Mole: Efficient Crash Reproduction in Android Applications With Enforcing Necessary UI Events

Maryam Masoudian, Heqing Huang, Morteza Amini, and Charles Zhang

***Abstract*—To improve the quality of Android apps, developers use automated debugging and testing solutions to determine whether the previously found crashes are reproducible. However, existing GUI fuzzing solutions for Android apps struggle to reproduce crashes efficiently based solely on a crash stack trace. This trace provides the location in the app where the crash occurs. GUI fuzzing solutions currently in use rely on heuristics to generate UI events. Unfortunately, these events often do not align with the investigation of an app's UI event space to reach a specific location of code. Hence, they generate numerous events unrelated to the crash, leading to an event explosion. To address this issue, a precise static UI model of widgets and screens can greatly enhance the efficiency of a fuzzing tool in its search. Building such a model requires considering all possible combinations of event sequences on widgets since the execution order of events is not statically determined. However, this approach presents scalability challenges in complex apps with several widgets. In this paper, we propose a directed-based fuzzing solution to reduce an app's event domain to the necessary ones to trigger a crash. Our insight is that the dependencies between widgets in their visual presentation and attribute states provide valuable information in precisely identifying events that trigger a crash. We propose an attribute-sensitive reachability analysis (ASRA) to track dependent widgets in reachable paths to the crash point and distinguish between events in terms of their relevancy to be generated in the crash reproduction process. With instrumentation, we inject code to prune irrelevant events, reducing the event domain to search at run time. We used four famous fuzzing tools, Monkey, Ape, Stoat, and FastBot2, to assess the impact of our solution in decreasing the crash reproduction time and increasing the possibility of reproducing a crash. Our results show that the success ratio of reproducing a crash has increased for *one-fourth* of crashes. In addition, the average reproduction time of a crash becomes at least 2x faster. Wilcoxon Mann-Whitney test shows this enhancement is significant when our tool is used compared to baseline and insensitive reachability analysis.**

***Index Terms*—Android fuzzing, bug reproducing, directed fuzzing, widget dependencies, crash stack trace.**

Manuscript received 8 June 2023; revised 1 July 2024; accepted 9 July 2024. Date of publication 15 July 2024; date of current version 15 August 2024. This work was supported by under Grant PRP/004/21FX from Hong Kong Innovation and Technology Commission and grants from Huawei and TCL. Recommended for acceptance by A. Zaidman. *(Corresponding author: Maryam Masoudian.)*

Maryam Masoudian is with Hong Kong University of Science and Technology, Hong Kong, China, and also with Sharif University of Technology, Tehran, Iran (e-mail: mamt@connect.ust.hk).

Heqing Huang is with the Faculty of Computer Science Department, City University of Hong Kong, Hong Kong, China (e-mail: heqhuang@cityu.edu.hk).

Morteza Amini is with the Faculty of Department of Computer Engineering, Sharif University of Technology, Tehran, Iran (e-mail: amini@sharif.edu).

Charles Zhang is with the Faculty of Computer Science Department, Hong Kong University of Science and Technology, Hong Kong, China (e-mail: charlesz@cse.ust.hk).

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2024.3428543, provided by the authors.

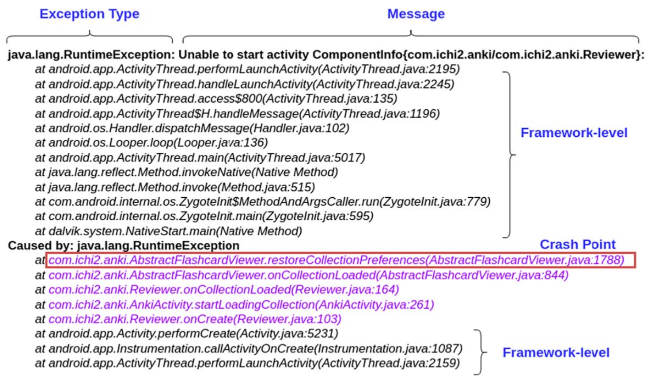
Digital Object Identifier 10.1109/TSE.2024.3428543

I. Introduction

ANDROID devices occupy over 80 percent of the mobile market share [1]. The quality of Android applications (apps) significantly influences user satisfaction, as studies [2], [3] show that over 80 percent of users stop using an app when it crashes. To prevent this from happening, developers must find and fix the root causes of the seen crashes. A crash stack trace (Fig. 1) is a common resource reported in issue tracking systems like GitHub [4] when a crash happens. It contains the crash type, the crash point, and a partial call stack to the crash point. Developers often begin by manually debugging or testing the app to determine whether a crash is reproducible. However, the manual process of looking for a particular crash is tedious and time-consuming, like finding a needle in a haystack.

To make the process more efficient, developers opt for using automated approaches that automatically examine an app to speed up the crash reproduction process. One of the appealing solutions is using state-of-the-art fuzzing techniques. Because of the *event-driven* nature of Android, most of the existing fuzzing tools [5], [6], [7], [8], [9], [10] focus on generating events [11] on UI widgets, which are the most common input type in Android apps [12]. These solutions are categorized as either **coverage-based** or **directed fuzzing**. The **coverage-based** solutions [5], [6], [7], [13], [14], [15] aim to explore more parts of an app code to find more crashes [16], using random or coverage-based heuristics (e.g., code coverage, widget coverage, activity transition coverage). Unfortunately, using these heuristics to generate UI events is not ***aligned*** with efficiently reproducing a crash.

Consider the motivating example shown in Fig. 2, in which the app has more than 100 activities. Here, the transition between two activities is required for triggering the crash. Using a fuzzing tool that leverages activity transition coverage (e.g., FastBot2 [13]) makes the search process longer and more complicated. This leads to the generation of many unrelated events according to our experiments (see Section II-B), which we call **event explosion**. Consequently, the crash reproduction takes a longer time to trigger the target crash.


Exception Type
Message
Framework-level
Crash Point
Framework-level

```
java.lang.RuntimeException: Unable to start activity ComponentInfo{com.ichi2.anki/com.ichi2.anki.Reviewer}:
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2195)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2245)
    at android.app.ActivityThread.access$800(ActivityThread.java:135)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1196)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:136)
    at android.app.ActivityThread.main(ActivityThread.java:5017)
    at java.lang.reflect.Method.invokeNative(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:515)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:779)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:595)
    at dalvik.system.NativeStart.main(Native Method)
Caused by: java.lang.RuntimeException
    at com.ichi2.anki.AbstractFlashcardViewer.restoreCollectionPreferences(AbstractFlashcardViewer.java:1788)
    at com.ichi2.anki.AbstractFlashcardViewer.onCollectionLoaded(AbstractFlashcardViewer.java:844)
    at com.ichi2.anki.Reviewer.onCollectionLoaded(Reviewer.java:164)
    at com.ichi2.anki.AnkiActivity.startLoadingCollection(AnkiActivity.java:261)
    at com.ichi2.anki.Reviewer.onCreate(Reviewer.java:103)
    at android.app.Activity.performCreate(Activity.java:5231)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1087)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2159)
```

Fig. 1. Example crash stack trace for issue ID 5756 in AnkiDroid. Framework and application-level API calls are specified in black and purple color, while the crash point is highlighted with a red box.

Directed fuzzing, another category of fuzzing solutions, explores the app code to reach a specific part of it, called target point [16]. A target point can be a specific statement (e.g., a crash point) or functionality of an app (e.g., an API function). A directed fuzzing solution initially performs a reachability analysis to identify all the feasible paths to the target point. A path is called feasible if it starts from an app's entry point and reaches the target point. In previous directed fuzzing studies [17], [18] for C/C++ programs, the value of input variables that result in the execution of the feasible paths were computed. Subsequently, proper code was injected into the app to allow exclusively these input values, thereby preventing the execution of irrelevant paths toward the target point. In Android, this can be achieved by identifying the UI events as they are the main input. Therefore, it is essential to first find a precise model of UI widgets and their event interactions due to the event-driven nature of Android apps. However, the mentioned studies could not identify suitable events in Android apps because they did not model the apps' event-driven nature.

Similarly, one can statically attempt to construct a precise static UI model. Due to numerous events in Android apps, this may confront the scalability issue. To construct this model, the app's control flow graph is analyzed to find feasible paths to the crash point. This graph must include all possible execution orders of events at run time, but the user actually determines their order at run time. Although there are *a few numbers of* widgets in each screen, considering all possible event sequences can lead to path explosion. For instance, all possible event sequences in the 1^{st} screen are 5! in our example crash in Fig. 2. This number is much larger considering the combinations of the event sequences in all of the screens, although each has less than ten widgets[1]. Hence, existing studies like Gator [19], WTG[2] [20], CrashScope [21], and GoalExplorer [22] construct their UI models in a high-level granularity like an activity. They only identify the events responsible for inter-component transitions and presume all the other events inside an activity constitute the whole event domain to explore. In this case, the crash reproduction process can waste enormous efforts by examining irrelevant events and becoming still deficient, finally leading into *event-explosion*.

Our insight to solve this problem is that the visual and functional dependencies between widgets in the feasible paths to the crash point tell us the correct order of events at run time. In Android, the developers use widget attributes to enforce dependencies between them. They accomplish a specific functionality in an app. In the example crash scenario depicted in Fig. 2, there is a fragment for holding the list of filtered decks (screen 7). This fragment becomes visible when at least one filtered deck is created by clicking on 'Create filtered deck' (screen 2). Here, the `VISIBILITY` attribute of the fragment is used to impose the dependencies between the two widgets/views. The fragment is initially invisible (screen 1). It becomes visible once the menu item 'Create filtered deck' is clicked. By tracking widget attribute states statically from the crash point in a backward manner, it is possible to determine the dependent widgets and, hence, a set of widgets that are more likely involved in a crash occurrence.

Widget dependencies are preserved in the code's logic by using Android API functions to set or access the attribute value states. For example, `getVisibility` and `setVisibility` are used for accessing and setting the value state of the visibility attribute. To find dependent widgets in each activity, one should track both setting and accessing states. Since attributes are not explicitly defined or accessed through variables in the API functions, inference of their value state is challenging. However, there are a limited number of attributes for all the existing widget types in Android that are *set* or *accessed* through documented API functions. Thus, we have proposed a specification to store the API functions associated with each attribute. We also designed an *abstraction* called *configuration* that enables us to track the value state of the widget attributes in the reachability analysis. We perform a flow, context, and path-sensitive reachability analysis armed with the *sensitivity* to the value states of the widgets' attributes and name it **Attribute-Sensitive Reachability Analysis (ASRA)**[3].

Once the widgets and their attribute states are collected in the feasible paths to the crash point, we can use such dependencies to distinguish between the necessary and irrelevant events statically. An event is handled through callback functions in the app code. We determine the *necessity* of an event in reaching the target point if there is at least one association with the collected widget attributes in the callback handling it. If there is no association, we consider the event to be *irrelevant*. We use instrumentation to add `assert(False)` in the code to prune such events to prevent the fuzzing solution from exploring its code at runtime. Eventually, we will have a precise set of necessary widgets, and the events that explore them can reproduce the crash.

In summary, the following contributions are made:

- Providing empirical evaluation to demonstrate the necessity of solving the event explosion problem in crash reproduction.

[1]The number of widgets are shown above each screen in Fig. 2.
[2]Window Transition Graph (WTG)

[3]Our tool is available at: https://github.com/maryammsd/Mole

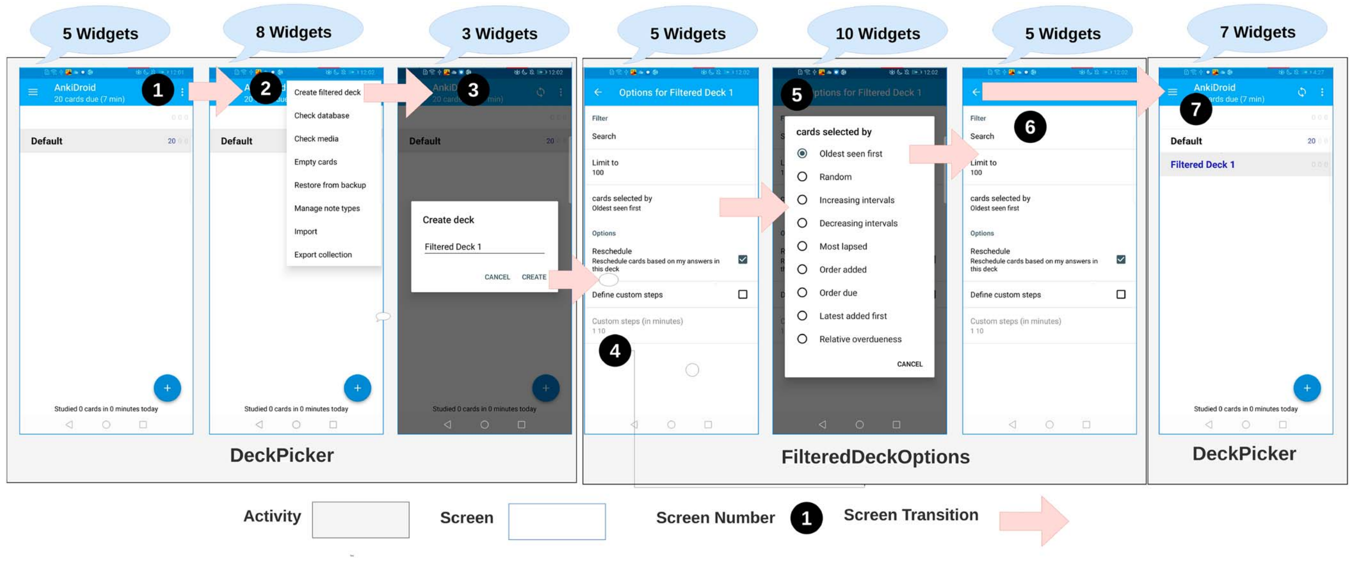


Fig. 2. The steps for reproducing the crash with issue ID 5756 in the AnkiDroid app are shown on seven screens from left to right. An event explosion is possible to happen in this example when performing coverage-based fuzzing since there are, in the worst case, $5! \times 8! \times 3! \times 5! \times 10! \times 5! \times 7!$ different cases of selecting widgets in the shown screens.

- Using the widgets' attribute value states to recognize the order and dependency between events in triggering a crash.
- Proposing a directed fuzzing approach that increases the possibility of reproducing a crash and speeding up the reproduction time.

The rest of the paper is organized as follows: In Section II, a brief overview of events and widgets in Android apps is given. Next, we discuss the problems when leveraging the existing fuzzing tools for reproducing crashes from stack traces in Section III. Later in this section, we use a motivating example to elaborate more on the existing challenges and our insight into solving them. Section IV outlines our proposed approach for reproducing the known crashes. Section V presents the evaluation results of our approach in terms of effectiveness and performance by using the success ratio and a discussion on future work. Finally, we survey existing work in crash reproduction in Section VI and conclude the paper in Section VII.

II. Background

To create a shared understanding of the method presented in this paper, we provide an overview of the event-based model of Android apps and the structure of widgets in these apps. Later on, we discuss existing studies that can be used for crash reproduction in Android apps with having the crash stack trace.

A. Preliminary

Event-driven Apps: An event is the main entry point of an app when the user interacts with it. To handle the event received from a user, a callback function must be declared in the app code. There are distinct UI-based callback functions for each event type in Android (e.g., `onClick` for handling *clicking* event), as shown in Table I. These functions are implemented in the activity components of an app. In Android, activities are designed to handle user interactions with the app. In this component, one or multiple screens[4] are created. Due to the transition of a user within screens, an activity has different states, including creation, (re)starting, pausing, resuming, stopping, etc. The state changes in an activity are also handled with some other callback functions (e.g., `onCreate`, `onPause`, etc.). The creation of the UI elements and assigning callback functions to them occur in an activity's creation, (re)starting and resuming states. Hence, an activity is the largest unit of an app's Graphical User Interface (GUI), and a UI element is the smallest in which the user interacts with the app by generating an event on it.

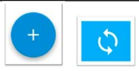
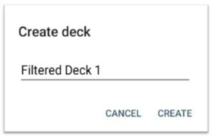
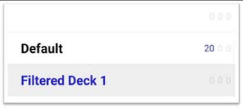
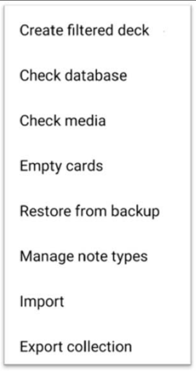
Widgets: A UI element, which is known as a widget, can be a single element (e.g., button, text box) or a grouped set (e.g., list items). Various types of widgets, including Button and TextView, are supported in Android. Since an event on a widget is regarded as an entry point to an app, it is essential to identify the widgets and callback functions used for handling the events for the fuzzing process. Table I provides a list of some of the existing widget types and callback function names defined in Android for handling events on them.

There are two resources in an app for designing widgets on a screen and declaring callbacks to handle events generated on them:

1) **XML layout files:** The layout structure of the widgets in a screen is stored in an *XML* file. In this file, distinct tags are used to differentiate between different widget types. This file also contains other widget properties via attributes such as a unique identifier, size, and position on the screen.

[4]Fragment is used for designing various screens separately in Android. For simplicity, we use the term *widget* to refer to a fragment and consider a grouped kind of widget as it is composed of several widgets inside.

TABLE I
LIST OF ANDROID WIDGETS AND ASSOCIATED CALLBACKS USED FOR EVENT HANDLING

Widget	**Callback(s)**	**Example**
Button	`onClick`	
AlertDialog	`onClick`	
ListView	`onListItemClick`	
Menu	`onOptionsItemSelected`	 Create filtered deck Check database Check media Empty cards Restore from backup Manage note types Import Export collection

2) **App source code:** The widget-typed variables correspond to the widget items in the existing layout files, which are declared in the app code.

Methods, such as *findViewById*, are used to map a variable to its equivalent widget in the layout file. Besides this way of creating widgets, one can dynamically add a new widget to a screen and set its attributes in the app code. Also, there exist some API functions (e.g., `getVisibility` or `setVisibility`) to access other attributes of a widget and set/modify their value in the code. Both the layout files and application code are valuable resources for identifying the widgets and their attribute value states in an application.

Callback Functions: In addition, callback functions are also implemented in the app code for handling events on the widgets. A callback function is assigned to a widget through *listeners* (e.g., `setOnClickListener`. Since a screen can contain several UI elements, an app does not have a single entry point from which to start the static analysis. Hence, callback functions do not have a direct edge in an Android app's Control Flow Graph (CFG). Instead, when an event is entered on a UI element, the Android operating system calls the corresponding callback function in the app code. FlowDroid [23] is a solution that constructs a *dummy main method* and overapproximates the execution order of the UI callback functions in an activity to aid in statically analyzing Android apps. An example of this method for `DeckPicker` activity is given in Fig. 5(a). As shown in this Figure, FlowDroid models the lifecycle of an activity's callback. This comes from the fact that the number of an activity's states and the transitions between them are few. Then, it directly *calls* the callbacks declared for handling UI events one after another in this method. Hence, static analysis solutions can use the dummy main method to analyze different properties on an Android app.

TABLE II
FREQUENCY DISTRIBUTION OF SUCCESS RATIOS FOR FUZZING TOOLS IN REPRODUCING CRASHES FROM THE THEMIS BENCHMARK (52 CRASHES) ACROSS TEN SEPARATE RUNS

Fuzzing Tool	**Success Ratio**				
	[0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1]
Monkey	74%	0	3%	7%	16%
Ape	72%	0	0%	0%	28%
Stoat	91%	0	0%	0%	9%
FastBot2	78%	4%	6%	2%	10%

B. Crash Reproduction in Android

Since a crash stack trace is unique and both coverage-based and directed Android fuzzing tools can access the log files, it is possible to leverage them to explore the app's UI elements, looking for a previously found crash. Coverage-based solutions utilize either random or other heuristics (e.g., code coverage, activity coverage, event count) to generate UI events to cover more parts of an app's code or UI events. In practice, we leveraged a set of critical[5] crashes collected in Themis benchmark [24] to assess the capability of coverage-based tools in crash reproduction. We selected four of the most well-known and effective coverage-based fuzzing tools for finding crashes, including:

- Monkey [25], a random-based fuzzing solution.
- Ape [5], a model-based fuzzing solution that uses a combination of random and greedy search to fuzz an app.
- Stoat [6], a stochastic model-based fuzzing solution.
- FastBot2 [13], a model-based fuzzing solution that uses reinforcement learning to generate events considering the activity transition.

We performed the fuzzing process with each tool in 10 separate runs in six hours[6]. Our experiments reveal that even the best coverage-based fuzzing tools are unable to reproduce critical crashes within a limited time. To provide more details, we have included Table II, which presents information about the success ratio of several fuzzing tools that we tested. The "success ratio" refers to the number of times a particular crash was successfully reproduced, divided by the total number of attempts made by the fuzzing tool. In this table, we have organized the success ratio into 5 different categories, or "buckets." For each bucket, we show the percentage of crashes with a success ratio within that range. The key takeaway from this table is that for around 70% of the crashes, the success ratio was less than 0.2. In other

[5]A bug is noted as critical if it affects a large number of the users and significant functionalities of the app are involved according to Themis [24].

[6]We used the Google Android emulator (Android 7.1 with API level 25) deployed on a 64-bit Ubuntu 18.04 machine (80 cores, Intel(R) Xeon(R) CPU E5-2698, and 250GB RAM). Each emulator was configured with 2GB RAM, 1GB SDCard, 1GB internal storage, and an X86 ABI image similar to the setting used by Themis [24].

words, the fuzzing tools reliably reproduced less than 30% of the critical crashes, even after multiple attempts.

On the contrary, directed solutions such as CrashScope [21] and GoalExplorer [22] focus only on generating the events necessary to reach a specific point in the app instead of exploring the entire event space. These approaches identify such events by building a UI model of the feasible paths leading to a particular part of the app. The precision of their model plays a crucial role in achieving faster results. Since widgets are the smallest UI objects in an app, a UI model with this level of granularity is ideal. However, statically extracting such a model can lead to scalability issues due to the large number of widgets in an app, each with numerous possible execution orders that must be analyzed. To address this issue, CrashScope and GoalExplorer construct a high-level UI model to guide fuzzers only in exploring the relevant code. However, their analyses are not scalable due to the event explosion problem. In fact, they have to sacrifice precision for efficiency by constructing the model using large units, such as activities and screens containing multiple widgets. Consequently, these larger units can still have several widgets and do not filter the infeasible program effectively at run time, leading to event explosion.

III. Motivating Example

To better illustrate the challenges of existing directed-based fuzzing studies in crash reproduction, we selected crash issue 5756 in AndkiDroid version 2.9.4. In this example, the crash occurs when the user creates a filtered deck for the first time and clicks on it. We highlighted the sequence of seven steps for reproducing this crash in Fig. 2.

CrashScope [21] attempts to find the set of activities involved in triggering a crash. To find a precise set of activities, this tool initially looks for them based on the contextual information inferred from the call stack in the crash stack trace (i.e., access to Wi-Fi and sensors.). This is mainly to enable the successful performance of such operations that are essential for triggering the crash. After finding the initial sets of activities, CrashScope starts fuzzing the app. Then, it dynamically builds an activity transition graph to track the activities that reach the target activity, which is located at the crash point. It keeps generating events in the target activity until the crash is found.

Although CrashScope attempts to find an accurate set of activities statically and dynamically, relying on contextual information limits its capability to reproduce different crash scenarios. For instance, CrashScope only statically identifies `DeckPicker` activity, and thus, cannot identify `FilteredDeckOptions` since the target crash locates in `DeckPicker`, and `FilteredDeckOptions` activity is created after it. Furthermore, since CrashScope does not keep track of widgets within activities, it cannot find out that the state of widgets in the former depends on the latter. Thus, it cannot perform steps ④ to ⑦ to create a deck filter to reproduce our target crash.

GoalExplorer [22], on the other hand, constructs the UI model at the screen level, which is more precise than an activity. Since an activity can have several screens, GoalExplorer statically identifies the screens and their transitions by performing a forward reachability analysis. To achieve this goal, it relies on the API functions used to create screens via fragments and inter-component communications. Fig. 2 illustrates some of the screens and activities found by GoalExplorer for our example crash. Even though GoalExplorer uses a smaller granularity, the number of widgets it finds is still large to explore. For instance, there are forty-three widgets in total in screens ① to ⑦ in our motivating example. Exploring all of them to perform the seven consecutive events for triggering our example crash requires many attempts. As a result, GoalExplorer could not reproduce the crash in six hours.

To overcome this issue, GoalExplorer uses heuristics in its fuzzing process by selecting each widget only once with the help of Stoat [6]. However, crashes may require complex conditions, such as executing the events in a specific order with certain repetitions. Consider that in our example crash, if the ‘cancel’ button in screen ③ is selected, it is not possible to create a filtered deck at all. The probability of generating an event only on this item in this screen is 0.5, but reaching this screen is dependent on the widgets selected in previous screens. Therefore, many attempts are needed to have all seven events generated in a specific order. Apart from the exploration strategy, GoalExplorer performs an intra-procedural callback analysis to find the screens. This makes its UI model inaccurate and imprecise when the screens are not directly created or accessed in the callback functions.

Overall, we summarize two challenges to overcome the illustrated problem below:

Challenge 1: How to construct a *precise and scalable UI model*? One of the challenges in constructing a precise and scalable UI model is the large number of possible sequences of generating events on the widgets, which can result in scalability issues or even explosion. To address this issue, we observe that the widget dependencies can help us find a more precise UI model of widgets involved in a crash. For example, the visibility attribute of the fragment holding a list of ‘filtered decks’ adds a dependency between the ‘filtered deck’ items in it (screen ⑦) and the ‘create filtered deck’ menu item (screen ②). The fragment must be visible so the newly created filtered deck is shown in screen ⑦. If we find such dependencies statically, we can deduce that the menu item and the newly created ‘deck filter’ must be selected. In addition to this, other widgets, such as the ‘setting’ button at the top left or the ‘add’ button at the bottom right, are extra, and performing an event on them is irrelevant. Overall, such dependencies allow us to classify the events in an app into two sets, necessary and irrelevant, for triggering a crash.

Challenge 2: How to identify the widget dependencies *statically*? The dependency between the widgets in an Android app is either due to the parent-child relationship or the widgets’ attribute states. The former dependency is obtained directly from the layout files. However, the latter is *enforced via widget attributes* in the app’s code. To find and track such dependency, the widget attributes and API functions with their states must be identified in the analysis. However, existing reachability analysis [19], [20], [22] in Android only retrieves the widgets

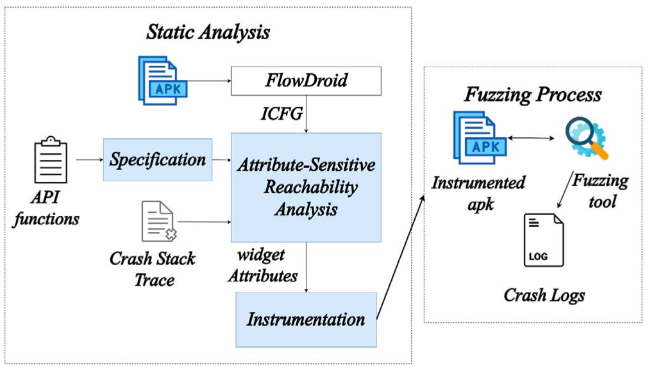


Fig. 3. Workflow of MOLE.

and their UI hierarchy in the code without tracking the widget attribute states since it is challenging to find the specification for each API function. Therefore, they cannot infer the widget dependencies precisely.

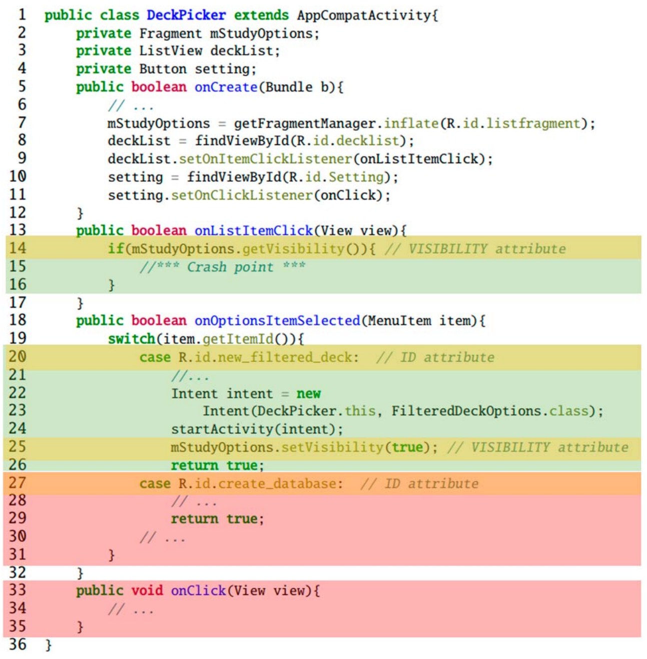
IV. PROPOSED APPROACH

The analysis performed by employing existing fuzzing tools shows the gap for efficiently reproducing a crash from the crash stack trace. To cover part of this gap, we introduced an approach that leverages existing fuzzing tools to search the smaller event domain on the app, which includes necessary events to reproduce a crash. To this end, the rest of this section first provides an overview of our approach and then delves into the details of our methodology.

A. Overview

Fig. 3 shows the workflow of our proposed approach named MOLE. MOLE is a directed fuzzing approach consisting of a static analysis and the fuzzing process. It accepts the *apk* file of an Android app and the *crash stack trace* as input. MOLE preprocesses the app through static analysis by performing an attribute-sensitive reachability analysis. It intends to identify the necessary and irrelevant events and enforce the execution of the former in the fuzzing process. Below, we elaborate on how MOLE handles each challenge in each step in detail using our motivating example:

Attribute-Sensitive Reachability Analysis: *To find a precise and scalable UI model of the necessary and irrelevant events, MOLE proposes a backward flow, context, path, and* ***attribute-sensitive reachability analysis*** *from the crash point.* Since statically determining all the event sequences confronts the scalability challenge, relying on the widget dependencies can precisely find the correct execution order of some of them. An attribute a applies the dependency between two widgets w_1 and w_2 if a's value is accessed upon generating an event on w_1 and is set with the same value state when generating an event on w_2. For example, the list of deck filters (w_1) is dependent on the menu item 'create deck filter' (w_2) in its visibility attribute (a) as shown in the motivating example in Fig. 2. Such a dependency is applied through the app's code. The visibility attribute of the



```
1   public class DeckPicker extends AppCompatActivity{
2       private Fragment mStudyOptions;
3       private ListView deckList;
4       private Button setting;
5       public boolean onCreate(Bundle b){
6           // ...
7           mStudyOptions = getFragmentManager.inflate(R.id.listfragment);
8           deckList = findViewById(R.id.decklist);
9           deckList.setOnItemClickListener(onListItemClick);
10          setting = findViewById(R.id.Setting);
11          setting.setOnClickListener(onClick);
12      }
13      public boolean onListItemClick(View view){
14          if(mStudyOptions.getVisibility()){ // VISIBILITY attribute
15              //*** Crash point ***
16          }
17      }
18      public boolean onOptionsItemSelected(MenuItem item){
19          switch(item.getItemId()){
20              case R.id.new_filtered_deck:  // ID attribute
21                  //...
22                  Intent intent = new
23                      Intent(DeckPicker.this, FilteredDeckOptions.class);
24                  startActivity(intent);
25                  mStudyOptions.setVisibility(true); // VISIBILITY attribute
26                  return true;
27              case R.id.create_database:  // ID attribute
28                  // ...
29                  return true;
30              // ...
31          }
32      }
33      public void onClick(View view){
34          // ...
35      }
36  }
```

Fig. 4. Snippet code of the DeckPicker activity. Red, green, and yellow colors indicate the feasible, irrelevant, and set/access statements.

fragment $mStudyOptions$, which holds the deck filter list, is accessed in a conditional statement (Line 14 in Fig. 4). This condition is satisfied if the attribute is set with value `true` (Line 25). Since these two statements are located in `onListItemClick` and `onOptionsItemSelected` callbacks, generating at least one event on their corresponding widgets (deck filter list and menu items) is necessary. To find these dependencies, MOLE leverages FlowDroid [23] to extract the ICFG of the app. Then, it uses the line number and the function where the crash point is located available in the crash stack trace to start the analysis.

API Specification: To track the widget attribute states, MOLE must be capable of identifying the statements where a widget attribute is set and accessed. In Android, several API functions exist to set and access the attributes programmatically. As Android has a finite number of attributes, we design a specification in MOLE to *store the API functions used for setting and accessing their states* as a solution to the second challenge. Hence, we can determine whether a statement is setting an attribute value or accessing it. We call these kinds of statements *def-site* and *use-site* respectively. We propose an abstraction, called *configuration*, to track the attribute states from the def-sites and use-sites in our backward analysis.

A configuration is a data structure that maps a widget element to a widget type variable in the code and stores its attributes' value states. Since the def-site and use-site relations must be preserved for each attribute, MOLE computes the value of an attribute at the reached use-site. Then, it searches through the activity for the def-site satisfying it. As each attribute has a finite value domain of boolean or numeric (explained in Section IV-D), we devise a set of *transfer function rules* in MOLE to compute the attribute values. For computing

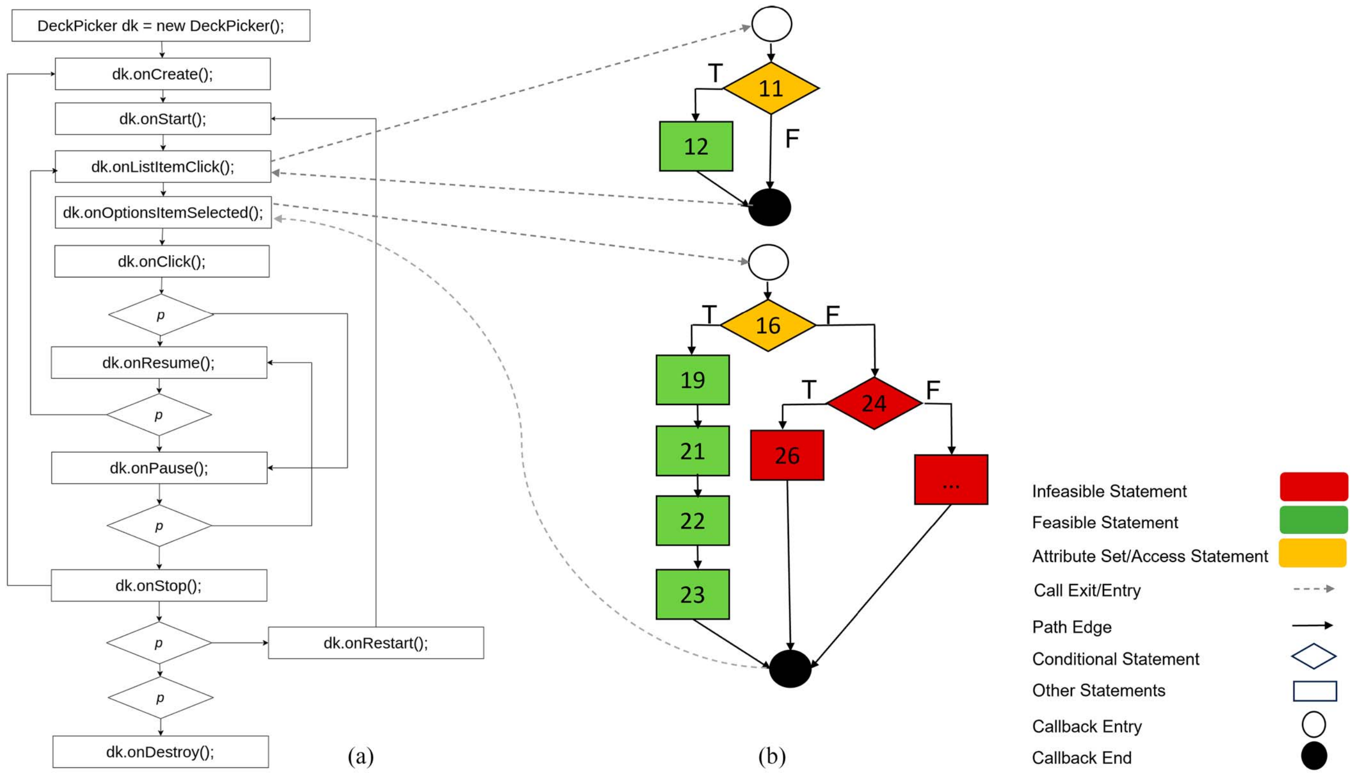


Fig. 5. (a) The control flow graph of the *dummy main method* built for `DeckPicker` activity. (b) The control flow graph of `onListItemClick` and `onOptionsItemSelected`. The numbers in each oval/rectangle present the line number in `DeckPicker` code.

attribute-related path conditions, our approach is similar to some studies [12], [18] that statically approximate the values of variables.

Instrumentation: After the attribute-sensitive reachability analysis is completed, it is time to identify the necessary events regarding the dependent widgets. The collected widget attributes are received in a set of configurations per each callback as input in this phase. Having at least one configuration in a callback indicates that this callback is dependent on at least one other callback to preserve an attribute state essential for reaching the crash point. Hence, it is *necessary* to generate the event that the callback handles. In addition, if there is no configuration in a callback, the event it handles might not take any role in the crash occurrence, and MOLE labels it as *irrelevant*.

Since some callback functions like `onOptionsItemSelected` handle events on several widgets, it is essential to identify the necessary events in such callbacks precisely. In these kinds of callbacks, the *ID* attribute of a widget, which is unique for each widget, is used to handle each event separately (Lines 20 and Line 27 in Fig. 4). The disjunctive path conditions are used in the app code to achieve this goal. We leverage this fact and inject the disjunctive path conditions at the entry point of each callback function to only accept necessary events and reject the irrelevant ones. This is equivalent to the weakest precondition [26], which is the weakest condition to be satisfied to reach a target statement. In our example crash, the def-site of attribute a is reached when the `case R.id.new_filtered_deck` is satisfied. The '$new\_filtered\_deck$' is the ID of an item in the deck filter list. As the path under `case R.id.create_database` is infeasible according to Fig. 5, generating an event on its widget is irrelevant and must be rejected. Hence, we inject the statements `assertTrue(item.getItemID() == R.id. new_filtered_deck || item.getItemID() != R.id.create_database);` at the entry of the callback function to prune the infeasible paths.

Fuzzing Process: In the last step, a fuzzing tool is selected, and **the fuzzing process** is initiated on the instrumented *apk* until the given time threshold is reached. The crash logs are recorded in a file during this process. This file is later used to search whether the target crash stack trace has been triggered or not.

B. Preliminary

Abstract Program: A simplified language is used to represent the target Android app as an abstract program in our proposed attribute-sensitive reachability analysis. As shown in Fig. 6, an abstract app a consists of one or more components q. Each component has one or more functions f. A function f accepts a vector of parameters $(v_1, v_2, ..., v_l)$ and returns a value in register r. It includes a set of statements s. A statement s is a sequence of statements that each can be any of an assignment $lhs \leftarrow rhs$, a conditional `if(b)`, a `goto`, or a `return` statement. In our simple language, $r[r]$ presents access to an object in an array. Access to an object's fields is also considered using the @ notation. In addition, we presume the binary or logical operations and comparisons on numerical values. It is worth

$$\begin{array}{lll}
\mathit{App}\ a & ::= & \mathit{Component}^{+} \\
\mathit{Component}\ q & ::= & \mathit{function}^{+} \\
\mathit{function}\ f & ::= & f : (v_1, ..., v_l) \rightarrow r\{s\} \\
\mathit{Statement}\ s & ::= & s_{prim} \mid s_{cond}.s \mid s; s \\
s_{cond} & ::= & if(b) \\
s_{prim} & ::= & lhs\ \leftarrow rhs \mid goto\ r \mid return\ r \\
 & \mid & invoke\ f(v_1, ..., v_l) \\
lhs & ::= & r \mid r[r] \mid @field \mid r.@field \\
rhs & ::= & lhs \mid invoke\ f(v_1, ..., v_l) \\
 & \mid & r \oplus r \mid r \mid \neg r \mid c \\
b & ::= & rhs\ \circledast\ rhs \mid \odot rhs \\
\oplus & ::= & + \mid - \mid ... \\
\odot & ::= & \neg \mid nill \mid \odot\odot \\
\circledast & ::= & = \mid \neq \mid > \mid < \mid \geq \mid \leq \\
c & ::= & n \mid true \mid false \qquad n \in \mathbb{Z}
\end{array}$$

Fig. 6. A simplified language for our proposed analysis.

noting that loop structures (like `for` and `while`) and more complicated conditional statements (like `switch-case`) can be abstracted as an `if` statement. In our reachability analysis, we consider execution (not an important one or more) or non-execution of the code placed in a loop based on the condition defined in the loop statement.

C. Attribute-Sensitive Reachability Analysis

To determine the necessary and irrelevant events by inferring dependent widgets, we explore the code for attribute states and dependent widgets. Specifically, we propose a new level of sensitivity, called *attribute-sensitivity*, and arm the backward flow, context, and path-sensitive analysis with it.

Definition IV.1: An *attribute-sensitive reachability analysis* tracks the states of the widget attributes on the feasible paths to the target statement as an indicator for finding dependent widgets.

Our analysis considers the crash point as the target statement s_t. To trace and collect the widget attribute states, we proposed a new data structure called *configuration* that tracks the widget attribute states in our analysis.

Definition IV.2: A *configuration* is a triple (p, t, r) that holds the set p consisting of the widget attribute states, the target label t, and the reachability state r collected at the entry and exit point of each statement (considering the effect of the statement itself) in the feasible paths to the target statement s_t in a backward manner.

In the configuration, we use the set p to store the necessary widget attribute states in a callback context. If the widget attribute state is checked in the conditional statement *if(b)* where *b* is a binary or comparison operation in a feasible path to s_t, it indicates that this state must be satisfied to trigger the crash at run time. We assume such an attribute state must be met. Hence, this set in the configuration c_{entry}, that is, the configuration at the entry point of a callback function, holds all the widget attribute states that must be satisfied in that callback.

Definition IV.3: A *widget attribute state* ρ is a four-dimensional vector $(var, id, attr, interval)$, that:

- var is the local or global variable used for representing a widget. A register *r* is used to store its value.
- id is a constant number used as an identifier for each widget in the layout file.
- att is a constant value used to represent an attribute of the widget.
- $interval$ is the expected range of values for the attribute *att* at the code.

Example IV.1: Consider that we want to extract the widget attribute `VISIBILITY` accessed at Line 14 in Fig. 4. At this line, the `VISIBILITY` attribute value state of the fragment $mStudyOptions$ is accessed in a conditional statement. Since the crash point is located in the `true` branch of this conditional statement, here b equals `mStudyOptions.getVisibility() == true`. Therefore, ρ_{11} will be $(mStudyOptions, null, VISIBILITY, [1, 1])$ for this statement, where 1 is equivalent to value $true$.

If p is not empty in a callback context, we can deduce the existence of a necessary widget attribute state to be met to trigger the crash in that callback. This set alone cannot be used to determine the necessity of a callback. In fact, the *dummy main method* of an activity has a `for` loop overapproximating the possible order of callbacks in the activity. Hence, p will be propagated to all the callbacks inter-procedurally. If we only rely on the set p, then all the callbacks are inferred as necessary even though some of them do not take part in triggering the crash. Alongside the collected use-sites, the callbacks containing the def-sites satisfying them should only be reachable at run time. Hence, we use another label named t to determine whether the crash point or a def-site corresponding to the collected widget attributes exists in a callback context. The combination of the sets p and t helps us to find all the paths that are set precisely and access the necessary widget attribute states among all the feasible paths.

Example IV.2: Consider that we want to identify the necessity and irrelevancy of the callbacks `onListItemClick` and `onClick` by using t and p. Since the crash point is located in `onListItemClick` callback, t is initialized with `true` in this callback. Starting from the crash point and reaching the Line 14, p is assigned with $\{\rho_{11}\}$ due to checking the visibility attribute state. According to the values of t and p, `onListItemClick` is a necessary callback. Since $mStudyOptions$ is a global variable, this p is propagated inter-procedurally in the *dummy main method*, while t is initialized for each function separately. In the `onClick` callback, however, there is no def-site corresponding to the collected attributes in p. Therefore, t remains `false` for this callback, and we can deduce this callback is irrelevant.

The label r in a configuration (p, t, r) indicates the reachability state of each statement. It can be either $true$ or $false$. Regarding the structure of the *dummy main method*, a reachability analysis marks all the callbacks and the events reachable if at least one feasible path exists.

Configuration Domain: According to the above definitions, the domain of a configuration (p, t, r) is $P \times 2^N \times \{true, false\}$ where N is the number of statements in a program. P is a power set of $Var \times ID \times Attr \times \mathbb{Z}$ in a program where

- Var is a set of all widget-typed variables defined in an app

Algorithm 1 Backward Attribute-Sensitive Reachability Analysis

```
Input: $s_t$: the crash point statement, $cfg$ : the CFG of function $f$ having all statements $Stmt$ as nodes and flows of statements as edges, $c^f_{exit}$: the configuration at the exit point of function $f$ in case there is a call-site of $f$
Output: $cl : Stmt \rightarrow Config \times Config$
1: procedure BACKWARDANALYSIS($s_t$, $cfg$, $c^f_{exit}$)
2:     for $s \in cfg$ do
3:         $c_{exit} \leftarrow (\emptyset, false, false)$
4:         $c_{entry} \leftarrow (\emptyset, false, false)$
5:         $cl(s) \leftarrow (c_{entry}, c_{exit})$
6:     end for
7:     if $s_t \in cfg$ then
8:         $(c_{entry}, c_{exit}) \leftarrow cl(s_t)$
9:         $c_{entry} \leftarrow (\emptyset, true, true)$
10:        $c_{entry} \leftarrow computeConfig(cfg, s_t, c_{entry})$
11:        $cl(s_t) \leftarrow (c_{entry}, c_{exit})$
12:        $StmtStack.push(s_t)$
13:        if $c^f_{exit} \neq (\emptyset, false, false)$ then
14:            $s_{last} \leftarrow getLastStatement(cfg)$
15:            $cl(s_{last}) \leftarrow ((\emptyset, false, false), c^f_{exit})$
16:        end if
17:    else
18:        if $c^f_{exit} \neq (\emptyset, false, false)$ then
19:            for $s \in cfg$ do
20:                $cl(s) \leftarrow (c^f_{exit}, c^f_{exit})$
21:            end for
22:        end if
23:    end if
24:    while StmtStack $\neq \emptyset$ do
25:        $s \leftarrow StmtStack.pop()$
26:        for $s' \in cfg.pred(s)$ do
27:            $(c_{entry}, c_{exit}) \leftarrow cl(s)$
28:            $c'_{exit} \leftarrow c_{entry}$
29:            $c'_{entry} \leftarrow computeConfig(cfg, s', c_{entry})$
30:            $(c'_{entry}, c'_{exit}) \leftarrow (c'_{entry}, c'_{exit}) \sqcup cl(s')$
31:            $cl(s') \leftarrow (c'_{exit}, c'_{entry})$
32:            $StmtStack.push(s')$
33:        end for
34:    end while
35: end procedure
```

- ID is a set of all id values of widgets defined in existing layouts of an Android app
- $Attr$ is a set of attributes of all the widget types in android
- $\mathbb{Z}$ is the domain of the attribute values that have a numerical domain

Algorithm: With the proposed configuration, finding a more precise set of feasible paths to the crash point is possible. The algorithm for performing the inter-procedural attribute-sensitive reachability analysis is shown in Algorithm 1. In this algorithm, the target statement s_t, the control flow graph of the function f, and the configuration received at the exit point of the call site f are provided as inputs. In addition, our algorithm computes the configuration c_{entry} and c_{exit} received at the entry and exit point of each statement s in the function f and stores them in cl.

Assuming that the target statement s_t exists in function f, the analysis is launched from the statement containing the call to function f. It calls the algorithm $Backward$-$Analysis(s_t, cfg, c^f_{exit})$. Here, the cfg is the control flow graph of the function f. Since the crash point is not reached yet, c^f_{exit} equals $(\emptyset, false, false)$. When the input is received in the algorithm, the configuration at each statement's entry and exit points s in the function f initialized with $(\emptyset, false, false)$ (Lines 2 to 6). Next, the configuration at the entry point of s_t is assigned with $(\emptyset, true, true)$ since s_t is located in function f (Line 9). Then, we compute the effect of the statement s_t according to the statement type. This computation is processed in the $computeConfig$ procedure with the help of designated transfer function rules. This procedure applies any of our proposed transfer function rules according to the type of the statement[7]. In Line 12, we store s_t in a stack to start the backward analysis from it. This initiates the process of looking for the necessary widget attributes in feasible paths to the crash point.

Afterward, we loop through all the predecessors of s_t and compute the effect on the propagated configuration until we reach the entry point of the function (Lines 24 to 34). In this analysis, the meet operation $\sqcup$ is used to join the configurations from multiple paths (Line 30). This operation is union $\cup$ for the path conditions p and the target label t and a logical disjunction $\vee$ for the reachability states r. Assume that s has two successors s' and s'' with c'_{entry} and c''_{entry} that are received at the exit point of the statement s, then its c_{exit} is:

$$c'_{entry} = (p, t, r) \ and \ c''_{entry} = (p', t', r')$$
$$\begin{aligned} c_{exit} &= c'_{entry} \sqcup c''_{entry} \\ &= (p, t, r) \sqcup (p', t', r') = (p \cup p', t \cup t', r \vee r') \end{aligned}$$

Transfer Function Rule: A transfer function rule $[R_i]$ is designed to compute c_{entry} from c_{exit} under condition c by considering the effect of the statement itself as below:

$$[R_i] \frac{\langle c_{exit}, s \rangle, \ c}{c_{entry}}$$

A set of twelve rules $R_1, ..., R_{12}$ are proposed for computing p, t and r and they are classified in the following groups:

1) Attribute Inference Rules for tracking widget attribute states (Rules $R_1 - R_5$)
2) Target Rules for tracking target statements in t (Rule $R_6 - R_7$)
3) Reachability Rule for following reachable statements (Rule R_8)
4) Inter-procedural Rules for propagating configurations from the caller to callee and vice versa (Rules $R_9 - R_{12}$)

The formal definition of these rules is given in Table III. The function map is used to map the parameters passed in the callee

[7]For simplicity, we only provide the transfer functions that we use in $computeConfig$ for checking the effect of each statement and do not add the algorithm of this function.

TABLE III
TRANSFER FUNCTION RULES

Rule Class	Semantics	Description
Attribute Inference Rules	$\text{R}_1\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}), x \leftarrow x'\rangle}{(p_{entry}, t_{exit}, r_{exit})}$	assigning the object x' to x: if both objects are widgets, all the configurations in p_{exit} must propagate to p_{entry} and the variable x must be updated with x' in widget states, so $p_{entry} = \bigcup_{(var,id,att,val)\in p_{exit} \wedge var\neq x}\{(var, id, att, val)\} \cup \bigcup_{(x,id,att,val)\in p_{exit}}\{(x', id, att, val)\}$
	$\text{R}_2\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}), r.@d \leftarrow c\rangle}{(p_{entry}, t_{exit}, r_{exit})}$	setting/updating the value of attribute attr(d) for widget r with the constant value c, so $p_{entry} = \bigcup_{(r,id,attr(d),val)\in p_{exit}}\{(var, id, att, c)\} \cup \bigcup_{(var,id,att,val)\in p_{exit}\wedge(var\neq r \vee attr(att)\neq attr(d))}\{(var, id, att, val)\}$
	$\text{R}_3\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), if(\odot r.@a)\rangle \\ eval(\odot r.@a) = \delta\end{array}}{(p_{exit} \cup \{(r, null, attr(a), [n, n])\}, t_{exit}, r_{exit})}$	accessing and checking the boolean-typed widget attribute attr(a) when the condition evaluates to δ, either `true` or `false`: $n = \begin{cases} 1 & \delta = true \\ -1 & \delta = false \end{cases}$
	$\text{R}_4\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), if(r.@a\ \circledast\ n)\rangle \\ eval(r.@a\ \circledast\ n)\end{array}}{(p_{exit} \cup \{(r, null, attr(a), [i, j])\}, t_{exit}, r_{exit})}$	accessing and checking the equivalence of numerical-typed widget attribute, in which: $[i, j] = \begin{cases} [n, +\infty) & \circledast := \geq \\ [n+1, +\infty) & \circledast := > \\ (-\infty, n-1] & \circledast := < \\ (-\infty, n] & \circledast := \leq \\ [n, n] & \circledast := = \\ (-\infty, n-1] \cup [n+1, +\infty) & \circledast := \neq \end{cases}$
	$\text{R}_5\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}), goto\ r\rangle}{(p_{exit}, t_{exit}, r_{exit})}$	changing the program execution to the address stored in register r
Target Rules	$\text{R}_6\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}), true\rangle}{(p_{exit}, t_{exit} \cup true, r_{exit})}$	updating the target label t_{entry} when reaching crash point
	$\text{R}_7\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), r@d \leftarrow c\rangle \\ \exists(r, id, attr(d), c) \in p_{exit}\end{array}}{(p_{exit}, t_{exit} \cup true, r_{exit})}$	reaching a def-site of an already found widget attribute attr(d) and updating the target label t_{entry}
Reachability Rule	$\text{R}_8\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}),\ s\rangle, s \in t_{exit}}{(p_{exit}, t_{exit}, r_{exit} \vee true)}$	checking the reachability state of the statement s
Inter-procedural Rules	$\text{R}_9\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), return\ r\rangle \\ lhs \leftarrow invoke\, f(p_1, ...p_l) \wedge f : (v_1, ..., v_l) \rightarrow r\end{array}}{(p_{entry}, \emptyset, r_{exit})}$	propagating all the widget attributes of *global declared*, passed through *arguments* and lhs received at the exit point of callee f *from* the *caller* to the *callee's* body, so $p_{entry} = \{(var, id, att, val) \mid (var, id, att, val) \in p_{exit} \wedge var \in GlobalVar\} \cup \{(v_i, id, att, val) \mid (p_i, id, att, val) \in p_{exit} \wedge map(p_i) = v_i\} \cup \{(r, id, att, val) \mid (lhs, id, att, val) \in p_{exit}\}\}$
	$\text{R}_{10}\ \dfrac{\langle(p_{exit}, t_{exit}, r_{exit}), return\rangle}{(p_{entry}, \emptyset, r_{exit})}$	propagating all the widget attributes of *global declared* or passed through *arguments* received at the exit point of callee f *from* the *caller* to the *callee's* body, so $pentry = \{(var, id, att, val) \mid (var, id, att, val) \in p_{exit} \wedge var \in GlobalVar\} \cup \{(v_i, id, att, val) \mid (p_i, id, att, val) \in p_{exit} \wedge map(p_i) = v_i\}$
	$\text{R}_{11}\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), lhs \leftarrow invoke\ f(p_1, ..., p_l)\rangle \\ [(p'_{entry}, t'_{entry}, r'_{entry}), r] \leftarrow compute(f)\end{array}}{(p_{entry}, t_{entry}, r_{exit} \vee r'_{entry})}$	propagating the widget attribute states received at the entry point of *callee* to the *caller* considering the effect of *global variables*, *arguments* and *return value*, so $t_{exit} \cup \{lhs \leftarrow invoke\ f(v_1, ..., v_l) \mid \lvert t'_{entry}\rvert \neq 0\}$ $p_{entry} \leftarrow \bigcup_{(var,id,att,val)\in p_{exit}\wedge var\notin\{lhs\}\cup GlobalVar}\{(var, id, att, val)\} \cup \bigcup_{(p_i,id,att,val)\in p'_{entry}\wedge map^{-1}(p_i)=v_i}\{(v_i, id, att, val)\}$ $\bigcup_{(r,id,att,val)\in p'_{entry}}\{(lhs, id, att, val)\}$
	$\text{R}_{12}\ \dfrac{\begin{array}{c}\langle(p_{exit}, t_{exit}, r_{exit}), invoke\ f(p_1, ..., p_l)\rangle \\ [(p'_{entry}, t'_{entry}, r'_{entry}), null] \leftarrow compute(f)\end{array}}{(p_{entry}, t_{entry}, r_{entry} \cup r'_{entry})}$	propagating the widget attribute states received at the entry point of *callee* to the *caller* considering the effect of *global variables* and *arguments*, so $t_{entry} = t_{exit} \cup \{invoke\ f(v_1, ..., v_l) \mid \lvert t'_{entry}\rvert \neq 0\}$ $p_{entry} \leftarrow \bigcup_{(var,id,att,val)\in p_{exit}\wedge var\notin\{lhs\}\cup GlobalVar}\{(var, id, att, val)\} \cup \bigcup_{(p_i,id,att,val)\in p'_{entry}\wedge map^{-1}(p_i)=v_i}\{(v_i, id, att, val)\}$

with its arguments. A complete description of these rules, along with examples, is given in the supplementary material.

Example IV.3: For each type of statement, one of the rules $[R_i]$ can be applied. For instance, the algorithm reaches a conditional statement in Line 14 in Fig. 4 in the callback `onListItemClick` immediately after finding the crash point. The algorithm checks whether the conditional statement performs a comparison on a widget attribute. The rules $[R_3]$ and $[R_4]$ are designed for checking the effect of `if` statement for different comparison operations. In our example statement, the function `getVisibility` is called on the variable $mStudyOptions$. Since the conditional statement checks the equality of the attribute with the value `true`, the rule $[R_3]$ is applied. According to the rule, the set p is updated with the visibility attribute state of $attr(getVisibility)$.

Summary: Since the *dummy main method* is a `for` loop, our analysis continues until a fixed point is reached. This means that the analysis continues until it finds all the def-sites

TABLE IV
List of Some of the Widget Attributes and API Functions Used for Setting and Accessing Their State

Type	API function	Attr
	Object findViewById(int id)	GETID
	int getId()	GETID
	void setVisibility(int visible)	VISIBILITY
android.view.View	int getVisibility()	VISIBILITY
	void setActivated(boolean visible)	ACTIVATE
	boolean isActivated()	ACTIVATE
	void setEnabled(boolean enable)	ENABLE
	boolean isEnabled()	ENABLE
android.view.MenuItem	int getItemId()	GETID
android.widget.CompundButton	Object findViewById(int id)	CMPBTN_DEF
	void setToggled(boolean toggle)	CMPBTN_TOGGLE
	int isToggled()	CMPBTN_TOGGLE
	void setChecked(boolean check)	MENU_CHECKED
	boolean isChecked()	MENU_CHECKED
android.widget.NumberPicker	int getValue()	NP_VALUE
android.widget.RadioGroup	void check(int checked)	RADIOBUTTON_CHECK
	int getCheckedRadioButtonId()	RADIOBUTTON_CHECK
	void setAction(int action)	MOTION_ACTION
android.view.MotionEvent	int getAction()	MOTION_ACTION

corresponding to the widget attributes accessed in reachable paths to the crash point. Eventually, our proposed approach ensures that all the found paths are feasible at run time since we collect

1) all the feasible paths to the crash point and
2) all the feasible paths to the def-sites of every widget attribute satisfying at least one of the conditions at the use-site of the same widget attribute in the feasible paths found in 1.

Hence, our analysis does not have any false negatives regarding the identification of necessary callbacks based on the collected widget attributes to trigger the crash point.

More importantly, due to the finite number of widget attributes in Android, the analysis time is greatly reduced compared to the time when all possible execution orders of UI events are considered. If there exist w widgets, each of which has k attributes on average with a numerical domain $\mathbb{Z}$, the number of iterations in our proposed reachability analyses is $|\mathbb{Z}|^{wk}$ in the worst case. In practice, our proposed analysis performs much fewer iterations since there are few conditional operations on the attributes of the widgets in Android. In addition, it eventually *terminates* due to the finite number of widgets, attributes, and attribute domains.

D. Widget Attribute Specification

In Android, `Widget` and `View` are two classes with different API functions for creating various types of graphical user interface elements (e.g., widgets and views). Some API functions implemented in these classes are used for setting or accessing the attributes of the widgets, as listed in Table IV. Since more than one variable is declared for some attributes, inferring the value state of an attribute from the variables with value-based static analysis is not easy. It requires knowing the logic behind the relation between the variables and their effect on the state of the attribute. This makes the automatic inference of them complicated. This information is not documented, and manual or automatic inspection is needed, which is out of the scope of our paper. Due to the finite number of widgets and attributes, we list all the attributes of the existing widgets and views in Android. We collected these API functions by studying the Android Development Documentations [27] for all attributes of each category of widget and view and collected the setting and accessing API functions. The attributes are all listed under the attribute category for each of them. The complete list of them is available in Tables X–XXII in online Appendix. Then, we design a simple specification to store the attributes and API functions declared for setting and accessing their values.

Our specification maintains a relationship between the widget type and its settings and accessing functions. We use the relation $attr(f) : f \rightarrow Attr$ to map the attributes with existing API function f used for setting or accessing its value state. For example, the `VISIBILITY` attribute is set via `setVisibility` and accessed by calling `getVisibility` functions. In addition, an attribute $Attr$ has one or more functions f used for setting or accessing its value state. Regarding the functions' roles, we classify them into two groups of *setting* and *accessing*. The first group refers to the functions that set the state of an attribute. The second group contains the functions that access the value state of an attribute. We use the function $type(f) : f \rightarrow \{1, 0\}$ to determine the role of a given function f. The output of the function $type$ is one or zero, indicating the setting or accessing kinds of a function.

E. Instrumentation

The collected configurations contain the necessary attribute states for reaching the target point. They are accumulated in the c_{entry} of the first statement in each callback. Since a callback function is the largest context for handling an event on a widget, we inject suitable code at the beginning of each callback to enforce the execution of necessary events. Based on the number of events a callback handles, there are two possible scenarios for pruning the irrelevant paths:

- If a callback handles only one event and the reachability state of its c_{entry} is false, or there are no target statements within its body, it is labeled as irrelevant. So, it is pruned by injecting an `assertTrue(false)` statement at the beginning of it. Otherwise, an `assertTrue(true)` is injected at the beginning of it to enforce its execution.
- If a callback handles several events on a grouped set of widgets, such as a *menu*, it is essential to identify the necessary and irrelevant ones in its context. The attribute *GETID* is used to distinguish between the events in a callback. Therefore, the set of collected states on this attribute helps us differentiate between them in terms of their necessity and relevancy. As events occur independently at run time, we leverage the disjunction operator between them and inject `assertTrue(cond)` at the beginning of the callback. The `cond` is a boolean expression that checks if the entered event is equal to the necessary ones and not equal to the irrelevant ones.

Apart from the UI callbacks used for handling the events, components include callbacks designed to handle different execution states. Since a component can only function correctly through the execution of all of its callbacks, all of a component's callbacks are regarded as necessary if at least one of those callbacks is reachable. On the contrary, if there is no reachable statement in a dummy main method of a component, all of its callbacks are regarded as irrelevant. Therefore, we use `assertTrue(false)` at the beginning of the callback functions of an unreachable component to reject its execution at run time.

V. Evaluation

This part aims to evaluate the effectiveness of MOLE in the reproduction process. In addition, it identifies any potential flaws that could compromise the precision and effectiveness of applying the static analysis and fuzzing process perspectives. To do so, we formulated the following questions:

- **RQ1:** How *effective* is MOLE for speeding up the crash reproduction compared to the baseline fuzzing tools?
- **RQ2:** How *common* are the widget dependencies imposed by collected widget attributes in Android applications?
- **RQ3:** What is the *accuracy* of MOLE in statically and dynamically detecting the reachable paths to the crash point?
- **RQ4:** How *scalable* is our proposed attribute-sensitive reachability analysis?
- **RQ5:** What is the *overhead* imposed by MOLE's instrumentation at run time?

A. Implementation

To perform the designated reachability analysis, MOLE is implemented over Vasco [28], a highly precise flow and context-sensitive non-distributive value-flow analysis in the presence of recursion. We leveraged FlowDroid [23] to construct the inter-procedural control flow graph of Android apps. We extended the list of callbacks in FlowDroid and used the SPARK algorithm to construct an app's call graph within a one-hour timeout. We assume the widget dependencies are imposed by using API functions, not data binding[8] [29]. We found 176 attributes in Android Development documentation [27]. We collected the setting and accessing API functions for widgets and views under `android.widget` and `android.view` packages in these documents. The complete list of them is available in the online Appendix.

Since setting or accessing attributes are widely performed programmatically through API functions rather than using data binding [29] in Android, our focus is on the analysis of attribute value states through API function calls. However, our solution also applies to the case when data binding is employed. In such a case, a pre-analysis is required to identify the variables used for each attribute in XML layout files and identify them in the app code as def-site or use-site.

B. Evaluation Setup

Benchmark: Since MOLE accepts information about the crash point from the crash stack trace, we selected Themis [24] benchmark for assessing our solution. This benchmark contains 52 critical crashes with their reproduction steps, crash stack traces, and the *apk* files of the target app. More importantly, the crashes listed in this benchmark vary in terms of the number of events that can be reproduced and the number of dependent and independent widgets among the events. In our study, only 20/52 of these crashes were successfully analyzed with MOLE. We explained the reasons for the failure of our tool in the analysis of the rest of the crashes in the results discussed for **RQ3**. We provided the list of these crashes, including the app's name, issue ID, and whether it is reproduced after analysis with our tool with at least one of the fuzzing tools we employed in Table V.

TABLE V
LIST OF THE CRASHES IN THEMIS THAT WE USED IN OUR EVALUATION. WE USE × AND ✓ TO SHOW IF THE CRASH WAS REPRODUCED OR NOT IN OUR EVALUATION

Application Name	Issue ID	Reproduced?
ActivityDiary	118	×
ActivityDiary	285	✓
AmazeFileManager	1558	✓
AmazeFileManager	1656	✓
AmazeFileManager	1796	✓
andBible	375	✓
AnkiDroid	4200	✓
AnkiDroid	4707	✓
AnkiDroid	4451	✓
AnkiDroid	5756	✓
AnkiDroid	4977	✓
AnkiDroid	6145	×
APhotoManager	116	✓
FirefoxLite	4881	✓
FirefoxLite	5085	✓
FirefoxLite	4992	×
OpenLauncher	67	✓
Osmeditor	637	×
Phonograph	112	×
ODC Collect	3222	×

Fuzzing tools Setup: We chose the best four coverage-based fuzzing tools: Monkey [25], Ape [5], Stoat [6], and FastBot2 [13] according to Themis [24] and FastBot2's [13] evaluation results from effectiveness and performance point of view for event generation. We also employed GoalExplorer [22] as the baseline, the only directed-based solution used for fuzzing an app to reach a target point to reproduce the crashes listed in our selected benchmark. We configured all the fuzzing tools to generate the event in the 6 hours continuously. For Monkey [25], Ape [5], and FastBot2 [13], we choose the default 200 and 500-millisecond delay between generating the events. Since Stoat [6] constructs a GUI model before the fuzzing process, we set 1 hour for the model construction and 5 hours for the event generation for it.

Environment Setting: For the attribute-sensitive analysis, we used a 64-bit Ubuntu 23.04 machine (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz and 24GB RAM) with a 20G Java heap size. For the fuzzing process, we ran the apps containing the crashes on Google Android emulators (Android 7.1 with API level 25) deployed on a 64-bit Ubuntu 18.04 machine

[8]Data Binding Library is a support library that allows developers to bind data sources to UI components in layouts using a declarative format rather than doing it programmatically.

(80 cores, Intel(R) Xeon(R) CPU E5-2698, and 250GB RAM). Each emulator is configured with 2GB RAM, 1GB SDCard, 1GB internal storage, and an X86 ABI image similar to the setting used by Themis [24].

C. RQ1: How Effective Is MOLE for Speeding Up the Crash Reproduction Compared to the Baseline Fuzzing Tools?

In order to evaluate the effectiveness of our approach, we leveraged two performance measures based on the best practices in a recent study [30]:

1) success ratio
2) reproduction time

First, we used the success ratio to assess the effectiveness of MOLE in reproducing the crashes. The success ratio is the proportion of successful runs in which a crash is reproduced among all the performed runs. Hence, it demonstrates the efficacy of our approach regarding non-determinism in the ample event space in Android apps. Since the success ratio follows a binomial distribution, we used the Fisher exact test to assess the enhancement of crash reproduction with our solution [31]. Hence, we performed ten runs with each fuzzing tool for each crash and compared our findings. We considered three different scenarios, in which,

- no Reachability Analysis (baseline),
- an Attribute-insensitive Reachability Analysis (RA), and
- an Attribute-sensitive Reachability Analysis (ASRA)

is conducted on an application before the fuzzing process. We used s_b, s_{RA} and s_{ASRA} to show the success ratio of each of the three scenarios correspondingly. We measured $p-value_1$ and $p-value_2$ in the Fisher exact test to assess the impact of ASRA with baseline and RA, respectively, in the reproduction process.

Second, we measured the crash reproduction time to evaluate the effect of our proposed approach on speeding up the reproduction process. We used t_b, t_r, and t_{Mole} to present the reproduction time when only baseline, RA, and ASRA were employed, correspondingly. Since the reproduction time does not follow a normal distribution, we used the Wilcoxon Mann-Whitney (WMW) test. Wilcoxon Mann-Whitney is a non-parametric statistical test that aims at comparing the distribution of two different samples [30], [31]. We measured $p-value_3$ and $p-value_4$ in this test to show the significance of our tool in decreasing the reproduction time compared to the crash reproduction process with baseline and RA. We also leveraged Delany Vargha A_{12} measure to show the effect size for the difference in every two samples of our tool with baseline and RA.

Results (Coverage-based Fuzzing Tools): Table VI presents the success ratio and the reproduction time of the crash reproduction in all three scenarios when using the coverage-based fuzzing tools. We highlight in bold the cases in which s_{ASRA} has increased in Table VI. We performed the Fisher exact test to assess the enhancement of s_{ASRA} with s_b and s_{RA}. We highlight the cases where s_{ASRA} is better than s_b with green color in Table VI. Similarly, we use yellow to show the cases where both ASRA and RA are indistinguishable in terms of their success ratio. Accordingly, there are only 9/25 cases in which MOLE has significantly enhanced the success ratio. Although the Fisher exact test shows not much difference between ASRA and RA, the reproduction time greatly decreased when MOLE was employed.

In addition, we utilized the average and median reproduction time to assess the crash reproduction speed. As shown in Table VI, the average reproduction time for all the crashes decreased when MOLE was employed when at least one fuzzing tool was chosen, which is highlighted with green color. Based on the results of the WMT test, $p-value_3$ and $p-value_4$ show that the reproduction was greatly enhanced with MOLE except in five cases. For issues `4977`, `116`, and `4881` in `AnkiDroid`, `AphotoManager` and `FirefoxLite`, the overhead imposed by MOLE negatively affects the reproduction time. In addition, there is more divergence between reproduction time in different runs due to the large number of widgets and their state transitions in `ActivityDiary` app for the crash with issue number `285`. Moreover, we calculated $A_{12_{RA}}$ and $A_{12_{ASRA}}$ to show the effect size of using ASRA compared to only using baseline or RA, respectively. If both scenarios are equivalent, A_{12} must be equal to 0.5. However, if the former yields a decrease in reproduction time, this measure will be higher than 0.5. Regarding our results, this measure is promising for all the cases where the crash is reproduced with MOLE except the five mentioned cases.

Based on our experiments, we noticed that the crash reproduction process had a higher success ratio when Monkey [25] was chosen compared to model-based fuzzing solutions. We observed that the high rate of events generated by Monkey are responded to by the app under analysis. This yields higher event coverage, increasing the possibility of triggering a crash again.

To explain this phenomenon, we used two other metrics: the average generated event rate ($event_g$) and the respondent event rate ($event_r$). The average generated event rate is the number of events a fuzzer generates per minute. The average respondent event rate indicates the number of events that an app successfully responds to in one minute. Since the selected fuzzing tools do not pause an app and generate events one after another immediately, an event can be skipped if the execution of the former lasts longer. Hence, the average generated and respondent event rates are not equal. The more events received in a shorter time, the more event coverage can be achieved, resulting in a better search process.

Monkey [25] outperforms model-based fuzzing solutions as its respondent event rate is higher. This tool generates random events within the same time interval. In contrast, the model-based fuzzing tools have a lower $event_g$, which results in lower $event_r$. Rather than immediately generating an event, these tools dynamically construct or update a GUI model. They retrieve an app's dynamic GUI model and construct/update it to generate the next event. This overhead decreases both the generated event and respondent event rates, affecting the reproduction process negatively. In Tables VIII and IX, we show both $event_g$ and $event_r$ for all the selected fuzzing tools when time intervals $200ms$ and $500ms$ were set.

Results (Directed Fuzzing Tool): Apart from the GUI fuzzing solutions, we leveraged GoalExplorer [22], one of the

TABLE VI
THE THREE VARIABLES s_b, s_{RA}, AND s_{ASRA} SHOW THE SUCCESS RATIO IN THREE SCENARIOS OF BASELINE, RA, AND ASRA. WE USE t_b, t_r AND t_{Mole} TO SHOW BOTH THE AVERAGE (AVG.) AND MEDIAN CRASH REPRODUCTION TIME FOR THE CRASHES REPRODUCED BY MONKEY (M), APE (A), STOAT (S) AND FASTBOT2 (F) WHEN BASELINE, RA OR ASRA WAS PERFORMED

Application Name	Issue ID	Fuzzing Tool	success ratio					Reproduction Time (hour)									
						Fisher exact		t_b		t_r		t_{Mole}		Wilcoxon Mann-Whitney			
			s_b	s_{RA}	s_{Mole}	$p-value_1$	$p-value_2$	Avg.	Median	Avg.	Median	Avg.	Median	$p-value_3$	$A_{12_{RA}}$	$p-value_4$	$A_{12_{ASRA}}$
ActivityDiary	285	**M**	0	0.9	1	$0.5e-5$	0.5	$\times$	$\times$	5.2	5.1	4.4	4.5	$0.6e-3$	0.5	$0.1e-3$	0.5
		A	0	0.4	0.1	0.13	0.5	$\times$	$\times$	3.81	6	5.4	6	0.1	0.3	0.7	0.45
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
AmazeFileManager	1558	**M**	0	1	1	$0.5e-5$	1	$\times$	$\times$	0.2	0.15	0.03	0.01	$0.1e-3$	0	$0.1e-3$	0
		A	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
AmazeFileManager	1656	**M**	0	0.6	1	$0.5e-5$	0.02	$\times$	$\times$	3.19	2	0.03	0.03	0.02	0.2	$0.1e-3$	0
		A	0	0.4	1	$0.5e-5$	0.04	$\times$	$\times$	3.38	2	0.04	0.03	0.02	0	$0.1e-3$	0.2
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0.8	$0.3e-3$	$0.3e-3$	$\times$	$\times$	$\times$	$\times$	2.1	1.15	1	0.5	$0.2e-2$	0.1
AmazeFileManager	1796	**M**	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		A	1	0.4	0.8	0.2	0.07	1.41	1.41	3.63	6	1.76	0.91	0.4	0.6	0.02	0.2
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0.3	0.4	1	$0.1e-2$	$0.5e-2$	4.4	6	3.64	6	0.13	0.13	0.4	0.4	$0.2e-2$	0.01
andBible	375	**M**	0	0.7	0.9	$0.5e-5$	0.24	$\times$	$\times$	2.78	1.5	1.65	1.15	$0.6e-2$	0.05	$0.8e-3$	0.05
		A	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
AnkiDroid	4200	**M**	0	0	0	1	0.5	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		A	0	0.2	0.8	$0.3e-4$	0.04	$\times$	$\times$	5.6	6	5.7	5.9	0.4	0.4	$0.2e-2$	0.25
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
AnkiDroid	4707	**M**	0	0.7	1	$0.5e-5$	0.1	$\times$	$\times$	3.44	0.13	1.33	0.66	$0.8e-2$	0.15	$0.1e-4$	0
		A	1	0.7	0.8	0.1	0.3	0.37	0.33	2.55	0.33	1.89	0.16	0.12	0.4	0.17	0.32
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0.1	0.5	0.9	$0.5e-3$	0.06	5.9	$\times$	4.13	5.13	2.32	2	0.08	0.27	$0.7e-3$	0.05
AnkiDroid	4451	**M**	0	0.9	0.9	$0.5e-5$	0.5	$\times$	$\times$	5.16	5.16	2.5	2.25	$0.6e-3$	0.05	$0.6e-3$	0.05
		A	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
AnkiDroid	5756	**M**	1	0.9	1	1	0.5	2.41	2.41	3.31	3.04	1.89	1.83	0.06	0.74	0.02	0.2
		A	1	1	0.6	0.04	0.04	0.35	0.33	4.02	0.2	2.84	0.09	0.5	0.6	$0.2e-3$	0.98
		S	1	1	0.8	0.2	0.2	1.35	1	1.3	1.3	2.11	1.3	0.4	0.6	0.4	0.6
		F	0.4	0.5	1	$0.5e-2$	0.01	3.8	6	3.51	3.75	1.2	2.1	0.1	0.5	0.1	0.29
AnkiDroid	4977	**M**	0	1	1	1	1	0.19	0.19	0.1	0.05	0.04	0.04	0.03	0.22	0.02	0.4
		A	1	1	1	1	1	0.08	0.08	0.16	0.09	0.1	0.06	0.6	0.63	0.8	0.57
		S	0.9	1	1	0.5	1	1.85	1.6	0.72	0.79	0.01	0.1	$0.3e-2$	0.19	0.01	0.11
		F	0	0.4	0.6	$0.5e-2$	0.2	$\times$	$\times$	4	6	2.8	2.1	0.1	0.3	0.02	0.2
APhotoManager	116	**M**	1	1	1	1	1	0.01	0.01	0.09	0.07	0.04	0.03	0.02	0.8	$0.8e-2$	0.8
		A	1	1	1	1	1	0.03	0.03	0.03	0.03	0.03	0.03	0.2	0.5	1	0.5
		S	0	1	1	$0.5e-4$	1	$\times$	$\times$	1.17	1.08	0.54	0.1	$0.1e-3$	0	$0.1e-3$	0
		F	1	1	1	1	1	0.01	0.01	0.02	0.03	0.02	0.03	0.05	0.75	0.05	0.75
FirefoxLite	4881	**M**	1	1	1	1	1	0.01	0.01	0.09	0.07	0.01	0.01	$0.1e-3$	0.5	$0.1e-3$	0.5
		A	1	1	1	1	1	0.03	0.03	0.03	0.03	0.03	0.03	0.1	0.03	1	0.5
		S	0	0	1	$0.5e-4$	$0.5e-4$	$\times$	$\times$	$\times$	$\times$	0.03	0.03	1	0.5	$0.1e-3$	0
		F	1	1	1	1	1	0.01	0.01	0.03	0.03	0.03	0.03	1	1	$0.1e-4$	1
FirefoxLite	5085	**M**	0	1	0.9	$0.5e-5$	0.5	$\times$	$\times$	1.64	1.66	1.27	0.74	$0.1e-3$	0	$0.6e-3$	0.05
		A	1	0.7	0.8	0.2	0.3	1.31	1.33	2.25	6	1.75	0.81	0.1	0.3	0.02	0.2
		S	0	0.1	0.9	$0.5e-4$	$0.5e-4$	$\times$	$\times$	$\times$	$\times$	1.14	1.15	1	0.5	$0.6e-4$	0.05
		F	0.3	0.4	1	$0.1e-2$	$0.5e-2$	4.5	6	3.82	6	1.85	0.45	0.4	0.39	$0.4e-2$	0.12
OpenLauncher	67	**M**	0	1	1	1	1	$\times$	$\times$	3.09	3.33	1.46	1.66	$0.5e-4$	0	$0.5e-4$	0
		A	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		S	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5
		F	0	0	0	1	1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1	0.5	1	0.5

[*] The symbol $\times$ indicates that the crash was not reproduced in all the ten runs.
[**] We use value 6 for the median or average reproduction time when the crash is not reproduced within our desired period.
[***] We highlight cases in which Mole is better than baseline and RA with green. We use yellow to show the other reproduced crashes that Mole and RA are not much different.

directed fuzzing approaches, to compare the effectiveness of our tool. Since GoalExplorer performs an intra-procedural static analysis only on the callback functions in an app's components, it can only locate crashes in which the crash point exists in such a callback function context. However, none of the crash points in the Themis benchmark were located in the context of the components' callbacks except for one crash (the crash occurred in `APhototManager` app). Therefore, this tool could only reproduce one crash in the selected benchmark.

D. RQ2: How Common Are the Widget Dependencies Imposed by Collected Widget Attributes in Android Applications?

To answer this question, we enumerated the widget attribute states found in the reachable paths to the listed crash points in Table V.

Results: We studied Android development documentation and collected a list of 176 API functions used for accessing and setting attribute values. Fig. 7 shows the percentages of

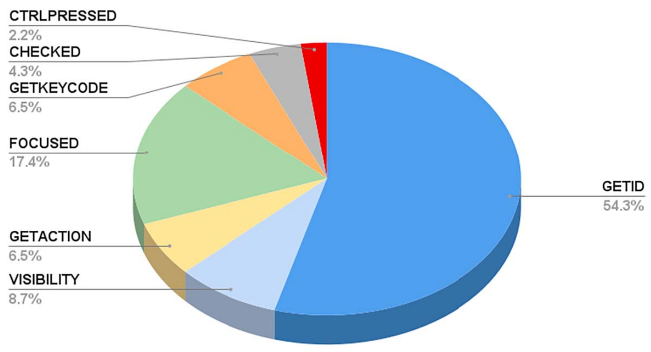


Fig. 7. Percentages of the widget attributes found by MOLE.

the widget attributes found in the crashes listed in Table V. Based on this figure, the attributes `GETID`, `FOCUSED`, and `VISIBILITY` are highly used. The former is for checking the widget ID in a callback to handle an event on it. The second is used to indicate that a user is interacting with the widget. The latter is used to control the visibility state of a widget. Since there is no ground truth, we cannot assess the sufficiency of the list of our collected attributes.

E. RQ3: What Is the Accuracy of MOLE in Statically and Dynamically Detecting the Reachable Paths to the Crash Point?

According to our results, MOLE could successfully perform the attribute-sensitive analysis on the 20/52 crashes in the Themis [24] benchmark. Notably, existing fuzzing tools failed to reproduce 6 of these 20 crashes. To validate the accuracy of our static analysis, we manually verified the reachable paths to the crash point for each crash. In the following sections, we delve into our tool's limitations and explore the sources of inaccuracy in both static and dynamic analysis.

Results: Below, we first enumerate the reasons behind the failure of MOLE in successfully performing ASRA on all the crashes listed in Themis [24] benchmark. Next, we discuss why the selected fuzzing tools could not reproduce some of the crashes listed in Table V.

MOLE *False Negative:* Our proposed tool could not successfully perform the attribute-sensitive reachability analysis on all the apps due to the following reasons:

1) Value-flow Analysis Limitations: While our approach performs inter-procedural value flow analysis, we only employ intra-procedural pointer analysis to identify the value of a variable, rather than a full-fledged inter-procedural value flow analysis (e.g., issue 112 in `Phonograph`).
2) Unsound ICFG: MOLE is dependent on the underlying ICFG construction algorithm for locating the crash points. For instance, it does not model the callback execution order for some Android objects (e.g., callbacks in `RecycleViewAdapter` and `Fragment`) in its call graph construction, and the crashes located in such callbacks are unreachable. Twenty-nine crashes were having this problem during performing ASRA (e.g., issue 1918 in `NextCloud`).
3) Library Code: MOLE only searches the app code while some crash points are located in the library code (e.g., the crash with issue ID 1837 in `AmazeFileManager` and 745 in `OmniNote`).

Fuzzing Process False Negative: According to Table V, only six crashes were not reproduced with the selected fuzzing tools due to the reasons enumerated below:

- Independent Widgets: When no dependent widgets are involved in triggering a crash, using our tool does not enhance the reproduction time. It results in a smaller number of irrelevant events. For instance, the crashes with issue numbers 112, 3222, and 4942 in `Phonograph`, `ODC collect`, and `FirefoxLite` did not have dependent widgets in triggering the crash. For `FirefoxLite` app having the crash with issue ID equal to 4992, the other case was held. In this crash, the crash point was located in an `onCreate` method in the app code, and no widget attribute state was available in the reachable paths to this point. Hence, the proportion of irrelevant events to the necessary ones was low. Therefore, a small part of the app code was pruned, our analysis is equivalent to an attribute-insensitive reachability analysis, and the event explosion problem is still held.
- Input Values: Some of the crashes are triggered when *specific input types (numerical, text) are entered.* Since the selected fuzzing tools generate GUI events only, they are incapable of triggering such crashes. For instance, the crash with an issue number equal to 637 in `Osmeditor` needs a specific value of '*' and '0' and cannot be reproduced by chosen GUI fuzzing tools. The crash with issue number 118 in `ActivityDiary` also required a photo as input to be uploaded to the app.
- Android Environment Setting: Some crashes are triggered under specific settings of an Android device, while selected fuzzing tools do not configure or fuzz environment settings. For instance, it is essential to change the language setting to a specific language to reproduce the crash with issue number 6145 in `AnkiDroid` app.

Moreover, we also observed that Ape [5] throws an error upon starting the fuzzing process on some apps like `Phonograph`, `OpenLauncher`, and `and-bible`.

F. RQ4: How Scalable Is Our Proposed Attribute-Sensitive Reachability Analysis?

To evaluate the scalability of our proposed attribute-sensitive reachability analysis, we measured the execution time for the benchmark apps, excluding the time taken for call graph construction.

Results: According to Table VII, our analysis takes 11 minutes in the worst case on `andBible` app with 500K lines of codes finding at most 74 widget attributes. Notably, the backward reachability analysis prunes a significant portion of

TABLE VII
ATTRIBUTE-SENSITIVE REACHABILITY ANALYSIS RESULTS

Application Name	Issue ID	Code size	Installs	% Necessary Components	% Necessary Callbacks	% Pruned Instructions	# Found Attributes	Analysis time (sec)
ActivityDiary	118	6K	1K	33	71	67	14	16
ActivityDiary	285	6K	1K	33	**65**	**59**	**26**	50
AmazeFileManager	1558	61K	1M	17	**82**	**85**	**12**	117
AmazeFileManager	1656	61K	1M	17	**82**	**85**	**12**	494
AmazeFileManager	1796	61K	1M	17	**80**	**85**	**240**	303
And-bible	375	32K	500K	29	**97**	**91**	**74**	653
AnkiDroid	4200	106K	10M	27	**7**	**98**	**44**	106
AnkiDroid	4451	106K	10M	27	**71**	**89**	**57**	204
AnkiDroid	4707	106K	10M	27	**48**	**81**	**44**	207
AnkiDroid	4977	106K	10M	28	88	65	63	127
AnkiDroid	5756	106K	10M	27	82	98	26	163
AnkiDroid	6145	106K	10M	27	50	80	70	252
AphotoManager	116	36K	-	33	100	76	53	54
FirefoxLite	4881	45K	100M	25	73	91	0	138
FirefoxLite	4942	45K	100M	19	50	15	0	122
FirefoxLite	5085	45K	100M	27	**74**	**92**	**35**	157
Openlauncher	67	45K	100M	25	81	87	20	23
Osmeditor	637	300K	50K	30	56	87	11	181
Phonograph	112	117K	1M	29	75	84	0	78
ODC Collect	3222	200K	1M	5	20	10	0	1962
Average				19	67.25	67.1	49.25	265.05
Min				5	20	10	0	16
Max				33	100	98	240	1962

[*] Here, code size is measured in a number of lines of code.

the application's code, eliminating an average of 81% of components. Consequently, integrating attribute sensitivity can enhance precision while maintaining scalability.

G. RQ5: What Is the Overhead of MOLE's Instrumentation at Run Time?

To quantify the runtime overhead of pruning irrelevant events, we measured the average time required to execute instrumented instructions. We utilized two metrics, $event_g$ and $event_r$, to gain a deeper understanding of the instrumentation's overhead. Specifically, $event_g$ represents the number of events generated by the fuzzer per minute, while $event_r$ denotes the number of events successfully received by the app within the same timeframe. The instrumentation overhead can cause significant delays in event execution, leading to app unresponsiveness and potential event loss. In turn, it can result in a disparity between $event_g$ and $event_r$, highlighting the negative impact of instrumentation on the app's runtime and fuzzing process.

Results: Our findings reveal that the overhead of checking an event varies from 0.005 ms to 2 seconds. This vast difference comes from the fact that a callback can handle more than one event. Hence, the number of instructions our tool injects in a callback is proportional to the number of events it handles. Moreover, existing fuzzing tools do not wait for an event to complete before generating the next one. This results in a significant disparity between the number of received events ($event_r$) and generated events ($event_g$). With this assumption, the overhead imposed on an event responded to by an app increases, affecting the fuzzing process efficiency.

Tables VIII and IX display the values of $event_r$ and $event_g$ for the selected fuzzing tools. According to these tables, there is a subtle difference between $event_g$ and $event_r$ in model-based fuzzing solutions like Stoat and FastBot2 for most of the crashes when we use longer time intervals $500ms$. However, there is a considerable discrepancy between $event_g$ and $event_r$ when a shorter time interval of $200ms$ is chosen when the same fuzzing tools were employed. For instance, the values of

TABLE VIII
AVERAGE (AVG.) RATE OF GENERATED EVENTS AND THE RESPONDENT EVENTS PER MINUTE WHEN THE TIME INTERVAL IS $200ms$ FOR GENERATING EVENTS. THE VARIANCE OF THE RATES FOR FIVE RUNS IS INCLUDED UNDER COLUMNS σ^2

Application Name	Issue ID	Monkey				Ape				Stoat				FastBot2			
		$event_g$		$event_r$		$event_g$		$event_r$		$event_g$		$event_r$		$event_g$		$event_r$	
		Avg.	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2
ActivityDiary	118	239	2	31	0.2	50	23	23	7	30	0.3	7	0.01	124	3	20	11
ActivityDiary	285	234	300	8	1	59	33	2.5	0.4	10	0.1	8	1	212	260	9	0.4
AmazeFilemnager	1558	74	27	13	7	49	25	31	14	9	28	4.5	0.04	114	2	60	131
AmazeFileManager	1656	171	727	71	131	52	3	31	22	56	190	15	0.5	134	6	66	73
AmazeFileManager	1796	64	1	5.5	7	52	30	33	44	7	0.1	6	5	141	12	50	190
and-bible	375	215	3	1	0	10	0.2	1.5	1	0.1	≈ 0[*]	0.1	≈ 0[*]	20	0.7	0.4	≈ 0[*]
AnkiDroid	4200	232	23.7	51	88	**50**	1	**23**	5	**57**	7	**5**	6	**131**	6.5	**27**	2.5
AnkiDroid	4707	234	14	143	46	**51**	13	**24**	2	**133**	195	**5**	0.2	**81.5**	8	**22**	5
AnkiDroid	4451	215	38	112	12	**50**	6	**27**	3	**109**	67	**11**	1	**148**	3	**39**	52
AnkiDroid	5756	205	33	17	5	**51**	9	**24.1**	34	**15**	9	**7**	0.05	**132.6**	5	**26**	23
AnkiDroid	4977	208	253	127	100	**52**	6	**37.**	6.4	**105**	363	**10.4**	2	**147**	3.5	**95**	88.5
AnkiDroid	6145	204.4	11	119	12.5	**52**	5	**32**	10	**52**	5	**4.4**	0.6	**150**	7	**97**	179
APhotoManager	116	229	129	70	80	46	5.5	48	21	64	3	6.5	0.02	146	15	26	117
FirefoxLite	4881	80	433	5	2	**22**	6	**14**	8	**47.5**	18	**8**	0.2	**107**	7	**17**	20
FirefoxLite	5085	169	266	57	55	**19**	3	**20**	1	**57**	51	**20**	11	**83.5**	17	**25**	6
FirefoxLite	4992	40	11	5.5	0.02	-	-	-	-	**144.5**	230	**7**	0.6	**57**	2	**50**	0.2
Openlauncher	67	250.4	504	210	542	60	10	32	1	212	504	7	0.2	242	8	114	1114
Osmeditor	637	249	25	156	510	49	7	31.5	174	164	318	13	1	132	9	101	4
Phonograph	112	49	41	2	0.02	-	-	-	-	52	4	6	0.5	53	2	21	0.4
ODC Collect	3222	45	30	51	144	32	3	14	5	22.7	≈ 0[*]	5	≈ 0[*]	140	28	52	136

[*] Here, ≈ 0 denotes values less than 0.009.
[**] We use '-' for issues 4992 and 112 in FirefoxLite and Phonograph apps since Ape throws an error upon analysis of them.

TABLE IX
AVERAGE (AVG.) RATE OF GENERATED EVENTS AND THE RESPONDENT EVENTS PER MINUTE WHEN THE TIME INTERVAL IS $500ms$ FOR GENERATING EVENTS. THE VARIANCE OF THE RATES FOR FIVE RUNSIS INCLUDED UNDER COLUMNS σ^2

Application Name	Issue ID	Monkey				Ape				Stoat				FastBot2			
		$event_g$		$event_r$		$event_g$		$event_r$		$event_g$		$event_r$		$event_g$		$event_r$	
		Avg.	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2	*Avg.*	σ^2
ActivityDiary	118	103	0.5	13	0.1	44	7	22	3	6	0.1	2	0.01	103	35	13	4
ActivityDiary	285	105	0.5	4	0.04	25	24	24	25	4.5	4	2	0.4	102	53	10	2
AmazeFilemnager	1558	88	117	69	491	42	1.4	26	5	3	0.4	3	0.02	84	10	56	80
AmazeFileManager	1656	79	361	75	637	44	22	42	3	4	1	2.5	0.2	69	6	62	4.6
AmazeFileManager	1796	56	29	13	0.5	46	28	42	3	4	0.2	2.5	0	64	11	55	23
and-bible	375	215	3	1	0	10	0.2	1.5	1.4	0.1	≈ 0[*]	0.1	≈ 0[*]	20	1	0.4	≈ 0[*]
AnkiDroid	4200	101	2	20	5	**53**	10	**25**	5	**5**	0.02	**5**	0.03	**58**	3	**19**	2
AnkiDroid	4707	102	2.4	53	25	**27**	4	**25**	2	**5**	1	**1**	0.14	**100**	44.5	**22**	0.16
AnkiDroid	4451	97	9	48	23	**51**	3	**24**	5.4	**6.5**	0.02	**5**	≈ 0[*]	**81**	21	**57**	62
AnkiDroid	5756	96	10	5	3.4	**46**	0.1	**25**	1	**6**	0.1	**6**	0.02	95	17	**22**	10
AnkiDroid	4977	98	20	51	31	**49**	21	**31**	13	**6.5**	0.03	**5**	0.1	**109**	47	**62**	8
AnkiDroid	6145	93	51	52	3	**47**	4	**17**	3	**15.4**	4	**1.4**	0.2	**97**	6	**57**	5
APhotoManager	116	99	1	28	8	42.4	17	51	2	6	0.05	3	0.05	120	82	38	7
FirefoxLite	4881	56	10.5	4	0.06	**25**	34.5	**10**	0	**3**	0.1	**2**	≈ 0[*]	**79**	10.5	**17**	2
FirefoxLite	5085	67	7	9	2.4	**45**	1	**8**	107	**3**	1	**1.5**	1	**103**	137	**23**	5
FirefoxLite	4992	33	1	5	0.02	43	2.5	39	0.2	**3**	≈ 0[*]	**2**	0.04	**98**	4	**45**	8
Openlauncher	67	104	0.5	77	19	56	3	47	2	5	0.01	1.6	0.1	106	9	57	1
Osmeditor	637	103	1	20	9	5	8	49	12.5	4.5	0.05	3	0.3	88	37	89	21
Phonograph	112	40	15	3	14	-	-	-	-	19	0.6	5	0.06	58	4	20	0.1
ODC Collect	3222	37	1	4	0	27	0.6	26	0	6	0.01	2	0.5	54	30	50	188

[*] Here, ≈ 0 denotes values less than 0.009.
[**] Ape throws an error upon analysis of the Phonograph app for the issue numbered 116.

$event_g$ and $event_r$ are highlighted in bold with their variances in red color for AnkiDroid and FirefoxLite apps in Tables VIII and IX. Comparatively, the difference between $event_r$ and $event_g$ is negligible when Ape is utilized. This is likely due to Ape's dynamic GUI model updates, which impose a burden on the app's runtime. In general, Ape's overall $event_r$ does not significantly change due to the burden of dynamic GUI model construction. Interestingly, choosing a longer time interval (e.g., 500ms) allows for parallel event generation and model construction, resulting in reduced variance between the two metrics. This also can reduce the effect of MOLE's instrumentation overhead. In summary, the overall respondent event rate becomes slower from `1.05X` to `71X` on average when MOLE is employed. Although this burden is imposed on an app's run time execution, our pruning solution is highly effective, allowing the reproduction of 14/20 crashes.

H. Discussion

Future Work Our findings show that the crash reproduction time depends on the fuzzing tool's event generation rate and its searching strategy. In addition, the strategy used by the coverage-based testing tools is not aligned with selecting widgets involved in reproducing a crash. Hence, designing a suitable fuzzing tool that is compliant with the crash reproduction goal to select widgets concerning its effect on crash detection is an open research question in future work. Moreover, balancing the event rates and coverage is very appealing when precise GUI models are built dynamically. Current tools do not effectively generate events because they only use time intervals in event generation.

In Android, an app's functionalities can depend on its environment settings, such as network connection, Bluetooth, etc. In case changing the environment setting is required along with generating GUI events in a crash [32], our tool is incapable of reproducing it. An interesting future direction is identifying what setting has affected a crash and having the crash stack trace. There can be many possible combinations of GUI event sequences and setting changes to trigger a crash. When the setting is recognized, the next challenge is to effectively change the setting state alongside the GUI event generation.

Threats to Validity: The external threat to our solution is the inability of our approach to find a precise set of necessary and irrelevant events due to the unsoundness of the underlying call graph construction algorithm. In addition to this, the primary internal threat to the impact of our approach is the completeness of the API functions used. We attempted to check Android documentation to collect all possible widget attributes manually. However, if the Android community adds any new widget or attribute, our approach cannot consider it unless it is added to our specification. Moreover, we cannot find the widgets and their dependencies used by developers from third-party libraries. The last internal threat in our solution arises from not considering the data dependencies between other variables in the pruning process. In our design, we relied on the best practices in Android [27] that encourage developers to instantiate the variables in `onCreate` callbacks of components. Suppose we do not prune the callbacks regarding the data dependencies between all the variables used in the app. In that case, it is possible to confront other crashes or mistakenly reproduce a resolved crash.

VI. Related Work

Crash Reproduction in Android Apps: When a crash is reported in an issue, the user might report the steps to reproduce differently through written explanations, uploaded screenshots, or recorded videos of the steps. Inferring each resource to automate the crash reproduction has different challenges. The textually written report contains the type of events and widget names involved in the crash occurrence. ReCDroid [33], ReCDroid+ [34], Yakusu [35], and ReproBot [36] aim to process the human-readable sentences. They leverage Natural Language Processing (NLP) to identify the widgets and input values from the crash report. Then, they examine the app by synthesizing the events from the collected information, looking for the crash.

In these studies, the quality of the crash report plays a vital role in finding all the involved widgets. ReCDroid [33] and ReCDroid+ [34] only analyze the crash reports containing chronological crash reproduction steps. However, the crash reports are sometimes incomplete or contain linking words connecting sentences explaining the steps. ReproBot [36] attempted to solve the second case by designing a wiser NLP phase and using Reinforcement Learning (RL) in the dynamic exploration part. It leverages a temporal normalization step to refine sentences when linking words exist in the crash report. It also takes advantage of reinforcement learning (RL) in dynamic exploration to choose the best matches of widgets and events.

Apart from the written reports, visually taken resources like screenshots or recorded videos of reproduction steps are another kind of available resource. GIFDroid [37] is a recent study trying to reproduce crashes from the recorded video. Similar to written crash reports, the uploaded videos can be incomplete. GIFDroid [37] attempts to solve this issue by searching for the missing steps. It identifies the frames in a video and utilizes image processing to recognize the widgets. Then, it constructs a UI transition graph from the found frames and searches for the missing steps statically.

The advantage of our tool is that it relies on crash stack trace only. It enables us to reproduce the crashes for which there are no textual or visual reproduction steps. As we attempt to reduce the search space by identifying necessary events and pruning irrelevant events at run time, we can employ different event-generation techniques to explore this reduced event domain.

Event Sequencing: Static vs. Dynamic: Analyzing event-driven programs like Android apps presents a challenge due to the "event explosion" – the vast number of possible execution paths. FlowDroid [23] addresses this by constructing a *dummy main method* that over-approximates all possible execution orders of component lifecycle and UI callbacks. Several studies [38], [39], [40], [41] have explored methods to predict precise execution orders for events, leveraging variable and field properties in `C++`, `Java`, and Android. These approaches, including Staiger [38], Arlt et al. [39], TrimDroid [40], and Columbus [41], utilize def-use chains of variables to achieve this. For Android apps specifically, TrimDroid [40] expands on this by considering not only variables but also fields within widget-typed variables (e.g., the visibility attribute), recognizing dependencies between widgets.

QADroid [42] takes a different approach to finding precise execution orders between callbacks. It tracks the invocation of listeners associated with widget event handlers, predicting their execution order. Bloub et al. [43] further explore event dependencies by identifying conditional statements on widget-typed variables within callback function contexts and analyzing the order of event occurrence. Cider [44] is another static solution that aims to precisely identify the correct execution order of callbacks in the presence of compatibility issues in different Android versions. Our approach, however, focuses exclusively on the visual properties of Android apps to predict the execution orders of GUI widgets. We statically extract a general model by identifying dependencies between widgets based on the def-use chains of their attribute value states. This approach overcomes the imprecision and scalability issues often encountered in traditional sound and precise data flow analysis, which typically rely on a broader range of data.

Apart from static approaches, several studies [45], [46], [47] try to get the correct execution order of the events dynamically at run time. Compared to static solutions, using dynamic analysis benefits in collecting a precise set of event sequences. Learning from the collected event sequences can help find a precise execution order for callback functions. Existing solutions use random testing or design test cases to fuzz an app, collect the event traces, and automatically construct a precise model of them afterward.

VII. Conclusion

While directed fuzzing is commonly used to reproduce crashes in conventional programs; we presented MOLE, a direct-fuzzing solution for reproducing crashes in Android apps efficiently. Compared to existing fuzzing approaches, MOLE leverages a fast, sound, and precise attribute-sensitive reachability analysis to prune infeasible paths meticulously. Our reachability analysis aims to find feasible paths to the crash point by taking advantage of the widget attribute states to avoid confronting the scalability issue. Since widget attributes are used to add dependencies between widgets, finding them helps us to identify the irrelevant and necessary events involved in the crash occurrence. After identifying these event types, we can inject code to selectively explore the necessary events at runtime and filter out the irrelevant ones. As a result, the fuzzing tools can focus on a smaller set of relevant events, significantly reducing the time required for the crash reproduction process.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful and valuable comments.

References

[1] "Share of global smartphone shipments by operating system from 2014 to 2023." Statista. Accessed: May 2023. [Online]. Available: https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/
[2] D. Knott, "What mobile users expect from mobile apps." Applause. Accessed: May 2023. [Online]. Available: https://www.applause.com/blog/what-mobile-users-expect-from-mobile-apps
[3] J. Simic, "Here's what you need to know about mobile app bugs." Shakebugs. Accessed: May 2023. [Online]. Available: https://www.shakebugs.com/blog/mobile-app-bugs/
[4] "GitHub, Let's build from here." (2023) GitHub. Accessed: Nov. 3, 2023. [Online]. Available: https://github.com/
[5] T. Gu et al., "Practical GUI testing of android applications via model abstraction and refinement," in *Proc. 41st Int. Conf. Softw. Eng., (ICSE)*. Piscataway, NJ, USA: IEEE Press, 2019, pp. 269–280, doi: 10.1109/ICSE.2019.00042.
[6] T. Su et al., "Guided, stochastic model-based GUI testing of android apps," in *Proc. 11th Joint Meeting Found. Softw. Eng., (ESEC/FSE)*, New York, NY, USA: ACM, 2017, pp. 245–256, doi: 10.1145/3106237.3106298.
[7] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2019, pp. 1070–1073.
[8] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 481–492.
[9] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proc. 25th Int. Symp. Softw. Testing Anal., (ISSTA)* 2016. New York, NY, USA: ACM, 2016, pp. 94–105, doi: 10.1145/2931037.2931054.
[10] C. Peng, Z. Zhang, Z. Lv, and P. Yang, "MUBot: Learning to test large-scale commercial android apps like a human," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2022, pp. 543–552.
[11] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)" in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2015, pp. 429–440.
[12] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2019, pp. 102–114.
[13] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng., (ASE)*, New York, NY, USA: ACM, 2023, doi: 10.1145/3551349.3559505.
[14] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight UI-guided test input generator for android," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, 2017, pp. 23–26.
[15] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal., (ISSTA)*, New York, NY, USA: ACM, 2020, pp. 153–164, doi: 10.1145/3395363.3397354.
[16] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, pp. 6–6, 2018.
[17] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur. 20)*, Berkeley, California, USA: USENIX Association, Aug. 2020, pp. 2255–2269. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong
[18] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON : Directed grey-box fuzzing with provable path pruning," in *Proc. IEEE Symp. Secur. Privacy (SP) (SP),* Los Alamitos, CA, USA: IEEE Comput. Soc., May 2022, pp. 104–118, doi: 10.1109/SP46214.2022.00007.
[19] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 89–99.
[20] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng., (ASE)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 658–668, doi: 10.1109/ASE.2015.76.
[21] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "CrashScope: A practical tool for automated testing of android applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, 2017, pp. 15–18.
[22] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2019, pp. 115–127.
[23] S. Arzt et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, (PLDI)*. New York, NY, USA: ACM, 2014, pp. 259–269, doi: 10.1145/2594291.2594299.
[24] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for android against real-world bugs," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., (ESEC/FSE)*. New York, NY, USA: ACM, 2021, pp. 119–130, doi: 10.1145/3468264.3468620.
[25] "UI/application exerciser monkey." Android. Accessed: Feb. 2, 2023. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey
[26] E. W. Dijkstra, "A discipline of programming," *Executional Abstraction*. Upper Saddle River, NJ, USA: Prentice-Hall, 1976.

[27] “Android development documentation.” Vrim Infotech. Accessed: Jan. 15, 2023. [Online]. Available: https://developer.android.com/develop

[28] R. Padhye and U. P. Khedker, “Interprocedural data flow analysis in soot using value contexts,” in *Proc. 2nd ACM SIGPLAN Int. Workshop State Art Java Program Anal., (SOAP)*, New York, NY, USA: ACM, 2013, pp. 31–36, doi: 10.1145/2487568.2487569.

[29] “Data binding in android,” Android. Accessed: Dec. 1, 2023. [Online]. Available: https://developer.android.com/codelabs/android-databinding#0

[30] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proc. Proc. ACM SIGSAC Conf. Comput. Commun. Secur., (CCS)*. New York, NY, USA: ACM, 2018, pp. 2123–2138, doi: 10.1145/3243734.3243804.

[31] A. Arcuri and L. Briand, “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Softw. Testing, Verification Rel.*, vol. 24, no. 3, pp. 219–250, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486

[32] J. Sun et al., “Characterizing and finding system setting-related defects in android apps,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2941–2963, Apr. 2023.

[33] Y. Zhao et al., “ReCDroid: Automatically reproducing android application crashes from bug reports,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 128–139.

[34] Y. Zhao et al., “ReCDroid+: Automated end-to-end crash reproduction from bug reports for android apps,” vol. 31, no. 3, Mar. 2022, doi: 10.1145/3488244.

[35] M. Fazzini, M. Prammer, M. d’Amorim, and A. Orso, “Automatically translating bug reports into test cases for mobile apps,” in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal., (ISSTA)*, New York, NY, USA: ACM, 2018, pp. 141–152, doi: 10.1145/3213846.3213869.

[36] Z. Zhang, R. Winn, Y. Zhao, T. Yu, and W. G. J. Halfond, “Automatically reproducing android bug reports using natural language processing and reinforcement learning,” 2023, *arXiv:2312.17733*.

[37] S. Feng and C. Chen, “GIFdroid: Automated replay of visual bug reports for android apps,” 2021, *arXiv:2001.04171*.

[38] S. Staiger, “Static analysis of programs with graphical user interface,” in *Proc. 11th Eur. Conf. Softw. Maintenance Reeng. (CSMR)*, 2007, pp. 252–264.

[39] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon, “Lightweight static analysis for GUI testing,” in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng.*, 2012, pp. 301–310.

[40] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, “Reducing combinatorics in GUI testing of android applications,” in *Proc. 38th Int. Conf. Softw. Eng., (ICSE),* New York, NY, USA: ACM, 2016, pp. 559–570, doi: 10.1145/2884781.2884853.

[41] P. Bose et al., “Columbus: Android app testing through systematic callback exploration,” in *Proc. 45th Int. Conf. Softw. Eng., (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1381–1392, doi: 10.1109/ICSE48619.2023.00121.

[42] A. Sharma and R. Nasre, “QADroid: Regression event selection for android applications,” ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 66–77, doi: 10.1145/3293882.3330550.

[43] A. Blouin, V. Lelli, B. Baudry, and F. Coulon, “User interface design smell: Automatic detection and refactoring of Blob listeners,” *Inf. Softw. Technol.*, vol. 102, pp. 49–64, Oct. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918300892

[44] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, “Understanding and detecting callback compatibility issues for android applications,” *(ASE)*, New York, NY, USA: ACM, 2018, pp. 532–542, doi: 10.1145/3238147.3238181.

[45] A. Radhakrishna et al., “DroidStar: Callback typestates for android classes,” in *Proc. 40th Int. Conf. Softw. Eng., (ICSE)*, New York, NY, USA: ACM, 2018, pp. 1160–1170, doi: 10.1145/3180155.3180232.

[46] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, “Understanding JavaScript event-based interactions,” in *Proc. 36th Int. Conf. Softw. Eng., (ICSE)*, New York, NY, USA: ACM, 2014, pp. 367–377, doi: 10.1145/2568225.2568268.

[47] S. Meier, S. Mover, and B.-Y. E. Chang, “Lifestate: Event-driven protocols and callback control flow,” in *Proc. 33rd Eur. Conf. Object-Oriented Program. (ECOOP),* A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 1: 1–1:29. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.1

Maryam Masoudian received the bachelor’s degree from Amirkabir University of Technology and the master’s degree from Sharif University of Technology, Tehran, Iran. She is currently working toward the Ph.D. dual-degree program between Sharif University of Technology and Hong Kong University of Science and Technology. Her research interest includes software testing and program analysis with the focus on Android applications.

Heqing Huang received the B.S. degree from Xidian University, the M.Sc. degree from Imperial College London, and the Ph.D. degree from Hong Kong University of Science and Technology. He is an Assistant Professor with the Department of Computer Science, City University of Hong Kong. His research interest includes enhancing software reliability and security with diverse hybrid program analysis techniques. He has published multiple papers with the premium venues of security, software engineering, systems, and programming languages. His research received many awards, including the ACM SIGPLAN Distinguished Paper Award, the Google Research Paper Award, and the Hong Kong Ph.D. fellowship. His research has had a significant industry impact, and it has been successfully deployed in Huawei and awarded the Distinguished Collaborator Award. His research also successfully detected hundreds of software vulnerabilities in open-source software and received many acknowledgments from various communities, such as GNU, theorem provers, and Linux Kernel.

Morteza Amini is currently an Associate Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran and he is also one of the Directors of Data and Network Security Lab (DNSL) in this department. His research interests include database security, intrusion detection and mitigation, Android security, and IoT security and privacy.

Charles Zhang received the B.Sc., M.Sc., and Ph.D. degrees with honors from the University of Toronto. He is a Professor and the Director of the Cybersecurity Lab in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). He worked as a Software Engineer with Motorola Inc., an Expert Advisor with Huawei, and an Expert Security Panelist with Hong Kong Monetary Authority. His research is supported by Research Grant Council, Innovation and Technology Fund, and grants from Huawei, Ant, Tencent, TCL, Microsoft, and IBM. His research interests include program analysis techniques to improve software reliability. He was an Associate Editor of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and served on many organizational and technical committees of leading international conferences. His research awards include the distinguished paper awards of PLDI, OOPSLA, and ICSE, as well as the ACM SIGSOFT Doctoral Dissertation Award. His notable industrial impact includes the commercialization of research through Sourcebrella, acquired by the Ant Group, the research collaboration award from Ant Group, and the first to win twice the Huawei Distinguished Collaborator Award.