Selective Control-Flow Abstraction via Jumping

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

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

Manu Sridharan
Samsung Research America, USA
m.sridharan@samsung.com

Abstract

We present *jumping*, a form of selective control-flow abstraction useful for improving the scalability of goal-directed static analyses. Jumping is useful for analyzing programs with complex control-flow such as *event-driven* systems. In such systems, accounting for orderings between certain events is important for precision, yet analyzing the product graph of all possible event orderings is intractable. Jumping solves this problem by allowing the analysis to selectively abstract away control-flow between events irrelevant to a goal query while preserving information about the ordering of relevant events. We present a framework for designing sound jumping analyses and create an instantiation of the framework for performing precise inter-event analysis of Android applications. Our experimental evaluation showed that using jumping to augment a precise goal-directed analysis with inter-event reasoning enabled our analysis to prove 90–97% of dereferences safe across our benchmarks.

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

Keywords Android static analysis; event-driven systems; control-flow abstraction

1. Introduction

We consider the problem of selective flow/path-sensitive static analysis of *event-driven* systems. These systems are becoming increasingly important due to their prevalent use in web and mobile applications. In event-driven systems, control-flow occurs via invocation of event callbacks that may or may not be ordered. *Inter-event* flow/path-sensitive reasoning is often important for precision, but such reasoning can be prohibitively expensive due to the large number of possible event orderings that must be considered.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814293

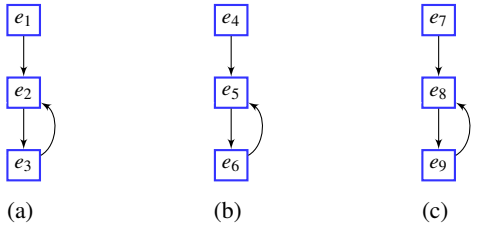


Figure 1: A simple event system containing three components with independent lifecycles.

To illustrate this problem, consider the simple event system in Figure 1. This system has three components (a), (b), and (c) with independent *event lifecycle graphs*. Events within an individual lifecycle graph are ordered by directed edges: $e_1 \rightarrow e_2$ specifies that e_1 must execute before event e_2. Otherwise, the events are unordered with respect to events in other lifecycle graphs. For example, the system specifies that any of events e_2, e_4–e_9 can execute immediately after e_1, but e_3 cannot. Event interleavings across lifecycle graphs are important to consider since all events may access shared mutable state.

The challenge in performing a flow/path-sensitive analysis of such systems is respecting intra-lifecycle ordering constraints while soundly accounting for interleavings of inter-lifecycle events. The obvious approach to achieving this result is to compute and analyze the product graph of all event lifecycle graphs in the event system. However, the number of edges in the product graph will be exponential in the number of components, and all such edges must be considered to perform a flow/path-sensitive analysis (even for the tiny system of Figure 1, the product graph contains 27 edges). For practical event systems with tens of components and hundreds of events like the Android applications we consider in Section 7, this graph quickly becomes intractable to represent—let alone analyze.

In practice, additional complications arise that make this problem even more difficult. Analyzing an individual event can be quite expensive because each event is essentially a standalone program—it may call thousands of procedures and contain loops and recursion. Component lifecycles can execute more than once, so the analysis may have to visit each edge in the product graph multiple times in order to compute a

fixed point. Finally, systems like Android implement lifecycle components and events via objects and instance methods, so the analysis may need to consider an unbounded number of instances of each lifecycle component.

Our insight is that although inter-event flow-sensitive reasoning is required to prove many properties of event-driven systems, most of these properties can be proven without considering all of the possible interleavings across component lifecycles. To leverage this insight to improve scalability, we need a *selective* form of control-flow abstraction flexible enough to apply flow/path-sensitive reasoning within an event lifecycle, but not across all edges in the lifecycle product graph. Though previous approaches to control-flow abstraction have effectively addressed the problem of selective context/object-sensitivity [26, 29, 30, 33], we are not aware of any previous work that can vary flow/path-sensitivity in the manner desired here. Previous flow-sensitive approaches to analyzing Android applications (e.g., [18]) have avoided this issue by assuming the lifecycles of different components cannot interleave. This is unsound and (as we will see in Section 2) can cause the analysis to miss real bugs.

In this paper, we tackle the challenge of selective flow/path-sensitive abstraction by introducing *jumping*, a form of goal-directed control-flow abstraction that enables the analysis to jump directly to code relevant to a particular goal query. Our key idea is that if we can identify the set of events that may affect the query at hand, we only need to reason about all possible orderings between these events in order to be sound. We have found that since the number of relevant events for a given query is typically small in practice, this approach is tractable even for large event systems. Our jumping framework allows us to limit the analysis to relevant events while retaining flow/path-sensitivity.

Our approach chooses locations to jump to by considering both data dependencies using a *data-relevance relation* and control dependencies using a notion of *control-feasibility*. The data-relevance relation enables the analysis to identify commands that may affect the current query, while control-feasibility information allows the analysis to consider event lifecycle constraints to preserve flow/path-sensitivity while jumping.

Though this paper focuses primarily on applying jumping to enable scalable analysis of event-driven systems, jumping is a general framework for selective control-flow abstraction whose utility goes beyond analyzing event-driven systems. Our framework provides an expressive way to describe a wide variety of sound control-flow abstractions that can be applied to any goal-directed backward abstract interpretation.

This paper makes the following contributions:

- We present a framework for *jumping analysis*, an approach for improving the effectiveness of goal-directed abstract interpretation via selective control-flow abstraction. We define soundness conditions for the relevance relation that enable jumping and prove the soundness of an analysis that performs jumps based on such a relation (Section 3).
- We devise an efficient points-to based technique for computing precise data-relevance information in heap-manipulating programs (Section 4).
- We give a semantics to the lifecycle graphs used by Android documentation to represent inter-event control-flow constraints, and we show how to specialize a general lifecycle graph to a specific subclass and object instance (Section 5).
- We use our jumping framework to design HOPPER, a practical jumping analysis for verifying safety properties in event-driven, heap-manipulating Android programs. HOPPER leverages our precise data-relevance relation along with control-feasibility information from Android lifecycle graphs to efficiently preclude consideration of concretely infeasible event orderings (Section 6).
- We evaluated HOPPER on the challenging client of checking null dereferences in event-driven Android programs (Section 7). Our results showed that adding jumping to a path-, flow-, and context-sensitive backward analysis decreased the number of unproven dereferences by an average of 54%. In addition, we found 11 real bugs in four different Android applications, nine of which have already been fixed via our submitted patches.

2. Overview

In this section, we motivate the difficulty and importance of verifying the absence of null dereferences in event-driven Android programs (Section 2.1) and show how jumping allows our analysis to prove the safety of dereferences without reasoning about an intractable number of event orderings (Section 2.2). Our running example (distilled from buggy code in the ConnectBot[1] app) will be attempting to prove the safety of the dereferences at lines 7 and 8 in Figure 2—that is, we want to show that `mService` and `mHostDb` (respectively) are always non-null at these lines. However, only the dereference at line 8 can be proven safe; the dereference at line 7 is buggy.

2.1 Motivation: Verifying Dereference Safety in Android

Null dereference errors (a Java `NullPointerException`, or NPE) are a major cause of failures in Android applications. In a search of the commit logs of the ten open-source Android apps used in our evaluation, we found 738 distinct commits containing the string "NPE" or "null," roughly 3% of all commits. Further, a recent paper on Facebook's INFER static analyzer reported that their internal database of production Android app crashes contained many null dereference errors [11]. Such errors cause crashes that stop the app and degrade user experience. Unlike crashes in web

[1] `https://github.com/connectbot/connectbot/pull/60`

```
class HostActivity extends onClickListener {
  // in method HostActivity.<init>
1   ManagerService mService = null;
2   HostDatabase mHostDb = null;

    void onCreate() {
      ServiceConnection cxn =
3       new ServiceConnection() {
          void onConnected(@Nonnull Service s) {
4           mService = (ManagerService) s;
          }
        };
      bindService(..., cxn);
5     findViewById(...).setOnClickListener(this);
6     mHostDb = new Database();
    }

    void onClick(View v) {
7     Host host = mService.getHost();
8     mHostDb.saveHost(host);
    }

    void onDestroy() {
9     mHostDb = null;
10    mService = null;
    }
}
```

Figure 2: A simple Android app with two components whose lifecycles are shown in Figure 3. The programmer uses correct reasoning about the lifecycle to ensure that the dereference at line 8 is safe, but mistaken assumptions about the lifecycle make the dereference at line 7 unsafe.

application code that can be fixed on the backend and pushed to the user when the page is refreshed, an app crash cannot be fixed until (1) an update fixing the bug is released and approved by the app store, and (2) the user elects to download the update, a process that can take weeks or even months [11]. Below, we discuss how subtleties of the Android app lifecycle can lead to null deference errors.

Bugs, Safety, and Lifecycles One reason that null dereferences in Android apps are easy to create and difficult to reason about is the complicated Android lifecycle. Though most events within the lifecycle of a single component are ordered, the lifecycles of different components can interleave arbitrarily and cause unexpected behavior. As a concrete example, Figure 3 shows the lifecycle graphs of the HostActivity and ServiceConnection classes from Figure 2. As in Section 1, a directed edge from event e_1 to e_2 means that e_1 must execute before e_2 is allowed to execute. The ε event represents a special "skip" event to soundly model the fact that user interaction events such as `onClick` may not be triggered. The edges between `onClick` and ε model the fact that a user can trigger the callback an arbitrary number of times.

The HostActivity and ServiceConnection components have independent lifecycles, but (as we can see from the code in Figure 2) they share the `mService` object. This leads to a null dereference at line 7 in the case that the `onClick` event fires before the `onConnected` event (since `mService` will still be null). This bug is due to faulty reasoning about the event-driven lifecycle of Android—the developer does not account for all possible interleavings between the HostActivity and ServiceConnection lifecycles.

On the other hand, the dereference at line 8 is safe because of ordering constraints in the HostActivity lifecycle. The developer delays initializing the `mHostDb` field to the heavyweight Database object until line 6 of the `onCreate` callback to avoid incurring the memory footprint of this object until it is needed. In addition, the developer assigns null to `mHostDb` at line 9 of the `onDestroy` event in order to relieve memory pressure as soon as possible (the enclosing HostActivity object may not become unreachable for some time after this event). These optimizations are safe because the lifecycle for HostActivity dictates not only that the `onCreate` event always executes after the constructor and before the `onClick` event, but also that the `onDestroy` event can only execute after all invocations of the `onClick` event.

Finding lifecycle sensitive bugs via testing is difficult given that (a) real apps have hundreds or thousands of events, (b) the developer must find the right combination of events that lead to a bug, and (c) exercising the app in a way that triggers the right events in the proper order is a tedious process. Thus, an effective static approach to this problem has the potential to significantly improve the state of affairs for Android app developers.

2.2 Proofs and Bug-Finding with Jumping Analysis

Though numerous static approaches to proving the absence of null dereferences have been proposed (e.g., [15, 22, 24, 25]), the key challenge in analyzing our motivating example does not concern the client of null dereferences specifically. Even a type-based approach with programmer-written nullness annotations would likely not work well. In Android, the nullness or non-nullness of a reference is frequently not a flow-insensitive invariant that holds at every program point or even an almost-everywhere invariant [12] that holds at nearly every program point. Instead, non-nullness (along with many other properties) holds during some phases of the lifecycle and not others.

Thus, the challenge for analyses (as we have explained in Section 1) is to perform precise reasoning about event orderings within a lifecycle without incurring the cost of reasoning about *all* event orderings. In what follows, we use the example in Figure 2 to demonstrate how jumping meets this challenge. This example has been simplified to contain only the events and instructions relevant to the two queries, but a real app would have many other lifecycle components whose event orderings the analysis might need to consider. In essence, the power of jumping is that it allows the analysis

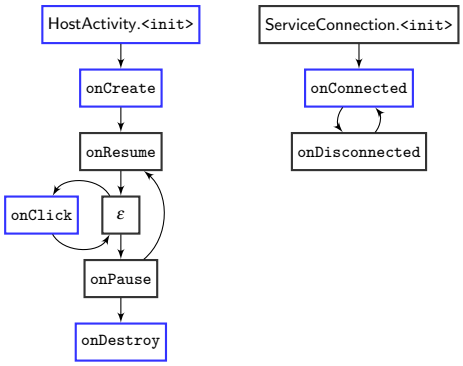


Figure 3: Lifecycle graphs for the HostActivity and ServiceConnection classes of Figure 2.

to soundly reduce a complex real-world event system into a simple system like the one in Figure 2.

Anatomy of a Jumping Analysis We consider extending THRESHER [7] (a path-, flow-, and context-sensitive backward analysis for *refuting* queries written in the symbolic heap fragment of separation logic [4] without inductive predicates) with the ability to jump, but jumping can just as easily be combined with any form of backward abstract interpretation (e.g., [9, 28]). At the intra-event level, the analysis behaves like THRESHER. When the backward analysis reaches an event boundary (that is, the entry of an application method that is invoked by the Android framework), the analysis chooses to compute a set of relevant events to jump to rather than continuing to follow backward control flow into the complex Android framework code.

From the entry of the current event e_{cur}, the analysis executes a jump by performing the following steps:

(1) **Identify important commands using data-relevance** For each constraint in the query, the analysis computes the set of *data-relevant* program commands whose concrete execution may produce a configuration in the concretization of the constraint. This process is similar to computing a partial slice that only considers immediately relevant commands (see Section 8 for a full discussion of the differences between our technique and slicing). Finding the set of relevant commands makes use of a global view of the program from a points-to graph computed by an up-front analysis.

(2) **Associate relevant commands to events** The analysis walks backward in the program's call graph from the containing method of each relevant command identified in step (1) and stops each time it hits an event boundary. This yields the set of events that may lead to the execution of relevant commands.

(3) **Order relevant events using control-feasibility information.** Though it would be sound to jump to all relevant events or even to jump directly to each relevant command, doing so loses information about the ordering of relevant events/commands, which is bad for analysis precision. In order to be precise, the analysis must account for the fact that only certain events are *control-feasible* with respect to the current event e_{cur} according to the Android lifecycle.

(4) **Jump to each control-feasible event.** The analysis forks a case split for each event that is both data-relevant and control-feasible, jumps to the exit of the event, and continues backward analysis for each case.

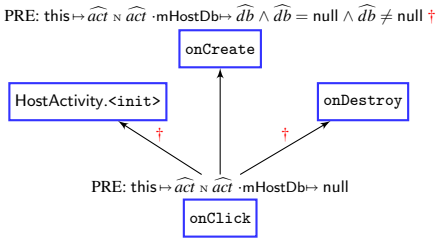


Figure 4: Proving safety of the dereference at line 8 of Figure 2 using jumping analysis.

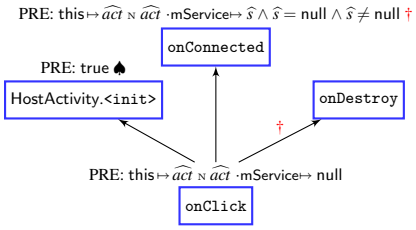


Figure 5: Failed safety proof for buggy dereference at line 7 of Figure 2 using jumping analysis.

Analyzing the Example App We now demonstrate how our jumping analysis uses the process described above to prove the safety of the dereference at line 8 and identify the bug at line 7. The analysis works in the style of THRESHER: it takes an initial query R that is a necessary precondition [13] for the bug to occur and attempts to prove safety (*refute* the query) by propagating the query backward from its initial program point ℓ in an attempt to derive a contradiction. This analysis is a form of proof by contradiction: it computes an over-approximation of the backward reachable states from the program point/query and refutes the query when it has derived the unreachability of (R, ℓ) (e.g., $\bot$—no possible concrete states) at a set of locations that together control-dominate the initial program point ℓ. For the dereference at line 8 to fail, the initial query is $\widehat{act}$ ·mHostDb↦ null[2] at program point 8, where $\widehat{act}$ is a symbolic variable representing a non-null HostActivity object and the ↦ edge denotes an *exact* points-to constraint in the sense of separation logic [27].

[2] All separation logic queries we write should be interpreted as describing a sub-heap (i.e., spatially conjoined with the predicate describing any heap).

The diagram in Figure 4 visualizes the process of proving the safety of this dereference by refuting the query. The analysis first uses THRESHER's transfer functions to propagate this precondition backward to the beginning of the `onClick` event, yielding the necessary bug precondition this$\mapsto \widehat{act}$ $\wedge$ $\widehat{act}$·mHostDb$\mapsto$ null shown in the figure.

At this point, the analysis chooses to perform a jump because it has reached an event boundary. The analysis decides which events to visit next using the four steps outlined above: first, it computes the data-relevant commands that may change the current bug precondition. This step yields the commands at lines 2, 6, and 9. Second, it uses the call graph to associate these relevant commands with their calling events, which yields the set of events HostActivity.`<init>`, `onCreate`, and `onDestroy`.

Third, the analysis uses the lifecycle graph for HostActivity in Figure 3 to perform control-feasibility filtering. The analysis determines that `onDestroy` is not control-feasible with respect to the current event `onClick` because `onDestroy` is not backward-reachable from `onClick` in the lifecycle graph for HostActivity. The analysis also determines that HostActivity.`<init>` is not control-feasible because it is *postdominated* by the relevant event `onCreate`—every feasible concrete execution reaching `onClick` visits `onCreate` between HostActivity.`<init>` and `onClick`.

Thus, the analysis concludes that it only needs to jump to `onCreate`. Figure 4 represents the decision not to jump to the data-relevant, but control-infeasible events HostActivity.`<init>` and `onDestroy` by marking the edges to these events with †'s. The directed edge from `onClick` to `onCreate` indicates that the analysis performs a jump from the entry of `onClick` to the exit of `onCreate` with the precondition shown for `onClick` as the abstract state.

When the analysis encounters the assignment at line 6 of the `onCreate` event, it refutes the query because there is an inconsistency between this command and the current abstract state: the points-to constraint $\widehat{act}$ ·mHostDb$\mapsto$ null says that the `mHostDb` field must hold the value null, but the command assigns a non-null Database value to this field. The analysis has therefore shown the safety of the dereference at line 8.

Figure 5 shows how the same analysis process (correctly) fails to prove the safety of the dereference at line 7. The analysis determines that the dereference is safe if the `onConnected` event executes before `onClick` and that the relevant event `onDestroy` is not control-feasible with respect to `onClick`, so it marks these paths as refuted (†). However, in the case that HostActivity.`<init>` is the last relevant event to fire before `onClick`, the command `mService = null` at line 1 discharges the precondition for `onClick`, leaving a necessary bug precondition of true. The analysis cannot hope to find a refutation given this precondition, so it gives up and reports the dereference at line 7 as a possible bug (as indicated by the ♠ symbol).

programs	$P,T ::= \{t_1,\ldots,t_n\}$
transitions	$t ::= \ell_1 \dashv[c]\!\rightarrow \ell_2$
commands	$c ::=$ `skip` $\mid$ `assume` $e \mid$ `call` $\ell \mid$ `return` $\ell \mid \ldots$
program labels	$\ell \in$ **Label**
call strings	$L \in$ **Strings** $::= [] \mid \ell::L$
abstract call strings	$\hat{L} \in$ **Strings**
concrete stores	$\rho \in$ **Store**
concrete states	$\sigma \in$ **State** $::= (\rho, L)$
abstract stores	$\hat{\rho} \in$ **Store**
abstract states	$R \in$ **State** $::= \top \mid \bot \mid (\hat{\rho},\hat{L}) \mid R_1 \vee R_2$
command semantics	$\langle \sigma, c\rangle \Downarrow \sigma'$
abstract semantics	$\vdash \{R_{\text{pre}}\}\ c\ \{R_{\text{post}}\}$
concretization	$\gamma :$ **State** $\rightarrow \wp($**State**$)$
invariant map	$I :$ **Label** $\rightarrow$ **State**

Figure 6: A language composed of atomic commands connected by unstructured control flow.

3. Jumping Analysis Framework

In this section, we formalize a framework for jumping analyses. The framework provides a mechanism by which any goal-directed, over-approximate, backward abstract interpretation can soundly perform "jumps" over irrelevant code given a *relevance relation* that meets certain soundness conditions. We will show how to instantiate this framework to perform tractable analysis of event-driven programs in Section 6.

3.1 Preliminaries: Commands and Transitions

We consider the imperative programming language of commands and unstructured control-flow presented in Figure 6. Our framework is parametric in the sub-language of commands and abstraction of concrete states chosen. A program in our language consists of a finite set of transitions t. We use P for the program of interest and T for a set of transitions in P. A transition $\ell_1 \dashv[c]\!\rightarrow \ell_2$ consists of a pre-label ℓ_1, a command c, and a post-label ℓ_2.

We assume that the concrete semantics of the commands are provided via a judgment form $\langle \sigma, c\rangle \Downarrow \sigma'$ that specifies how c transforms a concrete state σ to another state σ'. A transition relation for a small-step operational semantics of transition systems is given by a judgment form

$$\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle\ ,$$

defined by applying the concrete command semantics $\langle \sigma, c\rangle \Downarrow \sigma'$ to transition $t\colon \ell \dashv[c]\!\rightarrow \ell'$. A judgment $\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle$ is well-formed only if the pre- and post-labels of t are ℓ and ℓ', respectively.

Our model encodes conditional branching using an `assume` e command that blocks unless e evaluates to true and encodes looping using `assume` along with back edges in the transition relation. We represent procedure calls using call and return commands that are linked to (respectively) callee procedures and caller sites in the program transitions.

$$\boxed{I \vdash \ell_1 \dashv[c]\vdash \ell_2}$$

$$\textsc{A-Transition}\quad \frac{I(\ell_2) \models R \quad \vdash \{\, R' \,\}\ c\ \{\, R \,\} \quad R' \models I(\ell_1)}{I \vdash \ell_1 \dashv[c]\vdash \ell_2}$$

$$\boxed{I \vdash \ell}$$

$$\textsc{A-Jump}\quad \frac{I(\ell_{\text{post}}) \models R \quad \langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}} \quad I \vdash t \text{ for all } t\colon \ell_i \dashv[c_{ij}]\vdash \ell_j \in T_{\text{rel}} \quad R \models I(\ell_j) \text{ for all } \ell_j \quad I \vdash \ell_i \text{ for all } \ell_i}{I \vdash \ell_{\text{post}}}$$

Figure 7: Jumping analysis. The key A-JUMP rule expresses the ability to skip code based on a relevance relation.

Both commands manipulate a call string L composed of program labels. The concrete semantics for these commands are standard (they are given in Blackshear [6, Chapter 5]). For simplicity in presentation, we assume that all variables in the program are globally scoped and that parameter binding is accomplished via ordinary assignment.

We write R for an abstract state that over-approximates a set of concrete states defined by a concretization γ. Notationally, we use a semantic entailment relation $R_1 \models R_2$ defined over concretization as $\gamma(R_1) \subseteq \gamma(R_2)$. We write $\top$ for the state such that $\gamma(\top) \stackrel{\text{def}}{=} \textbf{State}$ and $\bot$ for the state such that $\gamma(\bot) \stackrel{\text{def}}{=} \emptyset$. Otherwise, a state is a finite disjunction of pairs of a store abstraction $\hat{\rho}$ and a call string abstraction $\hat{L}$. We leave the particular store and call string abstractions of interest unspecified. An abstract semantics for commands is given by the judgment form $\vdash \{\, R_{\text{pre}} \,\}\ c\ \{\, R_{\text{post}} \,\}$: a backward Hoare triple stating that for all concrete post-states in R_{post} in which c terminates for some concrete pre-state, then that concrete pre-state is in R_{pre}. More formally, the abstract semantics must satisfy the following soundness condition:

$$\begin{aligned}\text{If } \vdash \{\, R_{\text{pre}} \,\}\ c\ \{\, R_{\text{post}} \,\} \text{ and } \langle \sigma_{\text{pre}}, c \rangle \Downarrow \sigma_{\text{post}} \text{ such that} \\ \sigma_{\text{post}} \in \gamma(R_{\text{post}}), \text{ then } \sigma_{\text{pre}} \in \gamma(R_{\text{pre}}).\end{aligned} \tag{1}$$

As an informal shorthand, we say R_{post} is *may-witnessed* by executing the command c from R_{pre}. As a corollary of this soundness condition, if the analysis *refutes* an input query R_{post} (i.e., derives $\bot$ on all backward paths originating from R_{post}), then R_{post} represents a set of unreachable concrete states. However, the analysis may over-approximate by failing to refute R_{post} even if R_{post} does not represent any concretely reachable states.

3.2 Control-Flow Abstraction with Jumping

To describe static analysis of transition systems, we define an invariant map $I : \textbf{Label} \to \hat{\textbf{State}}$ that maps each program label ℓ to candidate invariants at ℓ given by an abstract state R. Our jumping analysis is defined by the judgment form $I \vdash \ell$ that asserts, "I over-approximates the concrete states from which ℓ can be reached in a state satisfying $I(\ell)$." As a shorthand, when the judgment $I \vdash \ell$ holds, we say that I *may-witnesses* $I(\ell)$, or simply I *may-witnesses* ℓ.

This judgment form relies on an auxiliary judgment form $I \vdash t$ that asserts, "For a transition $t\colon \ell_1 \dashv[c]\vdash \ell_2$, $I(\ell_1)$ over-approximates the concrete states from which executing c yields a state satisfying $I(\ell_2)$." As above, we say that I *may-witnesses* transition t when the judgment $I \vdash t$ holds.

In Figure 7, we define these two judgment forms. The A-TRANSITION rule defines $I \vdash t$, which is analogous to the consequence rule of standard Floyd-Hoare logic. The rule says that I may-witnesses $\ell_1 \dashv[c]\vdash \ell_2$ if there is a triple $\vdash \{\, R' \,\}\ c\ \{\, R \,\}$ that satisfies soundness condition (1), such that $I(\ell_2)$ is stronger than R and $I(\ell_1)$ is weaker than R'. This rule is essentially just a wrapper that lifts an abstract semantics for commands to an abstract semantics for transitions that is constrained by our invariant map I.

The key rule for our jumping analysis is A-JUMP, which decides the transitions that the analysis should visit next. This rule relies on a relevance relation written using the judgment form $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ that asserts, "Given an abstract state R at program label ℓ_{post}, the set of relevant program transitions is T_{rel}." Intuitively, the rule says to perform a backward jump from the current label ℓ_{post} to the post-label of each relevant transition in T_{rel}, skipping all transitions in between.

The rule's first two premises $I(\ell_{\text{post}}) \models R$ and $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ state that we compute a set of relevant transitions T_{rel} using some weakening of the query $I(\ell_{\text{post}})$. Allowing this weakening of the state abstraction is crucial, as weakening of the state can only make the set of relevant transitions T_{rel} smaller. The "$R \models I(\ell_j)$ for all ℓ_j" premise constrains the post-state of each relevant transition to be weaker than the current state R. Together, these two premises can be seen as consequence for the transitions skipped by the jump.

The premise "$I \vdash t$ for all $t\colon \ell_i \dashv[c_{ij}]\vdash \ell_j \in T_{\text{rel}}$" checks that I may-witnesses each relevant transition $t \in T_{\text{rel}}$—that is, it uses the auxiliary judgment form to abstractly execute each relevant transition that was jumped to. Finally, the remaining premise "$I \vdash \ell_i$ for all ℓ_i" recursively continues the backward analysis by checking that I may-witnesses the pre-label ℓ_i of each relevant transition that was jumped to.

Inference, Loops, and Recursion While the judgment form $I \vdash \ell$ is most easily read as a checking system for judging when I may-witnesses ℓ for a given I, we can obtain an inference system that computes an invariant map I with a standard post-fixed-point computation via abstract interpretation. We begin the abstract interpretation from a map I_0 initialized with the initial query at the start label ℓ and all other labels mapping to $\bot$. The analysis applies the A-JUMP rule to that start label and updates the invariant map with the inferred values for R. A weakening (as in premise $R' \models I(\ell_1)$ of A-TRANSITION) corresponds to an update to the invariant map with a join (i.e., $I_{i+1}(\ell_1) = I_i(\ell_1) \sqcup R'$) or widen $\triangledown$ as appropriate to break loops in the abstract interpretation. This

process continues with additional labels via the recursive invocation "$I \vdash \ell_i$ for all ℓ_i" in the A-JUMP rule until the invariant map computation reaches a fixed point.

In the analysis, an arbitrary context- and object-sensitivity policy can be implemented by the choice of the call string abstraction $\hat{L} \in \widehat{\textbf{Strings}}$ and the state abstraction $R \in \widehat{\textbf{State}}$. For example, a simple k-callstring context-sensitivity policy could keep a disjunct for distinct call strings up to length k (joining or widening abstract stores $\hat{\rho}$ as needed).

In our implementation, we uniformly handle all sources of looping and recursion by widening at targets of back edges. Our widening operator bounds the length of the materialized prefix of the abstract call string $\hat{L}$ (i.e., program labels $\ell_1 :: \cdots :: \ell_k :: \mathsf{anystring}$) and the number of materialized heap locations (i.e., $\mapsto$ constraints) in the abstract store $\hat{\rho}$.

3.3 Identifying Relevant Transitions

The A-JUMP rule is an extremely general rule that allows a wide variety of strategies for choosing the transitions that the analysis should jump to. All transitions not jumped to are skipped. But what transitions can the analysis soundly skip? A-JUMP allows the analysis to skip any transitions except for the set of transitions T_{rel} returned by the relevance relation, so the burden of ensuring soundness falls squarely upon this relation. In this subsection, we will first build intuition for what transitions can and cannot be skipped before formally defining the soundness conditions that must imposed on a relevance relation in order to ensure sound jumping.

Data-Relevance and Control-Feasibility For the relevance relation to be sound, it must not omit any important transitions that could be involved in may-witnessing the query of interest. There are many different strategies that the relevance relation can use to ensure this soundness property. We will show that each of these strategies can be thought of as (1) choosing a set of *data-relevant* transitions that would be sound to return on their own, then (2) soundly filtering this set using *control-feasibility* information. As a first consideration, consider a relevance relation that returns all transitions in the program as data-relevant and performs no filtering. This relevance relation is trivially sound: it cannot skip any important transitions because it does not skip any transitions at all. This corresponds to a fully flow- and context-insensitive view of the program, as every transition will be a jump target from every other transition.

The above strategy of taking all program transitions can be improved by considering a simple form of control-feasibility: postdominance in the control-flow graph. Intuitively, if transition t' postdominates transition t, there is no need to consider both t and t' as relevant, as all backward paths to t must go through t'. Hence, it is sufficient to only consider t' as relevant. Via this reasoning, we can conclude that instead of treating all transitions in the program as relevant, one can instead use just the immediate predecessor transitions of the current program label while remaining sound. We have simply recovered the standard approach taken by flow/path-sensitive analyses.

Treating just the immediate predecessors as relevant is quite precise, but it does not utilize the key strength of jumping: the ability to skip irrelevant transitions entirely. We can make better use of jumping by refining the set of transitions returned by the data-relevance step using information about the program's data-flow. We can leverage data-flow information by modifying the data-relevance step to return all transitions that may affect the query state as data-relevant. For example, if the abstract state R is $\mathtt{x} \geq 0$ for a program variable $\mathtt{x}$, then writes to any variable other than $\mathtt{x}$ clearly cannot affect the query state and can be soundly skipped. An interesting aspect of our framework is that it admits a more restrictive strategy in which only transitions that *weaken* the abstract state are considered relevant (further discussion in Blackshear [6, Chapter 5]).

This strategy of taking the set of data-relevant transitions is less precise than taking the set of the immediate predecessors. An analysis that uses the data-relevance strategy will lose flow-sensitivity while jumping because it does not take the program's control-flow into account. On the other hand, the data-relevance strategy is likely to be more efficient because it considers only the (typically small) set of transitions that may affect the query without reasoning about any of the other transitions in the program or the control-flow between them.

A very powerful strategy is combining the previous two: first identify a set of data-relevant transitions for the current query, then use control-feasibility information such as postdominance in the control-flow graph to filter this set as much as possible. Doing this allows us to get the best of both worlds while jumping: we can skip vast swaths of irrelevant code by limiting our consideration to the data-relevant transitions, and we can maintain flow-, path-, and context-sensitivity while jumping by filtering away control-infeasible transitions using information about the control-flow between data-relevant transitions. This is the approach that we take with the relevance relation we will define in Section 6.

Defining Relevance Soundness Motivated by the preceding discussion of sound strategies for selecting relevant transitions to explore, we define our soundness condition for a relevance relation in a way that permits all strategies to be thought of as computing data-relevant transitions, then filtering these conditions using control-feasibility:

Condition 1 (Relevance soundness)**.**
If $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$, $\langle \sigma, \ell_{\text{pre}} \rangle \xrightarrow[T]{}{}^{*} \langle \sigma', \ell_{\text{post}} \rangle$,
$t_{\text{irrel}} : \ \ell_1 \dashv [c] \mapsto \ell_2 \in P - T_{\text{rel}}$, *and* $\vdash \{ R' \}\ c\ \{ R \}$, *then either*
(a) $R' \models R$, *or*
(b) $\exists\, T_1, T_2$ *s.t.* $T = T_1 \hat{} T_2$, $t_{\text{irrel}} \notin T_2$ *and* $T_{\text{rel}} \cap T_2 \neq \emptyset$.

We write $\langle \sigma, \ell \rangle \xrightarrow[T]{}{}^{*} \langle \sigma', \ell' \rangle$ for the judgment form of multi-step concrete evaluation, that is, the reflexive-transitive closure of single-step concrete evaluation. This multi-step

concrete evaluation records each transition it visits between ℓ_{pre} and ℓ_{post} in the trace T. We denote trace concatenation with $T_1 \hat{} T_2$.

Condition 1(a) captures the soundness of returning data-relevant transitions by imposing restrictions on a transition t_{irrel} that is *not* returned by the relevance relation. It states that for any program transition t_{irrel} not included in the set of relevant transitions for state R, the pre-state R' with respect to the transition command c is at least as strong as R. From the analysis perspective, this means that it is sound to exclude t_{irrel} from the jump targets because it cannot possibly move R closer to being witnessed. This is a very general notion of data-relevance, as it captures relevance based on both modification and weakening.

Condition 1(b) captures the soundness of filtering data-relevant transitions based on control-feasibility information. It says that if a transition t_{irrel} is not included in the set of relevant transitions for program point ℓ_{post}, then we can decompose the trace of visited transitions T into a pre-trace T_1 and a post-trace T_2 such that the post-trace contains a relevant transition, but does not contain t_{irrel}. This means then some relevant transition $t_{\text{rel}} \in T_{\text{rel}}$ must always happen *between* t_{irrel} and ℓ_{post}. From the perspective of our backward analysis, this means that it is sound to exclude t_{irrel} from the set of jump targets because some relevant transition t_{rel} that will be jumped to postdominates t_{irrel} in the program control-flow.

Using data dependence analysis to skip analysis of clearly irrelevant code is an effective and commonly used technique in program analysis. The novelty of our framework for sound jumping analysis based on Condition 1 is the ability to *simultaneously* reason about data-relevance and control-feasibility information to filter away a larger set of transitions (i.e., transitions that are data-relevant, but not control-feasible). Our relevance soundness condition is both permissive and general: it allows all of the control-flow abstraction strategies we have discussed so far and opens the door for any strategy whose structure can be described as computing data-relevant transitions, then filtering using control-feasibility.

Soundness of Jumping Analysis Next, we state and prove a soundness theorem demonstrating that any relevance relation satisfying this soundness condition can be used to define a sound jumping analysis.

Theorem 1 (Soundness of jumping analysis)**.**
If $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ *and* $I \vdash \ell_{\text{post}}$ *such that* $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$*, then* $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$.

The theorem says that if the concrete state σ_{post} at program point ℓ_{post} is in the concretization of the abstract state stored at label ℓ_{post} of the invariant map, then σ_{dummy} is in the concretization of the abstract state stored at the pre-entry label ℓ_{dummy}. In the theorem, we write concrete state σ_{dummy} for a distinguished element of **State** that represents the uninitialized, "junk" state before beginning the execution of the program. The proof of this theorem can be found in Blackshear [6, Appendix A]. The primary challenge of this proof was in formulating Condition 1 to be strong enough to prove the theorem, but weak enough to permit a wide variety of strategies for control-flow abstraction (including all of the other strategies described in the "Data-Relevance and Control-Feasibility" discussion on page 7).

Instantiating the Framework This section has presented a general framework for performing control-flow abstraction in a goal-directed backward abstract interpretation. In subsequent sections, we will present a practical instantiation of this framework for effective goal-directed analysis of event-driven Android programs. Section 4 explains how we compute precise data-relevance information in the presence of heap-manipulating commands, Section 5 explains how we represent inter-event control in Android in a way that enables control-feasibility filtering, and Section 6 combines the previous two sections to define a relevance relation for efficient event-driven analysis that satisfies relevance soundness (Condition 1).

4. Computing Precise Data-Relevance Information for Separation Logic Constraints

In this section, we explain how we compute precise data-relevance information for heap constraints in a way that satisfies the data-relevance condition of relevance soundness (Condition 1(a)). The key challenge of computing data-relevance information in the presence of the heap is to compute a precise approximation of relevant writes (i.e., precisely identify commands that may write to portions of the heap described in abstract state). If the analysis is not effective at precisely identifying such commands, it may report too many commands as data-relevant and negate the scalability benefits of jumping. Our approach is to leverage a combination of up-front points-to information, instance-from constraints [7] relating object instances to abstract locations, and explicit disaliasing information to precisely identify heap dependencies.

4.1 Command Language, Abstract State, and Abstract Semantics

As explained in Section 3.1, our framework for jumping analysis can be used with any representation of abstract state, command language, and abstract semantics. Here, we will consider computing data-relevance information for the separation logic-based abstract representation of THRESHER [7]. Specifically, the relevance relation we define here complements the abstract semantics of THRESHER extended with the abstract semantics for performing procedure calls presented in Figure 10. We will carefully explain the the command language and abstract state used by THRESHER as well as the abstract semantics of procedure calls before defining our data relevance relation in Section 4.2.

commands	$c ::= x := e \mid x := \texttt{new}_a\,\tau() \mid x := y.f \mid x.f := y$
	$\mid \texttt{assume}\ e \mid \texttt{call}\ \ell \mid \texttt{return}\ \ell$
expressions	$e ::= x \mid \cdots$
abstract locations	a
program variables	x, y
object fields	f
types	τ

Figure 8: An imperative command language with dynamic memory allocation and heap reads/writes.

abstract states	$R ::= \top \mid \bot \mid (\hat{\rho}, \hat{L}) \mid R_1 \vee R_2$
abstract call strings	$\hat{L} ::= \mathsf{anystring} \mid \ell :: \hat{L}$
abstract stores	$\hat{\rho} ::= M \wedge F \mid \mathsf{false}$
memories	$M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v}.f \mapsto \widehat{u} \mid M_1 \ast M_2$
pure formulæ	$F ::= \mathsf{true} \mid F_1 \wedge F_2 \mid \widehat{v}\ \mathsf{from}\ \mathring{r} \mid \cdots$
points-to regions	$\mathring{r}, \mathring{s} ::= \{a_0, \ldots, a_n\}$
instances	$\widehat{v}, \widehat{u}, \widehat{o}$

Figure 9: Components of abstract state.

Command Language We consider a simple imperative command language with objects and dynamic memory allocation (Figure 8). Like in Figure 6, concrete states σ are pairs of a concrete store ρ and a concrete call string L. Commands interact with the program heap via reads, writes, and allocations. Programmers can create aliasing relationships using imperative assignment. We leave our language of expressions unspecified, but we assume that expressions are pure and include reads of variables. An abstract location a is named by a syntactic allocation site $x := \texttt{new}_a\,\tau()$ and represents a potentially unbounded number of object instances of type τ allocated from its naming site. The language also contains `call` and `return` commands for performing procedure calls (as explained in Section 3.1).

Abstract State The components of THRESHER's state are enumerated in Figure 9. The top-level analysis unit is an abstract state R which consists of an (abstract store, abstract call string) pair, a disjunction of abstract states, or the special $\top/\bot$ states representing all concrete states and an unreachable state (respectively). A symbolic instance $\widehat{v}$ represents a *single* instance of an object (unlike an allocation site, which may represent an unbounded number of objects).

Abstract call strings implement a simple call-site sensitive form of context-sensitivity. Call strings consist of either the special anystring call string representing any possible call string, or a known label ℓ prepended to an abstract call string.

A memory M consists of an arbitrary memory any, an exact points-to constraint, or a separating conjunction of two memories. We write any for an arbitrary memory instead of using the more traditional true because we use true to denote the boolean truth value. We interpret memory M as $M \ast \mathsf{any}$, but typically omit the any when writing a memory for the sake of conciseness.

Points-to edges in our analysis state are *exact* points-to constraints of the form $x \mapsto \widehat{v}$ or $\widehat{v}.f \mapsto \widehat{u}$. The form $x \mapsto \widehat{v}$ means that the program variable x contains a value represented by the symbolic instance $\widehat{v}$. Similarly, the form $\widehat{v}.f \mapsto \widehat{u}$ means the symbolic instance $\widehat{v}$ holds the value represented by the symbolic instance $\widehat{u}$ at the offset specified by its f field.

The spatial constraints in a memory M are conjoined with a pure formula F consisting of either the truth value true, a conjunction of pure formulæ, or a special instance-from constraint. Instance-from constraints of the form "$\widehat{v}$ from $\mathring{r}$" state that the instance $\widehat{v}$ must have been allocated from the region $\mathring{r}$ (a region $\mathring{r}$ is a set of allocation sites). The constraint $\widehat{v}$ from $\emptyset$ is equivalent to false since it means that $\widehat{v}$ could not have been allocated from any allocation site. Tracking instance-from constraints explicitly in the analysis enables deriving such contradictions, and have been shown to significantly decrease the number of aliasing case splits that a backward separation logic-based analysis must perform [7]. We leave the remaining forms of pure formulæ unspecified, but our implementation handles other pure constraints that can be easily represented and solved by an SMT solver (inequality, arithmetic constraints, etc.).

$\boxed{\vdash \{R'\}\ c\ \{R\}}$

A-RETURN

$$\frac{}{\vdash \{(\hat{\rho}, \ell :: \hat{L})\}\ \texttt{return}\ \ell\ \{(\hat{\rho}, \hat{L})\}}$$

A-CALL-OK

$$\frac{\ell = \ell_1}{\vdash \{(\hat{\rho}, \hat{L})\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho}, \ell :: \hat{L})\}}$$

A-CALL-REF

$$\frac{\ell \neq \ell_1}{\vdash \{\bot\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho}, \ell :: \hat{L})\}}$$

A-CALL-ANY

$$\frac{}{\vdash \{(\hat{\rho}, \mathsf{anystring})\}\ \texttt{call}\ \ell\ \{(\hat{\rho}, \mathsf{anystring})\}}$$

Figure 10: Abstract semantics of `call` and `return` commands.

Abstract Semantics for Procedure Calls Figure 10 defines the abstract semantics for `call` and `return` commands. We explained the meaning of the backward Hoare triple $\vdash \{R'\}\ c\ \{R\}$ in Section 3. The abstract semantics for commands follows those defined previously in Blackshear et al. [7, Figure 4], which include all command forms other than `call` and `return`. While the abstract semantics are not strictly nec-

essary for understanding the data-relevance relation that we will present in Section 4.2, we define the abstract semantics for `call` and `return` here for completeness.

The A-RETURN rule says that when the analysis moves backward across the statement `return` ℓ, the return label ℓ is prepended to the abstract call string. This constrains the abstract call string to reflect that any concrete execution could only have reached this program point if it previously visited a matching `call` ℓ instruction that pushed ℓ onto the call string. The A-CALL-OK rule expresses the case where the analysis subsequently encounters this matching call. If the label ℓ on top of the call string matches the label ℓ_1 of the call command, the analysis weakens the state by popping the label off of the call string. By contrast, the A-CALL-REF rule expresses the case where the analysis subsequently encounters a non-matching call. If the label on top of the call string ℓ does not match the label ℓ_1 of the call command, the analysis *refutes* the current path (derives $\bot$) since no concrete state could have a non-ℓ_1 label on top of its call string immediately after executing the command `call` ℓ_1. Finally, the A-CALL-ANY rule says that an unconstrained call string anystring can be propagated backward across a `call` command without any changes.

4.2 Creating a Data-Relevance Relation for Separation Logic Constraints

Figure 11 defines a data-relevance relation for the abstract state and semantics explained in Section 4.1. The top-level judgment form $R \rightsquigarrow T_{\text{rel}}$ asserts that the transitions in T_{rel} may be relevant to the abstract state R. The auxiliary judgment forms $\hat{\rho} \rightsquigarrow T_{\text{rel}}$ and $\hat{L} \rightsquigarrow T_{\text{rel}}$ assert the same thing for the two sub-components of an abstract state: an abstract store $\hat{\rho}$ and an abstract call string $\hat{L}$.

They key challenge in computing data-relevance information for separation logic is determining what commands might be relevant to a points-to constraint $\widehat{v}{\cdot}f \mapsto \widehat{u}$ stating that an object instance $\widehat{v}$'s field f contains the value $\widehat{u}$. For this constraint, we begin by considering commands that syntactically update field f (commands of the form $x.f := y$) as being possibly-relevant. Then, we can use the pure constraints in the abstract memory to further restrict relevant commands. In particular, if we have the instance-from constraint $\widehat{v}$ from $\mathring{r}$, we can restrict the relevant commands to those that satisfy the condition $\text{pt}(x) \cap \mathring{r} \neq \emptyset$. The function $\text{pt}(x)$ denotes the points-to set of x as computed by an up-front points-to analysis on the program P. The above condition ensures consistency between the points-to sets of x and the corresponding region $\mathring{r}$, rejecting any command that could not possibly yield the $\widehat{v}{\cdot}f \mapsto \widehat{u}$ points-to constraint because it writes to an object different from $\widehat{v}$.

This process of leveraging instance-from constraints and up-front points-to information to precisely identify heap writes is encoded in the R-WRITE rule of Figure 11. We can further restrict relevant writes based on disaliasing constraints (either explicitly given $\widehat{v} \neq \widehat{o}$ or implied by separation $\widehat{v}{\cdot}g \mapsto - \ast \widehat{o}{\cdot}g \mapsto -$), though these additional restrictions are not expressed in the rule for presentation. In other words, we consider all transitions that may modify $\widehat{v}{\cdot}f \mapsto \widehat{u}$ (syntactically) to be relevant and exclude a transition when we can prove that it does not modify $\widehat{v}{\cdot}f \mapsto \widehat{u}$ (semantically). This strategy satisfies the data-relevance condition of relevance soundness (Condition 1(a)), and we adopt a similar strategy to ensure soundness for the remaining rules.

The remaining rules for the judgment form $\hat{\rho} \rightsquigarrow T_{\text{rel}}$ define data-relevance for other store-manipulating command forms. The rules R-READ, R-NEW, and R-ASSIGN compute the relevant commands for a local points-to constraint $x \mapsto \widehat{v}$ by using program syntax to identify all possible writes to x. These rules essentially encode a flow-insensitive variation of reaching definitions. The R-SEP rule gives structure to the judgment form by recursively applying the relevance relation to each sub-memory of the store to find the relevant transitions for the entire store. It says that the set of relevant transitions for the store $\hat{\rho}$ is the union of the relevant transitions for each of its sub-memories.

The R-BOT, R-TOP, and R-ANY rules defines the base cases of relevance for an unreachable state, a state representing all concrete states, and the separation logic predicate any that is satisfied by any heap (respectively). Each of these rules returns only the special initial transition t_{init} representing the first transition in the program. We can think of these rules as saying that nothing is relevant to each of these states; the reason each rule returns t_{init} is that proving relevance soundness is much easier if we can maintain the invariant that the set of relevant transitions is never empty (more specifically, that it always contains t_{init}).

The R-CASES rule says that for a disjunction of abstract states $R_0 \vee R_1$, the set of relevant transitions is the union of the relevant transitions for R_0 and R_1. The R-SPLIT rule decomposes an abstract state into its store component and call string component, computes the relevant transitions for each component using the auxiliary relevance judgments, and returns the union of the relevant transitions.

Finally, the R-CALL rule says that a call command with return label ℓ_1 must be considered relevant to a call string with a label $\ell = \ell_1$ as its first label. In our backward analysis, the abstract semantics for `call` can weaken the abstract state by popping a label off of the call string, thereby creating a less constrained call string. Thus, we must consider all `call` instructions with labels matching the top of the call string to be relevant in order to be sound. However, at an event boundary our implementation always chooses to weaken the abstract call string to anystring at before computing data-relevance/control-feasibility information and jumping, so this rule is never applied in practice. Instead, R-ANYSTRING will always be applied. This rule is simply the call string analog of the R-ANY and R-TOP rules.

$\boxed{R \rightsquigarrow T_{\text{rel}}}$

R-CASES
$$\frac{R_0 \rightsquigarrow T_0 \quad R_1 \rightsquigarrow T_1}{R_0 \vee R_1 \rightsquigarrow T_0 \cup T_1}$$

R-BOT
$$\frac{}{\bot \rightsquigarrow \{t_{\text{init}}\}}$$

R-TOP
$$\frac{}{\top \rightsquigarrow \{t_{\text{init}}\}}$$

R-SPLIT
$$\frac{\hat{\rho} \rightsquigarrow T_1 \quad \hat{L} \rightsquigarrow T_2}{(\hat{\rho}, \hat{L}) \rightsquigarrow T_1 \cup T_2}$$

$\boxed{\hat{\rho} \rightsquigarrow T_{\text{rel}}}$

R-SEP
$$\frac{\hat{\rho} = M_1 \,\text{N}\, M_2 \wedge F \quad M_1 \wedge F \rightsquigarrow T_1 \quad M_2 \wedge F \rightsquigarrow T_2}{\hat{\rho} \rightsquigarrow T_1 \cup T_2}$$

R-ASSIGN
$$\frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := y]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \,\mathsf{from}\, \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}}$$

R-NEW
$$\frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := \texttt{new}_a\, \tau()]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \,\mathsf{from}\, \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}}$$

R-READ
$$\frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := y.f]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \,\mathsf{from}\, \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}}$$

R-WRITE
$$\frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x.f := y]\vdash \ell_j \text{ and } \text{pt}(x) \cap \mathring{r} \neq \emptyset \,\}}{\widehat{v}.f \mapsto \widehat{u} \wedge \widehat{v} \,\mathsf{from}\, \mathring{r} \wedge \widehat{u} \,\mathsf{from}\, \mathring{s} \wedge F \rightsquigarrow T_{\text{rel}}}$$

R-ANY
$$\frac{}{\mathsf{any} \wedge F \rightsquigarrow \{t_{\text{init}}\}}$$

$\boxed{\hat{L} \rightsquigarrow T_{\text{rel}}}$

R-CALL
$$\frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[\texttt{call}\ \ell_1]\vdash \ell_j \text{ and } \ell = \ell_1 \,\}}{\ell :: \hat{L} \rightsquigarrow T_{\text{rel}}}$$

R-ANYSTRING
$$\frac{}{\mathsf{anystring} \rightsquigarrow \{t_{\text{init}}\}}$$

Figure 11: A data-relevance relation that uses an up-front points-to analysis and instance-from constraints to precisely identify relevant commands for separation logic constraints.

4.3 Cost of Computing Data-Relevance

To conclude, we briefly comment on the cost of computing data-relevance information. Clearly, this process needs to be efficient in order for control-flow abstraction via jumping to enhance the scalability of the analysis. Our data-relevance relation's use of a precomputed points-to analysis typically allows us to compute relevance quite quickly.

One potential scalability concern is than many rules in Figure 11 quantify over every command in the program P. We note that in practice, we can often compute relevance much more efficiently by exploiting procedural abstraction and up-front points-to information. The R-ASSIGN, R-NEW, and R-READ rules compute the relevant statements for a constraint on some local variable x, so we only need to inspect each command in the method that x belongs to.

Shrinking the number of commands that the R-WRITE rule must consider is slightly more challenging, but we can do so using the points-to graph and instance-from constraints. Let $\mathring{E}$ be the edge set of the points-to graph, and let $x \Mapsto a$ denote a may-points to edge from the graph. Further, let the containing method of a local variable x be given by $method(x)$. Assuming that we are interested in determining the relevant write commands of the form $x.f := y$ for a heap constraint $\widehat{v}.f \mapsto \widehat{u} \wedge \widehat{v} \,\mathsf{from}\, \mathring{r} \wedge \widehat{u} \,\mathsf{from}\, \mathring{s}$, we can compute two sets of procedures: $P_{\widehat{v}} = \{\, method(x) \mid (x \Mapsto a) \in \mathring{E} \wedge a \in \mathring{r} \,\}$ and $P_{\widehat{u}} = \{\, method(y) \mid (y \Mapsto a) \in \mathring{E} \wedge a \in \mathring{s} \,\}$. These are the sets of methods containing locals that may point to $\widehat{v}$ and $\widehat{u}$ (respectively). Any method containing a write command that discharges the constraint $\widehat{v}.f \mapsto \widehat{u}$ must have a local variables pointing to both $\widehat{v}$ and $\widehat{u}$, so we only need to look at write commands from methods in the set $P_{\widehat{v}} \cap P_{\widehat{u}}$. In practice, this set is typically small enough to investigate efficiently.

5. Representing Inter-Event Control-Flow

In this section, we explain how we use *lifecycle graphs* to represent the inter-event control-flow in Android applications in a way that allows our analysis to address the challenges laid out in Section 1. We will make use of this information to check inter-event control-feasibility when we define a practical relevance relation for Android in Section 6.

For Android programs, we must consider two distinct kinds of control-flow information: intra-event control flow and inter-event control flow. Handling intra-event control-flow is the same as handling interprocedural control flow in an ordinary Java program, which is a well-understood problem. Control-flow between methods can be represented

using a call graph and control-flow within a method can be represented using a control-flow graph for the method.

Representing *inter-event* control-flow is more difficult because this information is not directly represented in the call graph. In fact, the logic for maintaining orderings among events lives in native code in the Android framework, so ordering information cannot be inferred by analyzing the Java portion of the framework alone.

Our approach to representing inter-event control-feasibility constraints is to formally define the meaning of the event ordering information that programmers have access to: the lifecycle documentation for Android components (e.g., the Activity lifecycle[3]). This documentation takes the form of *lifecycle graphs* where nodes are lifecycle event methods and directed edges express ordering constraints among the events. We have already seen how such graphs are useful in Section 2: Figure 3 specified the ordering of lifecycle events for the components used in the example and allowed the analysis to filter the set of relevant events to jump to.

To go from the documentation to a graph to a representation that can provide control-feasibility information during static analysis, we need the following:

(1) A well-defined *semantics* for Android lifecycle graphs. Our analysis can then use these graphs to filter out irrelevant transitions based on the control-feasibility condition of relevance soundness (Condition 1(b)).

(2) A specialization of generic lifecycle graphs of core Android components (e.g., Activity, Service) to a lifecycle graph for a specific *application subclass* of that component. This specialization resolves Java method overriding to make explicit the method code for each *application subclass*, and for precision, it incorporates other callbacks, such as those for handling user interface widgets.

(3) A way for the analysis to resolve lifecycle events on *object instances*. Since events in Android are methods on lifecycle objects, we need to prove that object instance $\widehat{o_1}$ *must-aliases* $\widehat{o_2}$ for two events $\widehat{o_1}.m_1$ and $\widehat{o_2}.m_2$ in order to show that $\widehat{o_1}$ and $\widehat{o_2}$ are constrained by the same lifecycle graph. If we cannot prove this fact, it is unsound to do any control-feasibility filtering because $\widehat{o_1}$ and $\widehat{o_2}$ could be different instances of the same lifecycle class (and therefore would have independent lifecycles).

Once a lifecycle graph has been specialized for a particular subclass and its events have been resolved to a particular object instance, it can be used directly by the analysis to perform control-feasibility filtering.

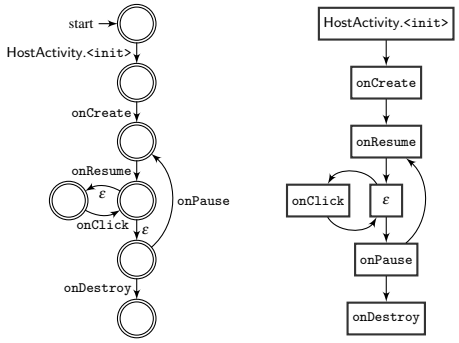
Giving Semantics to Android Lifecycle Graphs Consider the lifecycle graphs in Figure 3. These graphs specify the sequence of possible event traces for a particular Android lifecycle component (though the Android documentation never explicitly explains their meaning). If we think of the nodes of a lifecycle graph as labels for their outgoing edges,

[3] `http://developer.android.com/guide/components/activities.html#Lifecycle`

$$\begin{array}{ll} \text{lifecycle graphs} & G ::= \{\ldots, e_1 \to e_2, \ldots\} \\ \text{events} & e ::= C.m \mid \widehat{o}.m \mid \varepsilon \\ \text{classes} \quad C \qquad \text{methods} \quad m & \end{array}$$

Figure 12: Components of lifecycle graphs.

we can interpret a lifecycle graph as a nondeterministic finite automata (NFA) that accepts the language of all feasible concrete event traces for its lifecycle component. In order to account for partial traces (e.g., traces ending in an exception that interrupts the lifecycle), every node must be an accepting state. For example, the lifecycle graph for the `HostActivity` class from Figure 3 (reproduced below right for convenience) corresponds to the following NFA:



In order to connect the meaning of a lifecycle graph G to our model of concrete program execution, let us consider labeling an NFA edge not with the name of its corresponding event e, but with the entry transition of the event method, which we write as $\text{entry}(e)$. This means that the strings accepted by the lifecycle NFA (which we write as $\ulcorner G \urcorner$ for a lifecycle graph G) are strings of transitions t (i.e., traces T) rather than strings of events e. We can now state a soundness condition for lifecycle graphs.

Condition 2 (Lifecycle graph soundness)**.** *If concrete execution can reach event $e \in G$, the lifecycle graph G accepts the concrete trace projected onto the transitions of the lifecycle graph. More formally, if $\langle \sigma, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T \,\hat{}\, t]{}{}^{*} \langle \sigma', \ell' \rangle$ and event $e \in G$ where $t = \text{entry}(e)$, then $\ulcorner G \urcorner$ accepts $\text{events}(T \,\hat{}\, t, G)$.*

The function $\text{events}(T, G)$ simply projects a concrete trace T onto the transitions of a lifecycle graph G:

$$\text{events}(T, G) \stackrel{\text{def}}{=} \begin{cases} t \,\hat{}\, \text{events}(T_1) & \text{if } T = t \,\hat{}\, T_1 \text{ and } \exists\, e \in G.\ \text{entry}(e) = t \\ \text{events}(T_1) & \text{if } T = t \,\hat{}\, T_1 \text{ and } \nexists\, e \in G.\ \text{entry}(e) = t \\ [] & \text{if } T = [] \end{cases}$$

We assume that the lifecycle graphs specified in the Android documentation are sound.

Static Lifecycle Graphs: Specializing Lifecycle Graphs to Application Classes Android applications hook into the framework by subclassing special Android core components like Activity. Thus far, we have discussed events rather

abstractly, but events in Android correspond to methods on Java objects. We make this explicit by considering events as pairs of the method and the class in which it is defined (i.e., $C.m$) or as pairs of the method and the receiver object on which it is invoked (i.e., $\widehat{o}.m$) as shown in Figure 12. A well-formed lifecycle graph can consist of class-method events or object-method events; we call the former version a *static lifecycle graph* and the latter a *dynamic lifecycle graph*. We will explain the special event ε shortly.

The Android lifecycle documentation specifies the ordering of methods for core components, but we would like static lifecycle graphs specialized for the classes in the application under analysis. The specialization of lifecycle methods is straightforward by following the method resolution semantics of Java given a class hierarchy. Suppose we wish to specialize a general lifecycle graph G describing an Android core component C_{core} for an application subclass C_{app} (i.e., $C_{\text{app}} <: C_{\text{core}}$). For each event method node $C_{\text{core}}.m$ in G, we replace the node with $C.m$ where C is the class from which C_{app} inherits method m (e.g., $C = C_{\text{app}}$ if C_{app} overrides m).

An application class can also register custom callback methods that are triggered by external events such as user interaction. For example, the HostActivity class from Figure 2 extends the `OnClickListener` interface, overrides the `onClick` method, and registers itself as the listener for `onClick` events by calling `setOnClickListener(`**this**`)` at line 5. For soundness, we need to account for all such callback methods, which we could do simply by treating them as independent lifecycle components that have no ordering constraints. However, for precision it is important for the analysis to associate these callback methods with the appropriate component. The analysis should also understand that these user-triggered events can only occur during the "active" phase of the registering lifecycle component when the user can interact with the component. For Activity components, this active phase is the interval between `onResume` and `onStop`.

We incorporate custom callback events into the lifecycle graph with a simple flow-insensitive analysis. For an application class C_{app}, we consider its reachable methods in the call graph to determine what custom callbacks it may register. To represent the active phase of a class $C <:$ Activity, we introduce an ε event between `onResume` and `onClick` events as we saw in Figure 3. An ε event is a no-op event that translates to an ε-transition in the NFA formulation.

Once we have identified the set of callbacks $\{e_{\text{cb}}, \ldots\}$ that can execute during the active phase of the registering component, we "attach" each custom callback event e_{cb} to the active phase with edges $\varepsilon \to e_{\text{cb}}$ and $e_{\text{cb}} \to \varepsilon$. This models the fact that the user may or may not trigger an interaction event and that interaction events can be triggered an arbitrary number of times. This analysis is flow-insensitive because we do not consider the program point where registering methods like `setOnClickListener` are called. We also do not consider orderings between core lifecycle components (e.g., modeling the launching order of Activity's). Incorporating this information via techniques like those presented in Yang et al. [32] could improve the precision of our static lifecycle graphs.

Callback-registering methods like `setOnClickListener` may register any object with the appropriate method defined as the callback object (not just the **this** object). A common pattern is to use anonymous inner classes to implement these callbacks, as the anonymous ServiceConnection object created at line 3 of Figure 2 does. As a consequence, a lifecycle graph may need to contain methods invoked on multiple object instances (e.g., the **this** object and the anonymous inner class object). We consider this issue next.

Dynamic Lifecycle Graphs: Resolving Lifecycle Events on Object Instances A significant challenge in leveraging lifecycle information in a flow/path-sensitive analysis is to soundly account for the fact that the lifecycle applies to object instances at run time. Our approach is to resolve static lifecycle graphs to object instances in order to create a dynamic lifecycle graphs that can be directly used by the analysis. We perform this resolution on-the-fly during analysis.

To describe this approach more concretely, suppose our abstract memory is $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M$ for some memory M while currently analyzing code in some event method $C.m_2$; that is, we are in the lifecycle event $\widehat{o}_1.m_2$ in the corresponding dynamic lifecycle graph with some facts about objects $\widehat{o}_1$ and $\widehat{o}_2$. We would like to leverage an event-ordering constraint $C.m_1 \to C.m_2$ in the static lifecycle graph for C, but for soundness, we have to consider both $\widehat{o}_1.m_1$ and $\widehat{o}_2.m_1$ as possible events.

Our analysis handles this problem by performing an eager case split on aliasing (if we have no existing aliasing information on $\widehat{o}_1$ and $\widehat{o}_2$). That is, just before considering the event-ordering constraint, we split the abstract state into an aliased case $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M \wedge \widehat{o}_1 = \widehat{o}_2$ and a disaliased case $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M \wedge \widehat{o}_1 \neq \widehat{o}_2$. The aliased case gives us the must-alias fact that we need to soundly leverage the event-ordering constraint for control-feasibility filtering.

The eager case split means that we have a separate proof obligation for the disaliased case where we cannot use the event-ordering constraint in the static lifecycle graph. However, as we will see in more detail in Section 6, applying data-relevance often allows us to quickly rule out this case. In the common case that the relevant commands in $C.m_1$ and $C.m_2$ are writes to **this** of the lifecycle object, we can use data-relevance to rule out $\widehat{o}_2.m_1$ (in the disaliased $\widehat{o}_1 \neq \widehat{o}_2$ case). Even if the relevant writes are through non-**this** pointers (e.g., `p.f = ⋯`), our precise reasoning about aliasing and strong updates typically handles this disaliased case quickly.

6. A Jumping Analysis for the Heap and Events

In this section, we bridge the gap between theory and practice by combining the jumping framework from Section 3, the data-relevance relation for the heap from Section 4, and the formalization of Android lifecycle graphs in Section 5 to design HOPPER, a practical jumping analysis for analyzing event-driven Android programs. We achieve this by devising a sound relevance relation for THRESHER [7], a precise backward analysis that tightly integrates the results of an up-front points-to analysis to refute separation logic queries.

We have given the intuition for the jumping strategy that our relevance relation implements in Sections 1 and 2: when the analysis reaches an event boundary, it identifies events that contain data-relevant commands, filters the set of data-relevant events using control-feasibility constraints based on Android lifecycle information, then jumps to the remaining events. To realize this vision, we need to address one remaining issue: using the semantics of lifecycle graphs to perform control-feasibility filtering. We explain how we solve this issue in Section 6.1 before presenting an algorithm for computing relevant transitions that utilizes our solution in Section 6.2.

6.1 Using Lifecycle Graphs for Control-Feasibility Filtering

To utilize Android lifecycle information in our relevance relation, we must connect the meaning of lifecycle graphs from Section 5 (Condition 2) to the control-feasibility condition of relevance soundness (Condition 1(b)). To maintain precision while jumping from one event to another, we must ensure that we only perform jumps that respect the ordering constraints encoded in Android lifecycle graphs (while simultaneously considering the necessary interleavings in order to be sound). Our solution is to utilize the reachability and *post-dominance* information encoded in the lifecycle graph. Since our analysis is backward, only jumps from the current program point to a preceding transition in a concrete execution trace ending at the current program point are control-feasible. Thus, we can filter a set of possibly relevant transitions using control-feasibility by considering backward reachability in the lifecycle graph.

If some event e is not backward-reachable from the current event e_{cur}, then we know that no concrete trace ending in e_{cur} can possibly have visited e first (following the semantics of lifecycle graphs as concrete trace-accepting NFAs in Condition 2). Thus, we can prune all events that are not backward-reachable from e_{cur} in the lifecycle graph G to produce a pruned lifecycle graph G' where e_{cur} is a leaf node.

For example, we can use this technique to reason that if the analysis is currently in the `onClick` event of Figure 3, the `onDestroy` event has not yet occurred in the current lifecycle. If we prune nodes and edges not backward reachable from the current node `onClick`, we can prune the `onDestroy` event. We do not need to consider jumps from e_{cur} to pruned events.

The analysis can further refine the possible jump targets using postdominance on the pruned graph G'. Consider the postdominance tree rooted at e_{cur}; that is, a tree where each node is the immediate postdominator of its children. For any set of potentially relevant events E, we only need to consider the smallest set $E' \subseteq E$ such that E' postdominates E. As a consequence, for all $e \in E$, there is an $e' \in E'$ such that e' is between e_{cur} and e in the postdominator tree rooted at e_{cur}. The correctness of this reasoning follows directly from the meaning of lifecycle graphs and the definition of postdominance: if e_{cur} postdominates e' and e' postdominates e, we can conclude that every trace accepted by the lifecycle NFA that visits e_{cur} always visits e' beforehand without visiting e in between.

To give a more concrete example using the HostActivity lifecycle graph in Figure 3, we would like to able to determine that if the analysis is currently in the `onClick` event and we know that only the `onCreate` and HostActivity.`<init>` events are relevant, then we only need to jump to `onCreate`. We can derive this fact by demonstrating that `onClick` postdominates `onCreate` and `onCreate` postdominates HostActivity.`<init>` in G'.

6.2 An Effective Relevance Relation for Android

Finally, we present our algorithm for computing Android-specialized relevance information (Figure 13) and argue that our algorithm is sound with respect to relevance soundness (Condition 1). The algorithm implements the $\langle R, \ell \rangle \rightsquigarrow T_{\text{rel}}$ judgment form for relevance relations and is executed each time the A-JUMP rule is applied.

In the usual case where the current program label ℓ_{cur} is not the entry label of an event, the algorithm behaves like a standard path-sensitive backward analysis by choosing to visit the predecessor labels of the current program label next (lines 2–4). Clearly, this satisfies relevance soundness by satisfying the control-feasibility condition (Condition 1(b)).

In the case that the current program label is the entry label of an event, we perform jumps to a computed set of relevant transitions using the data-relevance and control-feasibility constraints described previously. Specifically, the algorithm computes the set of data-relevant events that may write to the current abstract state (lines 5–20) and then filters this set of events using control-feasibility information from the lifecycle graph of the current event (lines 22-37).

First, the algorithm computes the set of data-relevant transitions T_{rel} for R_{cur} using the points-to analysis, as we have explained in Section 4.2. In principle, the algorithm could return the set T_{rel} and still be sound by satisfying relevance soundness Condition 1(a), but this would be imprecise because it would not take the ordering of events in the lifecycle graph into account. The algorithm thus walks backward from the calling method of each relevant transition t_{rel} (given by method(t_{rel})) in the call graph until it reaches an event

Require: Current abstract state R_{cur}
Require: Current label ℓ_{cur}
Require: Program transition relation P
Require: Call graph CG
Ensure: Returned transition set T_{rel} satisfies Condition 1
1: // not at event entry, follow predecessors
2: **if** ℓ_{cur} is not the entry label of an event **then**
3: **return** preds(ℓ_{cur}, P)
4: **end if**
5: // at event entry, get data-relevant events
6: $T_{\text{rel}} \leftarrow$ dataRel(R_{cur}) // compute $R_{\text{cur}} \rightsquigarrow T_{\text{rel}}$
7: $E_{\text{rel}} \leftarrow \emptyset$ // events leading to a relevant transition
8: **for all** $t_{\text{rel}} \in T_{\text{rel}}$ **do**
9: $W \leftarrow$ [method(t_{rel})] // method worklist
10: $V \leftarrow \emptyset$ // track visited methods to handle CG cycles
11: **while** $W \neq \emptyset$ **do**
12: Remove m from W
13: **if** m is event **then**
14: $E_{\text{rel}} \leftarrow \{ m \} \cup E_{\text{rel}}$
15: **else if** $m \notin V$ **then**
16: Add preds(m, CG) to W
17: **end if**
18: $V \leftarrow \{ m \} \cup V$
19: **end while**
20: **end for**
21: // filter data-relevant events with a lifecycle graph
22: $e_{\text{cur}} \leftarrow$ event(ℓ_{cur})
23: $G \leftarrow$ specializeLifecyleGraph(class(e_{cur}), CG)
24: $E_{\text{inG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \in G \}$
25: $E_{\text{notinG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \notin G \}$
26: $E_{\text{feas}} \leftarrow \emptyset$ // data-relevant/control-feasible events in G
27: $W \leftarrow$ [e_{cur}] // lifecycle graph event worklist
28: $V \leftarrow \emptyset$ // track visited events to handle cycles in G
29: **while** $W \neq \emptyset$ **do**
30: Remove e from W
31: **if** $e \in E_{\text{inG}}$ **then**
32: $E_{\text{feas}} \leftarrow \{ e \} \cup E_{\text{feas}}$
33: **else if** $e \notin V$ **then**
34: Add preds(e, G) to W
35: **end if**
36: $V \leftarrow \{ e \} \cup V$
37: **end while**
38: **return** exitTrans($E_{\text{feas}} \cup E_{\text{notinG}}$)

Figure 13: An algorithm for selecting relevant transitions to visit in event-driven, heap-manipulating Android programs.

on each path (loop from lines 8–20). The resulting set of data-relevant events E_{rel} contains the set of all events whose execution might lead to a relevant transition. Returning the exit transition of each of these events E_{rel} would also satisfy relevance soundness via a combination of Condition 1(b) and (a) because by construction of E_{rel}, these exit transitions collectively postdominate all relevant transitions.

However, we can gain additional precision by removing events from T_{rel} based on control-feasibility information from the lifecycle graph, which is what the algorithm does next. Lines 22 and 23 compute a lifecycle graph specialized for the class of the current event e_{cur}, as we have described in Section 5. The algorithm then partitions the set of relevant events E_{rel} based on their presence in the lifecycle graph (lines 24–25). It does this because only the events E_{inG} that are in the lifecycle graph should be filtered in the subsequent step—the events in E_{notinG} are unordered with respect to e_{cur} and the algorithm must return all of them for soundness.

The loop from lines 29–37 performs control-feasibility filtering on nodes in the lifecycle graph. This loop computes a subset E_{feas} of E_{inG} that must be returned for soundness. The loop walks backward from the current event e_{cur} in the lifecycle graph G and stops each time it reaches a relevant event. The construction of E_{feas} ensures that at the end of the loop, relevant events that are not backward reachable from e_{cur} will be excluded from the set E_{feas}, and as will events postdominated by both e_{cur} and some other relevant event. We have argued for the soundness of excluding events based on backward reachability and postdominance in the lifecycle graph in Section 6.1.

Finally, the algorithm takes the union of the lifecycle graph control-feasible relevant events E_{feas} and the unordered relevant events E_{notinG} and returns their exit transitions as the set of transitions that must be visited (line 38). The set $E_{\text{feas}} \cup E_{\text{notinG}}$ is a subset of the set E_{rel} whose exit transitions already satisfy relevance soundness. We have only removed events from this set by soundly filtering based on lifecycle control-feasibility information, so returning the exit transitions of $E_{\text{feas}} \cup E_{\text{notinG}}$ also satisfies relevance soundness.

7. Empirical Evaluation

In order to evaluate the effectiveness of jumping analysis, we sought to test the following experimental hypothesis:

> *Jumping is a scalable approach to flow/path-sensitive inter-event analysis.*

We hypothesize that augmenting a state-of-the-art path-sensitive analysis with jumping increases precision by allowing the analysis to reason about event orderings, yet limits the number of event orderings that must be considered enough to make analysis tractable.

Experimental Setup In order to test our hypotheses, we chose to evaluate jumping analysis by attempting to prove the absence of null dereferences in event-driven Android programs. We chose this client because null dereferences are a common problem in real-world Android apps, and the event-driven nature of Android makes precisely verifying the absence of null dereferences a significant challenge for analyses (see Section 2.1 for a more detailed discussion of this client). We implemented the practical jumping analysis described in Section 6 in the HOPPER [1] tool, a variant of the THRESHER [7] tool that builds on the WALA [3] analysis infrastructure and the Z3 [14] SMT-solver. HOPPER extends THRESHER by adding the ability to perform jumps, but the tools are otherwise identical.

The core of THRESHER is an engine for refuting queries written in separation logic. Clients are implemented as lightweight add-ons that take a program as input and emit separation logic queries for the core refuter to process. We extended THRESHER/HOPPER with a new client for checking null dereferences. The client leverages `@Nonnull` annotations inferred by the NIT [2] tool to eliminate easy cases where non-nullness of fields, function return values, or function parameters is a flow-insensitive property. For each non-static field read/write `x.f` or function call `x.m()` in the the program, the client emits the necessary bug precondition $x \mapsto \mathsf{null}$ as a query to refute in order to prove dereference safety.

We gave THRESHER and HOPPER a maximum budget of 10 seconds to refute each query. This budget includes all aspects of analysis time except for the initial call graph construction (including HOPPER-specific aspects such as computing data-relevance, specialization of lifecycle graphs, etc.). We chose this budget through trial and error—we found that larger budgets did not allow either tool to find appreciably more refutations, whereas smaller budgets caused too many timeouts for both tools. In the case that a tool cannot refute a query within the budget, a timeout was declared and the dereference was reported as a potential bug. We ran all experiments in single-threaded configuration on a Mac desktop machine running Mac OS 10.10.2 with 64GB of RAM and 3.5GHz Intel Xeon processors.

Android applications can make use of concurrency—events execute atomically if they run on the same thread, but the execution of events can interleave if the events execute on separate threads. In addition, app developers can use Java threads for multithreaded execution in the usual way. THRESHER and HOPPER do not soundly account for either of these features, as both tools assumes that all events execute atomically on a single thread. Both tools also do not soundly handle reflection and native code for which we do not have handwritten stubs—these constructs are treated as no-ops.

Representing Android Event Dispatch Instead of analyzing a synthetic harness that models the event dispatch performed by the Android framework, THRESHER and HOPPER analyze the actual logic for event dispatch in the Android framework source code. To allow this, we pre-process each app we analyzed with DROIDEL [8], a tool that explicates key reflective calls in the Android framework by replacing them with calls to automatically generated, application-specific stub methods. We then use the `ActivityThread.main` method of the framework as a single entrypoint for call graph construction. There are several advantages to analyzing the actual event dispatch code instead of using a harness: (1) we do not have to worry about soundly and precisely modeling the execution context of events, which can be a significant challenge, and (2) generating a harness that precisely represents all ordering constraints is impractical, as we have already argued in Section 1.

7.1 Proving Dereferences Safe with Jumping

We ran both THRESHER and HOPPER on the corpus of ten open-source Android apps shown in Figure 14. The apps range in size from 3K source lines of code to 57K source lines of code. Since the primary challenge of analyzing these apps comes from considering interleavings of their component lifecycles, we also report the number of core lifecycle components (i.e., Application, Activity, Fragment, Service, and ContentProvider subclasses) and the total number of events in each app. Our analysis must consider the possibility of interleavings between events of different components for soundness, but must preserve the ordering of events within the lifecycle of a single component for precision. Recall from Section 1 that the size of a reified harness that considers the interleavings between just a single instance per component is exponential in the number of components.

The "Unsafe Derefs" columns of Figure 14 summarize the results of proving null dereference safety on our benchmark apps with NIT, THRESHER and HOPPER. Each column reports the number of unproven dereferences after running the tool (where 0 represents proving all dereferences safe, so lower is better). The results show that although about a third of the dereferences can be proven safe using the flow-insensitive analysis of NIT, the path-, flow-, and context-sensitive THRESHER analysis was significantly more precise (providing evidence that precision beyond flow-insensitivity is necessary for proving dereference safety in Android apps). The **Hop Impr** column gives the percentage reduction in unsafe dereferences achieved by running HOPPER (where 100% represents proving all remaining dereferences safe, so higher is better). HOPPER substantially improved on the already-significant precision of THRESHER—on average, HOPPER reduced the number of dereferences unproven by THRESHER by more than half.

The difference between HOPPER and THRESHER is that the jumping capability of HOPPER enables precise inter-event analysis, as we predicted in our experimental hypothesis. We noticed that when THRESHER reaches an event boundary without proving safety, it continues precise backward analysis of the event dispatch code of the Android framework and (almost always) times out without finding a proof. By contrast, HOPPER jumps from an event boundary to a (typically) small set of relevant events and is frequently able to prove safety based on precise and tractable inter-event reasoning.

The final **Total Hop Proven** column shows that for every benchmark, HOPPER proved at least 90% of the dereferences safe (92% safe on average). We note that previous state-of-the-art work in null dereference checking for ordinary, non-event-driven Java programs (e.g., [22–25]) reports proving 84-91% of dereferences safe on average. Achieving similar precision results in the presence of the formidable scalability challenges introduced by an event-driven setting is a significant advance.

Benchmark Size				Unsafe Derefs				HOPPER Effectiveness		
Bench	**KLOC**	**Com**	**Evt**	**Deref**	**Nit**	**Thr**	**Hop (Impr %)**	**Proven (%)**	**Time (s)**	**Time/deref (s)**
DrupalEditor	3	10	127	790	574	157	66 (58)	92	717	0.9
NPR	5	14	120	829	617	181	50 (72)	94	691	0.8
Last.fm♠	11	12	174	4840	3528	950	474 (50)	90	5922	1.2
DuckDuckGo	13	34	272	1901	1277	514	144 (72)	92	1751	0.9
GitHub	19	70	572	3603	2520	583	284 (51)	92	3749	1.0
SeriesGuide♠	32	80	871	8184	5438	987	625 (37)	92	13929	1.7
ConnectBot♠	33	13	201	2190	1562	316	75 (76)	97	880	0.4
TextSecure	38	63	588	5921	3643	621	272 (56)	95	2306	0.4
K-9Mail	55	52	750	19032	11968	3067	1988 (35)	90	27197	1.2
WordPress♠	57	98	1325	15000	9735	2402	1341 (44)	91	18596	0.9
Total	266	446	5000	62290	40862	9778	5319 (53)	92	75738	0.9

Figure 14: Proving dereference safety in event-driven Android apps with HOPPER. The "Benchmark Size" column grouping gives the number of (thousands of) lines of application source code (**KLOC**), lifecycle components (**Com**), and events (**Evt**) for each benchmark. The "Unsafe Derefs" column grouping lists the number of possibly-unsafe dereferences in each app before analysis (**Deref**) followed by the number remaining after running NIT (**Nit**), THRESHER (**Thr**), and HOPPER (**Hop**). The HOPPER column also lists the percentage reduction in unproven derefs of HOPPER over THRESHER (**Impr %**). The final column grouping gives the percentage of derefs proven safe by HOPPER (**Proven (%)**), the time taken to process all derefs (**Time (s)**), and the time taken per deref (**Time/deref (s)**). The "Total" row gives the sum of all numeric rows and the geometric mean of the **(Impr %)**, **Proven (%)**, and **Time/deref** rows. ♠'s indicate benchmarks where our partial manual triaging revealed a true bug.

7.2 Manual Triaging of Alarms

To understand why HOPPER sometimes fails to prove safety, we manually triaged a sample of 20 unproven dereferences from each of our 10 benchmark applications (a total of 200 alarms). We classified the unproven dereferences into three categories (a) true bugs, (b) scalability issues, or (c) precision issues. We placed a dereference into the true bugs category if we found a concretely feasible sequence of events would lead the application to throw a NullPointerException. We classified a dereference as a scalability issue if we determined that HOPPER possessed the necessary precision to prove the dereference safe, but was not able to do so within the 10 second budget. Finally, we labeled a dereference as a precision issue if HOPPER did not have the precision required to prove the query correct. This category includes both analysis imprecision (e.g., loop invariant inference, container abstraction) as well as modeling imprecision (e.g., Android UI models, Android/Java reflection and native code).

The results from our manual triaging are shown inset. In the 200 alarms we examined, most dereferences that cannot be proven safe are due to precision issues (172). Of these 172 alarms, 132 would require more precise modeling of the Android framework and 41 are due to more fundamental analysis imprecision. Many of Android modeling issues are additional constraints on the interaction between different lifecycle components that we do not account for. For example, proving safety of some dereferences required understanding details such as the order in which Activity's launch each other or the fact that a callback on a Button cannot be invoked if the `visible` attribute of the Button is set to false. Handling all of the corner cases of the complex Android framework is a challenging task that we leave to future work.

(a) Bug	(b) Scalability	(c) Precision
11	17	172

Nearly all of the the analysis imprecision issues stem from imprecise abstraction of containers and strings. Both of these precision problems are orthogonal to HOPPER's approach to analysis of event-driven programs and could in principle be addressed by enhancing HOPPER with better abstractions or solvers (e.g., [16] for containers and [20] for strings).

The fact that only 17 of the 200 unproven dereferences we examined could not be proven safe due to scalability issues strengthens our conviction that jumping is an effective approach for tractable analysis of event-driven systems. Though HOPPER is not perfect, it proves an impressive 92% of the dereferences it encounters. The vast majority of proof failures are due to our incomplete modeling of Android rather than scalability issues.

Bugs Found We found eleven bugs in four different apps: Last.fm (1), Seriesguide (5), ConnectBot (4) and WordPress (1). The bug in WordPress had already been eliminated by the developers, though in an indirect way (replacing the functionality in the buggy class with an entirely new class). We sent pull requests fixing the bugs in each of the remaining projects. The developer of SeriesGuide and ConnectBot accepted all of our pull requests. The developers of Last.fm have not yet responded to our pull requests. This project is updated infrequently and has a backlog of pending pull requests.

Of the eleven bugs that we found, five of them involved misunderstanding or misusing the Android lifecycle in some way. This strengthens our belief that the lifecycle is a source of confusion for developers that would be well-served by better analysis tools. We further note that four of the five bugs involved interactions between the lifecycles of different components. These bugs could not be found by an unsound approach that models the lifecycle of each component, but does not consider interleaving lifecycles of different components.

Other Instantiations of the Framework In addition to using jumping for effective analysis of event-driven programs, we have also considered an alternate instantiation of the jumping framework for analysis of ordinary (that is, non-event-driven) Java programs. This instantiation relies on data-relevance information alone and chooses to jump only when the safety of the query may rely on a flow-insensitive invariant established earlier in the program. We applied this instantiation to the problem of proving downcast safety in the DaCapo2006 [5] benchmarks and observed a similar precision improvement of HOPPER over THRESHER. See Blackshear [6, Chapter 6] for more details on this instantiation of the framework.

8. Related Work

Static Analysis of Android Applications Numerous techniques have considered static analysis of Android apps, but to the best of our knowledge, few have tackled the problem that we address in this paper: soundly modeling the interleaving of different component's lifecycles. The harness method generated by the state-of-the-art FLOWDROID [18] tool soundly models the sequential execution of component lifecycles, but not their interleaving. This unsound modeling avoids the cost of computing a product graph as described in Section 1, but will miss bugs like the one explained in Section 2 along with the five lifecycle-sensitive bugs we found in Section 7.

ANADROID [21] is the only tool we are aware of that explicitly claims to handle interleavings between lifecycles of different components. Their *entry point saturation* technique efficiently computes a fixed point over all possible event ordering. However, this computation does not take intra-lifecycle event orderings into account and thus will lose precision. We found this kind of precision to be crucially important in Section 7—HOPPER's improvement over THRESHER comes entirely from more precise and scalable inter-event reasoning.

Analysis of Asynchronous and Event-Driven Programs Identifying a small set of commands relevant to the query and their corresponding events using data-relevance exploits the fact that the data dependencies of a program are often less complex than its control dependencies in practice. Recent techniques for concurrent program verification [17] and bug finding [10] have used a similar insight: an effective way to prevent the complexity of a concurrent program analysis (static or dynamic) from growing exponentially in the number of threads is to design the analysis around tracking data dependencies rather than control dependencies. Jumping based on a relevance relation allows the analysis to exploit both data-relevance and control-feasibility information to improve scalability, and jumping can be applied in sequential, concurrent, and event-driven settings.

Jhala et al. show that the IFDS framework can be extended to enable analysis of event-driven programs and present a goal-directed algorithm for proving safety properties in their extended framework [19]. Their focus is on handling unordered events whose execution may interleave, whereas we focus on the problem of preserving the ordering between lifecycle events whose execution is atomic.

Program Slicing Identifying commands that may affect a query using a data-relevance relation is closely related to program slicing [31]. Our approach is most closely related to semantic slicing [9, 28], since we perform a slice with respect to an abstract state rather than a seed command. A key difference between our data-relevance relation and semantic slicing is that we only compute the first step of a slice (i.e., the *immediately* relevant commands) rather than computing a transitive closure of relevant commands as a complete slice does. In many cases, taking a complete slice includes the majority of the program and is prohibitively expensive to compute. Our approach is much more efficient than the obvious approach of taking a full slice with respect to the query and analyzing the sliced program.

9. Future Work

In future work, we plan to demonstrate the generality of our framework for jumping analyses by instantiating it with new data-relevance/control-feasibility relations and applying it to different problem domains. A logical next step would be trying to adapt jumping analysis to the analysis of concurrent programs (both event-driven and threaded). For example, we could combine the data-relevance relation from Section 4 with a control-feasibility filter that leverages information about **synchronized** blocks and graphs of thread spawning structure (in a manner similar to our utilization of Android lifecycle graphs in Section 6). We are hopeful that such a strategy would greatly enhance the scalability of the analysis by decreasing the number of thread interleavings that must be considered while jumping, just as the analysis described in this paper significantly reduces the number of event orderings that the analysis needs to consider.

10. Conclusion

We have presented *jumping*, a general framework for selective control-flow abstraction that can be applied to any goal-directed backward abstract interpretation. The key idea behind jumping is an intertwining of data-relevance and control-feasibility reasoning that enables an analysis to soundly jump

to a small set of relevant program transitions. We instantiated our jumping framework with HOPPER, a tool for performing precise and scalable reasoning of event-driven Android apps by leveraging heap dependencies for data-relevance and Android lifecycle graphs for control-feasibility. HOPPER proved the safety of 90–97% of dereferences in the apps it analyzed and allowed us to isolate eleven real bugs.

Acknowledgments

We thank Pavol Černý and the University of Colorado Programming Languages and Verification Group (CUPLV) for insightful discussions, as well as the anonymous reviewers for their helpful comments. This material is based on research sponsored in part by DARPA under agreement numbers FA8750-14-2-0039 and FA8750-14-2-0263, the National Science Foundation under CAREER grant number CCF-1055066, and a Facebook Graduate Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Hopper. `https://github.com/cuplv/hopper`, 2015.

[2] Nit: Nullability inference tool. `http://nit.gforge.inria.fr/`, 2015.

[3] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`, 2015.

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2005.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[6] Sam Blackshear. *Flexible Goal-Directed Abstraction*. PhD thesis, University of Colorado Boulder, 2015.

[7] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Programming Language Design and Implementation (PLDI)*, 2013.

[8] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to Android framework modeling. In *State of the Art in Program Analysis (SOAP)*, 2015.

[9] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Language Design and Implementation (PLDI)*, 1993.

[10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[11] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods (NFM)*, 2015.

[12] Devin Coughlin and Bor-Yuh Evan Chang. Fissile type analysis: Modular checking of almost everywhere invariants. In *Principles of Programming Languages (POPL)*, 2014.

[13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Programming Language Design and Implementation (PLDI)*, 2008.

[16] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Principles of Programming Languages (POPL)*, 2011.

[17] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *Principles of Programming Languages (POPL)*, 2013.

[18] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[19] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2007.

[20] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4), 2012.

[21] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS)*, 2013.

[22] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Gowri Nanda. Verifying dereference safety via expanding-scope analysis. In *Software Testing and Analysis (ISSTA)*, 2008.

[23] Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest preconditions analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[24] Amogh Margoor and Raghavan Komondoor. Two techniques to improve the precision of a demand-driven null-dereference verification approach. *Sci. Comput. Program.*, 98, 2015.

[25] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. In *International Conference on Software Engineering (ICSE)*, 2009.

[26] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[27] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.

[28] Xavier Rival. Understanding the origin of alarms in Astrée. In *Static Analysis (SAS)*, 2005.

[29] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[30] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Programming Language Design and Implementation (PLDI)*, 2006.

[31] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[32] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[33] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.