoleActivity` instance. Correspondingly, the resulting activity stack would be ($\xi_{\texttt{ConsoleActivity}}$, $\xi_{\texttt{HostListActivity}}$) with the `HostListActivity` instance reordered to the top and the `ConsoleActivity` instance still in the stack.

6.2 Semantics of Relevant Android Constructs

The exact runtime behavior of an activity launch or termination event is at the core of understanding the interactions between activities. As demonstrated in the earlier section, this behavior depends heavily on the activity stack. For example, the state of the stack determines whether a new activity instance needs to be created, which specific activity gets displayed next, and how data are communicated between which specific activities.

A static analysis to model the state and changes of the activity stack is highly desirable as foundation for many other compile-time analysis techniques such as static error checking, security analysis, test generation, and automated debugging. To the best of our knowledge, such a static analysis does not currently exist. The goal of this work is to develop semantic foundations and analysis algorithms for solving this problem. We propose a principled solution, starting with a definition of the semantics of relevant Android constructs (this section) and using it to define a static analysis for it (next section). As a first step towards solving the full problem, this solution focuses only on cases where the application runs in one single task; future work will consider the analysis of multiple tasks.

6.2.1 Syntax and Semantics of ALite^{Stk}

This section describes ALite^{Stk}, an extension of ALite (defined in Section 5.2.2) which considers additional Android constructs that are relevant to modeling of activity stacks. For the syntax and semantics of other constructs, the reader should refer to Section 5.2.

First, the semantic domain is extended to include the activity stack $\delta \in \mathit{Loc} \times \mathit{Loc} \times \ldots \times \mathit{Loc}$. Specifically, $\delta = (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n})$ is a sequence of activity instances representing an activity stack, where ξ_{c_1} and ξ_{c_n} are the bottom activity and the top activity, respectively. Recall that $\xi_c \in \mathit{Loc}$ is a heap location labeled with the class c it instantiates. Here we will focus only on the classes $c \in \mathit{ActivityClasses}$. If necessary, different instances of the same activity class are additionally labeled with superscripts (e.g., ξ_c^1 and ξ_c^2) to avoid confusion. Note that for any two elements ξ_{c_i} and ξ_{c_j} in the stack, they can be of the same activity class ($c_i = c_j$), but they cannot be the same instance ($\xi_{c_i} \neq \xi_{c_j}$). With this extension, the semantic rules are now written as $\langle s, \rho, \eta, \delta\rangle \rightarrow \langle \rho', \eta', \delta'\rangle$, which describes the changes in environment, heap and activity stack when statement s is executed. The rules introduced in Section 5.2.2 can be easily extended by noting that $\delta' = \delta$ because the operations discussed there do not change the activity stack.

The second aspect of the extension is to consider operations that do affect the activity stack. Such operations are calls to platform APIs that may trigger activity launch or termination. There are many variations of such API calls, but conceptually they can be abstracted as the following 6 types of operations: `standard`, `singleTop`, `standard-clearTop`, `singleTop-clearTop`, `reorderToFront`, `finish`. Examples of these operations have already been shown in Section 6.1.

Operation `standard` This operation starts a specified target activity and pushes it to the stack, regardless of the state of the original stack. It is the default and most commonly used way to start an activity. The semantic rule is

$$[\textsc{Standard}] \quad \langle m(x), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n})\rangle \rightarrow \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n}, \xi_{\rho(x)})\rangle$$

where method call $m(x)$ represents a `standard` operation to start an instance of the activity class c specified by x (i.e., $c = \rho(x)$). In a minor abuse of notation, here c denotes both the activity class c and its name.

Operation `finish` This operation terminates the current activity—the one on top of the stack—and returns to the previous activity. The semantic rule is

$$[\textsc{Finish}] \quad \langle m(), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n})\rangle \rightarrow \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{n-1}})\rangle$$

If the stack becomes empty due to this operation, an additional effect is to exit the application.

Operation `singleTop` This operation and the other three operations discussed below are all activity launch operations that first attempt to find an existing instance of the specified target activity class, and then decide what to do next based on the result of the search. The `singleTop` operation compares the type of the current top activity with the target activity class. If they are the same, no new activity instance is created and instead the current top activity is reused. When satisfying activity instance cannot be found, the effect of the operations is the same as the `standard` operation. Thus, the rule for a `singleTop` operation is

$$[\textsc{SingleTop}] \quad \langle m(x), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n})\rangle \rightarrow \begin{cases} \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n})\rangle, & \text{if } \rho(x) = c_n \\ \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_n}, \xi_{\rho(x)})\rangle, & \text{otherwise} \end{cases}$$

For the remaining three operations, the behavior is similar: if the search for an existing instance of the target activity class does *not* succeed, the effect is the same as `standard`. Otherwise, each operation has its own specific behavior. To simplify the presentation, we will describe the semantics of the remaining operations in cases when the search does find a matching activity instance.

Operation `standard-clearTop` A `standard-clearTop` operation searches the activity stack from top to bottom to find the first activity of the same class as the target activity class. Then, it removes all activity instances above the matched one, as well as the matched one itself. Next, a new instance of the target activity is created and pushed to the stack. This behavior is captured by rule

$$[\textsc{Standard-ClearTop}] \quad \langle m(x), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi_{c_i}, \ldots, \xi_{c_n})\rangle \rightarrow \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi'_{c_i})\rangle$$

where $\rho(x) = c_i$ and $c_i \neq c_j$ for $i+1 \leq j \leq n$. Note that ξ'_{c_i} is a newly created activity instance, and thus is different from ξ_{c_i} in the old stack.

Operation `singleTop-clearTop` This operation is very similar to the previous one, but differs in that it will reuse the matched activity rather than creating a new one. The semantic rule is

$$[\textsc{SingleTop-ClearTop}] \quad \begin{array}{l} \langle m(x), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi_{c_i}, \ldots, \xi_{c_n})\rangle \rightarrow \\ \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi_{c_i})\rangle \end{array}$$

where $\rho(x) = c_i$ and $c_i \neq c_j$ for $i + 1 \leq j \leq n$. The activity instance ξ_{c_i} is reused.

Operation `reorderToFront` Similar to the previous two operations, `reorderToFront` also searches the activity stack from top to bottom to find the first matched activity, one that is of the same type as the target. Then, it removes this instance from the middle of the stack and pushes it back to the top of the stack:

$$[\textsc{ReorderToFront}] \quad \begin{array}{l} \langle m(x), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi_{c_i}, \xi_{c_{i+1}}, \ldots, \xi_{c_n})\rangle \rightarrow \\ \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{i-1}}, \xi_{c_{i+1}}, \ldots, \xi_{c_n}, \xi_{c_i})\rangle \end{array}$$

where $\rho(x) = c_i$ and $c_i \neq c_j$ for $i + 1 \leq j \leq n$. The activity instance ξ_{c_i} is *reordered* to the top of stack and reused, and no new activity instance is created.

Activity lifecycle callbacks Activities have well-defined lifecycles, and a series of lifecycle callback methods are invoked at different stages. As the operations discussed above are executed, the activities being added to or removed from the stack will transition between these stages. So, in addition to the semantics defined by the rules, another aspect of the semantics is the invocation of specific sequences of lifecycle callbacks. Based on the state and changes of the activity stack, such invocation sequences can be easily defined. The details of all such sequences, for each of the six rules from above, are already demonstrated by the examples in Section 6.1. For example, for a `singleTop` operation in the case when the stack top matches the target activity class, the top activity instance is subjected to the sequence of lifecycle callbacks `onPause()`, `onNewIntent(...)`, `onResume()`.

Operations `rotate`***,*** `home`***, and*** `power` In addition to the operations discussed earlier in this section, there are other operations that can invoke sequences of activity lifecycle callback methods. These operations correspond to the ROTATE, HOME, and POWER transitions defined in Chapter 4, and are thus called `rotate`, `home`, and `power` operations. The `rotate` operation is triggered when the user rotates the device screen. When this happens, (1) the top activity on stack is popped and destroyed, and (2) a new instance of the same activity class is created and pushed to the stack. This behavior can be expressed by the following semantic rule

$$[\textsc{Rotate}] \quad \langle m(), \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{n-1}}, \xi_{c_n})\rangle \rightarrow \langle \rho, \eta, (\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_{n-1}}, \xi'_{c_n})\rangle$$

where ξ_{c_n} is the top activity before the operation and ξ'_{c_n} is the newly created instance from the same activity class c_n. Assuming the top activity ξ_{c_n} is referenced by variable `a`, the corresponding conceptual sequence of executed statements is "`a.onPause(); a.onStop(); a.onDestroy(); b = new` c_n`; b.onCreate(...); b.onStart(); b.onResume()`". The `home` operation occurs when the user goes to the HOME screen and immediately goes back; the `power` operation occurs when the user locks the screen and immediately unlocks it. These two operations do not change the activity stack. However, when a `home` or a `power` operation is triggered, the top activity instance in the activity stack is subjected to the sequence of lifecycle callbacks "`onPause()`, `onStop()`, `onRestart()`, `onStart()`, `onResume()`" or the sequence "`onPause()`, `onResume()`", respectively.

6.3 Static Analysis of the Activity Stack

Given the abstracted language A<span style="font-variant:small-caps">Lite</span>^{Stk}, we aim to develop a static analysis of the activity stack, modeling its state and changes. The input to such an analysis is an *activity transition graph* which encodes the transitions between activities. In this graph, each node corresponds to an activity, and each edge between two nodes corresponds to one possible transition between them. Similar representations can be found in existing work (e.g., [11]). However, we additionally annotate the edges with the abstracted operations for activity launch from ALITE^{Stk}, to enable an analysis of the activity stack. The `finish` operation is generally feasible for each activity, as a response to either pressing the BACK button or some other user event. However, we do not represent it as part of this input graph because the target of a $\xrightarrow{\texttt{finish}}$ edge depends on the state of the activity stack and thus these edges instead are actually part of the analysis output. Operations `rotate`, `home`, and `power` are also generally feasible for each activity. Although they are not included as part of the activity transition graph, we do consider them later for the analysis of the activity stack. An example of an activity transition graph is shown in Figure 6.1. In the rest of this section, we first discuss the construction of this input graph. Then we present the analysis algorithm which takes this graph as input and produces the possible states of the activity stack.

6.3.1 Construction of Activity Transition Graph

The starting point for the construction of the activity transition graph is the *main activity*. The main activity is the activity specified by the application to be displayed when it is first launched. The operation to start this activity is performed by the platform, and is thus not part of the analyzed code. Correspondingly, there will not be an incoming edge for the main activity node in the graph to represent such an activity launch operation. Starting from the main activity, it is desirable to find all possible target activities, and further to find their targets as well. During this process, nodes and edges are constructed as necessary, and added to the graph. The graph is built by repeating this process until no new edges can be constructed.

Given an activity, the set of target activities it can transition to are determined by the set of all reachable activity launch API calls, i.e. call statements in reachable methods. Recall that the static reference analysis introduced in Chapter 5 provides the ability to retrieve the set of GUI objects associated with each activity, and the set of event handler methods associated with each GUI object. We can leverage this existing analysis to find all the GUI objects associated with the given activity and their event handlers. Then the set of these event handlers and their transitive callees are the set of reachable methods for this activity. In the actual implementation, additional processing is performed to take into account the execution of activity lifecycle methods, and also methods made reachable due to menus and dialogs.

Next, for each source activity *src* and each reachable activity launch API call, the analysis needs to understand and extract the target activity *tgt* and the corresponding abstract operation *op*, and create a new edge $src \xrightarrow{op} tgt$. One commonality among all the activity launch API calls is that an *intent* object is used to specify the target activity and the launch configurations, which correspond to *tgt* and *op*, respectively. There are several existing techniques [11, 20, 42, 82] for analysis of intent objects. Typically, these existing techniques leverage a data-flow analysis to infer the content of an intent object at each activity launch call and to determine the target activity. Similar techniques can be used to additionally identify the launch configurations, since they are set up via API calls that are similar to those used to set up the target activity of an intent. These launch configuration parameters imply the type of the

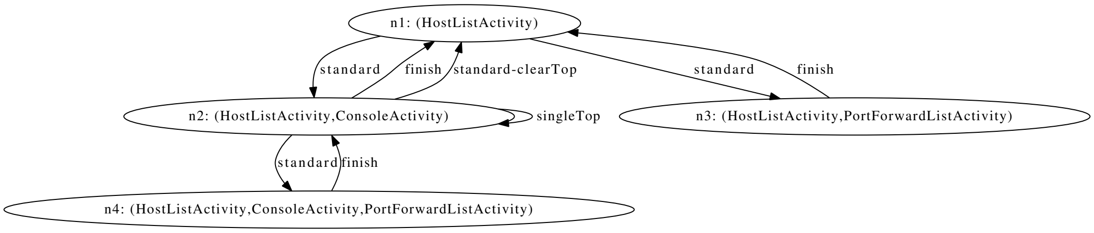

n1: (HostListActivity)
standard
finish
standard-clearTop
standard
finish
n2: (HostListActivity,ConsoleActivity)
singleTop
n3: (HostListActivity,PortForwardListActivity)
standard
finish
n4: (HostListActivity,ConsoleActivity,PortForwardListActivity)

Figure 6.2: Output stack transition graph for the input activity transition graph in Figure 6.1.

corresponding abstract operation *op*. Note that the type of abstract operation *op* also depends on the `launchMode` setting of the target activity defined in the Android manifest file. Our implementation handles such manifest settings as well.

6.3.2 Analysis Algorithm

With the activity transition graph, we can build an analysis to compute all possible activity stack states via graph traversal. For practical considerations, we further constrain the problem with the condition that each activity class has at most one instance in each of the valid stack states. In other words, for any two activity instances ξ_{c_i} and ξ_{c_j} in an activity stack, the activity classes c_i and c_j should be different ($c_i \neq c_j$).

The output of this analysis is a *stack transition graph* which encodes the possible stack states and the transitions between them. Figure 6.2 shows the output stack transition graph for the input activity transition graph shown in Figure 6.1. Each node in the stack transition graph represents one possible valid activity stack. Activity instances in the stack are simply represented by the activity class name since no two distinct instances would have the same name. Each edge between two nodes is labeled with the abstract operation that triggers the transition between the two corresponding

Algorithm 6.1: $\textsc{ActivityStackAnalysis}(ATG)$

```
Input: Activity transition graph $ATG = (N_{in}, E_{in})$, main activity node $main \in N_{in}$
Output: Stack transition graph $STG = (N_{out}, E_{out})$
// The initial stack contains only the main activity.  Act extracts the activity name from a
// $N_{in}$ node.  MakeNode finds or creates an $STG$ node for a given stack.
$initStack \leftarrow \textsc{MakeNode}(\langle \textsc{Act}(main) \rangle)$
// Initialize the output graph to contain the stack node representing the initial stack.
$N_{out} \leftarrow \{initStack\}$
$E_{out} \leftarrow \emptyset$
// MakeList creates a list containing only the specified element.  The created list supports
// RemoveFirst, removing the first element from the list, and AddFirst, inserting an element
// at the beginning of the list.
$worklist \leftarrow \textsc{MakeList}(initStack)$
while $worklist$ is not empty do
    $stack \leftarrow worklist.\textsc{RemoveFirst}()$
    $src \leftarrow stack.top()$
    // Considers the activity launch operations.
    foreach $src \xrightarrow{op} tgt \in E_{in}$ do
        // ConstructNewStack builds a new stack node based on the given input and the
        // semantic rules defined in Section 6.2.  null is returned if the new stack
        // contains two instances from the same activity class.
        $newStack \leftarrow \textsc{ConstructNewStack}(stack, op, tgt)$
        if $newStack \neq null$ then
            if $newStack \notin N_{out}$ then
                $worklist.\textsc{AddFirst}(newStack)$
                $N_{out} \leftarrow N_{out} \cup \{newStack\}$
            $E_{out} \leftarrow E_{out} \cup \{stack \xrightarrow{op} newStack\}$
    // Considers finish operation.  ConstructNewStack clones stack and pops the top element.
    $newStack \leftarrow \textsc{ConstructNewStack}(stack, \texttt{finish})$
    if $newStack \neq \emptyset$ then
        if $newStack \notin N_{out}$ then
            $worklist.\textsc{AddFirst}(newStack)$
            $N_{out} \leftarrow N_{out} \cup \{newStack\}$
        $E_{out} \leftarrow E_{out} \cup \{stack \xrightarrow{\texttt{finish}} newStack\}$
```

activity stacks. When multiple operations can trigger the same transition, multiple edges are created. Note that implicitly for each node, there are three default self-edges to represent the `rotate`, `home`, and `power` operations. For simplicity, we do not show these edges in the figure or in the algorithm described below.

Algorithm 6.1 performs an iterative depth-first traversal of the activity transition graph to implement the analysis. When the application first launches, the initial activity stack contains only the main activity and is the starting point of traversal.

Lines 1–4 reflect this initialization. Next, we consider both the activity launch operations (lines 8–14) and the activity termination operation (lines 15–20). A candidate stack *newStack* is first built from the current stack *stack*, the operation *op*, and the target activity (if it is a launch operation) using a helper function ConstructNewStack (lines 9 and 15). The helper function is straightforward to implement based on the semantic rules defined in Section 6.2. The candidate new stack is discarded if it is invalid, meaning that it contains multiple activity instances from the same class. When it is valid, a new edge $\mathit{stack} \xrightarrow{op} \mathit{newStack}$ should be added to the output graph (lines 14 and 20). Before doing that, if the node for *newStack* does not already exist in the output graph node set N_{out}, it should be created and saved in N_{out}, and also added to *worklist* (lines 11–13 and lines 17–19). When it already exists in N_{out}, the traversal does not continue along its direction, which is standard for depth-first traversal.

6.3.3 Analysis Implementation

Our implementation is based on the static reference analysis introduced in Chapter 5, which itself is based on the Soot analysis framework [109]. An extension needed for this work is the analysis of Android intent objects, in order to construct the activity transition graph. This extension uses techniques similar to those from existing work (e.g., [11, 20, 42, 82]), but focuses on explicit intents only, and implements the additional characterization of transition operation types (i.e., the transition edges are labeled with abstract operations such as `standard` and `singleTop`).

6.4 Control-Flow Analysis of Lifecycle Callbacks

In this section, we describe an extension of our technique which targets a standard and fundamental problem in static analysis: *control-flow analysis*. Since data-flow analysis has to model the program's control flow (in addition to the dataflow domain under consideration), control-flow analysis is also a key component of data-flow analysis.

Typically, control-flow analysis is performed on some representation of the program. The standard such representation is the interprocedural control-flow graph (ICFG). This graph combines the intraprocedural control-flow graphs (CFGs) of the program's procedures. Nodes correspond to statements, and intraprocedural edges show the control flow inside a procedure. The CFG for a procedure has a dedicated start node and a dedicated exit node. Each call is represented by two nodes: a call-site node c_i and a return-site node r_i. There is an interprocedural edge from a call-site node to the start node of the called procedure; there is a corresponding edge from the exit node to the return-site node. An ICFG path that starts from the entry of the main procedure is *valid* if the interprocedural edges along the path are matched [98]. One way to express the validity condition is through a standard context-free grammar [91]: $\textit{valid} \rightarrow c_i\ \textit{matched} \mid \textit{matched}$ where $\textit{matched} \rightarrow c_i\ \textit{matched}\ r_i\ \textit{matched} \mid \epsilon$. Here non-terminal *matched* represents paths along which each call-site node c_i is matched with the corresponding return-site node r_i. A valid path allows for yet-unmatched call-site nodes; these unmatched nodes correspond to the current state of the run-time call stack, and are often referred to as a *call string*.

The conceptual goal of control-flow analysis is to determine the set of all valid paths. In an actual static analysis, this goal needs to be refined: for example, since the number of valid paths is typically very large (and often infinite), some abstractions need to be defined based on the analysis goals. But, at its essence, the goal of control-flow analysis is to find all valid paths. The natural way to achieve this is to perform a graph traversal augmented with a call stack containing the currently-unmatched call-site nodes c_i. Whenever the traversal attempts to extend the current path with a new edge, the stack is consulted to determine if the extended path is valid (i.e., when the new edge leads to r_i, the top of the stack must be c_i). In addition, the stack is updated as necessary (e.g., by pushing c_i or popping r_i). This abstracted view of control-flow analysis is used throughout the rest of this section.

For a framework-based platform such as Android, the application does not contain a main procedure from which control-flow paths start. The interaction between an application and the platform is through callbacks: the high-level view of the control flow is as a sequence of calls from (unknown) platform code to specific application methods. In this section we are specifically interested in callbacks to activity lifecycle methods such as `onCreate` and `onDestroy`. (Section 6.1 provides several examples of such methods.) We focus on these methods because the proper management of the activity lifecycle is an essential concern for Android developers (e.g., to avoid leaks [33, 102, 122]) and these methods define the core "skeleton" of the application's control flow.

Based on these observations, we consider *abstracted* interprocedural paths in which the only calls to and returns from lifecycle callback methods are represented, and all other nodes are abstracted away. Furthermore, since a lifecycle method completes execution before any other lifecycle method starts, the abstracted paths are always of the form $c_i\ r_i\ c_j\ r_j\ c_k\ r_k \ldots$ and will be represented simply as $m_i\ m_j\ m_k \ldots$ where, for example, c_i is an unknown call-site node in the platform code that invokes application-defined callback method m_i. Thus, we are interested in a specialized version of control-flow analysis which produces the set of all valid sequences of invocations of application-defined lifecycle callback methods.

A traditional approach for performing such an analysis is to create an abstract main procedure which captures all possible behaviors of the platform code, and then to analyze the abstract main together with the application code using standard interprocedural control-flow analysis. As an example, consider FlowDroid [10], a flow- and context-sensitive taint analysis which models the effects of callbacks by creating a wrapper main method. An example of such a wrapper method for the example from Section 6.1 is shown in Figure 6.3. It is represented as a `while`-loop, in each iteration of which some of the activities in the app may get the opportunity to execute. For example, execution of `HostListActivity` is represented by lines 4–18. First, line 5 creates a `HostListActivity` object. This object creation is not a call to lifecycle callbacks, but is part of the generated wrapper method and included for completeness. Then, it continues with a call to `onCreate` and ends with a call to `onDestroy`. The region between `onCreate` and `onDestroy` represents the *entire lifetime* of this activity. The two loops at lines 7–16 and lines 10–14 represent the *visible lifetime* and *foreground lifetime* of the activity, respectively. Similarly, they are bounded by pairs of lifecycle callbacks, with the slight variation for visible lifetime that a call to `onRestart` may get executed before calling `onStart`. The execution of `onRestart` is conditional because it is not called the first time an activity is started. The innermost loop (line 12) is an event loop to handle user actions (e.g., click on a button), and it is represented as calls to event handler callbacks (e.g., `onClick`), details of which are

```
 1 void wrapperMainMethod() {
 2   while (...) {
 3     // assume activities can run in any order
 4     if (...) {
 5       HostListActivity n1 = new HostListActivity();
 6       n1.onCreate(...);
 7       while (...) { // loop for activity visible lifetime
 8         if (...) n1.onRestart();
 9         n1.onStart();
10         while (...) { // loop for activity foreground lifetime
11           n1.onResume();
12           while (...) { ... } // event loop to handle user actions
13           n1.onPause();
14         }
15         n1.onStop();
16       }
17       n1.onDestroy();
18     }

19     if (...) {
20       ConsoleActivity n2 = new ConsoleActivity();
21       ...  }

22     if (...) {
23       PortForwardListActivity n3 = new PortForwardListActivity();
24       ...  }  }  }
```

Figure 6.3: FlowDroid-based wrapper main method for the example shown in Section 6.1.

omitted for simplicity. Lines 19–24 are the generated code for the other two activities, which has similar structures and thus is not displayed in detail. The code for each activity is enclosed in an `if`-statement to express the intended behavior that each activity may or may not execute in one iteration of the outer `while`-loop, reflecting the assumption that activities may execute in an arbitrary order. In this example, we show calls to all the callback methods for each activity, while in the actual implementation of FlowDroid, calls are put in the wrapper method only if the resolved call target is defined in the application code. In other words, line 13 would not exist in a real wrapper method generated by FlowDroid because `HostListActivity` does not override the default `Activity.onPause` lifecycle callback method.

As discussed, such a wrapper method can capture the control flow of lifecycle callbacks for each individual activity. However, since this wrapper was designed for a particular form of interprocedural taint analysis, and not to express the general control flow, computing the abstracted interprocedural paths based on this approach has certain limitations. Specifically, one cannot apply standard control-flow analysis on this wrapper to determine all possible sequences of callbacks.

The key limitation is that this approach does not model transitions and interactions between activities. As demonstrated in Section 6.1, lifecycle callbacks from different activities could interleave because the activity lifecycles are nested. Paths with such interleaved calls are not represented by this wrapper method. Consider an application with only the main activity `A` and one other activity `B`, and a very simple execution which (1) launches the app and starts `A`, (2) starts `B` from `A`, (3) presses BACK and returns to `A`, and (4) exits the app. In this execution, the lifetime of `B` is entirely nested in the lifetime of `A`. The corresponding subsequences of lifecycle callbacks for these steps are

1. `A.onCreate(...)`, `A.onStart()`, `A.onResume()`
2. `A.onPause()`, `B.onCreate(...)`, `B.onStart()`, `B.onResume()`, `A.onStop()`
3. `B.onPause()`, `A.onRestart()`, `A.onStart()`, `A.onResume()`, `B.onStop()`, `B.onDestroy()`
4. `A.onPause() A.onStop() A.onDestroy()`

The complete sequence is the concatenation of these four subsequences. For the wrapper shown in Figure 6.3, there does not exist a control-flow path that expresses this start-to-end sequence of callbacks. Since this approach does not consider such interleaving behaviors for callbacks made on different activities, as a side effect, another important callback `onNewIntent` (due to abstract operations such as `singleTop`) is not considered either. Thus, a conservative solution to the control-flow problem cannot be computed using the wrapper main from FlowDroid. In addition, there is inherent imprecision in this approach, as it assumes that activities can be ordered arbitrarily, while in reality such ordering can be significantly constrained by the possible activity transitions and the configurations associated with them.

The technique introduced in the previous section models the semantics of the activity stack, and can be used to address the limitations of this wrapper-based approach for control-flow analysis of lifecycle callbacks. Given an output STG, there is a clear mapping between each STG edge and its corresponding sequence of lifecycle callbacks. Paths in the STG correspond to abstracted interprocedural paths. The number of STG paths is infinite, so the number of abstracted interprocedural paths (i.e., sequences of lifecycle callbacks) is also infinite. As typical in static analysis, this infinite set should be abstracted by a finite abstraction. The choice of this abstraction, of course, depends on the goals of the client control-flow analysis. An STG traversal algorithm can be defined based on this target abstraction, and the current STG path, together with the associated sequence of callbacks, can be maintained during the traversal. Note that when STG paths are constructed, their corresponding callback sequences need to also consider (1) the sequence of lifecycle callbacks to start the main activity during app launch, as well as (2) the callback sequence to exit the app when the only activity on the stack is terminated. These additional sequences correspond to implicit activity launch and activity termination STG edges.

App	ATG		STG	
	Nodes	Edges	Nodes	Edges
APV	4	6	6	10
Astrid	8	9	10	18
BarcodeScanner	2	1	2	2
Beem	5	5	6	10
ConnectBot	11	12	12	23
FBReader	14	113	195716	391430
K9	7	8	8	14
KeePassDroid	13	42	645	1288
Mileage	12	16	17	32
MyTracks	12	39	144	286
NPR	8	45	1424	2846
NotePad	4	6	4	7
OpenManager	4	3	4	6
OpenSudoku	9	11	9	16
SipDroid	3	3	4	6
SuperGenPass	2	1	2	2
TippyTipper	5	7	8	14
VLC	3	2	3	4
VuDroid	3	6	5	8
XBMC	19	101	64473	138132

Table 6.1: Measurements of activity transition graph and stack transition graph.

6.5 Evaluation

We evaluate the proposed analysis on the same set of 20 open-source Android applications used in Chapter 5. Figure 6.1 shows the measurements of activity transition graph and stack transition graph. The implicit STG self-edges due to `rotate`, `home`, and `power` operations are not included in these measurements. The running time of this analysis is comparable to that of the static reference analysis in Chapter 5, and thus omitted. The measurements of nodes and edges in the activity transition graph are with respect to the subgraph reachable from the main activity. The analysis of activity stacks is performed on this subgraph to produce the corresponding stack transition graph.

Not all activities in every application are reachable and included in the traversed activity transition graph because (1) we focus only on explicit intents, (2) the analysis models only behaviors involving a single activity stack, and (3) certain Android features (e.g., implicit activity launch triggered by `TabHost`) are not modeled. Work to handle these features is orthogonal to the core contributions of this chapter, which are a formal semantics to reason about the behavior of the activity stack, as well as an algorithm to compute the possible stack states given the activity transition relationships as the input.

The output of our analysis explicitly represents the behavior of the stack and the interaction between activities. This output can be used by a client static analysis to prune infeasible control-flow paths due to constraints of the activity stack behavior, or by an automated test generation tool to cover more code paths that would be missing without knowledge of the activity stack.

As a characterization of precision, we also compare our analysis with a hypothetical naïve stack analysis, which assumes arbitrary execution order of activities. The assumption implies that activities could appear in arbitrary positions of the activity stack, which is similar in spirit to the approach in FlowDroid [10]. As in our STG-based analysis, the hypothetical analysis considers only stacks in which each activity class has at most one instance. Figure 6.2 shows the result of this comparison. Column “STG-based” shows the numbers of stacks computed by our stack analysis. These are the same numbers as those in the “STG Nodes” column in Figure 6.1. Column “Arbitrary” shows the numbers of stacks that would have been computed by the hypothetical analysis. This number is defined by $\sum_{i=1}^{k} P(n,i)$, where n is the

App	STG-based	Arbitrary
APV	6	40
Astrid	10	400
BarcodeScanner	2	4
Beem	6	85
ConnectBot	12	1111
FBReader	195716	62618582404
K9	8	8659
KeePassDroid	645	1359245485
Mileage	17	13344
MyTracks	144	4765344
NPR	1424	109600
NotePad	4	16
OpenManager	4	16
OpenSudoku	9	3609
SipDroid	4	15
SuperGenPass	2	4
TippyTipper	8	205
VLC	3	9
VuDroid	5	15
XBMC	64473	196476518410399

Table 6.2: Numbers of possible activity stacks based on the proposed STG-based analysis and based on the assumption of arbitrary ordering.

number of reachable activities from the main activity (column "ATG Nodes" in Figure 6.1), k is the maximum depth of the activity stacks computed by our analysis, and $P(n,i) = \frac{n!}{(n-i)!}$. The value of $P(n,i)$ shows the number of distinct stacks of length i whose elements are selected from the n activities, assuming that no activity appears more than once in a particular stack. The summation simulates an approach which considers all activity stacks of length $1, 2, \ldots, k$. As the differences between the two columns clearly demonstrate, the hypothetical analysis performs much worse than our stack analysis. The fundamental problem is the assumption of arbitrary execution order, due to lack of modeling of ordering constraints. Our analysis specifically targets such constraints.

6.5.1 Case Study

The `standard` abstract operation is the default and the most common way to start an activity. To the best of our knowledge, existing static analysis techniques for Android have all treated an activity launch as a `standard` operation. When an activity is started in this way, the behavior of the activity stack is immediately clear without further analysis: (1) a new instance of the target activity is created and pushed to the stack; and (2) when a `finish` operation occurs for that new instance, it is popped off the stack and the control returns back to the previous activity, which is well-defined by the stack state before execution of `standard`. The ability to determine the `finish` transitions allows the augmentation of the activity transition graph to represent them compactly and to facilitate development of clients of our analysis (e.g., test generation, program understanding, and correctness checking). Specifically, for each edge $(\xi_{c_1}, \xi_{c_2}, \ldots, \xi_{c_m}) \xrightarrow{\texttt{finish}} (\xi_{c'_1}, \xi_{c'_2}, \ldots, \xi_{c'_n})$ in the stack transition graph, we can augment the activity transition graph with a corresponding edge $c_m \xrightarrow{\texttt{finish}} c'_n$. Considering again the example shown in Section 6.1, these augmented edges derived from `finish` transitions actually refer to the omitted $\xrightarrow{\texttt{finish}}$ edges in Figure 6.1. Therefore, one interesting question is the following: how often do the studied applications use the other "non-standard" ways of activity launch discussed in this chapter? Note that non-self `singleTop` transitions such as $c_1 \xrightarrow{\texttt{singleTop}} c_2$ where $c_1 \neq c_2$ behave similarly to `standard` and do not require additional analysis. In the context of this question, we treat such `singleTop` operations as if they were `standard`.

Our experimental measurements show that 3 out of the 20 applications make use of "non-standard" activity launches (e.g., `standard-clearTop`). For these apps

App	k=2		k=4		k=6		k=8	
	#Seq	Length	#Seq	Length	#Seq	Length	#Seq	Length
APV	4	4.00	14	8.93	54	14.20	214	19.54
Astrid	7	6.14	45	12.82	294	19.55	1923	26.29
BarcodeScanner	2	5.00	2	11.00	2	17.00	2	23.00
Beem	4	4.00	14	8.21	54	12.39	214	16.56
ConnectBot	9	6.44	84	13.38	795	20.35	7530	27.31
FBReader	11	7.73	527	15.31	20728	22.11	668642	28.50
K9	3	4.33	9	9.00	32	13.03	122	16.66
KeePassDroid	2	4.00	23	11.26	415	18.01	6506	24.33
Mileage	4	2.75	25	6.76	207	10.92	1790	14.95
MyTracks	9	7.22	109	15.01	1313	22.61	15354	30.08
NPR	7	7.14	139	15.18	2192	22.29	30341	29.10
NotePad	4	5.25	16	10.75	64	16.25	256	21.75
OpenManager	4	2.75	12	5.08	36	7.42	108	9.75
OpenSudoku	4	4.25	17	8.94	75	13.53	333	18.13
SipDroid	3	4.33	7	8.57	18	12.56	47	16.47
SuperGenPass	2	4.50	2	10.50	2	16.50	2	22.50
TippyTipper	4	5.50	16	11.75	74	17.86	358	23.91
VLC	3	5.00	6	11.50	12	18.00	24	24.50
VuDroid	3	3.33	8	8.50	24	13.50	72	18.50
XBMC	11	5.73	361	13.02	10906	19.82	299564	26.34

Table 6.3: Measurements of sequences of lifecycle callbacks.

(ConnectBot, NotePad, XBMC), we additionally examined the “non-standard” transitions $\delta_1 \xrightarrow{\text{“non-standard”}} \delta_2$ for the two stacks δ_1 and δ_2 in the stack transition graph to determine how many of them would not have a matching $\delta_2 \xrightarrow{\texttt{finish}} \delta_1$ transition. Absence of such $\xrightarrow{\texttt{finish}}$ transitions reveals limitations in existing static analysis approaches and illustrates one possible client of our activity stack analysis. For all of these three apps, we do observe such absence. For example, for XBMC, the largest one among the three apps, 12.5% of all activity launch transitions are “non-standard” ones that do not have a matching $\xrightarrow{\texttt{finish}}$ transition.

6.5.2 Sequences of Lifecycle Callbacks

The measurements in Figure 6.3 characterize the sequences of lifecycle callbacks that can be derived from STG paths. For a specified value of k, we consider STG paths starting with the implicit launch-main-activity STG edge and containing exactly k edges. To avoid skewing the results, implicit STG edges due to `rotate`, `home` and `power` operations are not considered in this experiment. Two measurements are reported—the number of sequences (columns "#Seq") and the average length of sequences (columns "Length"). As expected, these two measurements both increase as k increases. Each sequence computed for a smaller k (e.g., when $k = 2$) is a prefix of some sequences computed for a larger k (e.g., when $k = 8$). The increase in average length is roughly linearly proportional to the increase of k, while the increase in number of sequences is much faster.

6.6 Summary

In this chapter, we introduce the first static analysis to model the Android activity stack, the changes in this stack, and the interactions between activities. We extend the formal semantics developed in Chapter 5 to include abstractions to represent the state and changes of the activity stack. Based on the semantics, we encode relevant Android constructs in an *activity transition graph* and perform traversal on this graph to compute the set of all possible activity stack states. The output of the analysis is encoded in a *stack transition graph*, whose nodes represent stacks and edges represent abstract operations to trigger the transition between two stacks. As an extension to the proposed technique, we develop a control-flow analysis of activity lifecycle callbacks. This work is an important step toward fully modeling the control/data flow of an Android application. It can be leveraged by other researchers to prune infeasible control-flow paths in static analysis for Android, or to discover more paths that would be missing without modeling of the activity stack.

CHAPTER 7: Related Work

7.1 Software Bloat Analysis

A number of tools have been proposed to quantify various symptoms of bloat (e.g., [34,39,53,89]), without providing insights into the reasons why this bloat occurs. Mitchell et al. [76] consider the transformations of logical data in order to explain run-time behavior and to assist a programmer in deciding whether execution inefficiencies exist. The approach in this work is not automated. Their follow-up work [75] focuses on deciding whether data structures have unnecessarily high memory consumption. Work by Dufour et al. analyzes the use and shape of temporary data structures [35, 36]. Their approach is based on a blended analysis, where a run-time call graph is collected and a static analysis is applied based on this graph. JOLT [97] is a VM-based tool that uses a new metric to quantify *object churn* and identify regions that make heavy use of temporary objects, in order to guide method inlining inside a just-in-time compiler.

In general, existing bloat detection work can be classified into two major categories: manual tuning methods (i.e., mostly based on measurements of bloat) [35,36, 75, 76], and fully automated optimization techniques such as the entire field of JIT technology [9] and the research from [97]. We provide analyses to support manual tuning, guiding programmers to code where bloat is likely to exist, and then allowing human experts to perform the code modification. By doing so, we hope to help the programmers quickly get through the hardest part of the tuning process—finding the likely bloated regions—and yet use their (human) insights to perform application-specific optimizations.

More recent work on bloat detection includes techniques that focus on different bloat patterns (such as excessive copy activities [116] and inefficient use of data structures [115]) to help programmers identify performance bottlenecks. The previous work closest to our reference propagation technique described in Chapter 2 is the profiling of copy activities from [116]. While both techniques track the flow of data, the proposed reference propagation analysis is more general and powerful in several aspects. First, the analysis records much more detailed information on how objects propagate, including information that connects the propagation with the corresponding source code statements. This level of detail makes it easier to explain and fix a performance problem. Second, the abstractions used to represent the propagation are more powerful, since they are specific to a producer of references, while the profiling in [116] merges the flow from multiple producers. Third, our work reports potential problems together with indicators of the likely difficulty of explaining and eliminating them. This approach is based on properties of the propagation that capture the complexity of interprocedural control-flow and of interactions with heap data structures.

7.2 Memory Leak Detection

There exists a large body of work on memory leak detection, including both static [19, 21, 49, 50, 59, 83, 104, 114] and dynamic approaches [14, 38, 48, 53, 57, 74, 90, 117, 118]. LeakChecker introduced in Chapter 3 is the first practical static memory leak detector for managed languages.

Static analyses for memory leak detection Static analysis techniques [19, 21,49,50,59,83,104,114] have been widely used to detect memory leaks for unmanaged languages such as C and C++. The explicit memory management in such languages allows the formulation of leak detection as a reachability problem—any control flow path that creates an object but does not free it may reveal a leak and is thus reported to the user for inspection. Work in [19] defines a reachability problem on the program's guarded value flow graph, and detects leaks by identifying value flows from the source (malloc) to the sink (free). Saturn [114] reduces the problem of leak detection to a boolean satisfiability problem, and uses a SAT-solver to identify potential bugs. Shape analysis based on 3-valued logic [32] has been proposed to assert the absence of leaks in list manipulation functions. Hackett and Rugina [45] identify leaks with a shape analysis that tracks individual heap cells.

Orlovich and Rugina [83] use backward dataflow analysis to disprove the feasibility of potential leak errors. The Clouseau [49] leak detector employs pointer ownership to describe the responsibilities for freeing heap memory, and formulates leak analysis as an ownership constraint system. Work in [50] proposes a type system to describe object ownership for polymorphic containers, and uses type inference to detect constraint violations. These prior efforts target C and C++ program whereas we are interested in garbage-collected languages such as Java and C#. A reachability formulation cannot be adopted to find leaks for managed languages, because object deallocation is done automatically by GC. In contrast, developer insight is exploited by LeakChecker (Chapter 3) to identify leaking objects at a high, semantic level. Work in [96] presents a static live region analysis for Java to detect array-related memory leaks. The problem of detecting liveness regions of arrays is formulated using a constraint graph that models linear inequalities over variables. The approach from [30] uses separation logic and shape analysis to find unused objects. However, these two analyses can be extremely expensive and they have not been applied on large-scale applications.

Dynamic analyses for memory leak detection Heap analysis tools such as [38, 53] take heap snapshots and visualize the object graph to help the user find unnecessary references. However, they do not provide the ability to automatically pinpoint the cause of a memory leak. Work done in the research community uses either growing types [57, 74] (i.e., types with growing number of run-time instances) or object staleness [14, 48, 118] (i.e., the elapsed time since the last use of an object) to identify suspicious data structures that may be related to a memory leak. Recent work from [117] leverages high-level program semantics to detect leaks related to transactional code structures. All these existing dynamic analyses require appropriate executable programs and test inputs, and can detect problems only when leaks are triggered in a particular test execution. It may be very difficult to meet these requirements, especially during development and in-house testing. In addition, dynamic approaches cannot work for partial programs such as components, plugins, and mobile apps. LeakChecker, the static approach proposed in Chapter 3 does not have these limitations.

7.3 Testing and Analysis of Android Software

As a fast-growing leading platform for mobile computing, Android has attracted significant attention in the program analysis/testing research community.

Model-based GUI testing. Finite state machines and similar models for GUI testing have been used by a number of researchers (e.g., [1, 2, 11, 44, 67, 69, 70, 105, 112, 113, 124]). Given a GUI model, test cases can be generated based on various coverage criteria (e.g., [69]). In these approaches the focus is typically on functional correctness and the coverage criteria reflect this. In contrast, we are interested in non-functional properties, and the coverage categories we define explore specialized paths in the model (with multiple repetitions of a neutral cycle) in order to target common leak patterns. An alternative to model-based testing is random testing. For example, Hu and Neamtiu [51] use the Monkey tool [108] to randomly insert GUI events into a running Android application, and then analyze the execution log to detect faults. Random testing is highly unlikely to trigger the repeated behavior needed to observe sustained growth in resource usage, the goal of the work presented in Chapter 4.

Reverse engineering of GUI models has been studied by others (e.g., [44, 68, 70]) and has been applied to Android applications (e.g., [1, 2, 4, 11, 107, 124, 130]). Several techniques have been proposed to improve the precision of models and the test cases generated from them (e.g., [8, 125, 126]). Almost all of these approaches consider only the GUI, and do not relate back to the program code. The GUI testing strategy presented in Chapter 4 considers coverage of activities and activity lifecycle callback methods. This exploration of Android-specific feature leads to efficient test generation and shorter test execution time.

Testing and static checking for Android. Prior work has considered the use of concolic execution to generate sequences of events for testing of Android applications [3, 55]. Zhang and Elbaum [128] focus on testing of exception-handling code when applications are accessing unreliable resources. As an alternative to testing, static checking can identify various defects including invalid thread accesses [129], energy-related defects [86], and security vulnerabilities [82]. The basis for these approaches is static analysis of Android applications, either to assist code instrumentation or to identify statically certain targeted behavioral patterns. Our work on foundational Android static analysis techniques (Chapters 5 and 6) can be leveraged to improve these existing approaches.

Static analysis for Android. Static analysis to understand GUI-driven behavior is essential for modeling the control/data flow of Android applications. Early work by Chaudhuri [18] and follow-up work on the SCanDroid security analysis tool [42] formalizes aspects of the semantics and performs control-flow analysis and security permissions analysis. This effort focuses on activities and other Android components (e.g., background services). These components communicate through intents—objects that describe the operation to be performed—and the analysis models these intents and the inter-component control flow based on them. The implementation is evaluated on a number of synthetic examples. This work does not model the GUI objects, events, and handlers that trigger the inter-component transitions, and uses conservative assumptions about the GUI-related control/data flow. Later work on related security problems [20, 43, 82] has similar limitations.

The A^{3}E tool for automated run-time exploration of Android applications [11] takes advantage of SCanDroid's static analysis to achieve high coverage. Such run-time coverage is essential for a variety of dynamic analyses for profiling, energy analysis, security analysis, and systematic testing (e.g., [2, 40, 47, 62, 85, 111, 127]). The analysis from SCanDroid is used to construct a static activity transition graph, with nodes representing activities and edges showing the possible transitions between them;

this graph is then used to drive the run-time exploration. It is unclear how this static analysis approach handles the general case when arbitrary GUI objects are associated with an activity, their handlers (located outside of the activity class) are registered via set-listener calls, and those handlers trigger transitions to new activities. Similar considerations apply to a hybrid static/dynamic analysis of UI-based trigger condition in Android applications [130], where security-sensitive behaviors are triggered dynamically based on a static model of activity transitions. The model construction in this work is incomplete and can benefit from the general solution provided in Chapter 5. Furthermore, these two approaches do not model the behavior of the activity stack and thus cannot fully express the semantics of an activity termination operation. The analysis introduced in Chapter 6 can be used to address this limitation.

A similar model, in which nodes represent UI screens and edges show transitions based on GUI events, is used as input to an automated test generation approach based on concolic execution [55]. Essential information encoded in the model is the set of tuples (activity a, GUI object v, event e, handler method h), where v is visible when a is active, and event e on v is handled by h. In this prior work the models are constructed manually; the output of the static analysis from Chapter 5 can be directly used to automate the generation of these tuples. As indicated by the case studies presented in Section 5.4.3, this information about GUI structure and behavior can be inferred very precisely by our analysis. The same benefits can apply to other model-based testing techniques for Android [105, 122, 123].

Yang et al. [124] present a reverse-engineering tool that combines static and dynamic analysis to construct a model of the application's GUI for testing purposes. The static analysis component identifies the objects that can serve as listeners, and determines the view ids of the GUI objects associated with these listeners. The analysis does not model the actual GUI objects (inflated or explicitly created), does not capture the general flow of these objects through the constructs described in Section 5.2, and does not account for the flow of view ids. Using our work, the generality of this tool could be increased. Similar observations apply to prior work on a static error checker for GUIs [129]. This tool is based on analysis of call paths that lead to operations on GUI objects. The analysis takes into account the objects created through inflation, but does not model precisely the flow of views due to the operations outlined in Section 5.2. Similar features and limitations can be seen in another static checker for Android [87]. Employing the analysis presented in Chapter 5 could lead to improved generality and precision for these checkers.

FlowDroid [10] is a precise flow- and context-sensitive taint analysis which performs interprocedural control-flow and data-flow analysis for Android. As part of this approach, the effects of callbacks are modeled by creating a wrapper main method. Our handling of relevant callbacks is conceptually similar, but without explicitly creating a wrapper. In FlowDroid, placeholder GUI objects that may flow into these callbacks are created in the wrapper method, while our approach propagates to the callbacks the actual GUI objects (Algorithm 5.6). In FlowDroid, XML layout files are examined to identify potential taint sources and connect them with the statements that access them. It does not appear that the tool models the constructs discussed in Section 5.2 and the corresponding GUI-related flow. CHEX [61] employs a different approach to model Android control flow. For an Android app, each callback method and all its transitive callees are defined as a code split, and all permutations of these code splits are used to derive the set of possible control-flow paths. AsDroid [52] analyzes event handlers of GUI objects to detect stealthy behaviors, but does not systematically model the GUI objects and the GUI hierarchy. These existing techniques could be complemented by the approach from Chapter 5, which would add general modeling and tracking of GUI objects and their event handlers.

Understanding GUI objects and their event handlers is essential for various other analyses of Android applications. For example, an existing static detector of energy-related software defects [86] requires control-flow analysis of the possible execution orders of event handlers. In this work, programmer input is needed to specify these orders. Instead, it may be possible to develop an automated approach based on analysis of activities, GUI objects associated with them, and handlers for these objects; the analyses from Chapters 5 and 6 provide the starting point for such an approach.

CHAPTER 8: Conclusions

The computing industry has experienced fast and sustained growth in the complexity of software functionality, structure, and behavior. Increased complexity has led to new challenges in program analyses to understand software behavior, and in particular to uncover performance inefficiencies. The same challenges are present in both traditional and mobile object-oriented software. Static and dynamic analyses need to keep up with this trend, and this often requires novel technical approaches.

This work is based on three key observations. First, following strictly the low-level definitions of performance inefficiencies makes it very difficult to develop practical and effective analyses. For example, static leak detection based on object liveness does not scale to large programs. As another example, complex programs may not have hot spots to analyze/optimize deeply, making traditional profiling techniques ineffective. Understanding high-level behavioral patterns of performance inefficiencies and bringing these insights into analysis design is a promising approach to overcome these limitations. Second, modeling only low-level semantics is no longer sufficient to build a precise analysis. For example, traditional reference and control-flow analysis techniques are not effective for Android applications, whose behavior is heavily dependent on the platform code. The Android platform has a mixture of features such as customized inter-component communication, heavy use of native code, complex GUI hierarchies, and event-driven control flow, none of which could be understood and precisely modeled by an analysis based on low-level semantics. In particular, information about GUI structure and behavior is lost without modeling based on the high-level semantics. Third, pursuing across-the-board high analysis precision is usually infeasible. Instead, selectively increasing analysis precision on certain program entities and carefully spending the analysis budget have been shown practical and effective in our work. In an analysis of Android GUI objects, modeling precisely only the GUI-related APIs and objects helps make a precise analysis practical. In an analysis of memory leaks, the leak candidate objects are modeled context-sensitively while context-insensitive modeling of other (irrelevant) objects helps reduce analysis cost, without sacrificing the overall precision and effectiveness. In short, a selective subset of high-level behavioral patterns and program semantics must be leveraged in order to perform practical program analyses for modern software.

Based on these key observations, we develop several dynamic and static program analysis techniques to understand, detect, and remove performance inefficiencies for both traditional and mobile object-oriented programs. Programs studied by these techniques are all written in Java, but we believe the proposed techniques are general enough to be applied to systems written in other object-oriented languages as well.

Bloat—excessive memory usage and work to accomplish simple tasks—is an important source of inefficiencies. We propose a novel reference propagation profiling tool to uncover performance problems in Java applications. The tool reports to developers a ranked list of suspicious allocation sites, annotated with information about the likely ease of performing transformations for them. Interesting performance inefficiency patterns are discovered by this analysis, and the running time reduction achieved by optimizing suspicious allocation sites can be significant.

Memory leaks commonly exist in both traditional and mobile object-oriented programs. Due to their presence, programs can slow down or even crash. We propose LeakChecker, the first practical static memory leak detector for Java. Leak detection performed by LeakChecker is based on an important observation that an event loop is often the place where severe leaks occur, and these leaks are often caused by objects outside the loop keeping unnecessary references to objects created inside the loop. The experimental results show that LeakChecker successfully finds leaks in all of the eight evaluated large programs and the false positive rate is reasonably low.

Resource leaks (e.g., memory leaks) are an important hurdle for quality software. We develop LeakDroid, a systematic and effective technique for testing of resource leaks in Android applications. In this work, test cases are generated to cover neutral cycles—sequences of GUI events that should not lead to increases in resource usage. Evaluation of this approach indicates that complicated and diverse resource leaks can be exposed by the generated test cases.

The availability of a GUI model is important for test generation to uncover resource leaks as well as general correctness problems in Android applications. Motivated by this need, we propose the first static analysis to model GUI-related Android objects, their propagation through the application, and their structural and behavioral properties. The analysis enables static modeling of control/data flow that is the basis for many compile-time analyses, error checking, test generation, and automated debugging. In another contribution toward static analysis of control/data flow, we develop the first static analysis to model the Android activity stack and the changes in this stack during program execution. This allows precise modeling of the interactions between activities, and serves as a starting point for other static and dynamic analyses for Android.

Several case studies have been presented for all of these analysis techniques. These studies demonstrate the effectiveness of the proposed insights, algorithms, and tools. Our experience with these techniques and tools provides promising evidence of practical approaches that can be used in real-world software development to understand and improve software behavior and performance.

BIBLIOGRAPHY

[1] D. Amalfitano, A. Fasolino, and P. Tramontana. A GUI crawling-based technique for Android mobile application testing. In *International Workshop on Testing Techniques and Experimentation Benchmarks for Event-Driven Software (TESTBED)*, pages 252–261, 2011.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *International Conference on Automated Software Engineering (ASE)*, pages 258–261, 2012.

[3] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59:1–59:11, 2012.

[4] Android GUITAR. `sourceforge.net/projects/guitar`.

[5] Apache Derby. `db.apache.org/derby`.

[6] Apache log4j. `logging.apache.org/log4j/1.2`.

[7] APV PDF viewer. `code.google.com/p/apv`.

[8] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight static analysis for GUI testing. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 301–310, 2012.

[9] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, 2005.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[11] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 641–660, 2013.

[12] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, pages 221–239, 2006.

[13] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP)*, pages 27–38, 2012.

[14] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[15] M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

[16] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 405–422, 2007.

[17] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Annual Computer Security Applications Conference (ACSAC)*, pages 463–475, 2007.

[18] A. Chaudhuri. Language-based security on Android. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–7, 2009.

[19] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491, 2007.

[20] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 239–252, 2011.

[21] Clang Static Analyzer. `clang.analyzer.llvm.org`.

[22] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *International Conference on Automated Software Engineering (ASE)*, pages 283–292, 2007.

[23] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[24] J. Clause and A. Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *International Conference on Software Engineering (ICSE)*, pages 515–524, 2010.

[25] comScore, Inc. The great American smartphone migration, 2012. `www.comscore.com/Press_Events/Press_Releases`.

[26] ConnectBot: Secure shell (SSH) client for the Android platform. `code.google.com/p/connectbot`.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, 2nd ed., 2001.

[28] *DaCapo Benchmarks*. `www.dacapo-bench.org`.

[29] W. DePauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.

[30] D. Distefano and I. Filipović. Memory leaks detection in Java by bi-abductive inference. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 278–292, 2010.

[31] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, 2000.

[32] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symposium (SAS)*, pages 115–134, 2000.

[33] P. Dubroy. Google I/O: Memory management for Android apps, 2011. `dubroy.com/blog/google-io-memory-management-for-android-apps`.

[34] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, 2003.

[35] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007.

[36] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[37] Eclipse project. `www.eclipse.org`.

[38] Eclipse Memory Analyzer Tool. `www.eclipse.org/mat`.

[39] ej-technologies GmbH. JProfiler. `www.ej-technologies.com`.

[40] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–6, 2010.

[41] FindBugs. `findbugs.sourceforge.net`.

[42] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

[43] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[44] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2012.

[45] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[46] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, 2005.

[47] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *International Conference on Software Engineering (ICSE)*, pages 92–101, 2013.

[48] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, 2004.

[49] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, 2003.

[50] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006.

[51] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *International Workshop on Automation of Software Test (AST)*, pages 77–83, 2011.

[52] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *International Conference on Software Engineering (ICSE)*, pages 1036–1046, 2014.

[53] Java Heap Analyzer Tool (HAT). `hat.dev.java.net`.

[54] Java Grande Forum Benchmark Suite. `www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`.

[55] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2013.

[56] Jikes Research Virtual Machine. `jikesrvm.org`.

[57] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[58] M. Jump and K. S. McKinley. Detecting memory leaks in managed languages with Cork. *Software: Practice and Experience*, 40(1):1–22, 2010.

[59] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *International Symposium on Memory Management (ISMM)*, pages 131–140, 2008.

[60] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, pages 153–169, 2003.

[61] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, pages 229–240, 2012.

[62] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 224–234, 2013.

[63] Managing the Activity lifecycle. `developer.android.com/training/basics/activity-lifecycle`.

[64] M. Marron, M. Mendez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 43–49, 2008.

[65] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology*, 19(2):1–33, 2009.

[66] Mckoi SQL database. `www.mckoi.com`.

[67] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[68] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Working Conference on Reverse Engineering (WCRE)*, pages 260–269, 2003.

[69] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 256–267, 2001.

[70] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, Oct. 2005.

[71] Memory analysis for Android applications. `goo.gl/VYHNKF`.

[72] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[73] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[74] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003.

[75] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[76] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[77] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338, 2007.

[78] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[79] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 65–74, 2007.

[80] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2005.

[81] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 6:1–6:11, 2012.

[82] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. le Traon. Effective inter-component communication mapping in Android with Epicc. In *USENIX Security*, 2013.

[83] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Static Analysis Symposium (SAS)*, pages 405–424, 2006.

[84] Overview of sensors in Android. `developer.android.com/guide/topics/sensors/sensors_overview.html`.

[85] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2012.

[86] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 267–280, 2012.

[87] E. Payet and F. Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[88] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[89] Quest Software. JProbe memory debugging. `www.quest.com/jprobe`.

[90] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *International Conference on Automated Software Engineering (ASE)*, pages 194–203, 2007.

[91] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[92] Robotium testing framework. `code.google.com/p/robotium`.

[93] A. Rountev, K. Van Valkenburgh, D. Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential Java programs. In *International Conference on Software Maintainance (ICSM)*, page 9, 2010.

[94] A. Rountev and D. Yan. Static reference analysis for GUI objects in Android software. In *International Symposium on Code Generation and Optimization (CGO)*, pages 143–153, 2014.

[95] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC)*, pages 126–137, 2003.

[96] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *International Conference on Compiler Construction (CC)*, pages 50–66, 2000.

[97] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[98] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[99] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[100] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[101] Standard Performance Evaluation Corporation. SPECjvm98 benchmark set. `www.spec.org/jvm98`.

[102] Stopping and restarting an activity. `developer.android.com/training/basics/activity-lifecycle/stopping.html`.

[103] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[104] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–264, 2012.

[105] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 377–386, 2011.

[106] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX Annual Technical Conference (USENIX)*, pages 307–320, 2008.

[107] P. Tramontana. Android GUI Ripper. `wpage.unina.it/ptramont/GUIRipperWiki.htm`.

[108] UI/Application exerciser monkey. `developer.android.com/tools/help/monkey.html`.

[109] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[110] VuDroid project. `code.google.com/p/vudroid`.

[111] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer profiling of Android applications. In *International Conference on Mobile Computing and Networking (MobiCom)*, pages 137–148, 2012.

[112] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 110–121, 2000.

[113] Q. Xie and A. M. Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodology*, 18(2):7:1–7:35, 2008.

[114] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 115–125, 2005.

[115] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[116] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[117] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011.

[118] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[119] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, pages 121–136, 2006.

[120] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering (ICSE)*, pages 134–144, 2012.

[121] D. Yan, G. Xu, S. Yang, and A. Rountev. LeakChecker: Practical static memory leak detection for managed languages. In *International Symposium on Code Generation and Optimization (CGO)*, pages 87–97, 2014.

[122] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in Android applications. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 411–420, 2013.

[123] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Workshop on Engineering Mobile-Enabled Systems (MOBS)*, pages 1–6, 2013.

[124] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 250–265, 2013.

[125] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.

[126] X. Yuan and A. M. Memon. Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.

[127] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 183–192, 2014.

[128] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *International Conference on Software Engineering (ICSE)*, pages 595–605, 2012.

[129] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 243–253, 2012.

[130] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104, 2012.