Thresher: Precise Refutations for Heap Reachability

Sam Blackshear
University of Colorado Boulder
samuel.blackshear@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder
evan.chang@colorado.edu

Manu Sridharan
IBM T.J. Watson Research Center
msridhar@us.ibm.com

Abstract

We present a precise, path-sensitive static analysis for reasoning about *heap reachability*; that is, whether an object can be reached from another variable or object via pointer dereferences. Precise reachability information is useful for a number of clients, including static detection of a class of Android memory leaks. For this client, we found that the heap reachability information computed by a state-of-the-art points-to analysis was too imprecise, leading to numerous false-positive leak reports. Our analysis combines a symbolic execution capable of path-sensitivity and strong updates with abstract heap information computed by an initial flow-insensitive points-to analysis. This novel *mixed* representation allows us to achieve both precision and scalability by leveraging the pre-computed points-to facts to guide execution and prune infeasible paths. We have evaluated our techniques in the THRESHER tool, which we used to find several developer-confirmed leaks in Android applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords heap reachability; path-sensitive analysis; symbolic execution

1. Introduction

Static reasoning about nearly any non-trivial property of modern programs requires effective analysis of heap properties. In particular, a heap analysis can be used to reason about *heap reachability*—whether one heap object is reachable from another via pointer dereferences at some program point. Precise heap reachability information improves heap-intensive static analyses, such as escape analysis, taint analysis, and cast checking. A heap reachability checker would also enable a developer to write statically checkable assertions about, for example, object lifetimes, encapsulation of fields, or immutability of objects.

Our interest in heap-reachability analysis arose while developing a tool for detecting an important class of memory leaks in Android applications. Briefly, such a leak occurs when an object of type Activity remains reachable from a static variable after the end of its life cycle and thus cannot be freed by the garbage collector (explained further in Sections 2 and 4). For this client, we found that highly precise reasoning about heap reachability, including flow-, context-, *and* path-sensitivity *with* materialization [43], was required to avoid emitting too many spurious warnings. We are unaware of an existing analysis that can provide such precision for heap-reachability queries while scaling to our target applications (40K SLOC with up to 1.1M SLOC of libraries). While approaches based on predicate abstraction or symbolic execution [2, 4, 11, 31] could provide the necessary precision in principle, to our best knowledge such approaches have not been shown to handle heap-reachability queries for real-world object-oriented programs (further discussion in Section 5).

We present an analysis for precise reasoning about heap reachability via on-demand refinement of a flow-insensitive points-to analysis. When the points-to analysis cannot refute a heap-reachability query Q, our technique performs a backwards search for a path program [7] witnessing Q; a failed search means Q is refuted. Our analysis is flow-, context-, and path-sensitive with location materialization, yielding the precision required by clients like the Android memory leak detector in an on-demand fashion.

In contrast to previous approaches to refinement-based or staged heap analysis [25, 26, 28, 36, 44], our approach refines points-to facts directly and is capable of path-sensitive reasoning. Also, unlike some prior approaches [28, 36, 44], our analysis does *not* attempt to refine the heap abstraction of the initial points-to analysis. Instead, we focus on the orthogonal problem of on-demand flow- and path-sensitive reasoning given a points-to analysis with a sufficiently precise heap abstraction.

Our analysis achieves increased scalability through two novel uses of the initial points-to analysis result. First, we introduce a *mixed symbolic-explicit* representation of heap reachability queries and transfer functions that simultaneously enables strong updates during symbolic execution, exploits the initial points-to analysis result, and mitigates the case split explosion seen with a fully explicit representation. Crucially, our representation allows the analysis to avoid case-splitting over entire points-to sets when handling heap writes, key to scaling beyond small programs. We also maintain *instance constraints* for memory locations during analysis, based on points-to facts, and use these constraints during query simplification to obtain contradictions earlier, improving performance.

Second, our analysis utilizes points-to facts to handle loops effectively, a well-known source of problems for symbolic analyses. Traditionally, such analyses either require loop invariant annotations or only analyze loops to some fixed bound. We give a sound algorithm for *inferring loop invariants over points-to constraints on path programs* during backwards symbolic execution. Our algorithm takes advantage of the heap abstraction computed by the up-front points-to analysis to effectively over-approximate the effects of the loop during the backwards refinement analysis (Section 3.3). Our technique maintains constraints from path conditions separately from the heap reachability constraints, enabling heuristic "dropping" of path constraints at loops to avoid divergence without significant precision loss in most cases.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Contributions. This paper makes the following contributions:

- We describe a technique for refining a points-to analysis with strong updates, path sensitivity, and context sensitivity to improve precision for heap-reachability queries. Our technique centers on enriching a backwards symbolic execution with the over-approximate information from the up-front points-to analysis to guide the execution and prune infeasible paths.
- We introduce a *mixed symbolic-explicit* representation of heap reachability queries and corresponding transfer functions that enable strong updates and precise disaliasing information during symbolic execution while reducing case splitting compared to a fully explicit approach. This representation is important both for finding contradictions early and for collapsing paths effectively.
- We present a method for *inferring loop invariants for heap-reachability queries* during symbolic execution based on over-approximating the heap effects of loops and dropping path constraints that may lead to divergence.
- We demonstrate the usefulness of our analysis by applying it to the problem of finding memory leaks in Android applications. Our tool refuted many of the false alarms that a flow-insensitive analysis reported for real-world Android applications and finished in less than twenty minutes on most programs we tried. We used our tool to successfully identify real (and developer-confirmed) memory leaks in several applications.

2. Overview

Here we present a detailed example to motivate our precise heap-reachability analysis and illustrate our technique. The example is based on real code from Android applications and libraries, and verifying the desired heap-reachability property requires strong updates, context sensitivity, *and* path sensitivity, which our technique provides in an on-demand fashion.

Our techniques were motivated by the need to detect leaks of Activity objects in Android applications. Every Android application has least one associated Activity object to control the user interface. Android development guidelines state that application code should not maintain long-lived pointers to Activity objects, as such pointers prevent the objects from being garbage collected at the end of their lifetimes, causing significant memory leaks (we discuss this issue further in Section 4). To detect such leaks in practice, it is sufficient to verify that Activity objects are *never* reachable from a static field via object pointers.

Figure 1 is a simple application that illustrates the difficulties of precisely checking this heap reachability property (ignore the boxed assertions for now). The `Main` class initializes and starts the application's Activity, the `Act` class. The `Vec` class captures the essence of a list data structure, as implemented in Android. This example is free of the leak described above, as the `Act` object allocated on line 1 is never made reachable from a static field.

For this example program, flow-insensitive points-to analysis techniques cannot prove the desired heap (un)reachability property due to the manner in which the `Vec` class is implemented. In the inset, we show a heap graph obtained by applying Andersen's analysis [1] with one level of object sensitivity [40] to the example. Graph nodes represent classes or abstract locations (whose names are shown at the corresponding allocation site in Figure 1), and edges represent possible values of field pointers. Object-sensitive abstract locations are named appropriately, for example, $\mathsf{vec}_0.\mathsf{arr}_1$ for arr_1 instances allocated when `Vec.push(−)` is invoked on instances of vec_0. Each edge indicates a may points-to relationship, written as $a_1 \cdot f \Rightarrow a_2$, meaning there may be an execution where field f of abstract location a_1 contains the address of location a_2. The graph imprecisely shows that

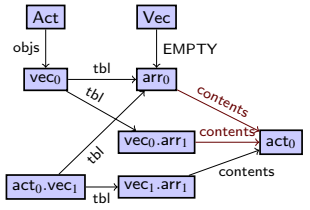

Figure 2.

```
   public class Main {
     public static void main(String[] args) {
1      Act a = new_act0 Act(); a.onCreate();
2    }
   }
   public class Act extends Activity {
     private static final Vec objs = new_vec0 Vec();
     public void onCreate() {
3      Vec acts = new_vec1 Vec();
4                  [ v̂^int_sz < v̂^int_cap, acts·sz ↦ v̂^int_sz, acts·cap ↦ v̂^int_cap,
                     this ↦ this, acts ↦ acts, acts·tbl ↦ arr0, this ↦ act0 ]
       acts.push(this
5                  [ v̂^int_sz < v̂^int_cap, this·sz ↦ v̂^int_sz, this·cap ↦ v̂^int_cap,
                     this ↦ vec1, vec1·tbl ↦ arr0, val ↦ act0 ]
       );
6      ...
7      objs.push("hello"
8                  [ v̂^int_sz < v̂^int_cap, this·sz ↦ v̂^int_sz, this·cap ↦ v̂^int_cap,
                     this ↦ vec0, vec0·tbl ↦ arr0, val ↦ act0 ] †
       );
9    }
   }
   public class Vec {
     private static final Object[] EMPTY = new_arr0 Object[1];
     private int sz; private int cap; private Object[] tbl;

     public Vec() {
10     this.sz = 0; this.cap = -1; this.tbl = EMPTY;
11                 [ v̂^int_sz < v̂^int_cap, this·sz ↦ v̂^int_sz, this·cap ↦ v̂^int_cap,
                     this ↦ this, this·tbl ↦ arr0 ] †
     }

     public void push(Object val) {
12     Object[] oldtbl = this.tbl;
13                 [ v̂^int_sz < v̂^int_cap, this·sz ↦ v̂^int_sz, this·cap ↦ v̂^int_cap,
                     this ↦ this, this·tbl ↦ arr0, val ↦ act0 ]
       if (this.sz >= this.cap) {
14       this.cap = this.tbl.length * 2;
15       this.tbl = new_arr1 Object[this.cap];
16                 [ this ↦ this, this·tbl ↦ arr0, val ↦ act0 ] †
         for (int i = 0; i < this.sz; i++) {
17         this.tbl[i] = oldtbl[i]; // copy from oldtbl
18       }
19     }
20                 [ this ↦ this, this·tbl ↦ arr0, val ↦ act0 ]
       this.tbl[this.sz] = val;
21                 [ arr0·contents ↦ act0 ]
       this.sz = this.sz + 1;
22   }
   }
```

Figure 1. Refuting a false alarm with context-sensitive, path-sensitive witness search. We show witness queries in the boxes. The † indicate refuted branches of the witness search.

Activity object act_0 is reachable from both static fields Act·objs and Vec·EMPTY, hence falsely indicating that a leak is possible.

The root cause of the imprecision in Figure 2 is the edge $\mathsf{arr}_0 \cdot \mathsf{contents} \Mapsto \mathsf{act}_0$, which indicates that the array assigned to Vec·EMPTY may contain the Activity object. `Vec` is implemented using the *null object pattern* [45]: rather than allocating a separate internal array for each empty `Vec`, all `Vec` objects initially use Vec·EMPTY as their internal array. The code in `Vec` is carefully written to avoid ever adding objects to Vec·EMPTY while also avoiding additional branches to check for the empty case. But, a flow-insensitive points-to analysis is incapable of reasoning precisely about this code, and hence it models the statement **this**`.tbl[`**this**`.sz] = val` on line 20 as possibly writing to Vec·EMPTY, polluting the points-to graph. Real Android collections are implemented in this way.[1] Note that a more precise heap abstraction would not help in this case—because the (concrete) null object is shared among *all* instances the `Vec` class, *no refinement on the heap abstraction* alone would be sufficient to rule out this false alarm.

More precise analysis of this example requires reasoning about the relationship between the `sz` and `cap` fields of each `Vec`. This relationship is established in the `Vec`'s constructor and must be preserved until its `push` method is called. Though there is a large body of work focused on the important problem of refining heap abstractions (e.g., [36, 44]), this example shows that doing so alone is sometimes not sufficient for precise results. An analysis that lacks path sensitivity and strong updates will be unable to prove that `Vec`'s never write into the shared array and must therefore conflate the contents of all `Vec` objects. The *witness-refutation* technique that we detail in this paper enables after-the-fact, on-demand refinement to address this class of *control-precision* issues.

Refinement by Witness Refutation. We refine the results of the flow-insensitive points-to analysis by attempting to refute all executions that could possibly witness an edge involved in a leak alarm. The term "witness" is highly overloaded in the program analysis literature, so we begin by carefully defining its use in our context. We first define a *path* as a sequence of program transitions. A *path witness* for a *query* Q is a path that ends in a state that satisfies Q. Such a path witness may be concrete/under-approximate/must in that it describes a sequence of program transitions according to a concrete semantics that results in a state where Q holds (e.g., a test case execution). Analogously, an abstract/over-approximate/may path witness is such a sequence over an abstract semantics (e.g., a trace in an abstract interpreter). Building on the definition of a path program [7], we define a *path program witness* for a query Q as a path program that ends in a state satisfying Q. A *path program* is a program projected onto the transitions of a given execution trace (essentially, paths augmented with loops). Note that a path program witness may be under- or over-approximate in its handling of loops. In this paper, we use the term "witness" to refer to over-approximate path program witnesses unless otherwise stated, as our focus is on sound refutation of queries.

Our analysis performs a goal-directed, backwards search for a path program witness ending in a state that satisfies a query Q. We *witness* a query by giving a witness that produces it, or we *refute* a query by proving that no such witness can exist. Our technique proceeds in three phases.

Obtain a Conservative Analysis Result. First, we perform a standard points-to analysis to compute an over-approximation of the set of reachable heaps, such as the points-to graph in Figure 2.

[1] In fact, we discovered buggy logic in the actual Android libraries that allowed writing to a null object, thereby polluting all empty containers! The bug was acknowledged and fixed by Google (`https://code.google.com/p/android/issues/detail?id=48055`).

Formulate Queries. Second, we formulate queries to refute alarms generated using the points-to analysis result. For the Activity leak detection client, an alarm is a points-to path between a **static** field and an Activity object. For example, the following points-to path from the graph in Figure 2 is a (false) leak alarm:

$$\mathsf{Act{\cdot}objs} \Mapsto \mathsf{vec}_0, \mathsf{vec}_0 \cdot \mathsf{tbl} \Mapsto \mathsf{arr}_0, \mathsf{arr}_0 \cdot \mathsf{contents} \Mapsto \mathsf{act}_0$$

To refute an alarm, we attempt to refute each individual edge in the corresponding points-to path. If we witness all edges in the path, we report a leak alarm. If we refute some edge e in the path, we delete e from the points-to graph and attempt to find another path between the source node and the sink node. If we find such a path, we restart the process with the new path. If we refute enough edges to disconnect the source and sink in the points-to graph, we have shown that the alarm raised by the flow-insensitive points-to analysis is false.

For our client, we wish to show the flow-insensitive property that a particular points-to constraint cannot hold at any program point. Thus, for each points-to edge e to witness, we consider a query for e at every program statement that could produce e. This information can be obtained by simple post-processing or instrumentation of the up-front points-to analysis [8].

Search for Witnesses. Finally, given a query Q at a particular program point, we search for path program witnesses on demand. In Figure 1, we illustrate a witness search that produces a refutation for the points-to constraint $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$ holding at program point 21. That is, we prove that the points-to constraint is unrealizable at that program point. By starting a witness search from each statement that potentially produces the edge, we will see that $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$ is in fact unrealizable at any program point.

Notationally, we use a single arrow $\mapsto$ to denote an *exact points-to constraint*, whose source and sink are symbolic values typically denoting addresses of memory cells, and a double arrow $\Mapsto$ to denote a may points-to edge between abstract locations (cf., Section 3.1). For example, the exact points-to constraint $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$ describes a single memory cell whose address is some instance in the concretization of arr_0 and contents is some instance in the concretization of act_0. This distinction is critical for enabling strong updates in a backwards analysis.

2.1 Mixed Symbolic-Explicit Queries

We illustrate the witness search by showing the sub-queries (boxed) that arise as the search progresses. Moving backwards from our starting point at line 21, the sub-query at program point 20 says that we need the following heap state at that point:

$$\mathsf{this} \mapsto \widehat{this}, \widehat{this} \cdot \mathsf{tbl} \mapsto \mathsf{arr}_0, \mathsf{val} \mapsto \mathsf{act}_0 \qquad (\dagger)$$

where $\widehat{this}$ is a *symbolic variable* that represents the receiver of the method. A symbolic variable (written as a hatted letter $\widehat{v}$) is an existential standing for an arbitrary instance drawn from some definite set of abstract locations. Here, $\widehat{this}$ represents some instance drawn from the points-to set of local variable this, which is $\{\mathsf{vec}_0, \mathsf{vec}_1\}$. We represent this fact with an *instance constraint*:

$$\widehat{this}\ \mathsf{from}\ \{\mathsf{vec}_0, \mathsf{vec}_1\} \qquad (\ddagger)$$

that we track as part of the query at program point 20. For the moment, we elide such 'from' constraints and discuss them further in Section 2.2 and Section 3. This sub-query conjoining the heap state from (†) and the instance constraint from (‡) is an example of a *mixed* symbolic-explicit state because we introduce a fresh symbolic variable for the contents of **this**, but also have named abstract locations arr_0 and act_0. We say that a query is fully *explicit* if all of its points-to constraints are between named abstract locations from the points-to abstraction. A named abstract location can be seen as a symbolic variable that is constrained to be from a singleton

abstract location set. This connection to the points-to abstraction in an explicit query enables our witness search to prune paths that are inconsistent with the up-front points-to analysis result, as we demonstrate in Section 2.2.

Backwards Path-By-Path Analysis. Returning to the example, the path splits into two prior to program point 20, one path entering the **if** control-flow branch at point 19, the other bypassing the branch to point 13. We consider both possibilities and indicate the fork in Figure 2 by indenting from the right margin. For the path into the branch, the loop between program points 16 and 19 has no effect on the query in question from point 20, so it simply continues backward to program point 16. Observe that because we are only interested in answering a specific query, this irrelevant loop poses no difficulty. At program point 16, we encounter a refutation for this path: the preceding assignment statement writes an instance of arr_1 to the *this*·tbl field, which contradicts the requirement that *this*·tbl hold an instance of arr_0 (underlined). Thus, we have discovered that no concrete program execution can assign a newly allocated array to **this**.tbl at line 15, that is, an instance of arr_1 and then place an Activity object in the EMPTY array at line 20 because **this**.tbl will point to that newly allocated array by then.

Resuming the path that bypasses the **if** branch, the analysis at program point 13 determines that entering the **if** branch changed the query and thus adds a *control-path constraint* to the abstract state indicating that the value of the **this**.sz field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}}$) must be less than the value of the **this**.cap field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$). As we will see, tracking the path constraint above is critical to obtaining a refutation for the example. From here, this path reaches the method boundary, leading the analysis to process the possible call sites at program points 8 and 5. The path at program point 8 can be refuted at this point, as the query requires that the val parameter be bound to an instance of act_0 (underlined), but the actual argument is the string `"hello"`. Thus, we have identified that this call to `push` cannot be the reason that an Activity object is placed into the EMPTY array because it is pushing a string, not an Activity.

The other path from the `acts.push` call site (i.e., program point 5) can continue. The query at program point 4 before the call simply changes the program variables of the callee to those of the caller. Continuing this path, we enter the constructor of `Vec` at program point 11. Here, we discover that the values of the `sz` and `cap` fields as initialized in the constructor contradict the control constraint $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}} < \widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$. Intuitively, the witness search has observed the invariant that a `Vec`'s `tbl` field cannot point to the `EMPTY` array after a call to its `push` method. We have refuted the last path standing, and so we have shown that the statement at line 21 cannot produced the edge $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$.

In the above, we have been rather informal in describing why certain points-to facts can be propagated back unaffected and why producing certain facts can be done with a strong update-style transfer function. Furthermore, in the example program from Figure 1, there is one (and only one) more statement that could produce the constraint $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$, which is the assignment at line 17 inside the copy loop. A witness search for this query starting at line 17 leads to a refutation similar to the one described above, but to discover it we must first infer a non-trivial loop invariant. Because we are interested in an over-approximate path program witness-refutation search, we have to obtain loop invariants. We consider these issues further in Section 3.

2.2 Taming Path Explosion From Aliasing

In the previous subsection, we focused on how refutations can occur. For example, backwards paths were pruned at line 15 and at program point 8 because we reach an allocation site that conflicts with our instance constraints in the query (e.g., we need an arr_0 not an arr_1 at line 15). In this section, we present a simple example to explain how our mixed symbolic-explicit representation enables our analysis to derive such contradictions earlier and thus mitigates the *aliasing path explosion problem*.

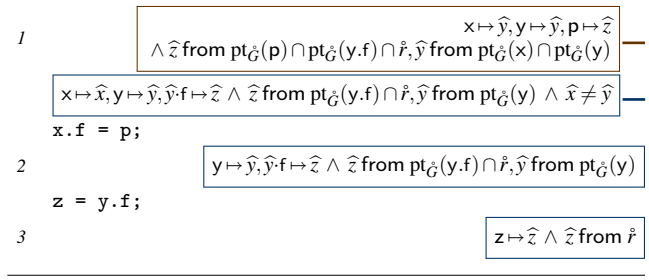

Figure 3. An example illustrating how 'from' constraints help tame path explosion from aliasing. Note that the set of abstract locations to which symbolic variable $\widehat{z}$ might belong is restricted each time we observe $\widehat{z}$ flow through a variable or field.

An over-approximate backwards symbolic executor that lacks information about aliasing will be forced to fork a number of cases to account for aliasing at every field write, quickly causing a case explosion that is worst-case exponential in the number of field writes. This case explosion is independent of (but compounded by) the well-known scalability problems caused by conditional branching in a path-sensitive analysis.

To address this aliasing path explosion problem, the key observation that we make is that contradictions from instance constraints can be derived *before* the allocation site by exploiting information from the up-front points-to analysis. In particular, the set of possible allocation sites for any instance can be restricted as we reason about how they flow into and out of program variables and heap locations. This observation motivates our mixed symbolic-explicit representation, which we demonstrate with a simple example shown in Figure 3. Our initial query at point 3 asks if program variable `z` can point to an instance $\widehat{z}$ from some set of abstract locations $\mathring{r}$—we call this a *points-to region*. Moving backwards across the statement `z = y.f`, we derive a pre-query at point 2 that says our original query can be witnessed if `y` points to some instance $\widehat{y}$ and that instance points to $\widehat{z}$ through its `f` field (i.e., $\mathsf{y} \mapsto \widehat{y}, \widehat{y}{\cdot}\mathsf{f} \mapsto \widehat{z}$). Additionally, we now know that the instance $\widehat{z}$ must be drawn from the *intersection* of $\mathring{r}$ and the abstract locations in the points-to set of y.f, which we write as $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$. If we can use the points-to graph $\mathring{G}$ to determine that no such abstract location exists (e.g., if we had $\mathring{r} = \{a_0, a_2\}$ and $\mathrm{pt}_{\mathring{G}}(\widehat{y}.\mathsf{f}) = \{a_1\}$), then we have refuted this query and can prune this path immediately.

Assuming $\mathring{r}$ and $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$ are not disjoint (i.e., $\mathring{r} \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \neq \emptyset$), we proceed with our backwards analysis to the field write `x.f = p`. We must consider two cases at program point 1: one where `x` and `y` are aliased (the top query) and one where they are not (the bottom query). In the aliased case, we can further constrain the instance $\widehat{u}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y})$. Some previous tools have used an up-front, over-approximate points-to analysis as an aliasing oracle to rule out aliased cases like this one (e.g., PSE [39])—if `x` and `y` cannot possibly point to the same abstract location (i.e., $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y}) = \emptyset$), this aliased case is ruled out. Our approach generalizes this kind of aliasing check by explicitly introducing 'from' constraints that are incrementally restricted. For example, we also constrain $\widehat{z}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{p}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}$, where the additional restriction is $\mathrm{pt}_{\mathring{G}}(\mathsf{p})$. This constraint says that the field write `x.f = p` produced the query in question only if the instance $\widehat{z}$ is drawn from some abstract location shared by these three sets.

Finally, we consider the case where x and y are not aliased (i.e., $\widehat{x} \neq \widehat{y}$). Here, the only change to the query is the addition of the constraints $\mathsf{x} \mapsto \widehat{x}$ and $\widehat{x} \neq \widehat{y}$. This disequality further constrains the query so that if we later discover that x and y are in fact aliased, we can refute this query. Accumulation of this kind of disaliasing constraint is common (e.g., [11]), but expensive (cf., Section 3.3).

We remark that the instance 'from' constraints can be viewed as a generalization of a fully explicit representation. To represent 'from' constraints explicitly, instead of a symbolic points-to constraint $\mathsf{x} \mapsto \widehat{x}$, we disjunctively consider all cases where we replace the symbolic variable $\widehat{x}$ with an abstract location from $\mathrm{pt}_{\mathring{G}}(\mathsf{x})$ (the points-to set of x) For example, suppose $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) = \{a_1, a_2\}$; then we case split and consider two heap states: (1) $\mathsf{x} \mapsto a_1$ and (2) $\mathsf{x} \mapsto a_2$. This representation corresponds roughly to a backwards extension of *lazy initialization* [33] over abstract locations instead of types. Note that while PSE-style path pruning only applies to ruling out the aliased case in field writes, the explicit representation of 'from' constraints permits the same kind of flow-based restriction shown in Figure 3. However, the cost is case splitting a separate query for each possible abstract location from which each symbolic variable is drawn (e.g., $|\mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}| \cdot |\mathrm{pt}_{\mathring{G}}(\mathsf{y})|$ queries at program point 2).

3. Refuting Path Program Witnesses

We formalize our witness-refutation analysis and argue that our technique is *refutation sound*—that we only declare an edge refuted when no concrete path producing that query can exist. The language providing the basis for our formalization is defined as follows: This language is a standard imperative programming language with object fields and dynamic memory allocation.

statements	$s ::= c \mid \mathtt{skip} \mid s_1 \,;\, s_2 \mid s_1 \,[]\, s_2 \mid \mathtt{loop}\, s$
commands	$c ::= x := y \mid x := y.f \mid x.f := y \mid x := \mathtt{new}_a\, \tau() \mid \mathtt{assume}\, e$
expressions	$e ::= x \mid \cdots$
types	$\tau ::= \{f_1, \ldots, f_n\} \mid \cdots$
program variables x, y	object fields f abstract locations a

Atomic commands c include assignment, field read, field write, object allocation, and a guard. For the purposes of our discussion, it is sufficient if an object type is just a list of field names. We leave unspecified a sub-language of pure expressions, except that it allows reading of program variables. The label a on allocation names the allocation site so that we can tie it to the points-to analysis. Compound statements include a do-nothing statement, sequencing, non-deterministic branching, and looping. Standard `if` and `while` statements can be defined in the usual way (i.e., $\mathtt{if}\,(e)\, s_1\, \mathtt{else}\, s_2 \stackrel{\text{def}}{=} (\mathtt{assume}\, e \,;\, s_1) \,[]\, (\mathtt{assume}\, !e \,;\, s_2)$ and $\mathtt{while}\,(e)\, s \stackrel{\text{def}}{=} \mathtt{loop}\,(\mathtt{assume}\, e \,;\, s) \,;\, \mathtt{assume}\, !e$).

For ease of presentation, our formal language is intraprocedural. However, our implementation is fully interprocedural. We handle procedure calls by modeling parameter binding using assignment and keeping an explicit abstraction of the call stack. The call stack is initially empty representing an arbitrary calling context but grows as we encounter call instructions during symbolic execution. If we reach the entry block of a function with an empty call stack, we propagate symbolic state backwards to all possible callers of the current function. We determine the set of possible callers using the call graph constructed alongside the points-to analysis.

3.1 Formulating a Witness Query over Heap Locations

Given a program, we first do a standard points-to analysis to obtain a points-to graph $\mathring{G}$: $\langle \mathring{V}, \mathring{E} \rangle$ consisting of a set of vertices $\mathring{V}$ and a set of edges $\mathring{E}$ (e.g., Figure 2). A vertex represents a set of memory addresses, which include program variables $x \in \mathbf{Var}$ and abstract locations $a \in \mathbf{AbsLoc}$ (i.e., $\mathring{V} \supseteq \mathbf{Var} \cup \mathbf{AbsLoc}$). An abstract location a abstracts non–program-variable locations (e.g., from dynamic memory allocation). We do not fix the heap abstraction, such as the level of context sensitivity, but we do assume that we are given the abstract location to which any `new` allocation belongs (via the subscript annotation). A points-to edge from $\mathring{E}$ is either of the form $x \Mapsto a$ or $a_0 \cdot f \Mapsto a_1$. The form $x \Mapsto a$ means a concrete memory address represented by the program variable x may contain a value represented by abstract location a. We write $a_0 \cdot f \Mapsto a_1$ to denote that f is the label for the edge between nodes a_0 and a_1. This edge form means that $a_0 \cdot f$ is a field of an object in the concretization of a_0 that may contain a value represented by abstract location a_1. A **static** field in Java can be modeled as a global program variable.

Our analysis permits formulating a query Q over a finite number of heap locations along with constraints over data fields:

queries	$Q ::= M \wedge P \mid \mathsf{false}$
memories	$M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v}.f \mapsto \widehat{u} \mid M_1 * M_2$
pure formulæ	$P ::= \mathsf{true} \mid P_1 \wedge P_2 \mid \widehat{v}\ \mathsf{from}\ \mathring{r} \mid \cdots$
points-to regions	$\mathring{r}, \mathring{s} ::= a \mid \mathsf{data} \mid \mathring{r}_1 \Cup \mathring{r}_2$
instances	$\widehat{v}, \widehat{u}$
refutation states	$R ::= Q \mid R_1 \vee R_2 \mid \exists \widehat{v}. R$

We give a heap location via an *exact points-to edge constraint* having one of two forms: $x \mapsto \widehat{v}$ or $\widehat{v}.f \mapsto \widehat{u}$. Recall that in contrast to points-to edges that summarize a *set* of concrete memory cells, an exact points-to constraint expresses a single memory cell. The first form $x \mapsto \widehat{v}$ means a memory address represented by the program variable x contains a value represented by a symbolic variable $\widehat{v}$ (and similarly for the second form for a field). Since we are mostly concerned with memory addresses for concrete object instances, we often refer to symbolic variables as *instances*. The memory any stands for an arbitrary memory.

We introduce one non-standard pure constraint form: the *instance constraint* $\widehat{v}\ \mathsf{from}\ \mathring{r}$ says the symbolic variable $\widehat{v}$ is an instance of a points-to region $\mathring{r}$ (i.e., is in the set of values described by region $\mathring{r}$). A *points-to region* is a set of abstract locations a or the special region data. For uniformity, the region data is used to represent the set of values that are not memory addresses, such as integer values. As we have seen in Section 2.2, instance constraints enable us to use information from the up-front points-to analysis in our witness-refutation analysis. As an example, the informal query $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$ from Section 2 is expressed as follows:

$$\widehat{v}_1 \cdot \mathsf{contents}_{[\widehat{v}_3]} \mapsto \widehat{v}_2 \wedge \widehat{v}_1\ \mathsf{from}\ \{\mathsf{arr}_0\} \wedge \widehat{v}_2\ \mathsf{from}\ \{\mathsf{act}_0\} \wedge \widehat{v}_3\ \mathsf{from}\ \mathsf{data}$$

where $\widehat{v}_3$ stands for the index of the array. This query considers existentially an instance of each abstract location arr_0 and act_0.

When writing down queries, we assume the usual commutativity, associativity, and unit laws from separation logic [41]. Since we are interested in witnessing or refuting a subset of edges corresponding to part of the memory, we interpret any memory M as $M * \mathsf{any}$ (or intuitionistically instead of classically [32, 41]).

3.2 Witness-Refutation Search with Instance Constraints

As described in Section 2, we perform a path-program–by–path-program, backwards symbolic analysis to find a witness for a given query Q. A refutation state R is simply a disjunction of queries, which we often view as a set of candidate witnesses. We include an existential binding of instances $\exists \widehat{v}. R$ to make explicit when we introduce fresh instances, but we implicitly view instances as renamed so that they are all bound at the top-level (i.e., all formulæ are in prenex normal form). Informally, a path program witness is a query Q_{wit} bundled with a sub-program examined so far s_{wit} and a sub-program left to be explored s_{pre}.

Definition 1 (Path Program Witness)**.** A *path program witness* for an input statement-query pair $\langle s, Q \rangle$ is a triple $\langle s_{\text{pre}}, Q_{\text{wit}}, s_{\text{wit}} \rangle$ where (1) $s \equiv s_{\text{pre}} \,;\, s_{\text{post}}$, and (2) s_{wit} is a sub-statement of s_{post} such that (a) if an execution of s_{post} leads to a store σ_{post} satisfying the input query Q, then it must be from a store σ_{wit} satisfying Q_{wit} and (b) executing s_{wit} from σ_{wit} also leads to σ_{post}.

$$\boxed{\vdash \{R\}\, s\, \{Q\} \qquad \vdash \{R'\}\, s\, \{R\}}$$

$$\textsc{WitRefuted}\ \frac{}{\vdash \{\mathsf{false}\}\, s\, \{\mathsf{false}\}} \qquad \textsc{WitCases}\ \frac{\vdash \{R_1'\}\, s\, \{R_1\} \qquad \vdash \{R_2'\}\, s\, \{R_2\}}{\vdash \{R_1' \vee R_2'\}\, s\, \{R_1 \vee R_2\}}$$

$$\textsc{WitSkip}\ \frac{}{\vdash \{\mathsf{any} \wedge \mathsf{true}\}\, s\, \{\mathsf{any} \wedge \mathsf{true}\}} \qquad \textsc{WitFrame}\ \frac{\vdash \{\bigvee_i M_i' \wedge P_i'\}\, s\, \{M \wedge P\} \quad s \text{ must not modify } M_{\mathrm{fr}}}{\vdash \{\bigvee_i (M_{\mathrm{fr}} * M_i') \wedge P_i'\}\, s\, \{(M_{\mathrm{fr}} * M) \wedge P\}}$$

$$\boxed{\vdash \{R\}\, c\, \{Q\}}$$

$$\textsc{WitNew}\ \frac{}{\vdash \{\mathsf{any} \wedge \ \widehat{v}\,\mathsf{from}\, a \cap \mathring{r}\ \wedge P\}\ x := \mathtt{new}_a\, \tau()\ \{x \mapsto \widehat{v} \wedge \widehat{v}\,\mathsf{from}\, \mathring{r} \wedge P\}}$$

$$\textsc{WitAssign}\ \frac{}{\vdash \{y \mapsto \widehat{v} \wedge \ \widehat{v}\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{r}\ \wedge P\}\ x := y\ \{x \mapsto \widehat{v} \wedge \widehat{v}\,\mathsf{from}\, \mathring{r} \wedge P\}}$$

$$\textsc{WitRead}\ \frac{P' = \widehat{u}\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(y) \wedge \ \widehat{v}\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(y.f) \cap \mathring{r}\ \wedge P}{\vdash \{\exists \widehat{u}.\, y \mapsto \widehat{u} * \widehat{u}.f \mapsto \widehat{v} \wedge P'\}\ x := y.f\ \{x \mapsto \widehat{v} \wedge \widehat{v}\,\mathsf{from}\, \mathring{r} \wedge P\}}$$

$$\textsc{WitWrite}\ \frac{\begin{array}{c} M_i = x \mapsto \widehat{v}_i * y \mapsto \widehat{u}_i * (\mathop{\Large *}_{j \neq i} \widehat{v}_j.f \mapsto \widehat{u}_j \wedge \widehat{v}_j\ \mathsf{from}\, \mathring{r}_j \wedge \widehat{u}_j\ \mathsf{from}\, \mathring{s}_j) \\ Q_i = M_i \wedge\ \widehat{v}_i\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r}_i\ \wedge\ \widehat{u}_i\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{s}_i\ \wedge (\bigwedge_{j \neq i} \widehat{v}_j \neq \widehat{v}_i) \wedge P \\ Q = \exists \widehat{x}.\, x \mapsto \widehat{x} * (\mathop{\Large *}_i \widehat{v}_i.f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\, \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\, \mathring{s}_i \wedge \widehat{v}_i \neq \widehat{x}) \wedge \widehat{x}\,\mathsf{from}\, \mathrm{pt}_{\mathring{G}}(x) \wedge P \end{array}}{\vdash \{Q \vee \bigvee_i Q_i\}\ x.f := y\ \{(\mathop{\Large *}_i \widehat{v}_i.f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\, \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\, \mathring{s}_i) \wedge P\}}$$

Figure 4. Witness-refutation search is a path-program–by–path-program backwards analysis. Boxed terms emphasize opportunities for refuting paths using instance constraints.

Any intermediate state in our backwards analysis is such a path program witness. Intuitively, the statement s_{wit} captures the path sub-program identified by the backwards analysis that is relevant to producing the input query Q (so far). A refutation occurs when Q_{wit} is false, that is, we have discovered that it is not possible to end up in a state satisfying Q. A "full" witness is when Q_{wit} is any; that is, we can no longer find a refutation. A "partial" witness is a witness where Q_{wit} is a query other than any or false.

We formalize a backwards path program enumeration transforming queries into sub-queries to eventually produce an any or a false refutation state. To describe the analysis, we define the judgment form $\vdash \{R'\}\ s\ \{R\}$ in Figure 4. This judgment form is a standard Hoare triple, but because our analysis is backwards, we read this judgment form from right-to-left. It says, "Given a post-formula R, we find a pre-formula R' such that executing statement s from a state satisfying R' yields a state satisfying R (up to termination)." Conceptually, the post-formula R is a (disjunctive) set of queries, and the pre-formula R' is the set of sub-queries. The path program s_{wit} can be obtained by a simple instrumentation of the rules similar to our prior work [8].

Deriving Refutations. At any point, we can extend the witness-refutation search for some disjunct. Here, we express this step with WitCases, which says a disjunctive refutation state $R_1 \vee R_2$ can be derived by finding a witness for R_1 and R_2. We make this system algorithmic for an implementation by representing cases as a disjunctive set of pending queries $\cdots \vee Q_i \vee \cdots$ that we extend individually. Rule WitRefuted simply says if we have derived false in the post-state for a statement, then we have false in the pre-state as well. To scale beyond the tiniest of programs, we need to be able to refute queries quickly so that the number of queries to consider, that is, the number of symbolic execution paths to explore, remains small. We have three tools for refuting queries: (1) separation (i.e., a query where a single memory cell would need to point to two locations simultaneously), (2) instance constraints (i.e., a query where an instance cannot be in any points-to region), and (3) other pure constraints (i.e., a query with pure constraints that are unsatisfiable, such as, from detecting an infeasible control-flow path). In our inference rules, we assume a refutation state R is always normalized to false if the formula is unsatisfiable. Since we are interested in sound refutations and over-approximate witnesses, for scaling, we can also weaken queries at the cost of potentially losing precision. We revisit this notion in Section 3.3.

Instance constraints are pure constraints that tie the exact points-to constraints in the query to the accumulated information from the up-front points-to analysis and the flow of the symbolic variables as discussed in Section 2.2. They can axiomatized as follows:

$$\widehat{v}\,\mathsf{from}\, \emptyset \iff \mathsf{false} \tag{1}$$

$$\widehat{v}\,\mathsf{from}\, \mathring{r}_1 \wedge \widehat{v}\,\mathsf{from}\, \mathring{r}_2 \iff \widehat{v}\,\mathsf{from}\, \mathring{r}_1 \cap \mathring{r}_2 \tag{2}$$

$$\mathsf{true} \iff \widehat{v}\,\mathsf{from}\, \mathbf{AbsLoc} \cup \{\mathsf{data}\} \tag{3}$$

In particular, we derive a contradiction when we discover an instance that can be drawn from any abstract location (axiom 1). Though our formalism groups instance constraints and other pure constraints together, our implementation keeps them separate for simplicity in checking. Instance constraints are checked using basic set operations and other pure constraints are checked with an off-the-shelf SMT solver, though it should be possible to encode instance constraints into the solver using this axiomatization.

We include a frame rule on M, WitFrame, to simplify our presentation, which together with WitSkip allow us to isolate the parts of the query that a block of code may affect and to ignore irrelevant statements. In other words, any statements that cannot affect the memory state in the query can be skipped. We can thus focus our discussion on an auxiliary judgment form $\vdash \{R\}\ c\ \{Q\}$ that describes how the assignment commands affect a query and its point-to and instance constraints. Informally, it says, "We can witness a query Q after assignment command c by executing c if we can also witness one of the sub-queries R before c." From an algorithmic perspective, we can view $\vdash \{R\}\ c\ \{Q\}$ as generating sub-queries that when combined with a frame may yield additional contradictions. We assume this judgment implicitly has access to the points-to graph $\mathring{G}$: $\langle \mathring{V}, \mathring{E} \rangle$ computed by the up-front analysis. We use the $\mathrm{pt}_{\mathring{G}}^{\circ}(\cdot)$ function to get the points-to set of a program variable via $\mathrm{pt}_{\mathring{G}}^{\circ}(x) \stackrel{\text{def}}{=} \{a \mid (x \Mapsto a) \in \mathring{E}\}$ or a field of a program variable via $\mathrm{pt}_{\mathring{G}}^{\circ}(y.f) \stackrel{\text{def}}{=} \{a_j \mid a_i \in \mathrm{pt}_{\mathring{G}}^{\circ}(y) \text{ and } (a_i.\mathsf{f} \Mapsto a_j) \in \mathring{E}\}$.

Backwards Transfer Functions and Instance Constraints. Rule WitNew says that the exact points-to constraint $x \mapsto \widehat{v}$ can be witnessed, or *produced*, by the allocation command $x := \mathtt{new}_a\, \tau()$ if instance $\widehat{v}$ may have been created at allocation site a. Following our axiomatization, the instance constraint $\widehat{v}\,\mathsf{from}\, a \cap \mathring{r}$ may immediately reduce to false if $a \notin \mathring{r}$. Or, we can drop it (without loss of precision) because this instance cannot exist before its allocation at this statement. Such a contradiction is precisely the reason for refuting the path at the new arr_1 allocation (program point 16) in our motivating example (Figure 1).

Now, consider rule WitAssign: it says that the exact points-to constraint $x \mapsto \widehat{v}$ can be produced by the assignment command $x := y$ if $y \mapsto \widehat{v}$ can be witnessed before this assignment *and* the instance $\widehat{v}$ can come from a region common to the points-to set of y and the region $\mathring{r}$. If the points-to set of y and the region $\mathring{r}$ are disjoint, we can derive a contradiction because the instance $\widehat{v}$ cannot come from any allocation site. Observe that WitAssign leverages the 'from' constraint and the up-front points-to analysis result to eagerly discover that *no* allocation site satisfies the conditions

required for a witness (rather than observing that a *particular* allocation site does not satisfy the conditions required for a witness, as in WITNEW). To get a sense for why these eager refutations are critical for scaling, consider the path refutation due to the binding of `val` to `"hello"` at the `objs.push` call site (program point 8). In our example, this refutation is via WITNEW because `"hello"` is a `String` allocation, but we can easily imagine a variant where `objs.push` is called with a program variable y that can conservatively point-to a large set of non-Activity objects. For such a program, WITASSIGN would allow us to discover a path refutation at the assignment corresponding to the binding rather than requiring us to continue exploration across the potentially exponential number of paths to the allocation sites that flow to y.

The WITREAD rule is quite similar to WITASSIGN except that we existentially quantify over the instance to which y points (i.e., $\widehat{u}$). We set up an initial points-to region constraint for the fresh symbolic variable $\widehat{u}$ based on the points-to set of y and narrow the points-to region of $\widehat{v}$ using the points-to set of $y.f$. As in WITASSIGN, we can derive a contradiction based on this narrowing if the region $\mathring{r}$ and the points-to set of $y.f$ are disjoint. Here, we have taken some liberty with notation placing, for example, 'from' constraints under the iterated separating conjunctions; we recall that $*$ collapses to $\wedge$ for pure constraints.

In the WITWRITE rule, the post-formula consists of two cases for each edge $\widehat{v}_i.f \mapsto \widehat{u}_i$ in the pre-formula: (1) the field write $x.f := y$ did not produce the edge $\widehat{v}_i.f \mapsto \widehat{u}_i$ (the first disjunct Q), or (2) the field write did produce the edge (the second set of disjuncts over all Q_i). If the write $x.f := y$ did produce the points-to edge $\widehat{v}_i.f \mapsto \widehat{u}_i$, then the points-to regions of $\widehat{v}_i$ and $\widehat{u}_i$ are restricted based on the points-to sets of x and y, respectively. The "not produced" case represents the possibility that this write updates an instance other than a $\widehat{v}_i$ (as reflected by the $x \mapsto \widehat{x}$ and $\widehat{v}_i \neq \widehat{x}$ conditions).

While WITWRITE can theoretically generate a huge case split, we have observed that the combination of instance constraints and separation typically allow us to find refutations quickly in practice (see Section 4). In particular, the "not written" case can often be immediately refuted by separation. For example, we end up with a contradictory query where a local variable `x` has to point to two different instances simultaneously (i.e., $x \mapsto \widehat{v} * x \mapsto \widehat{u} \wedge \widehat{v} \neq \widehat{u}$).

Guards and Control Flow. Except for loops (see Section 3.3), the remaining rules mostly relate to control flow and are quite standard (shown inset). To discover a contradiction on pure constraints, we state that the guard condition of an `assume` must hold in the pre-query. We write $e[M]$ for interpreting the program expression e in the memory state M.

The WITCHOICE rule analyzes each branch independently. Our implementation avoids path explosion due to irrelevant path sensitivity by adding pure constraints from `if`-guards only when the queries on each side of the branch are different (as in previous work [18, 39]), though our rules do not express this.

WITASSUME
$$\frac{}{\vdash \{M \wedge P \wedge e[M]\}\ \texttt{assume}\, e\ \{M \wedge P\}}$$

WITSEQ
$$\frac{\vdash \{R''\}\ s_1\ \{R'\} \quad \vdash \{R'\}\ s_2\ \{R\}}{\vdash \{R''\}\ s_1\ ;\ s_2\ \{R\}}$$

WITCHOICE
$$\frac{\vdash \{R_1\}\ s_1\ \{R\} \quad \vdash \{R_2\}\ s_2\ \{R\}}{\vdash \{R_1 \vee R_2\}\ s_1\ [\!]\ s_2\ \{R\}}$$

3.3 Loop Invariant Inference and Query Simplification

In this section, we finish our description of witness-refutation search by discussing our loop invariant inference scheme.

Roughly speaking, we infer loop invariants by repeatedly performing backwards symbolic execution over the loop body until we reach a fixed point over the domain of points-to constraints. To ensure termination, we drop all pure constraints affected by the loop body and fix a static bound on the number of instances of each abstract location to materialize. In our experiments, a static bound of one has been sufficient for precise results. The WITLOOP rule (shown inset) simply states that if the loop body has no effect on the query, the loop has no effect on it.

WITLOOP
$$\frac{\vdash \{R\}\ s\ \{R\}}{\vdash \{R\}\ \texttt{loop}\, s\ \{R\}}$$

By itself, this rule only handles the degenerate case where a loop can be treated as `skip` with respect to the query. For this case, the disjunctive set of queries R is trivially a loop invariant. For more interesting cases, we use WITCASES to consider each query in the refutation state individually so that we can infer an over-approximate loop invariant for each one. Thus, we infer a loop invariant on-the-fly for each *path program* rather than joining all queries at the loop exit and then inferring an invariant for all backwards paths into the loop (similar to [35] but with heap constraints).

WITABSTRACTION
$$\frac{R'_2 \models R'_1 \quad \vdash \{R'_2\}\ s\ \{R_2\} \quad R_1 \models R_2}{\vdash \{R'_1\}\ s\ \{R_1\}}$$

To preserve refutation soundness, we want to ensure that a contradiction false is only derived when there does not exist a concrete path witnessing the given query and thus must over-approximate loops. A sound, backwards over-approximation can be obtained by weakening the post-loop query Q_i. Since an individual query is purely conjunctive, we can weaken it quite easily by "dropping" constraints (i.e., removing conjuncts). Intuitively, dropping constraints is refutation-sound because it can only make it more difficult to derive a contradiction. This over-approximation is captured by the WITABSTRACTION rule. The rule says that at any program point, we can drop constraints, and doing so preserves refutation soundness (Theorem 1).

We write $R_1 \models R_2$ for the semantic entailment and correspondingly rely on a sound decision procedure in our implementation (used in WITABSTRACTION). Entailment between a finite separating conjunction exact points-to constraints can be resolved in a standard way by subtraction [5]. Without inductive predicates, the procedure is a straightforward matching. Entailment between the 'from' instance constraints can be defined as follows:

$$(\widehat{v}_1\ \mathsf{from}\ \mathring{r}_1) \models (\widehat{v}_2\ \mathsf{from}\ \mathring{r}_2) \quad \text{iff} \quad \widehat{v}_1 = \widehat{v}_2 \text{ and } \mathring{r}_1 \subseteq \mathring{r}_2 \qquad (\S)$$

As previously mentioned, 'from' constraints are represented as sets associated with a symbolic variable and solved with ordinary set operations. We discharge other pure constraints using an off-the-shelf SMT solver, so precision of reasoning about those constraints is with respect to the capabilities of the solver.

With these tools, our loop invariant inference is a rather straightforward fixed-point computation. For a loop statement `loop`s and a post-loop query Q, we iteratively apply the backwards predicate transformer for the loop body s to saturate a set of sub-queries at the loop head. Let R_0 be some refutation state such that $\vdash \{R_0\}\ s\ \{Q\}$, and let $R_{i+1} = R_i \vee R'$ where $\vdash \{R'\}\ s\ \{R_i\}$. We ensure that the chain of $R_0 \models R_1 \models \cdots$ converges by bounding the number of instances or materializations from the abstract locations. Since there are a finite number of abstract locations, the number of points-to constraints in any particular query is bounded by the number of edges in the points-to graph (i.e., $|\mathring{E}|$). For the base domain of pure constraints, widening [14] can be used to ensure convergence. Our implementation uses a trivial widening that drops pure constraints that may be modified by the loop.

Query Simplification with Disaliasing. The WITABSTRACTION rule captures backwards over-approximation by saying that at any point, we can weaken a refutation state without losing refutation soundness. Conceptually, we can weaken by replacing any symbolic join $\vee$ with an over-approximate join $\sqcup$. We perform one such join by replacing the refutation state $Q_1 \vee Q_2$ with Q_2 if $Q_1 \models Q_2$. Note that this join does not lose precision. Intuitively, for a refutation state R, we are interested in witnessing *any* query in R or refuting *all* queries in R. Here, a refutation of query Q_2 implies a refutation of Q_1, so we only need to consider Q_2.

To enable this join to apply often, we enforce a normal form for our queries by dropping certain kinds of constraints. As formalized in Figure 4, the backwards transfer functions for assignment commands c are as precise as possible, including the generation of disequality constraints in WITWRITE. These disequality constraints are needed locally to check for refutations due to separation, as detailed in Section 3.2. However, if this check passes, we drop them before proceeding and instead keep only the disaliasing information implied by separation and the instance from constraints. While this weakening could lose precision (e.g., if the backwards analysis would later encounter an `if`-guard for the aliasing condition), we hypothesize that this situation is rare and that the most useful disaliasing information is captured by separation and instance constraints.

In our implementation, we are much closer to a path-by-path analysis than our formalization would indicate. Refutation states are represented as a worklist of pending (non-disjunctive) queries to explore. To apply the simplification described above, we must keep a history of queries seen at a given program point: if we have previously seen a weaker query at this program point, then we can drop the current query. We keep a query history only at procedure boundaries and loop heads. This simplification has been especially critical for procedures.

Soundness. We define a concrete store σ be a finite mapping from variables or address-field pairs to concrete values (i.e., $\sigma : \mathbf{Var} \uplus (\mathbf{Addr} \times \mathbf{Field}) \rightharpoonup_{\text{fin}} \mathbf{Val}$) and give a standard big-step operational semantics to our basic imperative language The judgment form $\sigma \vdash s \downarrow \sigma'$ says, "In store σ, statement s evaluates to store σ'." Furthermore, we write $\sigma \models R$ to say that the store σ is in the concretization of the refutation state R. The definition of $\sigma \models R$ is as would be expected in comparison to separation logic. We need to utilize two other concrete semantic domains: a valuation η that maps instances to values (i.e., $\eta : \mathbf{Instance} \rightarrow \mathbf{Val}$) and a *regionalization* ρ that maps abstract locations to sets of concrete addresses (i.e., $\rho : \mathbf{AbsLoc} \rightarrow \wp(\mathbf{Addr})$). The regionalization gives meaning to the 'from' instance constraint. With these definitions, we precisely state the soundness theorem.

Theorem 1 (Refutation Soundness)**.** *If* $\vdash \{R_{\text{pre}}\}\ s\ \{R_{\text{post}}\}$ *and* $\sigma_{\text{pre}} \vdash s \downarrow \sigma_{\text{post}}$ *such that* $\sigma_{\text{post}} \models R_{\text{post}}$*, then* $\sigma_{\text{pre}} \models R_{\text{pre}}$*. As a corollary, refutations (i.e., when* R_{pre} *is* false*) are sound.*

Interestingly, the standard consequence rule from Hoare logic states the opposite in comparison to WITABSTRACTION by permitting the strengthening of queries. Doing so would instead preserve witness precision; that is, any path program witness exhibits some witness path (up to termination).

4. Case Study: Activity Leaks in Android

We evaluated our witness-refutation analysis by using it to find Activity leaks, a common class of memory leaks in open-source Android applications. We explain this client in more detail below. Our experiments were designed to test two hypotheses. The first and most important concerns the *precision* of our approach: we hypothesized that witness-refutation analysis reports many fewer false alarms than a flow-insensitive points-to analysis. We tried using a flow-insensitive analysis to find leaks, but found that the number of alarms reported was too large to examine manually. To be useful, our technique needs to prune this number enough for a user to effectively triage the results and identify real leaks. Our second hypothesis concerns the *utility* of our techniques: we posited that (1) our mixed symbolic-explicit is an improvement over both a fully explicit and a fully symbolic representation, (2) our query simplification significantly speeds up analysis, and (3) our on-the-fly loop invariant inference is needed to preserve precision in the presence of loops.

Client. Activity leaks occur when a pointer to an Activity object can outlive the Activity. The operating system frequently destroys Activity's when configuration changes occur (e.g., rotating the phone). Once an Activity is destroyed, it can never be displayed to the user again and thus represents unused memory that should be reclaimed by the garbage collector. However, if an application developer maintains a pointer to an Activity after it is destroyed, the garbage collector will be unable to reclaim it. In our experiments, we check if *any* Activity instance is ever reachable from a static field, a flow-insensitive property. Though a developer could safely keep a reference to an Activity object via a static field that is cleared each time the Activity is destroyed, this is recognized as bad practice.

Activity leaking is a serious problem. It is well-documented that keeping persistent references to Activity's is bad practice; we refer the reader to an article[2] in the Android Developers Blog as evidence. The true problem is that it is quite easy for developers to inadvertently keep persistent references to an Activity. Sub-components of Activity's (such as Adapter's, Cursor's, and View's) typically keep pointers to their parent Activity, meaning that any persistent reference to an element in the Activity's hierarchy can potentially create a leak.

Precision of Our Techniques: Threshing Alarms. We implemented our witness-refutation analysis in the THRESHER tool, which is publicly available.[3] Additional details on our implementation are included at the end of this section. All of our experiments were performed on a machine running Ubuntu 12.04.2 with a 2.93 GHz Intel Xeon processor and 32GB of memory. Though our analysis is quite amenable to parallelization in theory, our current implementation is purely sequential.

To evaluate the precision of our approach, we computed a flow-insensitive points-to graph for each application and the Android library (version 2.3.3) using WALA's 0-1-Container-CFA pointer analysis (a variation of Andersen's analysis with unlimited context sensitivity for container classes). For each heap path from a static field f to an Activity instance A in the points-to graph, we asked THRESHER to witness or refute each edge in the path from source to sink. If we refuted an edge in the heap path, we searched for a new path. We repeated this until THRESHER either witnessed each edge in the heap path (i.e., confirmed the flow-insensitive alarm) or refuted enough edges to prove that no heap path from f to A can exist (i.e., filtered out the leak report). We allowed an exploration budget of 10,000 path programs for each edge; if the tool exceeded the budget, we declared a timeout for that edge and considered it to be not refuted. On paths with call stacks of depth greater than three, we soundly skipped callees by dropping constraints that executing the call might produce (according to a mod/ref analysis computed alongside the points-to analysis). We limited the size of the path constraint set to at most two. Allowing larger path constraint sets slowed down the symbolic executor without increasing precision. We ran in two configurations: one with the Android library as-is (**Ann?**=N), and one where we added a single annotation to the HashMap class to indicate that the shared `EMPTY_TABLE` field can never point to anything (**Ann?**=Y). We did this because we observed that the use of the null object pattern in the HashMap class was a major source of imprecision for the flow-insensitive analysis (cf. Figure 1), but we wanted to make sure that it was not the only one our tool was able to handle.

[2] `http://developer.android.com/resources/articles/avoiding-memory-leaks.html`

[3] `https://github.com/cuplv/thresher`

	Benchmark *Size*			*Filtering* Effectiveness						Computational *Effort*			
Benchmark	**SLOC**	**CGB**	**Ann?**	**Alrms**	**RefA(%)**	**TruA(%)**	**FalA(%)**	**Flds**	**RefFlds**	**RefEdg**	**WitEdg**	**TO**	**T (s)**
PulsePoint♠	no src	198K	N	24	16 (67)	8 (33)	0 (0)	3	2	47	40	1	750
			Y	16	8 (50)	8 (50)	0 (0)	2	1	40	31	0	95
StandupTimer♣	2K	240K	N	25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1199
			Y	25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1068
DroidLife♠	3K	132K	N	3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
			Y	3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
OpenSudoku	6K	229K	N	7	1 (14)	0 (0)	6 (86)	1	0	2	21	1	1596
			Y	0	0 (0)	0 (0)	0 (0)	0	0	0	0	0	0
SMSPopUp♠	7K	232K	N	5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	49
			Y	5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	46
aMetro♠	20K	326K	N	144	18 (12)	36 (25)	90 (63)	8	1	62	66	3	4226
			Y	54	18 (33)	36 (67)	0 (0)	3	1	55	24	0	18
K9Mail♠	40K	394K	N	364	78 (21)	64 (18)	222 (61)	14	3	141	106	1	1130
			Y	208	130 (63)	64 (49)	14 (7)	8	5	124	80	0	374
Total	78K	1751K	N	572	129 (22)	115 (20)	332 (58)	33	9	280	287	6	8991
			Y	311	172 (55)	115 (37)	24 (8)	20	10	247	189	0	1602

Table 1. This table characterizes the size of our benchmarks, highlights our success in distinguishing false alarms from real leaks, and quantifies the effort required to find refutations. ♠'s indicate a benchmark in which we found an observable leak, and ♣ indicates a latent leak. The *Size* column grouping gives the number of source lines of code **SLOC** and the number of bytecodes in the call graph **CGB** for each app as well as the annotation configuration **Ann?**=Y/N. The *Filtering* column grouping characterizes the effectiveness of our approach for filtering false alarms. The first four columns list the number of (static field, Activity) alarm pairs reported by the points-to analysis **Alarms**, number of alarms refuted by our approach **RefA**, number of true alarms **TruA**, and the number of false alarms **FalA**. The final two columns of this group give the number of leaky fields reported by the points-to analysis **Fields** and the number of these fields **RefFlds** that we can refute (i.e., prove that the field in question cannot point to *any* Activity). The *Effort* columns describe the amount of work required by our filtering approach. We list the number of edges refuted **RefEdg**, edges witnessed **WitEdg**, edge timeouts **TO**, and the time **T (s)** taken by the symbolic execution phase in seconds. This number does not include points-to analysis time, which ranged from 8–46 seconds on all benchmarks.

Table 1 shows the results of this experiment. We first comment on the most interesting part of the experiment: the *filtering effectiveness* of our analysis. As we hoped, our analysis is able to refute many of the false alarms produced by the flow-insensitive points-to analysis. Overall, we refute 129/457 = 28% of these false alarms in the un-annotated configuration and 172/196 = 87% of these false alarms in the annotated configuration. Contrary to our expectations, we found many *more* refutations in the **Ann?**=Y configuration, confirming that our technique can indeed remedy imprecision other than the pollution caused by HashMaps.

Unfortunately, this also means that our analysis is not always able to remedy the imprecision caused by HashMaps. The major problem is that the **Ann?**=N configuration fails to refute many of the HashMap-related edges due to timeouts. In fact, most of the false alarms that are not common to both configurations stem from (soundly) not considering timed-out edges to be refuted. We observed that a timeout commonly corresponds to a refutation that the analysis was unable to find within the path program budget. This is not surprising; finding a witness for an edge only requires finding a single path program that produces the edge (which we can usually do quickly), but to find a refutation we must refute *all* path programs that might produce an edge (which is slow and sometimes times out, potentially causing precision loss).

For example, the single timeout in the **Ann?**=N run of K9Mail occurs on a HashMap-related edge that is refutable, but quite challenging to refute. As it turns out, refuting this edge is extremely important for precision—upon further investigation, we discovered that the analysis would have reduced the number of false alarms reported by over 100 if it had been able to refute it! In the **Ann?**=Y configuration, this edge disappears from the flow-insensitive points-to graph. We can see that this increases the number of alarms we are able refute from 78 to 130 even though the number of alarms reported by the flow-insensitive points-to analysis falls from 364 to 208.

```
public class EmailAddressAdapter extends ResourceCursorAdapter {
  private static EmailAddressAdapter sInstance;
  public static EmailAddressAdapter getInstance(Context context) {
    if (sInstance == null)
      sInstance = new_adr0 EmailAddressAdapter(context);
    return sInstance;
  }
  private EmailAddressAdapter(Context context) { super(context); }
}
```

Figure 5. A confirmed Activity leak discovered in K9Mail.

We now comment on the *computational effort* required to refute/witness edges. We first observe that the number of edges refuted is almost always greater then the number of alarms refuted, indicating that it is frequently necessary to refute several edges in order to refute a single (source, sink) alarm pair. For example, in the un-annotated run of aMetro, we refute 62 edges in order to refute 18 alarms. This demonstrates that the flow-insensitive points-to analysis is imprecise enough to find many different ways to produce the same false alarm.

We note that the running times are quite reasonable for an analysis at this level of precision, especially in the annotated configuration. No benchmark other than aMetro takes more than a half hour. Our tool would be fast enough to be used in a heavyweight cloud service or as part of an overnight build process.

Real* Activity *Leaks. As hypothesized, our tool's precision enabled us to ignore most false alarms and focus on likely leaks. We found genuine leaks in PulsePoint, DroidLife, SMSPopUp, aMetro, and K9Mail. Many of the leaks we found would only manifest under specialized and complex circumstances, but a few of the nastiest leaks we found would almost always manifest and are due to the same simple problem: an inappropriate use of the singleton pattern. We briefly explain one such leak from the K9Mail app.

In the code in Figure 5, the developer uses the singleton pattern to ensure that only one instance of EmailAddressAdapter is ever created. The leak arises when `getInstance()` is called with an Activity instance passed as the `context` parameter (which happens in several places in K9Mail). The Activity instance is passed backwards through the constructors of two superclasses via the `context` parameter until it is finally stored in the `mContext` instance field of the CursorAdapter superclass. For *every* Activity act_0 that calls `getInstance()`, the flow-insensitive points-to analysis reports a heap path $\mathsf{EmailAddressAdapter{\cdot}sInstance} \Mapsto \mathsf{adr}_0, \mathsf{adr}_0{\cdot}\mathsf{mContext} \Mapsto \mathsf{act}_0$. When the Activity instance is destroyed, the garbage collector will never be able to reclaim it because none of the pointers involved in the leak are ever cleared.

We found this leak in a version of K9Mail that was downloaded in September 2011 (all versions of the benchmarks we used are available in project's GitHub repository). We looked at the current version and noticed that the EmailAddressAdapter class had been refactored to remove the singleton pattern. We found the commit that performed this refactoring and asked the developers of K9Mail if the purpose of this commit was to address a leak issue; they confirmed that it was.[4]

We also discovered a very simple latent leak in StandupTimer that was also due to a bad use of the singleton pattern. We noticed that several of the path programs THRESHER produced for a field in this app would be a full witness for a leak if a single boolean flag `cacheDAOInstances` were enabled. Our tool correctly recognizes that this flag cannot ever be set and refutes the alarm report, but a modification to the program that enabled this flag would result in a leak. The path program witnesses our tool produces are always helpful in triaging reported leak alarms, but in this case even the *refuted* path program witness provided useful information that allowed us to identify an almost-leak. With a less constructive refutation technique, we might have missed this detail.

Utility of Our Techniques. To test our second set of hypotheses, we ran THRESHER on the benchmarks from Table 1 without using each of three key features of our analysis: mixed symbolic-explicit query representation, query simplification, and loop invariant inference. We hypothesized that: (1) using an alternative query representation would negatively affect scalability and/or performance, (2) not simplifying queries would negatively affect scalability and/or performance, and (3) the absence of loop invariant inference would negatively affect precision.

To test hypothesis (1), we implemented a fully symbolic query representation. In a fully symbolic representation, we do not track the set of allocation sites that a variable might belong to. We have up-front points-to information, but use it only to confirm that two symbolic variables are *not* equal (i.e., to prevent aliasing case splits in the style of [39]) and to confirm that a symbolic variable was allocated at a given site (as in WITNEW). This precludes both pruning paths based on the boxed 'from' constraints in Figure 4 and performing the entailment check between symbolic variables defined in Equation § in Section 3.3.

Using this fully symbolic representation, our analysis ran slower and timed out more often, but did not refute any fewer alarms than the run with the mixed representation. We observed several cases where a timeout caused the fully symbolic representation to miss refuting an edge that the mixed representation was able to refute, but in each case the edge turned out not to be important for precision (that is, it was one of many edges that needed to be refuted in order to refute an alarm, but both representations failed to refute all of these edges).

Benchmark	Ann?	T (slowdown)	TO (Δ)
PulsePoint	N	1237 (1.6X)	7 (+6)
	Y	220 (1.9X)	3 (+3)
StandupTimer	N	4946 (4.1X)	4 (+4)
	Y	4104 (3.8X)	4 (+4)
OpenSudoku	N	2984 (1.9X)	4 (+3)
	Y	-	-
SMSPopUp	N	95 (1.9X)	0 (+0)
	Y	76 (1.7X)	0 (+0)
aMetro	N	6863 (1.6X)	5 (+2)
	Y	18 (1X)	0 (+0)
K9Mail	N	990 (0.9X)	2 (+1)
	Y	454 (1.2X)	0 (+0)

Table 2. Performance of the fully symbolic representation as compared to the mixed symbolic-explicit representation.

The results of this experiment are shown in Table 2. We omit the results for DroidLife since they were unaffected by the choice of representation. For every other benchmark, we give the time taken with a fully symbolic representation, the number of times slower than the mixed representation this was (**T (slowdown)**), the number of edges that timed out, and how many timeouts were added over the mixed representation (**TO (Δ)**).

We can see that in both the annotated and un-annotated configurations, most benchmarks run at least 1.6X slower and time out on at least one more edge than they did with the mixed representation. The anomalous behavior of K9Mail in the un-annotated configuration occurs because the mixed representation is able to refute an edge that the fully symbolic representation times out on. Ultimately, this leads the mixed representation to make more progress towards (but ultimately fail in) refuting a particular alarm. The fully symbolic representation declares this particular alarm witnessed after the edge in question times out, which allows it to skip this effort and finish faster. Thus, hypothesis (1) seems to hold: using a fully symbolic representation negatively affected both performance and scalability as predicted, but choosing a fully symbolic representation did not ultimately affect the precision of the analysis in terms of alarms filtered.

To test hypothesis (2), we re-ran THRESHER on our benchmarks using the annotated Android library without performing any query simplification at all. This significantly hurt the performance of THRESHER on PulsePoint (102.4X slower), K9Mail (3.2X slower), and SMSPopUp (4.3X slower), but did not change the number of alarms refuted or witnessed for these benchmarks. On StandupTimer, not performing simplification caused the tool to run out of memory before completing the analysis, thus affecting both precision and performance. The performance of the tool on other apps was not significantly affected. Thus, hypothesis (2) seems to hold for the benchmarks that require significant computational effort.

Finally, to test hypothesis (3), we implemented a simple loop invariant inference that simply drops all possibly-affected constraints at any loop. With only this simple inference, the analysis was unable to refute some critical HashMap-related edges (using the unannotated library). This meant that the analysis could never distinguish the contents of different HashMap objects. This imprecision

[4] `https://groups.google.com/forum/?fromgroups=#!topic/k-9-mail/JhoXL2c4UfU`

prevented the analysis from refuting leak reports involving multiple HashMap's even on small, hand-written test cases. Our full loop invariant inference (Section 3.3) handled the hand-written cases precisely, but due to unrelated analysis limitations, it did not achieve any fewer overall refutations on our real benchmarks. Nevertheless, our testing confirmed hypothesis (3): our loop invariant inference was clearly necessary to properly handle Android HashMap's and similar data structures.

Implementation. THRESHER is built on top of the on the WALA program analysis framework for Java and uses the Z3 [19] SMT solver with JVM support via ScalaZ3 [34] to determine when path constraints are unsatisfiable. Like most static analysis tools that handle real-world programs, our tool has a few known sources of unsoundness. We do not reason about reflection or handle concurrency. We have source code for most (but not all) non-native Java library methods. In particular, the Android library custom implementations of core Java library classes (including collections) that we analyze. To focus our reasoning on Android library and application code, we exclude classes from Apache libraries, `java/nio/Charset`, and `java/util/concurrent` from the call graph. Though we track control flow due to thrown exceptions, we do not handle the **catch**`()` construct; instead, we assume that thrown exceptions are never caught.

Android apps are event-driven and (in general) Android event handlers can be called any number of times and in (almost) any order. We use a top-level harness that invokes every event handler defined for an application. Our harness allows event handlers to be invoked in any order, but insists that each handler is called only once in order to prevent termination issues. In our experiments, we did not observe any unsound refutations due to these limitations.

We do not do any modeling for special Android components such as Intent's and BroadcastReceiver's. Since most special components are used for communication between applications that run in separate memory spaces, we would not expect THRESHER to miss any memory leaks due to this modeling issue.

5. Related Work

Dillig et al. present precise heap analyses for programs manipulating arrays and containers [21, 22], with path and context sensitivity [20]. Our analysis introduces path and context sensitivity via on-demand refinement, in contrast to their exhaustive, summary-based approach. Our symbolic variables are similar to their index variables [21, 22] in that both symbolically represent concrete locations and enable lazy case splits. Unlike index variables, our symbolic variables do not distinguish specific array indices or loop iterations, since this was not required for our memory leak client. Also, our analysis does not require container specifications [22]; instead, we analyze container implementations directly. Hackett and Aiken [29] present a points-to analysis with intra-procedural path sensitivity, which is insufficient for our needs.

Several previous systems focused on performing effective backward symbolic analysis. The pioneering ESC/Java system [27] performed intra-procedural backward analysis, generating a polynomially-sized verification condition and checking its validity with a theorem prover. Snugglebug [11] performed inter-procedural backward symbolic analysis, employing directed call graph construction and custom simplifiers to improve scalability. Cousot et al. [17] present backward symbolic analysis as one of a suite of techniques for transforming intermittent assertions in a method into executable pre-condition checks. PSE [39] used backward symbolic analysis to help explain program failures, but for greater scalability, it did not represent full path conditions. Our work is distinguished from these previous systems by the integration of points-to analysis information, which enables key optimizations like mixed symbolic-explicit states and abstraction for loop handling.

Our analysis can be seen as refining the initial flow-insensitive abstraction of the points-to analysis based on a "counterexample" reachability query deemed feasible by that analysis. However, instead of gradually refining this abstraction as in, for example, counterexample-based abstraction refinement (CEGAR) [13] and related techniques [16] , our technique immediately employs concrete reasoning about the program, and then re-introduces abstraction as needed (e.g., to handle loops). In general, the above predicate-abstraction-based approaches have not been shown to work well for proving properties of object-oriented programs, which present additional challenges due to intensive heap usage, frequent virtual dispatch, etc. Architecturally, our system is more similar to recent staged analyses for typestate verification [25, 26], but our system employs greater path sensitivity and more deeply integrates points-to facts from the initial analysis stage. A path program [7] was originally defined in the context of improving CEGAR by pruning multiple counterexample traces through a loop at once. SMPP [30] performs SMT-based verification by exhaustively enumerating path programs in a forward-chaining manner (in contrast to our goal-directed search). The recent DASH system [4] refines its abstraction based on information from dynamic runs and employs dynamic information to reduce explosion due to aliasing.

Our witness-refutation search uses the "bounded" fragment of separation logic [41] and thus has a peripheral connection to recent separation-logic–based shape analyzers [6, 12]. In contrast to such analyzers, we do not use inductive summaries and instead use materializations from a static points-to analysis abstraction. Shape analysis using bi-abductive inference [10] enables a compositional analysis by deriving pre- and post-conditions for methods in a bottom-up manner and making a best effort to reach top-level entry points. The derivation of heap pre-conditions is somewhat similar to our witness-refutation search over points-to constraints, but our backwards analysis is applied on demand from a flow-insensitive query and is refined by incorporating information from an up-front, whole program points-to analysis. Recent work [24] has applied bi-abduction to detect real Java memory leaks in the sense of an object that is allocated but never used again. In contrast, our client is a flow-insensitive heap reachability property that over-approximates a leak that is not explicit in the code, but is realized in the Android run-time.

Similar to our path program witnesses, other techniques have aimed to either produce a concrete path witness for some program error or help the user to discover one. Bourdoncle [9] presents a system for "abstract debugging" of program assertions, in which the compiler aims to discover inputs leading to violations statically. Rival [42] presents a system based on combined forward and backward analysis for elucidating and validating error reports from the Astrée system [15]. Work by Ball et al. [3] observes that for showing the *existence* of program errors (as opposed to verifying their absence), a non-standard notion of abstraction suffices in which one only requires the existence of a concrete state satisfying any particular property of the corresponding abstract state (as opposed to all corresponding concrete states satisfying the property). We observe an analogous difference between refutation and witness discovery in Section 3. Similar notions underlie the "proof obligation queries" and "failure witness queries" in recent work on error diagnosis [23].

Previous points-to analyses have included refinement to improve precision. Guyer and Lin's client-driven pointer analysis [28] introduced context and flow sensitivity at possibly-polluting program points based on client needs. Sridharan and Bodik [44] presented an approach for adding field and context sensitivity to a Java points-to analysis via refinement. Recently, Liang et al. [36–38] have shown that highly-targeted refinements of a heap abstraction

can yield sufficient precision for certain clients. Unlike our work, none of the aforementioned techniques can introduce path sensitivity via refinement. A recent study on Andersen's analysis [8] used dependency rules akin to a fully-explicit analog of our mixed symbolic-explicit transfer functions in a flow-insensitive context.

6. Conclusion

We have presented THRESHER, a precise static analysis for reasoning about heap reachability with flow-, context-, and path-sensitivity and location materialization. THRESHER introduces such precision in an on-demand manner after running a flow-insensitive points-to analysis. By integrating flow-insensitive points-to facts directly into a mixed symbolic-explicit representation of the program state and computing sufficiently strong loop invariants automatically, our techniques scale well while maintaining good precision. In our evaluation, we applied THRESHER to the problem of detecting an important class of Android memory leaks and discovered real leaks while significantly improving precision over points-to analysis alone.

Acknowledgments. We thank Sriram Sankaranarayanan and the CUPLV group for insightful discussions, as well as the anonymous reviewers for their helpful comments. This research was supported in part by NSF under grant CCF-1055066.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.

[3] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *CAV*, 2005.

[4] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, 2008.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[7] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.

[8] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *SAS*, 2011.

[9] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*, 1993.

[10] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[11] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, 2009.

[12] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, 2005.

[16] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, 2007.

[17] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, 2011.

[18] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[19] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[20] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, 2008.

[21] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.

[22] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[23] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, 2012.

[24] D. Distefano and I. Filipović. Memory leaks detection in Java by bi-abductive inference. In *FASE*, 2010.

[25] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, 2004.

[26] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[27] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[28] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2), 2005.

[29] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, 2006.

[30] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[32] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[33] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

[34] A. S. Kksal, P. Suter, and V. Kuncak. Scala to the Power of Z3: Integrating SMT and Programming. In *CADE*, 2011.

[35] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, 2005.

[36] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *PLDI*, 2011.

[37] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, 2010.

[38] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, 2011.

[39] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *FSE*, 2004.

[40] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1), 2005.

[41] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[42] X. Rival. Understanding the origin of alarms in Astrée. In *SAS*, 2005.

[43] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1), 1998.

[44] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[45] B. Woolf. Null object. In *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.