Testing Android Apps via Guided Gesture Event Generation

Xiangyu Wu[†], Yanyan Jiang[§], Chang Xu*[‡], Chun Cao[‡], Xiaoxing Ma[‡], Jian Lu[‡]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
[†]shawnwu20147@gmail.com, [§]jiangyy@outlook.com, [‡]{changxu, caochun, xxm, lj}@nju.edu.cn

***Abstract*—Mobile applications (apps) are mostly driven by touch gestures whose interactions are natural to human beings. However, generating gesture events for effective and efficient testing of such apps remains to be a challenge. Existing event generation techniques either feed the apps under test with random gestures or exhaustively enumerate all possible gestures. While the former strategy leads to incomplete test coverage, the latter suffers from efficiency issues. In this paper, we study the particular problem of gesture event generation for Android apps. We present a static analysis technique to obtain the gesture information: each UI component's potentially relevant gestures, so as to reduce the amount of gesture events to be delivered in the automated testing. We implemented our technique as a prototype tool GAT and evaluated it with real-world Android apps. The experimental results show that GAT is both effective and efficient in covering more code as well as detecting gesture-related bugs.**

***Index Terms*—Android app, gesture, static analysis, testing**

I. Introduction

Mobile devices and apps are becoming increasingly popular in our daily lives. Different from desktop apps, which are mainly operated by keyboard and mouse, mobile apps are mostly driven by touch gestures (*gestures* for short), e.g., scales, double taps, scrolls, pinches, and long presses. Gesture, determined by a touch-screen event trace in a short period of time [1], is a powerful way to express the interactions that are natural to human beings.

However, gestures make efficient and thorough testing challenging. As mobile apps are usually released timely, developers often do not have sufficient budget for a thorough manual testing and testing labors are typically conducted by an automated *event generator* [2]–[8]. These techniques do not directly focus on gesture event generation and cannot easily be extended to support complicated gestures. Most existing work focuses on a small subset of gestures [2]–[4] or only considers click gestures [5]–[8].

The key problem of facilitating effective and efficient testing of gesture-intensive apps is to obtain each UI component's relevant gestures (we name it *gesture information*). With such information, one can feed the app under test with relevant gestures at runtime, avoiding feeding a UI component with irrelevant gestures. We focus our discussion on the Android system, which is one of the most prevalent mobile platforms for smartphones and tablets. Nevertheless, the techniques and methodologies discussed in this paper can also be extended to other platforms.

We address the problem of effective and efficient testing of gesture-intensive apps by a three-step approach. First, we conducted a mini empirical study to find the categories of gesture recognition in Android apps, which demonstrates how to connect UI components with their relevant gestures. Second, for each category of gesture recognition, we conducted a case analysis and designed an algorithm to statically extract its corresponding gesture information. Third, we integrated such extracted gesture information in testing by only feeding the UI components with their relevant gestures at runtime.

We implemented our guided gesture event generation tool GAT (Gesture-aware Testing) based on the depth-first search algorithm of A3E [3] and evaluated it using popular real-world open-source Android apps. We compared our guided gesture event generation technique (DFS-rg) with three baselines (DFS that only generates the click gesture, DFS-eg that exhaustively enumerates all gestures in a library, and Monkey [9] that randomly generates gestures). Evaluation results show that (1) DFS-rg has the similar capability of detecting real-world gesture-related bugs as DFS-eg; (2) DFS-rg outperforms DFS and Monkey in covering gesture-related code; and (3) DFS-rg achieves the same code coverage much faster than DFS-eg.

In summary, this paper makes the following contributions:

1) We studied and categorized how gestures are recognized in Android apps.
2) We developed a static gesture analysis technique to extract a UI component's relevant gestures for each category of gesture recognition.
3) We implemented a prototype tool GAT to integrate gesture information in event generation and experimentally evaluated the effectiveness and efficiency of GAT.

The remainder of this paper is organized as follows. Section II introduces the background and overview of our work. Section III presents our study results of how gestures are recognized in Android apps. Section IV and Section V elaborate on the static gesture extraction and gesture-guided event generation of GAT. Section VI details the implementation including some design issues. Section VII experimentally evaluates GAT against code coverage and crash triggering

*Chang Xu is the corresponding author.

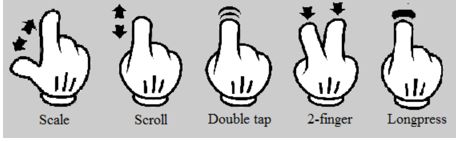

Fig. 1: Illustration of several widely used gestures [10]

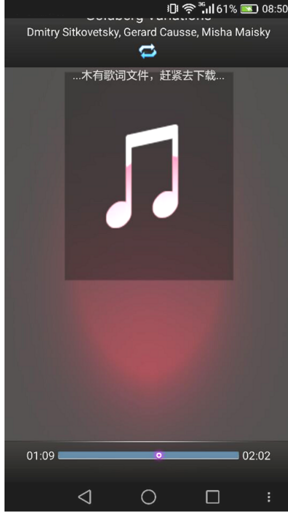

Fig. 2: A scroll gesture causes an unhandled exception and crashes the GestureMusic app

capability. Section VIII discusses related work and finally Section IX concludes this paper.

II. Background and Overview

Gestures facilitate natural interactions between a mobile device and its user. Figure 1 illustrates some widely used gestures: scales, scrolls, double taps, pinches, and long presses. A gesture is first *recognized* from raw system touch-screen events (each touch-screen event contains its coordinates, timestamp, and action, like DOWN, UP, MOVE, etc.) and then *handled* by its corresponding handler. Developers usually use the system libraries to recognize and handle gestures. Occasionally, they develop their customized gesture recognition algorithms.

Gestures make efficient and thorough automated testing challenging. The core of automated app testing is event generation at runtime [2]–[8]. However, existing techniques do not directly focus on gesture event generation and cannot easily be extended to support complicated gesture recognition. Most existing work focuses on a small subset of gestures [2]–[4] and some even generates only click gestures during testing [5]–[8]. Concolic testing [5] barely scales in solving complex path constraints that are generated by composite gestures (e.g., a multi-finger touch).

To illustrate existing techniques' limitation, we present a real-world bug example from GestureMusic, a music player app (Figure 2) whose most interactions (e.g., volume/play control) are realized by gestures. However, the developer incorrectly implemented the logic of music switching and the app may crash simply with a scrolling left/right gesture (meant for switching music). Existing techniques may analyze the app's UI layout and find that a touch event is relevant to this particular UI, but such analysis is not sufficient for inferring the corresponding relevant gestures. Dynodroid [2] cannot trigger this bug since it contains no scroll gestures in its test library. If one enhances this library by adding more gestures, the testing performance would largely degrade. On the other hand, random event generators (e.g., Monkey [9]) may have a chance to trigger the bug in theory, but they provide no bug detection guarantee due to their random nature.

Clearly, effective and efficient gesture event generation is challenging. One must determine each UI component's relevant gestures, i.e., *gesture information*, which may not be available (e.g., it is generally impossible to infer what gesture is relevant to a piece of customized gesture recognition code). We address such challenge by augmenting the existing GUI-model-based testing technique [3] by a best-effort static analysis that can infer most UI components' relevant gestures and use such information to guide automated testing. Our approach works as follows:

Study of Gesture Recognition in Android apps. The key to obtain gesture information is understanding how gestures are recognized (i.e., *gesture recognition*) by app developers, because the code of gesture recognition is the only place that associates UI components with their relevant gestures.

Therefore, we conducted a mini empirical study to figure out how gestures are recognized in Android apps. We found that gesture recognition falls into one of the following four categories: (1) using the `GestureDetector`, (2) parsing the touch events using customized code, (3) built-in gesture recognition of a UI component, or (4) using the `GestureOverlayView` and `GestureLibrary`.

Extracting Gesture Information. Based on the study results, we extract each UI component's relevant gestures by a case analysis for each category. For category (1), we develop a static taint analysis technique to trace a `GestureDetector` object's corresponding `GestureListener` object and its actual type, which contains handlers of gestures. For category (2), we extract relevant APIs for multi-touch gestures and statically check whether they are invoked by a UI component's touch-screen event handler. For category (3), we beforehand annotate the relevant gestures for each UI component that has built-in gesture recognition. For category (4), we treat such gestures as multi-touch gestures.

Integrating Gesture Information in Event Generation. Finally, we exploit the gesture information to guide event generation in app testing. We demonstrate how to integrate gesture information with the depth-first search algorithm [3]. In testing, instead of randomly feeding gestures or exhaustively enumerating all gestures from a gesture library, we only enumerate those gestures that are potentially relevant to a UI component. This treatment usually narrows down the enumeration to a few gestures, achieving both high effectiveness and high efficiency in app testing.

III. Gesture Recognition in Android Apps

To investigate how gestures are recognized in Android apps, we studied technical documents (API documentations, books, and tutorials) as well as open-source projects (from F-Droid and GitHub). In those open-source projects, we study code that is related to gesture recognition and handling. The details of the study are available on our website [11].

We summarize the four most popular categories of gesture recognition as follows, in the order of decreasing popularity:

1) Using the `GestureDetector` [12] library. To use `GestureDetector` for recognizing gestures, the developer provides a gesture listener object (of a subclass of `GestureListener`) containing overridden methods indicating its relevant gestures.
2) Parsing the `MotionEvent` (each represents a touch-screen event) objects with customized code. Within this category, gesture recognition can be as simple as a threshold checking (e.g., detecting a scroll gesture by checking the differential of x and y coordinates), or as complex as invoking a pre-trained classifier.
3) Built-in gesture recognition of a UI component. Some types of `Views` defined in the Android SDK are bound with specific gestures and the gestures are automatically recognized by the Android system.
4) Using `GestureOverlayView` and `GestureLibrary` [12]. Gestures drawn on a `GestureOverlayView` object can be captured and stored in a `GestureLibrary` object, which can be used for later recognition. Within this category, gestures can be in arbitrary shapes.

To the best of our knowledge, all gesture recognition ways in Android can be classified as one of the four categories. We discuss how gesture information is extracted for each category in Section IV.

IV. Extracting Gesture Information

Our goal is to determine the relevant gestures for each UI component (particularly, *activities* or *views* in Android [12]) in the app. For each gesture recognition site, we determine its relevant gestures by a case analysis depending on its category:

A. GestureDetector-based Gesture Recognition

`GestureDetector` recognizes gestures when its `onTouchEvent` method is invoked. However, its relevant gestures (gesture information) are not directly contained in the invocation site (Lines 12 and 22 in Figure 3). To obtain the gesture information, we should first find the instantiation site of `gd` at Line 2 as well as the argument `gl`, which is a gesture listener object whose actual type contains the gesture information (Lines 4–7). `gl` should again be traced back to its instantiation site at Line 1 and the overridden methods contain the gesture information. In Figure 3, since `onFling` and `onDoubleTap` are overridden by `MyGestureListener`, fling gestures and double tap gestures are relevant to `gl` and `gd`.

```
1  GestureListener gl = new MyGestureListener(...);
2  GestureDetector gd = new GestureDetector(...gl...);
3
4  class MyGestureListener extends GestureListener{
5  @Override onFling(){...}
6  @Override onDoubleTap(){...}
7  }
8
9  // Case 1: overriding the onTouchEvent method
10 ...
11 @Override public boolean onTouchEvent(MotionEvent e){
12   gd.onTouchEvent(e);
13   return super.onTouchEvent(e);
14 }
15 ...
16
17 // Case 2: setting the OnTouchListener
18 ...
19 View a = (...)findViewById(R.id.XXX);
20 a.setOnTouchListener(new View.onTouchListener(){
21 @Override public boolean onTouch(MotionEvent e){
22   gd.onTouchEvent(e);
23   return super.onTouchEvent(e);
24 }
25 });
26 ...
```

Fig. 3: Code snippet of gesture usage

```
int android.view.MotionEvent.findPointerIndex(int pointerId)
int android.view.MotionEvent.getPointerCount()
int android.view.MotionEvent.getPointerId(int pointerIndex)
float android.view.MotionEvent.getX(int pointerIndex)
float android.view.MotionEvent.getY(int pointerIndex)
```

Fig. 4: Multi-touch related APIs

To obtain gesture information of d, we first trace back to the instantiation site of d, which contains a gesture listener object ℓ (of type `GestureListener`). The gesture information can be derived from ℓ's actual type: ℓ's relevant gestures are implemented as overridden methods of the `GestureListener` base class, which again can be obtained at ℓ's instantiation site.

We use static taint analysis to decide an object's all possible instantiation sites, as listed in Algorithm 1. The algorithm iteratively finds the fixed point of T, the objects' instantiation sites. The algorithm displays the case analyses for assignments, method invocations and returns. We also maintain each object's possible aliases. At Lines 16 and 20 of Algorithm 1, when two objects are sure to be aliases, we update the alias information. Then, to determine whether an object c belongs to T, we check all aliases of c in A.

With the taint analysis, we first find all possible instantiation sites of ℓ to get ℓ's potentially actual types. Then, we find all possible instantiation sites of d. Since the instantiation sites of d now hold the information of ℓ's actual types, relevant gestures can be resolved. The gesture information of d is the union of all such gesture listeners' relevant gestures.

B. Customized Gesture Recognition

Customized gesture recognition parses `MotionEvent` objects by arbitrary code. Generally, it is impossible to tell what kind of gestures are relevant: the code may arbitrarily parse historical events or use machine-learning algorithms that even

Algorithm 1: Tracing an object's instantiation sites

```
Input: The control-flow graph G of the app, object use site d
Output: I be all possible instantiation sites of d
1  T ← ∅; // propagating objects' instantiation sites
2  M ← ∅; // map from an object to its instantiation sites
3  A ← ∅; // aliases information
4  // Pa(f, a) denotes method f's parameter a
5  // Rv(f) denotes method f's return value
6  while T is not fixed do
7      for every statement s ∈ G do
8          switch s do
9              case (instantiation) new(a)
10                 if type(a) = type(d) then
11                     T ← T ∪ {a} ;
12                     M[a] ← {a} ;
13             case (use site) d
14                 I ← M[d];
15             case (assignment) b := a ∧ a ∉ T
16                 A ← A ∪ {(a, b)};
17             case (assignment) b := a ∧ a ∈ T
18                 T ← T ∪ {b};
19                 M[b] ← M[b] ∪ M[a];
20                 A ← A ∪ {(a, b)};
21             case (method invocation) b.f(...a...) ∧ a ∈ T
22                 T ← T ∪ {Pa(f, a)};
23                 M[Pa(f, a)] ← M[Pa(f, a)] ∪ M[a];
24             case (method return) Rv(f):=a ∧a ∈ T
25                 T ← T ∪ {Rv(f)};
26                 M[Rv(f)] ← M[Rv(f)] ∪ M[a];
27 return I;
```

further complicate the case.

To generate gesture events in this case, our basic observation is that such cases are either simple gestures or multi-touch gestures. Therefore, we first conservatively assume that a certain set of gestures are relevant (particularly, scroll and scale gestures, as they are widely used and easy to be recognized). Then we scan all reachable gesture recognition code for the multi-touch gesture-related APIs listed in the white list in Figure 4. If we find any such API is invoked in the code, we further assume that multi-touch gestures are relevant. Generation of multi-touch gesture events is further discussed in Section V.

C. Built-in Gesture Recognition

This case is relatively trivial: a UI component's gesture information can be directly extracted either from its class signature or via the runtime UI layout, both of which are obtained by looking up in a set of predefined rules.

Algorithm 2: Gesture-guided depth-first search testing

```
1  S ← ∅; // explored states
2  function dfs(u, π) // u is the current state, π is a list
   containing "historical" events and states to reach state u
3  begin
4      for g ∈ getGestures(u) do
5          performGesture(g);
6          v ← getCurrentState();
7          if v ∉ S then
8              S ← S ∪ {v};
9              dfs(v, π :: ⟨g, v⟩) // "::" denotes list
               concatenation;
10         if v ≠ u then
11             backtrack(v, u, π); // ensure we come back to
               state u.
12 function backtrack(v, u, π)
13 begin
14     pressBackButton();
15     v ← getCurrentState();
16     if there is ⟨g_i, s_i⟩ ∈ π ∧ s_i = v then
17         for ⟨g_j, s_j⟩ ∈ π ∧ i < j ≤ |π| do
18             performGesture(g_j);
19     else
20         restartApp();
21         for ⟨g_i, s_i⟩ ∈ π do
22             performGesture(g_i);
```

We manually build the database of each UI component's built-in recognized gestures. Some views in Android (`ViewPager`, `Gallery`, etc.) automatically receive specific gestures because those classes override `onTouchEvent` to recognize gestures on their own. At runtime, if a specific view is on the screen, we look up the view type in the database to obtain its relevant gestures. We also check whether a view registers long click listeners, drag listeners, etc. Each of such registered listeners denotes a relevant gesture.

D. GestureOverlayView-based Recognition

In this case, the gesture patterns are pre-drawn by the users (or the developers). This is similar to the case of customized gesture recognition where relevant gestures are difficult to obtain. Therefore, we take a similar approach by treating it as relevant to multi-touch gestures in customized gesture recognition (Section IV-B).

V. GESTURE-GUIDED EVENT GENERATION

Our gesture-guided testing is based on the framework of the A3E [3] depth-first search algorithm. Particularly, we consider two app states to be equivalent if they are at the same activity and have the same UI layout so as to avoid generating redundant events. Therefore, the app can be treated as a graph where activities (associated with their views) are vertices and

events are edges. The testing procedure resembles a depth-first traversing of the graph. The gesture information is used for narrowing down the scope of gestures to be performed on a UI component and can be integrated into other existing event generation techniques [2]–[4], [6], [7] to avoid generating irrelevant gesture events.

The depth-first search is detailed in Algorithm 2. Starting from the initial state u, it enumerates all possible outgoing transitions from u and recursively explores any newly discovered state. If performing an event leads to an explored state (Line 10), it backtracks to state u (Line 11). The backtrack function first attempts to press the "back" button (Line 14). If pressing it yields an app state from which we have a known trace to u (Line 16), we partially replay the trace (Lines 17–18), otherwise the app is restarted and we conduct a full trace replay to state u.

getGestures (Line 4) differentiates how gesture events are generated. We study three particular strategies:

- DFS, the baseline depth-first search algorithm that does not use any gesture library. DFS assumes that each UI component is only relevant to the click gesture.
- DFS-eg (DFS-Enumerate-Gesture), in which getGestures returns *all* gestures defined in a predefined gesture library for all UI components.
- DFS-rg (DFS-Relevant-Gesture), our gesture-guided testing, which is described as follows.

DFS-rg handles different categories of gestures separately. As to predefined gestures (e.g., gestures recognized by built-in mechanism or `GestureDetector`), we directly send such gestures by existing tools. As to multi-touch gestures (Sections IV-B and IV-D), we record a series of typical multi-touch gestures (multi-finger scrolls and scales) plus some random gestures (multiple fingers simultaneously moving on the screen, which may cover different paths of the parsing code due to the random nature). When delivering such recorded gestures, we scale them to fit the UI component's boundary.

DFS-rg strikes a balanced point that is both efficient (avoiding exhaustive enumeration of irrelevant gestures) and thorough (testing each UI component with all its relevant gestures). In contrast, DFS is an efficient but non-thorough baseline, while DFS-eg is a thorough but inefficient baseline. We further demonstrate the power of this trade-off by comparing DFS-rg with DFS and DFS-eg in our evaluation (Section VII).

VI. Implementation

We implemented GAT, a prototype tool for gesture information extraction and gesture-guided testing. The architecture of GAT is shown in Figure 5.

A. Extracting Gesture Information

The gesture information extraction is built on top of the static analysis suite Soot [13].

For category (1), the static analysis finds a `GestureDetector` object's possible gesture listening classes and extracts the relevant gestures by examining

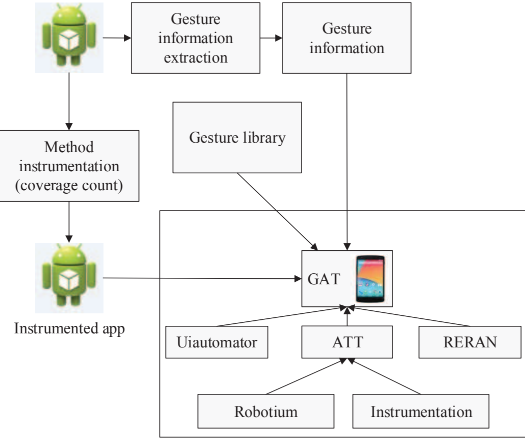

Fig. 5: The GAT architecture

TABLE I: Methods' relevant gestures

Methods	Gestures
onSingleTapUp, onSingleTapUpConfirmed, onShowPress	click
onLongPress	long click
onDoubleTap,onDoubleTapEvent	double tap
onScroll	scroll
onFling	fling
onScale,onScaleBegin,onScaleEnd	scale

overridden methods by Table I. For example, if the method `onFling` is overridden, a fling gesture is relevant.

We may fail to obtain a `GestureDetector` object's instantiation site due to implicit data flow. For example, a `GestureDetector` object may be put into a `Bundle` object which is associated with an `Intent` object. In such conditions, the data flow is almost impossible to be statically analyzed. If we fail to trace a `GestureDetector` object's instantiation site, we conservatively assume that the UI component is relevant to all gestures in Table I to ensure the UI component's relevant gestures are exercised in testing. We also found that developers usually follow simple patterns of recognizing gestures and the code style is relatively simple: GAT can obtain precise gesture information in most conditions.

B. Testing with Guided Gesture Event Generation

We implemented DFS, DFS-rg, and DFS-eg described in Section V based on our Android testing framework ATT [14] for a fair comparison. At runtime, ATT can obtain each on-screen UI component's boundary and actual type. We deliver different gestures at runtime based on the algorithm (DFS delivers clicks, DFS-eg enumerates all gestures in the library, and DFS-rg delivers only extracted relevant gestures).

Our gesture library contains a set of predefined gestures and the code to deliver them at runtime. The gesture library contains simple gestures (clicks, long clicks, double taps, etc.), composite gestures (scrolls, flings, drags, scales, etc.), and customized multi-touch gestures (Section V). Simple and composite gestures are hard-coded and delivered by Uiautomator [15]. Customized multi-touch gestures are first recorded and stored in a library by the record-and-replay tool RERAN [16]. We

TABLE II: List of evaluated buggy widget demos. The “Results” column shows the bugs found by Monkey / DFS / DFS-rg / DFS-eg, respectively.

Name	Revision	LOC	# Bugs	Results
Coverflow	#52604	860	2	1 / 0 / 2 / 2
SideMenu	#5422f	1,045	1	0 / 0 / 1 / 1
Flotandroid Chart	#c5532	6,140	2	2 / 1 / 2 / 2
Androviews	#e82df	593	1	0 / 0 / 1 / 1
Swipeable Cards	#e1269	717	1	0 / 0 / 1 / 1

recorded the touch-screen events (coordinates and timestamps) for each multi-touch gesture. A gesture is delivered by replaying the recorded touch event sequence with scaled coordinates.

VII. Evaluation

A. Methodology

We evaluated the effectiveness and efficiency of our DFS-rg technique. We divided the experimental subjects into two groups: buggy widget demos and real-world apps. We evaluated them separately using four techniques: DFS, DFS-rg and DFS-eg discussed in Section V as well as Monkey, the most prevalent automated event generator in practice[1]. Particularly, we try to answer the following research questions:

RQ1 (effectiveness): *Can DFS-rg cover more code or reveal more bugs than Monkey and DFS within a reasonable resource budget?*

RQ2 (efficiency): *Compared to DFS-eg, does DFS-rg achieve the same testing effect faster?*

The experimental setups are described as follows.

1) Buggy Widget Demos: We searched Github for high-starred widget demo projects that have reported gesture-related bugs. The first three columns of Table II show the basic information of these widgets.

We use these known bugs to compare the bug revealing capability of the evaluated techniques, i.e., effectiveness (RQ1). All demos are of small footprints (DFS series are ran until completion and the code coverage for Monkey is not increasing within a few minutes) and thus we do not study RQ2 for these subjects.

2) Real-world Apps: We also conducted evaluation using real-world gesture-intensive apps. We first selected a series of candidate apps, extracted gesture information of apps by the static analysis (Section IV), and chose the most gesture-intensive (many UI components are associated with gestures) apps, as listed in Table III.

To answer RQ1, we ran DFS-rg, Monkey, and DFS on the real-world apps and compare their method coverage. We also measured the coverage of gesture-related methods. A method is gesture-related if it is transitively reachable from an `onTouchEvent`, `onTouch` method, or in a `GestureListener` object.

[1]We were to compare DFS-rg with existing techniques that can generate gesture events. However, Dynodroid [2] and GUIRipper [6] cannot test our subjects because these subjects require Android 4.0 or higher. SwiftHand [4] also encountered problems in running the apps. Therefore, we implemented different versions of depth-first search using the same platform for a fair comparison.

TABLE III: List of evaluated real-world apps

Name	Version	Category	Source
Blue-infinity	1.0.27	Image	Google Play
BrighterBigger	1.01	Camera	Google Play
Gyeongbokgung	1.0.0	Travel	Google Play
Focal	1.0	Camera	F-Droid
Penroser	1.2	Wallpaper	F-Droid
VimTouch	1.7	Editor	F-Droid
Zoompage	1.0	Image	Github
GestureMusic	1.0	Media	Anzhi
Touch Calendar	1.1.29	Calendar	Anzhi

We run DFS and DFS-rg until termination and terminate Monkey if its coverage has not been increased for a relatively long period (5 minutes). All experimental results of Monkey are average of 10 runs. We did not evaluate DFS-eg because it usually cannot terminate within a reasonable amount of time. Such efficiency issue is studied in RQ2.

To answer RQ2, for each real-world app, we set a four-hour time limit and compare the coverage trends for DFS-rg and DFS-eg over time.

All experiments were performed on a desktop machine with 8 GB memory running Ubuntu Linux 12.04 and a Nexus 5 device running Android 5.0.

B. Evaluation Results

1) Buggy Widget Demos: Column 4 of Table II shows the number of known gesture-related bugs while Column 5 shows the number of bugs found by Monkey, DFS, DFS-rg and DFS-eg, respectively. The results indicate that without the support of gesture information, gesture-related bugs can rarely be revealed: DFS only discovered 1 out of 7 known gesture-related bugs; Monkey, though detected more gesture-related bugs than DFS (3 out of 7), is still not capable of detecting the remaining 4 bugs that must be triggered by a multi-touch gesture. On the other hand, with our gesture information, DFS-rg performed as good as enumerating all gestures in the library (DFS-eg): it successfully revealed all previously known gesture-related bugs.

2) Real-world Apps: To answer RQ1, the effectiveness evaluation of coverage is listed in Table IV. While we only know the existance of a known gesture-related bug in the Zoompage app, we should highlight the result that DFS-rg found two gesture-related bugs that are believed to be previously unknown (VimTouch and GestureMusic). Touching the screen with multi-fingers quickly can crash the VimTouch app, while scrolling left or right as the music is playing can crash the GestureMusic app. The issues are reported to the developers.

The general belief that a systematic search is more effective than random technique is validated in the evaluation results (DFS outperforms Monkey for 8 out of 9 subjects for the overall method coverage). The exceptional case is Zoompage because it consists of a single activity in which Monkey nearly exhaustively enumerated all possible built-in actions. For the similar reason, Monkey covered more gesture-related methods than DFS (7 out of 9).

TABLE IV: The main evaluation results. A/G denotes all methods/gesture-related methods in an app.

App name	# of A/G	Monkey			DFS			DFS-rg		
		time (s)	coverage (A/G)	bugs	time (s)	coverage (A/G)	bugs	time (s)	coverage (A/G)	bugs
Blue-infinity	1,026/98	1,233	35%/56%	0	3,843	58%/60%	0	5,089	66%/74%	0
BrighterBigger	367/6	505	70%/67%	0	790	80%/33%	0	836	83%/100%	0
Gyeongbokgung	3,704/61	1,582	8%/5%	0	7,613	14%/5%	0	7,794	16%/10%	0
Focal	1,515/76	812	35%/61%	0	1,647	37%/39%	0	2,018	43%/74%	0
Penroser	392/20	664	31%/30%	0	1,590	64%/20%	0	2,468	67%/60%	0
VimTouch	646/83	872	51%/61%	0	1,350	54%/59%	0	2,395	57%/69%	1*
Zoompage	96/21	420	73%/76%	1	143	62%/67%	0	216	78%/86%	1
GestureMusic	365/12	487	36%/50%	0	885	57%/33%	0	1,502	61%/83%	1*
Touch Calendar	728/29	625	26%52%	0	1,741	52%/48%	0	2,675	56%/66%	0

* Previously unknown bugs arisen from gestures.

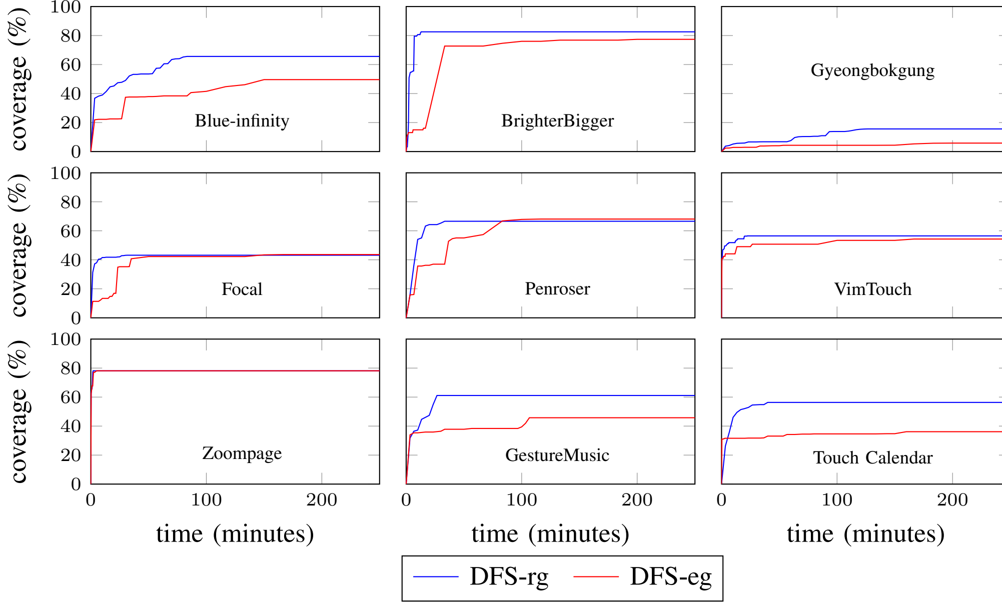

Fig. 6: Coverage trends for evaluated real-world apps

DFS-rg achieved the highest coverage for all evaluated subjects, both for the overall method coverage and the gesture-related method coverage. Furthermore, recall that we ran both DFS and DFS-rg until completion. With our guided gesture event generation, DFS-rg only consumed 2%–77% (mean 41%) more time than DFS, which is considered acceptable in practice. We also found that coverage of Gyeongbokgung is low for all evaluated testing techniques. This is due to its code obfuscation that yields a large amount of unreachable code. A manual testing of the app (trying to cover all possible functions of the app) achieved a method coverage of 13% and thus we believe that DFS-rg did thorough testing in this case.

Finally, one might argue that the coverage gain of DFS-rg is marginal compared with DFS (2%–16%, mean 5.4%). This is because complex gesture recognition and handling is only a small part of a large app (most app states can be manifested by click gestures). In fact, DFS-rg covered 5%–67% (mean 29%) more gesture-related methods than DFS, which is significant. Since covering gesture handling code is challenging and of great importance (we indeed found two previous unknown gesture-related bugs), we believe that it worths paying a small amount of time cost to automatically test apps in a gesture-sensitive way.

To answer RQ2, we plot the cumulative method coverage over time for DFS-rg and DFS-eg in Figure 6. DFS-rg terminated within the time-limit for all evaluated subjects and achieved its peak coverage much faster than DFS-eg. In contrast, the coverage of DFS-eg increased much slower than DFS-rg because time was wasted trying useless gestures that are not relevant to any UI component (6 out of 9 are terminated early due to time-outs).

For the two subjects (Focal and Penroser) that DFS-eg terminated within the time-limit, DFS-eg covered slightly more methods than DFS-rg (less than 1%). We believe that this is not due to our gesture information (recall that DFS-rg and DFS-eg detected the same gesture-related bugs for buggy widget demos). Rather, it is because the depth-first search strategy treats all app states reached by executing the same activity sequences to be equivalent, which may not reflect the actual model of the app under test. Therefore, DFS-eg is able to explore more app states due to its enumerative nature, yielding slightly higher method coverage. However,

this is a limitation of DFS-based algorithms and we believe that it should be addressed by more advanced event generation techniques [2], [5], [17].

Even though the static analysis of GAT is best-effort, the evaluation results show that the extracted gesture information is useful in facilitating more effective and efficient testing of an app. Furthermore, if a tester has the knowledge of the source code (which is mostly the case), such issues can be further alleviated by a few lines of annotations.

VIII. Related work

Gestures in automated app testing. Choudhary et al. [18] presented an empirical study of the mainstream testing techniques and proposed future research directions. We address the particular challenge of automatically mocking gesture events as existing techniques [3], [4], [6], [17], [19], [20] mainly focus on state-space exploration and do not pay enough attention to gestures. Dynodroid [2] and A3E [3] handle gestures by enumerating all gestures in a library (like DFS-eg) and are inefficient as the gesture library grows. SwiftHand [4] takes use of scroll gestures to minimize the number of restarts during the testing process. CONTEST [5] adopts concolic testing to generate events and is able to handle gestures. However, it faces scalability issues. Our previous work UGA [7] takes an alternative approach by amplifying a small amount of manual test inputs. However, as long as the user trace does not contain a gesture event, it can never be manifested.

Static analysis in Android. Android apps, though implemented in Java, have their own characteristics and can be statically analyzed. Our gesture information extraction also aligns with such work. Li et al. conducted a systematic literature review on static analysis of Android apps [21]. Some representative work includes FlowDroid [22] , Chex [23], Bartel et al. [24], Li et al. [25] and so on. Yang et al. [26] proposed a program representation for capturing sequences of lifecycle callbacks and event handler callbacks. DPartner [27] conducts static analysis to find code that can be migrated to a server for computation offloading. ASYNCHRONIZER [28] uses a points-to analysis to help developers automatically refactor long-running operations to `AsyncTask`.

IX. Conclusion

We present GAT, an automated Android testing technique that addresses the challenge of effective and efficient generation of gesture events. We use static analysis to infer each UI component's relevant gestures and use such gesture information to guide a systematic testing. Evaluation results show that GAT paid an affordable overhead over gesture-insensitive testing technique to achieve higher test coverage and it detected previously unknown gesture-related bugs.

X. Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61472174, #91318301, #61321491) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] "Some gestures," http://www.tutorialspoint.com/android/android_gestures.htm.
[2] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. of FSE*, 2013, pp. 224–234.
[3] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. of OOPSLA*, 2013, pp. 641–660.
[4] W. Choi, G. Necula, and K. Sen, "Guided gui testing of Android apps with minimal restart and approximate learning," in *Proc. of OOPSLA*, 2013, pp. 623–640.
[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. of FSE*, 2012, p. 59.
[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. of ASE*, 2012, pp. 258–261.
[7] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Li, "User guided automation for testing mobile apps," in *Proc. of APSEC*, 2014, pp. 27–34.
[8] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for Android applications," in *Proc. of ISSRE*, 2015, pp. 461–471.
[9] "Monkey," http://developer.android.com/tools/help/monkey.html.
[10] "Some gestures," https://pixabay.com.
[11] "A study of gesture recognition," http://moon.nju.edu.cn/spar/people/wxy/sogr.html.
[12] "Android classes," http://developer.android.com/reference/.
[13] "Soot," https://sable.github.io/soot/.
[14] Z. Meng, Y. Jiang, and C. Xu, "Facilitating reusable and scalable automated testing and analysis for Android apps," in *Proc. of Internetware*, 2015.
[15] "Uiautomator," http://developer.android.com/tools/testing-support-library/index.html#UIAutomator.
[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for Android," in *Proc. of ICSE*, 2013, pp. 72–81.
[17] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of Android apps," in *Proc. of FSE*, 2014, pp. 599–609.
[18] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proc. of ASE*, 2015, pp. 429–440.
[19] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proc. of FASE*, 2013, pp. 250–265.
[20] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. of Mobisys*, 2014, pp. 204–217.
[21] L. Li, F. B. Tegawende, P. Mike, R. Siegfried, B. Alexandre, O. Damien, K. Jacques, and L. T. Yves, "Static analysis of Android apps: A systematic literature review."
[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. of OOPSLA*, vol. 49, no. 6, 2014, pp. 259–269.
[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. of CCS*, 2012, pp. 229–240.
[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to Android," in *Proc. of ASE*, 2012, pp. 274–277.
[25] D. Li, A. H. Tran, and W. G. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Proc. of ICSE*, 2014, pp. 527–538.
[26] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *Proc. of ICSE*, 2015, pp. 89–99.
[27] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android Java code for on-demand computation offloading," in *Proc. of OOPSLA*, vol. 47, no. 10, 2012, pp. 233–248.
[28] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for Android applications through refactoring," in *Proc. of FSE*, 2014, pp. 341–352.