Targeted and Depth-first Exploration for Systematic Testing of Android Apps

Tanzirul Azim
University of California, Riverside
mazim002@cs.ucr.edu

Iulian Neamtiu
University of California, Riverside
neamtiu@cs.ucr.edu

Abstract

Systematic exploration of Android apps is an enabler for a variety of app analysis and testing tasks. Performing the exploration while apps run on actual phones is essential for exploring the full range of app capabilities. However, exploring real-world apps on real phones is challenging due to non-determinism, non-standard control flow, scalability and overhead constraints. Relying on end-users to conduct the exploration might not be very effective: we performed a 7-user study on popular Android apps, and found that the combined 7-user coverage was 30.08% of the app screens and 6.46% of the app methods. Prior approaches for automated exploration of Android apps have run apps in an emulator or focused on small apps whose source code was available. To address these problems, we present A^{3}E, an approach and tool that allows substantial Android apps to be explored systematically while running on actual phones, yet without requiring access to the app's source code. The key insight of our approach is to use a static, taint-style, dataflow analysis on the app bytecode in a novel way, to construct a high-level control flow graph that captures legal transitions among activities (app screens). We then use this graph to develop an exploration strategy named *Targeted Exploration* that permits fast, direct exploration of activities, including activities that would be difficult to reach during normal use. We also developed a strategy named *Depth-first Exploration* that mimics user actions for exploring activities and their constituents in a slower, but more systematic way. To measure the effectiveness of our techniques, we use two metrics: activity coverage (number of screens explored) and method coverage. Experiments with using our approach on 25 popular Android apps including BBC News, Gas Buddy, Amazon Mobile, YouTube, Shazam Encore, and CNN, show that our exploration techniques achieve 59.39–64.11% activity coverage and 29.53–36.46% method coverage.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Validation; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools,Tracing

General Terms Languages, Reliability, Verification

Keywords Google Android, GUI testing, Systematic exploration, Test case generation, Code coverage, Greybox testing, Dynamic analysis, Taint analysis

1. Introduction

Users are increasingly relying on smartphones for computational tasks [1, 2], hence concerns such as app correctness, performance, and security become increasingly pressing [6, 8, 30, 31, 38]. Dynamic analysis is an attractive approach for tackling such concerns via profiling and monitoring, and has been used to study a wide range of properties, from energy usage [38, 39] to profiling [40] and security [31]. However, dynamic analysis critically hinges on the availability of test cases that can ensure good coverage, i.e., drive program execution through a significant set of representative program states [36, 37].

To facilitate test case construction and exploration for smartphone apps, several approaches have emerged. The Monkey tool [15] can send random event streams to an app, but this limits exploration effectiveness. Frameworks such as Monkeyrunner [24], Robotium [18] and Troyd [20] support scripting and sending events, but scripting takes manual effort. Prior approaches for automated GUI exploration [9–12, 17, 34] have one or more limitations that stand in the way of understanding how popular apps run in their natural environment, i.e., on actual phones: running apps in an emulator, targeting small apps whose source code is available, incomplete model extraction, state space explosion.

For illustration, consider the task of automatically exploring popular apps, such as Amazon Mobile, Gas Buddy, YouTube, Shazam Encore, or CNN, whose source code is not available. Our approach can carry out this task, as shown in Section 6, since we connect to apps running naturally on the

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509549

phone. However, existing approaches have multiple difficulties due to the lack of source code or running the app on the emulator where the full range of required sensor inputs (camera, GPS, microphone) or output devices (e.g., flashlight) is either unavailable [32] or would have to be simulated.

To tackle these challenges, we present Automatic Android App Explorer (A^3E), an approach and open-source tool[1] for systematically exploring real-world, popular apps Android apps running on actual phones. Developers can use our approach to complement their existing test suites with automatically-generated test cases aimed at systematic exploration. Since A^3E does not require access to source code, users other than the developers can execute substantial parts of the app automatically. A^3E supports sensors and does not require kernel- or framework-level instrumentation, so the typical overhead of instrumentation and device emulation can be avoided. Hence we believe that researchers and practitioners can use A^3E as a basis for dynamic analyses [36] (e.g., monitoring, profiling, information flow tracking), testing, debugging, etc.

In this paper, our approach is focused on improving coverage at two granularity levels: *activity* (high-level) and *method* (low-level). Activities are the main parts of Android apps—an activity roughly corresponds to a different screen or window in traditional GUI-based applications. Increasing activity coverage means, roughly, exploring more screens. For method coverage we focus on covering app methods, as available in the Dalvik bytecode (compiled from Java), that runs on the Dalvik VM on an actual phone; an activity's implementation usually consists of many methods, so by improving method coverage we allow the functionality associated with each activity to be systematically explored and tested. In Section 2 we provide an overview of the Android platform and apps, we define the graphs that help drive our approach, and provide definitions for our coverage metrics.

To understand the level of exploration attained by Android app users in practice, we performed a user study and measured coverage during regular interaction. For the study, we enrolled 7 users that exercised 28 popular Android apps. We found that across all apps and participants, on average, just 30.08% of the app screens and 6.46% of the app methods were explored. The results and reasons for these low levels of coverage are presented in Section 3.

In Section 4 we present our approach for automated exploration: given an app, we construct systematic exploration traces that can then be replayed, analyzed and used for a variety of purposes, e.g., to drive dynamic analysis or assemble test suites. Our approach consists of two techniques, *Targeted Exploration* and *Depth-First Exploration*. Targeted Exploration is a directed approach that first uses static bytecode analysis to extract a Static Activity Transition Graph and then explore the graph systematically while the app runs on a phone. Depth-First Exploration is a completely dynamic approach based on automated exploration of activities and GUI elements in a depth-first manner.

In Section 5 we provide an overview of A^3E's implementation: hardware platform, tools and measurement procedures. In Section 6 we provide an evaluation of our approach on 25 apps (3 apps could not be explored because they were written mainly in native code rather than bytecode). We show that our approach is effective: on average it attains 64.11% and 59.39% activity coverage via Targeted and Depth-first Exploration, respectively (a 2x increase compared to what the 7 users have attained); it also attains 29.53% and 36.46% method coverage via Targeted and Depth-first Exploration, respectively (a 4.5x increase compared to the 7 users). Our approach is also efficient: average figures are 74 seconds for Static Activity Transition Graph construction, 87 minutes for Targeted Exploration and 104 minutes for Depth-first Exploration.

In summary, this work makes the following contributions:

- A qualitative and quantitative study of coverage attained in practice by 7 users for 28 popular Android apps.
- Two approaches, Targeted Exploration and Depth-first Exploration, for exploring substantial apps running on Android smartphones.
- An evaluation of the effectiveness of Targeted and Depth-first Exploration on 25 popular Android apps.

2. Android Activities, Graphs and Metrics

We have chosen Android as the target platform for our A^3E implementation as it is currently the leading mobile platform in the US [4] and worldwide [3]. We now describe the high-level structure of Android platform and apps; introduce two kinds of Activity Graphs that define the high-level workflow within an app; and define coverage based on these graphs.

2.1 Android App Structure

Android platform and apps. Android apps are typically written in Java (possibly with some additional native code). The Java code is compiled to a `.dex` file, containing compressed bytecode. The bytecode runs in the Dalvik virtual machine, which in turn runs on top of a smartphone-specific version of the Linux kernel. Android apps are distributed as `.apk` files, which bundle the `.dex` code with a "manifest" (app specification) file named `AndroidManifest.xml`.

Android app workflow. A rich application framework facilitates Android app construction, as it provides a set of libraries, a high-level interface for interaction with low-level devices, etc. More importantly, for our purposes, the application framework orchestrates the workflow of an app, which makes it easy to construct apps but hard to reason about control flow.

A typical Android app consists of separate screens named *Activities*. An activity defines a set of tasks that can be grouped together in terms of their behavior and corresponds

[1] `http://spruce.cs.ucr.edu/A3E/`

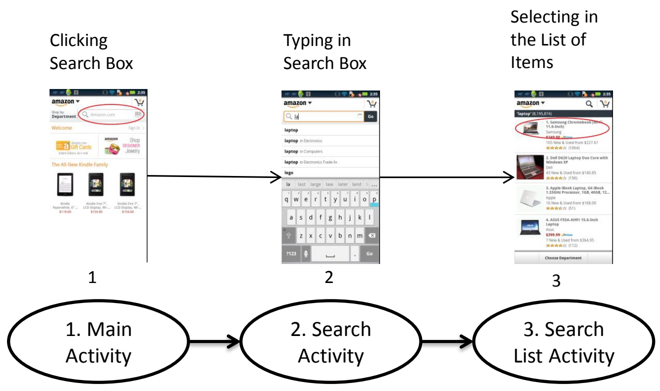

Figure 1. An example activity transition scenario from the popular Android app, Amazon Mobile.

to a window in a conventional desktop GUI. Developers implement activities by extending the android.app.Activity class. As Android apps are GUI-centric, the programming model is based on callbacks and differs from the traditional main()-based model. The Android framework will invoke the callbacks in response to GUI events and developers can control activity behavior throughout its life-cycle (create, paused, resumed, or destroy) by filling-in the appropriate callbacks.

An activity acts as a container for typical GUI elements such as toasts (pop-ups), text boxes, text view objects, spinners, list items, progress bars, check boxes. When interacting with an app, users navigate (i.e., transition between) different activities using the aforementioned GUI elements. Therefore in our approach activities, activity transitions and activity coverage are fundamental, because activities are the main interfaces presented to an end-user. For this reason we primarily focused on activity transition during a normal application run, because its role is very significant in GUI testing.

Activities can serve different purposes. For example in a typical news app, an activity home screen shows the list of current news; selecting a news headline will trigger the transition to another activity that displays the full news item. Activities are usually invoked from within the app, though some activities can be invoked from outside the app if the host app allows it.

Naturally, these activity transitions form a graph. In Figure 1 we illustrate how activity transitions graphs emerge as a result of a user interaction in the popular Android app, Amazon Mobile. On top we have the textual description of users' actions, in the middle we have an actual screen shot, and on the bottom we have the activities and their transitions. Initially the app is in the Main Activity; when the user clicks the search box, the app transitions to the Search Activity (note the different screen). The user searches for items by typing in item names, and a textual list of items is presented. When the user presses "Go", the screen layout changes as the app transitions to the Search List Activity.

App	Type	Category	Size		# Down-
			Kinst.	KBytes	loads
Amazon Mobile	Free	Shopping	146	4,501	58,745
Angry Birds	Free	Games	167	23,560	1,586,884
Angry Birds Space P.	Paid	Games	179	25,256	14,962
Advanced Task Killer	Free	Productivity	9	75	428,808
Advanced Task Killer P.	Paid	Productivity	3	99	4,638
BBC News	Free	News&Mag.	77	890	14,477
CNN	Free	News&Mag.	204	5,402	33,788
Craigslist Mobile	Free	Shopping	56	648	61,771
Dictionary.com	Free	Books&Ref.	105	2,253	285,373
Dictionary.com Ad-free	Paid	Books&Ref.	49	1,972	2,775
Dolphin Browser	Free	Communication	248	4,170	1,040,437
ESPN ScoreCenter	Free	Sports	78	1,620	195,761
Facebook	Free	Social	475	3,779	6,499,521
Tiny Flashlight + LED	Free	Tools	47	1,320	1,612,517
Movies by Flixster	Free	Entertainment	202	4,115	398,239
Gas Buddy	Free	Travel&Local	125	1,622	421,422
IMDb Movies & TV	Free	Entertainment	242	3,899	129,759
Instant Heart Rate	Free	Health&Fit.	63	5,068	100,075
Instant Heart R.-Pro	Paid	Health&Fit.	63	5,068	6,969
Pandora internet radio	Free	Music&Audio	214	4,485	968,714
PicSay - Photo Editor	Free	Photography	49	1,315	96,404
PicSay Pro - Photo E.	Paid	Photography	80	955	18,455
Shazam	Free	Music&Audio	308	4,503	432,875
Shazam Encore	Paid	Music&Audio	308	4,321	18,617
WeatherBug	Free	Weather	187	4,284	213,688
WeatherBug Elite	Paid	Weather	190	4,031	40,145
YouTube	Free	Media&Video	253	3,582	1,262,070
ZEDGE	Free	Personalization	144	1,855	515,369

Table 1. Overview of our examined apps.

We now proceed to defining the activity transitions graphs that form the basis of our work.

2.2 Static Activity Transition Graph

The *Static Activity Transition Graph* (SATG) is a graph $G_S = (V_S, E_S)$ where the set of vertices, V_S, represents the app activities, while the set of edges, E_S, represents possible activity transitions. We extract SATG's automatically from apps using static analysis, as described in Section 4.1.

Figure 2 shows the SATG for the popular shopping app, Craigslist Mobile; the reader can ignore node and edge colors as well as line styles for now. Note that activities can be called independently, i.e., without the need for entering into another activity. Therefore, the SATG can be a disconnected graph. SATG's are useful for program understanding as they provide an at-a-glance view of the high-level app workflow.

2.3 Dynamic Activity Transition Graph

The *Dynamic Activity Transition Graph* (DATG) is a graph $G_D = (V_D, E_D)$ where the set of vertices, V_D, represents the app activities, while the set of edges, E_D, represents actual activity transitions, as observed at runtime.

A DATG captures the footprint of dynamic exploration or user interaction in an intuitive way and is a subgraph of the SATG. Figure 2 contains the DATG for the popular shopping app, Craigslist Mobile: the DATG is the subgraph consisting of solid edges and nodes. Paths in DATG's illustrate sequences of actions required to reach a particular state of an app, which is helpful for constructing test cases or repro-

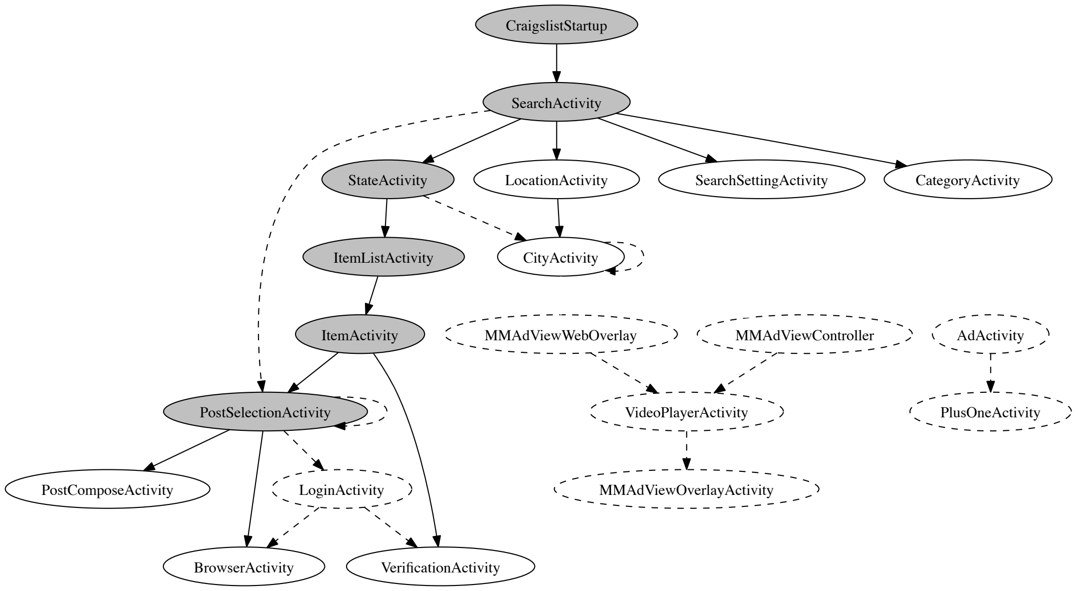

Figure 2. Static Activity Transition Graph extracted automatically by our approach from the Craigslist Mobile app. Grey nodes and associated edges have been explored by users. Solid-contour nodes (grey or white) and solid-line edges were traversed dynamically by our exploration. Dashed-contour nodes and dashed-line edges remained unexplored. Activity names are simplified for legibility.

ducing bugs. In the Appendix (Figure 7) we present a second DATG example based on runs from 5 different users, which is illustrates how different users exercise the app differently.

2.4 Coverage Metrics

We chose two coverage metrics as basis for measuring and assessing the effectiveness of our approach: activity coverage and method coverage. We chose these metrics because they strike a good balance between utility and collection overhead: first, activities and methods are central to app construction, so the numeric values of activity and method coverage are intuitive and informative; second, the runtime performance overhead associated with collecting these metrics is low enough so that user experience and app performance are not affected. We now proceed to defining the metrics.

Activity coverage. We define *activity coverage* (AC) as the ratio of activities reached during execution (AR) to the total number of activities defined in the app (AT), that is, $AC = \frac{AR}{AT}$. Intuitively, the higher the AC for a certain run, the more screens have been explored, and the more thorough and complete the app exploration has been. We retrieve the AR dynamically, and the AT statically, as described in Section 5.2.

Method coverage. Activity coverage is intuitive, as it indicates what percentage of the screens (that is, functionality at a high level) are reached. In addition, users might be interested in the thoroughness of exploration measured at a lower, method-level. Hence we use a finer-grained metric—what percentage of methods are reached—to quantify this aspect. We define *method coverage* (MC) as the ratio of methods called during execution (ME) to the total number of methods defined in the app (MT), that is, $MC = \frac{ME}{MT}$.

We found that all the examined apps, except Advanced Task Killer, ship with third-party library code bundled in the app's APK file; we exclude third-party methods from ME and MT computations as these methods were not defined by app developers hence we consider that including them would be misleading. We measured the ME using runtime profiling information and the MT via static analysis, as described in Section 5.2.

3. User Study: Coverage During Regular Use

One possible approach to exploration is to rely on (or at least seed the exploration with) actual runs, i.e., by observing how end-users interact with the app. Unfortunately, this approach is not systematic: as our measurements indicate, during normal user interaction, coverage tends to be low, as users explore just a small set among the features and functionality offered by the app. Therefore, relying on users might have limited utility. To quantify the actual coverage attained by end-users, we have performed a user study, as described next.

App dataset. As of March 2013, Google Play, the main Android app market, lists more than 600,000 apps. We selected a set of 28 apps for our study; the apps and their characteristics are presented in Table 1. The selection was based on several criteria. First, we wanted a mix of free and paid apps, so for 7 apps we selected both the free and the paid versions (column 2). Second, we wanted representation across different categories such as productivity, games, entertainment, news; in total, our dataset has apps from 17 different categories (column 3). Third, we wanted substantial apps; the sizes of our selected apps, in thousands of bytecode instructions and KB, respectively, are shown in columns 4 and 5. Finally, we wanted to investigate popular apps; in the last column we show the number of downloads as listed on Google Play as of March 28, 2013; the number of downloads varied from 2,775 to 6,499,521. We believe that this set covers a good range of popular, real-world mobile apps.

Methodology. We enrolled 7 different users in our study; one high-coverage minded user (called User 1) and six "regular" users (User 2–User 7). Each app was exercised by each user for 5 minutes, which is far longer than the typical average app session (71.56 seconds) [35]. To mirror actual app use "in the wild," the six regular users were instructed to interact with the app as they normally would; that is, regular users were not told that they should try to achieve high coverage. However, User 1 was special because the user's stated goal was to achieve maximum coverage within the time limit. For each run, we collected runtime information so we could replicate the experiment later. We then analyzed the 192 runs[2] to quantify the levels of activity coverage (separate screens) and method coverage attained in practice.

3.1 Activity Coverage

We now turn to discussing the levels of activity coverage that could be attained based on end-user coverage (separate and combined across users) for each metric.

Cumulative coverage. As different users might explore different app features, we developed a technique to "merge" different executions of the same app. More specifically, given two DATG's G_1 and G_2 (as defined in Section 2.3), we construct the union, i.e., a graph $G = G_1 \cup G_2$ that contains the union of G_1 and G_2's nodes and edges. This technique can potentially increase coverage if the different executions explore different parts of the app. We use this graph union-based *cumulative coverage* as a basis for comparing manual exploration with automated exploration.

Results. In Table 2, we present the activity count and a summary of the activity coverage achieved manually by the 7 users. Column 2 presents the number of activities in each app, including ads. Column 3 presents the number of activities, excluding ads (hence these numbers indicate the maximum the number of activities users can explore without clicking on ads). Column 4 shows the cumulative activity coverage, i.e., when combining coverage via graph union. The percentages are calculated with respect to column 3, i.e., non-ad activities; we decided to exclude ads as they are not related to core app functionality. The complete dataset (each app, each user) is available in Table 5 in the Appendix.

We can see that in regular exploration cumulative coverage is quite low across users: mean[3] cumulative coverage is 30.08% across all apps. We now proceed to explain why that is the case.

Why are so few activities explored? The "Missed activities" group of columns in Table 2 shows, for each app, the number of activities that *all* users missed (first column in the group), and the reason why these activities were missed (the remaining 6 columns in the group). We were able to group the missing activities into the following categories:

- **Unexplored features.** Specific features can be missed because users are not aware of/interested in those features. For example, apps such as Dictionary.com or Tiny Flashlight + LED, provide a "widget" feature, i.e., an app interface typically wider than a desktop icon to provide easy to access functionality. Another example is "voice search" functionality in the Dolphin Browser browser, which is only explored when users search by voice.
- **Social network integration.** Many apps offer the option to share information on social networking sites—third-party sites such as Facebook or Twitter, or the app's own network, e.g., Shazam. During normal app use, users do not necessarily feel compelled to share information. These missed activity types appear in the "social" column.
- **Account.** Many apps can function, e.g., watch videos on YouTube, without the user necessarily logging-in. If an user logs into her account, she can see her profile and have access to further activities, e.g., account settings or play-lists on YouTube. In those cases where users did not have (or did not log into) an account, account-specific activities were not exercised.
- **Purchase.** E-commerce apps such as Amazon Mobile offer functionality such as buy/sell items. If test users do not conduct such operations, those activities will not be explored.
- **Options.** When users are content with the default settings of the app and do not change settings, e.g., by accessing the "options" menu, options activities are not exercised.
- **Ads.** Many free apps contain ad-related activities. For example, in Angry Birds, all the activities but one (play game) were ad-related. Therefore, in general, free apps contain more activities than their paid counterparts—see Angry Birds, Advanced Task Killer, Dictionary.com. When users do not click on ads, the ad-related activities are not explored.

3.2 Method Coverage

Since activity coverage was on average about 30%, we would expect method coverage to be low as well, as the

[2] We had access to 192 ($28 \times 7 - 4$) instead of 196 runs; due to the unavailability of two user study subjects, we could not collect app execution data for two users for the apps IMDb Movies & TV and BBC News.

[3] We use geometric mean for all mean computations due to large standard deviations.

App	Activities		Activity coverage (%)	Missed activities							Methods	Method coverage (%)
	Total #	Excluding ads	Users 1–7 (cumulative)	# Missed	Features	Social	Account	Purchase	Options	Ads		Users 1–7 (cumulative)
Amazon Mobile	39	36	25.64	30	●		●	●			7,154	4.93
Angry Birds	8	1	100	6						●	6,176	10.98
Angry Birds Space Premium	1	1	100	0							7,402	0.68
Advanced Task Killer	7	6	70	3	●				●		3,836	11.46
Advanced Task Killer Pro	6	6	57	2	●				●		427	21.32
BBC News	10	10	52.34	3	●				●		257	7.69
CNN	42	39	19.05	10	●		●				7,725	4.97
Craigslist Mobile	17	15	42	35	●	●	●				2,095	10.76
Dictionary.com	22	18	61	11	●	●				●	2,784	13.83
Dictionary.com Ad-free	15	15	73.33	4	●						1,272	19.10
Dolphin Browser	56	56	12.5	49	●	●					13,800	13.26
ESPN ScoreCenter	5	5	60	2					●		4,398	1.35
Facebook	107	107	5.60	95	●				●		21,896	1.69
Tiny Flashlight + LED	6	4	66.67	4	●					●	1,578	15.91
Movies by Flixster	68	67	23.3	48	●	●				●	7,490	5.32
Gas Buddy	38	33	30.2	29	●	●	●			●	5,792	9.13
IMDb Movies & TV	39	37	25.64	30		●				●	8,463	4.60
Instant Heart Rate	17	14	29.4	15	●	●				●	2,002	4.60
Instant Heart Rate - Pro	17	16	13.2	16	●	●				●	1,927	5.13
Pandora internet radio	32	30	12.5	30			●		●	●	7,620	3.21
PicSay - Photo Editor	10	10	10	9					●	●	1,580	4.39
PicSay Pro - Photo Editor	10	10	33.33	9					●		-[a]	-
Shazam	38	37	15.8	36	●		●			●	9,884	9.43
Shazam Encore	38	37	22.3	33	●	●	●			●	9,914	9.32
WeatherBug	29	24	29	24	●	●				●	7,948	8.15
WeatherBug Elite	28	28	14.30	24	●	●					8,194	6.39
YouTube	18	18	27.77	17	●		●				11,125	5.13
ZEDGE	34	34	38.9	18	●					●	6287	9.27
Mean			*30.08*									*6.46*

[a] We could not get method profiling data for PicSay Pro as the profiler could not analyze it.

Table 2. The results of our user study.

methods associated with unexplored activities will not be invoked. The last group of columns in Table 2 shows the total number of methods for each app, as well as the percentages of methods covered by Users 1 and 2–7, respectively. We can see that method coverage is quite low: 6.46% is the mean cumulative coverage for users 1–7. The complete dataset (each app, each user) is available in Table 6 in the Appendix. In Section 6.1 we provide a detailed account of why method coverage is low.

4. Approach

We now present the two thrusts of our approach: Targeted Exploration, whose main goal is to achieve fast activity exploration, and Depth-first Exploration, whose main goal is to systematically explore app states. The two strategies are not complementary; rather, we devised them to achieve specific goals. Depth-first Exploration tests the GUI similarly to how an user would, i.e., clicking on objects or editing text boxes, going to newer activities and then returning to the previous ones via the “back” button. Targeted Exploration is designed to handle some special circumstances: it can list all the activities which can be called from other apps or background services directly without user intervention, and generates calls to invoke those activities directly. The Targeted strategy was required because not all activities are invoked through user interaction. Both strategies can start the exploration in the

```
//class NewsListActivity extends TitleActivity
public void onItemClick (...)
{
  Intent localIntent = new Intent(this, NewsStoryActivity.class);
  ...
  startActivityWithoutAnimation( localIntent );
}

//class TitleActivity extends RootActivity
public startActivityWithoutAnimation(Intent paramIntent)
{ super.startActivityWithoutAnimation(paramIntent); }

//class RootActivity
protected void startActivityWithoutAnimation(Intent paramIntent)
{ startActivity (paramIntent );...}
```

Figure 4. Intent passing through superclasses in NPR News.

app entry points, inject user-like GUI actions and generate callbacks to invoke certain activities.

To illustrate the main principles behind these strategies, let us get back to the flow of the Amazon Mobile app shown in Figure 1. In Targeted Exploration, the SATG is constructed via static analysis, and our exploration focuses on quickly traversing activities in the order imposed by the SATG—in the Amazon Mobile case, we quickly and automatically move from Main Activity to Search Activity to Search List Activity. In Depth-first Exploration, we use the app entry points from the SATG (that is, nodes with no incoming edges) to start the exploration. Then, in each activity, we retrieve the GUI elements and exercise them systematically. In the Amazon Mobile case, we start with the Main Activity and exercise all its contained GUI elements systematically (which will lead to eventually exploring Search Activity and from there, Search List Activity); this is more time-consuming, but significantly increases method coverage.

We first discuss how the SATG is constructed (Section 4.1), then show how it drives Targeted Exploration (Section 4.2); next we present Depth-first Exploration (Section 4.3) and finally how our approach supports test case generation and debugging (Section 4.4).

4.1 SATG Construction

Determining the correct order in which GUI elements of an Android app should be explored is challenging. The main problem is that the control flow of Android apps is non-standard: there is no main(), but rather apps are centered around callbacks invoked by the Android framework in response to user actions (e.g., GUI events) or background services (e.g, GPS location updates). This makes reasoning about control flow in the app difficult. For example, if the current activity is A and a transition to activity B is possible as a result of user interaction, the methods associated with A will not directly invoke B. Instead, the transition is based on a generic intent passing logic, which we will discuss shortly. We realized that intent passing and consequently SATG construction can be achieved via data-flow analysis, more specifically taint tracking. *Hence our key insight is that SATG construction can be reduced to a taint-tracking problem.*

Coming back to our prior example with the A and B activities, using an appropriately set-up taint analysis, we taint B; if the taint can reach an actual invocation request from A, that means activity B is reachable from A and we add an A→B edge to the SATG. The "glue" for activity transitions is realized by objects named *Intents*. Per the official Android documentation [33], an Intent is an "abstract description of an operation to be performed." Intents can be used to start activities by passing the intent as an argument to a startActivity -like method. Intents are also used to start services, or send broadcast messages to other apps. Hence tracking taint through intents is key for understanding activity flow.

We now provide several examples to illustrate SATG construction using taint analysis over the intent passing logic. In Figure 3, on the left we have valid Android Java code for class A that implements an activity. Suppose the programmer wants to set up a transition to another activity class, B. We show three examples of how this can be done by initializing the intent initialized in a method of A, say A.foo(), and coupling it with the information regarding the target activity B. In Example 1, we make the A→B connection by passing the class names to the intent constructor. In Example 2, the connection is made by setting the B's class as the intent's class. In Example 3, B is set to be called as a component of the intent. Our analysis will tag these Intent object declarations (**new** Intent()) as *sources*. Next, the taint analysis will look for *sinks*; in our example, the tagged sinks are startActivity , startActivityForResult , and startActivityIfNeeded . Of course, while here we show the Java code for clarity, our analysis operates on bytecode. Taint tracking is shown in Figure 3 (center): after tagging sinks and sources, the taint analysis will propagate dataflow facts through the app code, and in the end check whether tainted sources reach sinks. For all (source, sink) pairs for which taint has been detected, we add an edge in the SATG (Figure 3 (right)). Hence the general principle for constructing the SATG is to identify Intent construction points as sources, and activity start requests as sinks.

A more complicated, real-world example, of how our analysis tracks taint through a class hierarchy is shown in Figure 4, a code snippet extracted from the NPR News app. An Intent is initialized in NewsListActivity.onItemClick (...) , tagged as a source, and passed through the superclass TitleActivity to its superclass RootActivity. The startActivity (on the last line) is tagged as a sink. When the analysis concludes, based on the detected taint, we add an edge from NewsListActivity to NewsStoryActivity in the SATG.

4.2 Targeted Exploration

We now proceed to describing how we perform Targeted Exploration using the SATG as input. Figure 5 provides an overview. The automatic explorer, running on a desktop or

Tagging sources and sinks

```
public class A extends Activity {
 ...
public void foo()
{
  /* Example 1 */
  Intent intent1 = new Intent(A, B); //tagged as source
   ...
   startActivity (intent1); //tagged as sink

  /* Example 2 */
  Intent intent2=new Intent(); //tagged as source
  intent.setClass(this, B.class);
   ...
   startActivityForResult (intent2, requestCode); //tagged as sink

  /* Example 3 */
  Intent intent3 = new Intent(); //Intent object tagged as source
  intent.setComponent(new ComponentName("package.name", "B"));
   ...
   startActivityIfNeeded (intent3, requestCode); //tagged as sink
}}
```

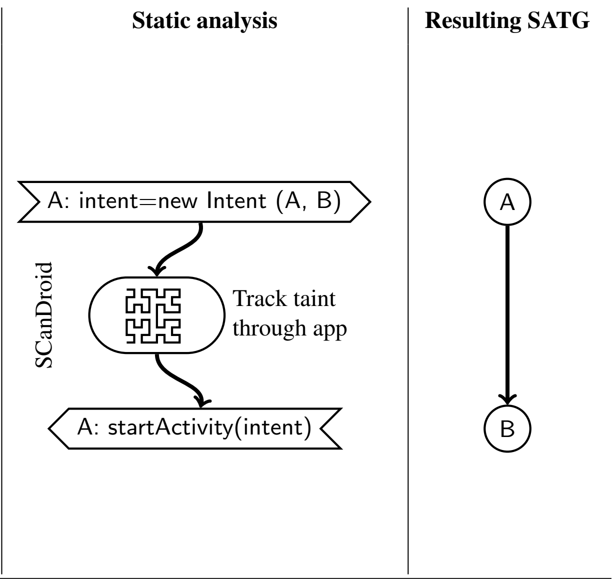

Figure 3. Constructing SATG's with taint analysis: sources and sinks are tagged automatically (left), taint is tracked by SCanDroid (center); the resulting SATG (right).

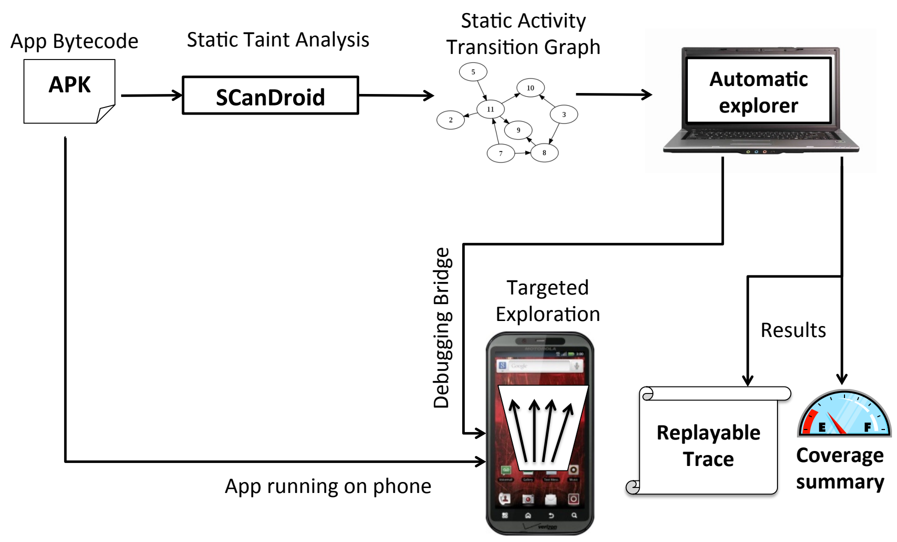

Figure 5. Overview of Targeted Exploration in A^{3}E.

laptop, orchestrates the exploration. The explorer first reads the SATG constructed by SCanDroid (a static dataflow analyzer that we customized to track intent tainting, as described in Section 4.1) from the app's bytecode, and then starts the app on the phone. Our SATG construction algorithm lists all the exported activities, and entry point activities. Exported activities are activities that can be independently called from within or outside the app; they are marked as such by setting the parameter `exported=true` in the manifest file. Note that not all activities can be called from outside—some have to be reached by the normal process, primarily for security reasons and to maintain application workflow. For example, when an activity can receive parameters from a previous activity, the parameters may contain security information that is limited to the application domain. Therefore, we cannot just "jump" to any arbitrary activity.

Next, the explorer runs the Targeted Exploration algorithm, which we will describe shortly. The explorer controls the app execution and communicates with the phone via the Android Debugging Bridge. The result of the exploration consists of a replayable trace—a sequence of events that can be replayed using our RERAN tool [7]—as well as coverage information.

We now proceed to describing the algorithm behind targeted exploration; parts of the algorithm run on the phone, parts in the automatic explorer. In a nutshell, the SATG contains edges A→B indicating legal activity transitions. Assuming we are currently exploring activity A, we have two

Algorithm 1 Targeted Exploration

Input: SATG $G_S = (V_S, E_S)$

```
 1: procedure TARGETEDEXPLORATION($G_S$)
 2:     for all nodes $A_i$ in $V_S$ that are entry points do
 3:         Switch to activity $A_i$
 4:         $currentActivity \leftarrow A_i$
 5:         for all edges $A_i \rightarrow A_j$ in $E_S$ do
 6:             if $A_j$ is exportable then
 7:                 Switch to activity $A_j$
 8:                 $currentActivity \leftarrow A_j$
 9:                 $G'_S \leftarrow$ subgraph of $G_S$ from starting node $A_j$
10:                 TARGETEDEXPLORATION($G'_S$)
11:             end if
12:         end for
13:         $guiElementSet \leftarrow$ EXTRACTGUIELEMENTS($currentActivity$)
14:         for each $guiElement$ in $guiElementSet$ do
15:             exercise $guiElement$
16:             if there is an activity transition to not-yet-explored activity $A_n$ then
17:                 $G'_S \leftarrow$ subgraph of $G_S$ from starting node $A_n$
18:                 $currentActivity \leftarrow A_n$
19:                 TARGETEDEXPLORATION($G'_S$)
20:             end if
21:         end for
22:     end for
23: end procedure
```

cases for B: (1) B is "exportable",[4] that is, reachable from A but not a result of local GUI interaction in A; or (2) B is reached from A as a result of local GUI interaction in A. In case (1) we switch to B directly, and in case (2) we switch to B when exploring A's GUI elements. In either case, exploration continues recursively from B.

Algorithm 1 provides a precise description of the Targeted Exploration approach. The algorithm starts with the SATG as input. First, we extract the app's entry point activities from the SATG (line 2) and start exploration at one of these entry points A_i (lines 3–4). We look for all the exportable activities A_j that have an incoming edge from A_i (lines 5–6). We then switch to each of these exportable activities and invoke the algorithm recursively from A_j (lines 7–10). Activities A_n that are not exportable but reachable from A_i will be switched to automatically as a result of local GUI exploration (lines 13–16) and then we invoke the algorithm recursively from A_n (lines 17–19).

The advantage of Targeted Exploration is that it can achieve activity coverage fast—we can switch to exportable activities without firing GUI events.

[4] The list of exportable activities is available in the `AndroidManifest.xml` file included with the app.

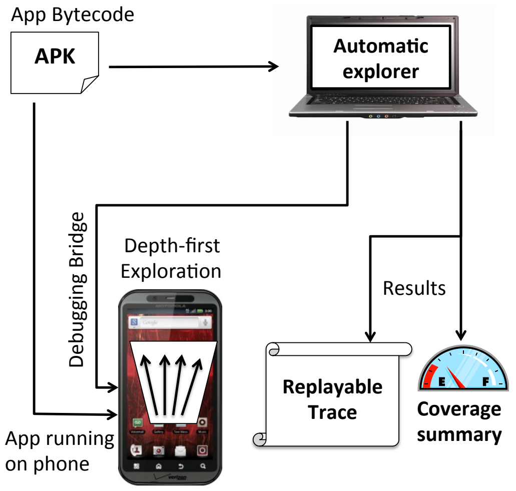

Figure 6. Overview of Depth-first Exploration in A^{3}E.

Algorithm 2 Depth-First Exploration

Input: Entry point activities $|A|$

```
 1: procedure DFE($|A|$)
 2:     for all nodes $A_i$ in $|A|$ do
 3:         Switch to activity $A_i$
 4:         DEPTHFIRSTEXPLORATION($A_i$)
 5:     end for
 6: end procedure
 7:
 8: procedure DEPTHFIRSTEXPLORATION($A_i$)
 9:     $guiElementSet \leftarrow$ EXTRACTGUIELEMENTS($A_i$)
10:     for each $guiElement$ in $guiElementSet$ do
11:         excercise $guiElement$
12:         if there is an activity transition to not-yet-explored activity $A_n$ then
13:             DEPTHFIRSTEXPLORATION($A_n$)
14:             Switch back to activity $A_i$
15:         end if
16:     end for
17: end procedure
```

4.3 Depth-First Exploration

We now proceed to presenting Depth-First Exploration, an approach that takes more time but can achieve higher method coverage. As it is a dynamic approach, Depth-First Exploration can be performed even when the tester does not have activity transition information (i.e., the SATG) beforehand. As the name suggests, this technique employs depth-first search to mimic how an actual user would interact with the app.

Figure 5 provides an overview. In this case, no SATG is used, but the automatic explorer runs a different, Depth-first Exploration algorithm, which we will describe shortly. The rest of the operations are identical with Targeted Exploration, that is, the explorer orchestrates the exploration and the results are a replayable trace and coverage information.

Algorithm 2 provides the precise description of the Depth-first Exploration approach. Similar to Targeted Exploration, we first extract the entry point activities from the app's APK; these activities will act as starting points for the exploration. We then choose a starting point A_i and start depth-first exploration from that point (lines 1–5). For each activity A_i, we extract all its GUI elements (line 9). We then systematically exercise the GUI elements by firing their corresponding event handlers (lines 10–11). Whenever we detect a transition to a new activity A_n, we apply the same algorithm recursively on A_n (line 13). This process continues in a depth-first manner until we do not find any transition to a newer activity after exercising all the GUI elements in that screen. We then go back to the previous activity and continue exploring its view elements (line 14).

4.4 Replayable Test Cases and Debugging

During exploration, A^{3}E automatically records the event stream using RERAN, a low-overhead record-and-replay tool [7], so that the exploration, or parts thereof, can be replayed. This feature helps users construct test cases that can later be executed via RERAN's replay facility. In addition, the integration with RERAN facilitates debugging—if the app crashes during exploration, we have the exact event stream that has led to the crash; assuming the crash is deterministic, we can reproduce it by replaying the event stream.

5. Implementation

We now proceed to presenting the experimental setup, implementation details, and measurement procedures used for constructing and evaluating A^{3}E.

5.1 Setup and Tools

The smartphones used for experiments were Motorola Droid Bionic running Android version 2.3.4, Linux kernel version 2.6.35. The phones have Dual Core ARM Cortex-A9 CPUs running at 1GHz. We controlled the experiments from a MacBook Pro laptop (2.66 GHz dual-core Intel Core i7 with 4GB RAM), running Mac OS X 10.8.3.

For the user study, we used RERAN, a tool we developed previously [7] to record user interaction so we could replay and analyze it later.

SCanDroid is a tool for static analysis on Dalvik bytecode developed by other researchers and us [25]. For this work we extended SCanDroid in two directions: (1) to tag intents and activity life-cycle methods as sinks and sources so we can construct the SATG, and (2) to list all the app-defined methods—this information was used for method coverage analysis.

5.2 Measuring Coverage

Activity coverage. The automatic explorer keeps track of AR, the number of successfully explored activities, via the `logcat` utility provided by Android Debug Bridge (`adb`) tool from the Android SDK.

The total number of activities, AT, was obtained offline: we used the open source `apktool` to extract the app's manifest file from the APK and parsed the manifest to list all the activities. From the AT and AR we exclude "outside" activities, as those are not part of the app's code base. Examples of outside activities are ad-related activities and external system activities (browser, music player, camera, etc.)

Method coverage. Android OS provides an Application Manager (`am`) utility that can create method profiles on-the-fly, while the app is running. To measure ME, the number of methods called during execution, we extracted the method entries from the profiling data reported by `am`. We measured MT, the total number of methods in an app, via static analysis, by tailoring SCanDroid to find and list all the virtual and declared method calls within the app. Note that third-party code is not included in ME and MT computation (Section 3.2).

5.3 Automatic Explorer

GUI element extraction and exercising is required for both Targeted and Depth-first Exploration. To explore GUI elements, A^{3}E "rips" the app, i.e., extracts its GUI elements dynamically using the Troyd tool [20] (which in turn is based on Robotium [18]). Robotium can extract and fire event handlers for a rich set of GUI elements. This set includes lists, buttons, check boxes, toggle buttons, image views, text views, image buttons, spinners, etc. Robotium also provides functionality for editing text boxes, clearing text fields, clicking on text, clicking on screen positions, clicking on hardware home menu, and back button. Troyd allows developers to write Ruby scripts that can drive the app using the aforementioned functionality offered by Robotium (though Troyd does not require access to the app's source code).

A^{3}E is built on top of Troyd. We modified Troyd to allow automatic navigation through the app, as follows. Each Android screen consists of GUI elements linked via event handlers. Simply invoking all the possible event handlers and firing the events associated with them would be incorrect—the app has to be in a state where it can accept the events, which we detected by interacting with the live app. Hence A^{3}E relies on live extraction of the GUI elements that are actually present on the screen (we call a collection of such elements a *view*). We then systematically map the related event handlers, and call them mimicking a real user. This run-time knowledge of views was essential for our automated explorer to work. Once we get the information of the views, we can systematically fire the correct actions.

As described in Section 3.1, our test users tended to skip features such as options, ads, settings, or sharing via social networks. To cover such activities and functionality, A^{3}E employs several strategies. A^{3}E automatically detects activities related to special responsibility, such as log in screen, social networking, etc. We created sets of credential information (e.g., username/password pairs) that A^{3}E then sends to the app just like a user would do to get past the screen, and continues the exploration from there.

As we implemented our approach on top of the Robotium testing framework, we had to compensate for its limitations. One such limitation was Robotium's inability to generate and send gestures, which would leave many kinds of views incompletely exercised when using Robotium alone. To address this limitation we wrote a library of common simple gestures (horizontal and vertical swipes, straight line swipes, scrolling). We leave complex multi-touch gestures such as pinching and zooming to future work; as explained in our prior work [7], synthesizing complex, multi-touch gestures is non-trivial.

In addition to the gesture library and log-on functionality, A^{3}E also supports microphone, GPS, compass, and accelerometer events. However, certain apps' functionality required complex inputs, e.g., processing a user-selected file. Feeding such inputs could be achieved via OS-level record and replay, a task we leave to future work.

With this library of input events, and GUI-element invocation strategies at hand, A^{3}E uses the appropriate exploration algorithm depending on the kind of exploration we perform, as described next.

5.4 Targeted Exploration

Section 4.1 described the intent passing logic among internal app activities, and Targeted Exploration uses the pre-constructed SATG to explore these intra-app activity paths. However, the Android platform also allows activities to be called from external apps through implicit messaging by intents, as follows: apps can define intent "filters" to notify the OS that certain app activities accept intents and can be started when the external app sends such an intent. Therefore, when systematically exercising an app's GUI elements, there is a chance that these externally-callable activities are missed if they are not reachable by internal activities. In addition to intent filters, developers can also identify activities as "exported," by defining `android:exported="true"` in the manifest file. This will allow activities to be called from outside the app. When implementing Targeted Exploration we also invoked these externally-callable activities so we do not miss them when they are not reachable from internal activities.

5.5 Depth-First Exploration

With the infrastructure for dynamic GUI element extraction and event firing at hand, we implemented Depth-first Exploration using a standard depth-first strategy: each time we find a transition to a new activity, we switch to that activity and thus we enter deeper levels of the activity hierarchy. This process continues until we cannot detect any more transition from the current activity. At this point we recursively go back to the last activity and start exploring from there.

6. Evaluation

We now turn to presenting experimental results. From the 28 examined apps presented in Section 3, we were able to explore 25; 3 apps could not be explored (Angry Birds, Angry Birds Space premium, Facebook) because they are written primarily in native code, rather than bytecode, as explained in Section 6.2.

We first evaluate our automated exploration techniques on these apps in terms of effectiveness and efficiency (Section 6.1), then discuss app characteristics that make them more or less amenable to our techniques (Section 6.2).

6.1 Effectiveness and Efficiency

Activity coverage. We present the activity coverage results in Table 3. Column 2 shows the number of nodes in the SATG, that is, the number of activities in each app, excluding ads. The grouped columns 3–5 show the activity coverage in percents, in three scenarios: the cumulative coverage for users 1–7, coverage attained via Targeted Exploration, and coverage attained via Depth-first Exploration. We make several observations. First, systematic exploration increases activity coverage by a factor of 2x, from 30.08% attained by 7 users cumulatively to 64.11% and 59.39% attained by Targeted and Depth-first Exploration, respectively. *Hence our approach is effective at systematically exploring activities.* Second, SATG construction pays off; because it relies on statically-discovered transitions, Targeted Exploration will be able to fire transitions in cases where Depth-first Exploration cannot, and overcome a limitation of dynamic tools that start the exploration inside the app.

Method coverage. The method coverage results are shown in the last columns of Table 3. The methods column shows the number of methods defined in the app, excluding third-party code. The next columns show activity coverage in percents, in three scenarios: the cumulative coverage for users 1–7, coverage attained via Targeted Exploration, and coverage attained via Depth-first Exploration. We make several observations. First, systematic exploration increases method coverage by about 4.5x, from 6.46% attained by 7 users cumulatively to 29.53% and 36.46% attained by Targeted and Depth-first Exploration, respectively. *Hence our approach is effective at systematically exploring methods.* Second, the lengthier, systematic exercising of each activity element performed by Depth-first Exploration translates to better exploration of methods associated directly (or transitively) with that activity.

Exploration time. In Table 4 we show the time required for exploration. Column 2 contains the static analysis time, which is required for SATG construction; this is quite efficient, at most 10 minutes but typically 4 minutes or less. We measured exploration time by letting both Targeted and Depth-first explorations run to completion, that is until we have explored all entry point activities and activities we

App	Acti-vities	Activity coverage (%)			Methods	Method coverage (%)		
		Users 1–7 (cumulative)	Targeted	Depth-first		Users 1–7 (cumulative)	Targeted	Depth-first
Amazon Mobile	36	25.64	63.90	58.30	7,154	4.93	28.1	45.10
Angry Birds	-	100	-	-	-	10.98	-	-
Angry Birds Space Premium	-	100	-	-	-	0.68	-	-
Advanced Task Killer	6	70	83.33	83.33	420	11.46	59.76	62.86
Advanced Task Kill. P.	6	57	83.30	83.30	257	21.32	39.30	73.20
BBC News	10	52.34	80.00	80.00	3,836	7.69	31.80	37.40
CNN	39	19.05	69.23	61.54	9,269	4.97	29.88	29.97
Craigslist Mobile	15	42	73.30	66.70	2,095	10.76	30.50	41.10
Dictionary.com	18	61	83.33	72.22	3,881	13.83	44.29	44.62
Dictionary.com Ad Free	15	73.33	100	80	1,846	19.10	47.72	49.13
Dolphin Browser	56	12.50	42.86	37.50	17,007	13.26	42.92	43.37
ESPN ScoreCenter	5	60	80.00	80.00	4,398	1.35	16.10	31.20
Facebook	107	5.60	-	-	-	1.69	-	-
Tiny Flashlight + LED	4	66.67	75	75	1,837	15.91	28.03	47.63
Movies by Flixster	67	23.30	77.60	61.20	10,151	5.32	29.50	31.80
Gas Buddy	33	30.20	72.70	63.60	5,792	9.13	31.40	47.80
IMDb Movies & TV	37	25.64	54.10	62.10	11,950	4.60	29.80	32.40
Instant Heart Rate	14	29.40	42.86	35.71	2,407	4.60	20.40	23.18
Instant Heart Rate - Pro	16	13.20	37.50	37.50	2,514	5.13	26.05	26.21
Pandora internet radio	30	12.50	80.0	76.70	7,620	3.21	21.10	31.70
PicSay - Photo Editor	10	10	50	40	1,458	4.39	25.58	27.43
PicSay Pro - Photo Editor	10	33.33	50	40	-	-	-	-
Shazam	37	15.80	45.95	45.95	12,461	9.43	34.74	35.67
Shazam Encore	37	22.30	45.90	51.40	9,914	9.32	29.10	36.30
WeatherBug free	24	29	54.17	45.83	7,744	8.15	40.05	40.33
WeatherBug Elite	24	14.30	91.70	87.50	7,948	6.39	17.20	25.70
YouTube	18	27.77	55.56	50	14,550	5.13	26.95	26.99
ZEDGE	34	38.90	67.60	67.60	6,287	9.27	16.60	24
Mean		*30.08*	*64.11*	*59.39*		*6.46*	*29.53*	*36.46*

Table 3. Evaluation results: activity and method coverage.

could transitively reach from them. We imposed no timeout. Columns 3 and 4 show the dynamic exploration time, 18–236 minutes for Targeted Exploration and 39–239 minutes for Depth-first Exploration. *Hence our approach performs systematic exploration efficiently*. As expected, Targeted Exploration is faster, even after adding the SATG construction time, because it can fire activity transitions directly (there were two exceptions to this, explained shortly). We believe that these times are acceptable and well worth it, considering the provided benefits: a replayable trace that can be used as basis for dynamic analysis or constructing test cases.

Targeted vs. Depth-first Exploration. While the two exploration techniques implemented in A^{3}E have similar goals (automated exploration) they have several differences.

Targeted Exploration requires a preliminary static dataflow analysis stage to construct the SATG. However, once the SATG is constructed, targeted exploration is fast, especially if a high number of activities are exportable, hence we can quickly achieve high activity coverage by directly switching to activities.

Depth-first Exploration does not require a SATG, but the exploration phase is slower—systematically exercising all GUI elements will prolong exploration. However, this prolonged exploration could be a worthy trade-off since Depth-first Exploration achieves higher method coverage.

We now present some examples that illustrate tradeoffs and shed light on some counterintuitive results. For Advanced Task Killer and ESPN ScoreCenter we attain similar activity coverage but for ESPN ScoreCenter method coverage is significantly lower. The reason for this is app structure:

App	Time		
	SATG (seconds)	Targeted (minutes)	Depth-first (minutes)
Amazon Mobile	222	123	131
Advanced Task Killer	39	41	47
Advanced Task Kill. P.	24	27	58
BBC News	68	18	52
CNN	14	158	161
Craigslist Mobile	43	83	91
Dictionary.com	66	113	131
Dictionary.com Ad Free	45	153	156
Dolphin Browser	595	171	179
ESPN ScoreCenter	42	22	44
Tiny Flashlight + LED	52	33	39
Movies by Flixster	53	228	219
Gas Buddy	157	109	124
IMDb Movies & TV	107	135	126
Instant Heart Rate	56	47	51
Instant Heart Rate - Pro	50	48	49
Pandora internet radio	92	89	111
PicSay - Photo Editor	36	119	121
PicSay Pro - Photo Ed.	40	112	129
Shazam	64	236	239
Shazam Encore	248	188	230
WeatherBug free	120	69	107
WeatherBug Elite	119	115	124
YouTube	200	131	135
ZEDGE	124	97	114
Mean	*74*	*87*	*104*

Table 4. Evaluation results: time taken by SATG construction and exploration. Note the different units across columns (seconds vs. minutes).

ESPN ScoreCenter employs complex activities, i.e., different layouts within an activity with more features to use. The Targeted algorithm quickly switches the activities without exploring layout elements in depth, while Depth-first takes longer to exercise the layout elements thoroughly. For the same app structure reason, Targeted exploration finishes significantly faster for Advanced Task Killer Pro and BBC News.

Most apps show better activity coverage for Targeted than Depth-first Exploration. This is primarily because they have multiple entry points, or they have activities with intent filters to allow functionality to be invoked from outside the app—starting the exploration from within the app for intent-filter based activities which are not invoked from within the app will fail to discover those activities. For instance, Amazon Mobile has a bar-code search activity which was missed during the Depth-first search, but the Targeted Exploration succeeded to call the activity with the information from the SATG. An exception from this were IMDb Movies & TV and Shazam Encore: both apps have lower activity coverage in Targeted Exploration than Depth-first. After investigation we found that some activities could be invoked by Targeted Exploration using intent filters with parameters, but Targeted Exploration failed to exercise the activities; this was due to specific input parameters what Targeted exploration failed to produce. The Depth-first search exercised the app as a user would and landed on those particular pages from the right context with correct input parameters, achieving higher coverage. For example the "play trailer" activity in IMDb Movies & TV was not run during Targeted Exploration as it does not have the required parameter, in this case the video file name or location.

The exploration times depicted in Table 4 also have some interesting data points. The exploration time largely depends on the size and GUI complexity of the app. Normally, Depth-first Exploration is slower than Targeted because of the switch back (line 14 in Algorithm 2). Two apps, though, Movies by Flixster and IMDb Movies & TV do not conform to this. Upon investigation, we found that activities in these apps form (almost) complete graphs, with many activities being callable from both inside the app or outside the app (but when called from outside, parameters need to be set correctly). Depth-first reached the activities naturally with the correct parameters, whereas Targeted had to back off repeatedly for some activities when attempting to invoke those activities before parameters were properly constructed.

6.2 Automatic Exploration Catalysts and Inhibitors

We now reflect on our experience with the 28 apps and discuss app features that facilitate or hinder exploration via A^{3}E. The reasons that prevent us from achieving 100% coverage are both fundamental and technical. Fundamental reasons include the presence of activities that cannot be automatically explored due to their nature, and for a good reason. For example, in Amazon Mobile, purchase-related activities could not be explored because server-side validation would be required: to go to the "purchased" state, we would first need to get past the checkout screen where credentials and credit card are verified.

Complex gestures and inputs. Our techniques can get good traction for apps built around GUI elements provided by the Android SDK, such as text boxes, buttons, images, text views, list views, check boxes, toggle buttons, spinners, etc. However, some apps rely on complex, app-specific gestures. For example, in the PicSay-Photo Editor app, most of the view elements are custom-designed objects, e.g., artwork and images that are manipulated via specific gestures. Our gesture and sensor libraries (Section 5.3) partially address this limitation.

Task switching. Another inhibitor is task switching: if our app under test loses focus, and another app gains focus, we cannot control this second app. For example, if an app invokes the default file browser to a specific file location, the A^{3}E explorer will stop tracking, because the file browser runs in a different process, and will resume tracking when the app under test regains focus. This is for a good reason, as part of the Android app isolation policy, but we cannot track GUI events in other apps.

Service-providing apps. Some apps can act as service providers by implementing services that run in the background. For example, WeatherBug runs in the background reporting the weather; Advanced Task Killer runs in the background and kills apps which were not accessed for a specific amount of time. Hence for such apps we cannot explore methods that implement background service-providing functionality because they are not reachable from the GUI.

Native code. Finally, our static analysis works on Dalvik VM bytecode. If the app uses native code instead of Dalvik bytecode to handle GUI elements, we cannot apply our techniques. For example, the two Angry Birds apps use native Open GL code to manage drawing on the canvas, hence our GUI element extraction does not have access to the native GUI information. We could not explore the Facebook app for the same reason.

7. Related Work

The work of Rastogi et. al. [10] is most closely related to ours. Their system, named Playground, runs apps in the Android emulator on top of a modified Android software stack (TaintDroid); their end goal was dynamic taint tracking. Their work introduced an automated exploration technique named intelligent execution (akin to our Depth-first Exploration). Intelligent execution involves starting the app, dynamically extracting GUI elements and exploring them (with pruning for some apps to ensure termination) according to a sequencing policy authors have identified works best—explore input events first, then explore action providers such as buttons. They ran Playground on an impressive 3,968 apps and achieved 33% code coverage on average. The are several differences between their approach and A^{3}E. First, they run the apps on a modified software stack on top of the Android emulator, whereas we run apps on actual phones using an unmodified software stack. The emulator has several limitations [32], e.g., no support for actual calls, USB, Bluetooth, in addition to lacking physical sensors (GPS, accelerometer, camera), which are essential for a complete app experience. Second, Playground, just like our Depth-first Exploration, can miss activities, as Table 4 shows—hence our need for Targeted Exploration which uses static analysis to find all the possible activities and entry points. Third, their GUI element exploration strategy is based on heuristics, ours is depth-first; both strategies have advantages and disadvantages. Fourth, since we ran our experiments on actual phones with unmodified VMs we could not collect instruction coverage, so we cannot directly compare our coverage numbers with theirs.

Memon et. al.'s line of work on GUI testing for desktop applications [12–14] is centered around event-based modeling of the application to automate GUI exploration. Their approach models the GUI as an *event interaction graph* (EIG); the EIG captures the sequences of user actions that can be executed on the GUI. While the EIG approach is suitable for devising exploration strategies for GUI testing in applications with traditional GUI design, i.e., desktop applications, several factors pose complications when using it for touch-based smartphone apps. First, and most importantly, transitions associated with non-activity elements cannot be easily captured as a graph. There is a rich set of user input features associated with smartphone apps in general (such as gestures—swipes, pinches and zooms) which are not tightly bound to a particular GUI object such as a text box or a button, so there is not always a "current node" as with EIG to determine the next action. For example, if the GUI consists of a widget overlapped on a canvas, each modeled as graphs, then the graph corresponding to the widget and the canvas combined has a set of nodes of size proportional to the product of number of nodes in the widget and canvas graphs; this quickly becomes intractable. Moreover, the next user action can affect the state of the canvas, widget, both, or neither, which again is intractable as it leads to an explosion in the number of edges in the combined graph. For example, activity *com.aws.android.lib.location.LocationListActivity* in the WeatherBug app contains different layouts, each containing multiple widgets; a horizontal swipe on any widget can change the layout, hence with EIGs we would have to represent this using a bipartite graph with a full set of edges among widgets in the two layouts. Second, as mobility is a core feature of smartphones, smartphone apps are built around multimodal sensors and sensor event streams (accelerometer, compass, GPS); these sensor events can change the state of the GUI, but are not easily captured in the EIG paradigm—many sensors do not exist on desktop systems and their supported actions are far richer than clicks or drags. Modeling such events to permit GUI exploration requires a different scheme compared to EIG; our event library (Section 5.3) and dynamic identification of next possible states allows us to generate multimodal events to permit systematic exploration. Third, Android app GUI state can be changed from outside the app, or by a background service. For example, an outside app can invoke an activity of another app through a system-wide callback which in the EIG model would a spontaneous transition into a node with no incoming edge. The behavior of the callback requests can certainly modify GUI states. Hence creating lists of action sequences that can be executed by a user on an interface will lead to exploring only a subset of GUI states. This is the reason why, while constructing the SATG, we analyze activities that accept intent filters and take appropriate action to design exploration test cases automatically. GUITAR [16] is a GUI testing framework for Java and Windows applications based on EIG. Android GUITAR [17] applies GUITAR to Android by extending Android SDK's MonkeyRunner tool to allow users to create their own test cases with a point-and-click interface that captures press events. In our approach, test case creation is automated.

Yang et. al. [11] implemented a tool for automatic exploration called ORBIT. Their approach uses static analysis on the app's Java source code to detect actions associated with GUI states and then use a dynamic crawler (built on top of Robotium) to fire the actions. We use static analysis on app bytecode to extract the SATG, as activities are stable, but then use dynamic GUI exploration to cope with dynamic layouts inside activities. They achieved significant statement coverage (63%–91%) on a set of 8 small open source apps; exploration took 80–480 seconds. We focus on a different problem domain: large real-world apps for which the source code is not available, so exploration times and coverage are not directly comparable.

Anand et al. [34] developed an approach named ACTEVE for concolic generation of events for testing Android apps whose source code is available. Their focus is on covering branches while avoiding the path explosion problem. ACTEVE generated test inputs for five small open source in 0.3–2.7 hours. Similarly, Jensen et al. [5] have used concolic execution to derive event sequences that can lead to a specific target state in an Android app, and applied their approach to five open source apps (0.4–33KLOC) and show that their approach can reach states that could not be reached using Monkey. Our focus and problem domains are different: GUI and sensor-driven exploration for substantial, popular apps, rather than focusing on covering specific paths. We believe that using concolic execution would allow us to increase coverage (especially method coverage), but it would require a symbolic execution engine robust enough to work on APKs of real-world substantial apps.

Monkey [15] is a testing utility provided by the Android SDK that can send a sequence of random and deterministic events to the app. Random events are effective for stress testing and fuzz testing, but not for systematic exploration; deterministic events have to be scripted, which involves effort, whereas in our case systematic exploration is automated. MonkeyRunner [24] is an API provided by the Android SDK which allows programmers to write Python test scripts for exercising Android apps. Similar to Monkey, scripts must be written to explore apps, rather than using automated exploration as we do.

Robotium [18] is a testing framework for Android that supports both black-box and white-box testing. Robotium facilitates interaction with GUI components such as menus, toasts, text boxes, etc., as it can discover these elements, fire related events, and generate test cases for exercising the elements. However, it does not permit automated exploration as we do.

Troyd [20] is a testing and capture-replay tool built on top of Robotium that can be used to extract GUI widgets, record GUI events and fire events from a script. We used parts of Troyd in our approach. However, Troyd cannot be used directly for either Targeted or Depth-first Exploration, as it needs input scripts for exercising GUI elements. Moreover, in its unmodified form, Troyd had a substantial performance overhead which slowed down exploration considerably—we had to modify it to reduce the performance overhead.

TEMA [41] is a collection of model-based testing tools which have been applied to Android. GUI elements form a state machine and basic GUI events are treated as keywords like events. Within this framework, test scripts can be designed and executed. In contrast, we extract a model either statically or dynamically and automatically construct test cases.

Android Ripper [9] is a GUI-based static and dynamic testing tool for Android. It uses a state-based approach to dynamically analyze GUI events and can be used to automate testing by separate test cases. Android Ripper preserves the state of the application where state is actually a tuple of a particular GUI widget and its properties. An input event triggers the change in the state and users can write test scripts based on the tasks that can modify the state. The approach works only on the Android emulator and thus cannot mimic sensor events properly like a real world application.

Several commercial tools provide functionality somewhat related to our approach, though their end-goals differ form ours. *Testdroid* [26] can record and run the tests on multiple devices. *Ranorex* [27] is a test automation framework. *Eggplant* [29] facilitates writing automated scripts for testing Android apps. *Framework for Automated Software Testing (FAST)* [28] can automate the testing process of Android apps over multiple devices.

Finally, there exist a variety of static [19, 21, 23, 25] and dynamic [22, 31] analysis tools for Android, though these tools are only marginally related to our work. We apply static analysis for SATG construction but our end goal is not static analysis. However, our replayable traces can fit very well into a dynamic analysis scenario as they provide significant coverage.

8. Conclusions

We have presented $\mathsf{A}^3\mathsf{E}$, an approach and tool that allow Android apps to be explored systematically. We performed a user study that has revealed that users tend to explore just a small set of features when interacting with Android apps. We have introduced Targeted Exploration, a novel technique that leverages static taint analysis to facilitate fast yet effective exploration of Android app activities. We have also introduced Depth-first Exploration, a technique that does not use static analysis, but instead performs a thorough GUI exploration which results in increased method coverage. Our approach has the advantage of permitting exploration without requiring the app source code. Through experiments on 25 popular Android apps, we have demonstrated that our techniques can achieve substantial coverage increases. Our approach can serve as basis for a variety of dynamic analysis and testing tasks.

Acknowledgments

We thank Lorenzo Gomez, Gavin Huang, Shashank Reddy Kothapalli, and Xuetao Wei for their assistance with the user study; Steve Suh and Jinseong Jeon for their valuable suggestions on the implementation; and the anonymous referees for their helpful comments. This research was supported in part by NSF grant CNS-1064646.

References

[1] Gartner. Inc. Gartner Says Worldwide PC Shipment Growth Was Flat in Second Quarter of 2012. July, 2012. URL `http://www.gartner.com/it/page.jsp?id=2079015`.

[2] Gartner. Inc. Gartner Highlights Key Predictions for IT Organizations and Users in 2010 and Beyond. January, 2010. URL `http://www.gartner.com/it/page.jsp?id=1278413`.

[3] IDC. Android and iOS Surge to New Smartphone OS Record in Second Quarter, According to IDC. August, 2012. URL `http://www.idc.com/getdoc.jsp?containerId=prUS23638712`.

[4] CNET. Android reclaims 61 percent of all U.S. smartphone sales. May, 2012. URL `http://news.cnet.com/8301-1023_3-57429192-93/android-reclaims-61-percent-of-all-u.s-smartphone-sales/`.

[5] Casper S Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67-77.

[6] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *AST '11*, pages 77-83.

[7] L. Gomez, I. Neamtiu, T.Azim, and T. Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *ICSE '13*.

[8] Jessica Guyn. Facebook users give iPhone app thumbs down. In *Los Angeles Times*. July, 2011. URL `http://latimesblogs.latimes.com/technology/2011/07/facebook-users-give-iphone-app-thumbs-down.html`, July 21.

[9] Domenico Amalfitano, Anna Rita Fasolino and Salvatore De. Using GUI Ripping for Automated Testing of Android Applications. In *ASE'2012*, pages 258-261.

[10] Vaibhav Rastogi, Yan Chen and William Enck. AppsPlayground: automatic security analysis of smartphone applications. In *CODASPY'2013*, pages 209-220.

[11] Wei Yang, Mukul Prasad and Tao Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *FASE'13*, pages 250-265.

[12] Xun Yuan and Atif M. Memon. Generating Event Sequence-Based Test Cases Using GUI Run-Time State Feedback. In *IEEE Transactions on Software Engineering*, 2010, pages 81-95.

[13] Xun Yuan and Atif M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE '07*, pages 396-405.

[14] Atif M. Memon. An event-flow model of GUI-based applications for testing. In *Software Testing, Verification and Reliability*, 2007, pages 137-157.

[15] Android Developers. UI/Application Exerciser Monkey. August, 2012. URL `http://developer.android.com/tools/help/monkey.html`.

[16] Atif Memon. GUITAR. August, 2012. URL `guitar.sourceforge.net`.

[17] Atif Memon. Android GUITAR. August, 2012. URL `http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR`.

[18] Google Code. Robotium. August, 2012. URL `http://code.google.com/p/robotium/`.

[19] SONY. APK Analyzer. January, 2013. URL `http://developer.sonymobile.com/knowledge-base/tool-guides/analyse-your-apks-with-apkanalyser/`.

[20] J. Jeon and J. S. Foster. Troyd: Integration Testing for Android. Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park, August 2012.

[21] Google Code. Androguard. January, 2013. URL `http://code.google.com/p/androguard/`.

[22] Google Code. Droidbox. January, 2013. URL `http://code.google.com/p/droidbox/`.

[23] Google Code. Android Assault. January, 2013. URL `http://code.google.com/p/android-assault/`.

[24] Android Developers. MonkeyRunner. August, 2012. URL `http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html`.

[25] Various Authors. SCanDroid. January, 2013. URL `https://github.com/scandroid/scandroid`.

[26] Bitbar. Automated Testing Tool for Android - Testdroid. January, 2013. URL `http://testdroid.com/`.

[27] Ranonex. Android Test Automation - Automate your App Testing. January, 2013. URL `http://www.ranorex.com/mobile-automation-testing/android-test-automation.html`.

[28] W. River. Wind River Framework for Automated Software Testing. January, 2013. URL `http://www.windriver.com/announces/fast/`.

[29] TestPlant. eggPlant for mobile testing.. January, 2013. URL `http://www.testplant.com/products/eggplant/mobile/`.

[30] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu and Sai Charan Koduru. An Empirical Analysis of the Bug-fixing Process in Open Source Android Apps. In *CSMR'13*.

[31] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010, pages 393-407.

[32] Android Developers. Android Emulator Limitations. March, 2013. URL `http://developer.android.com/tools/`

devices/emulator.html#limitations.

[33] Android Developers. Android Intents. March, 2013. URL http://developer.android.com/reference/android/content/Intent.html.

[34] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE '12*, pages 1-11.

[35] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer. Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage. In *MobileHCI '11*, pages 47-56.

[36] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. In *Software Engineering, IEEE Transactions on*, 2009, pages 684-702.

[37] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*, May 9, pages 24-27.

[38] S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating Android applications' CPU energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, 2012, pages 1-7.

[39] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys '11*, pages 335-348.

[40] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: multi-layer profiling of android applications. In *Mobicom '12*, pages 137-148.

[41] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *ICST '11*, pages 377-386.

Appendix

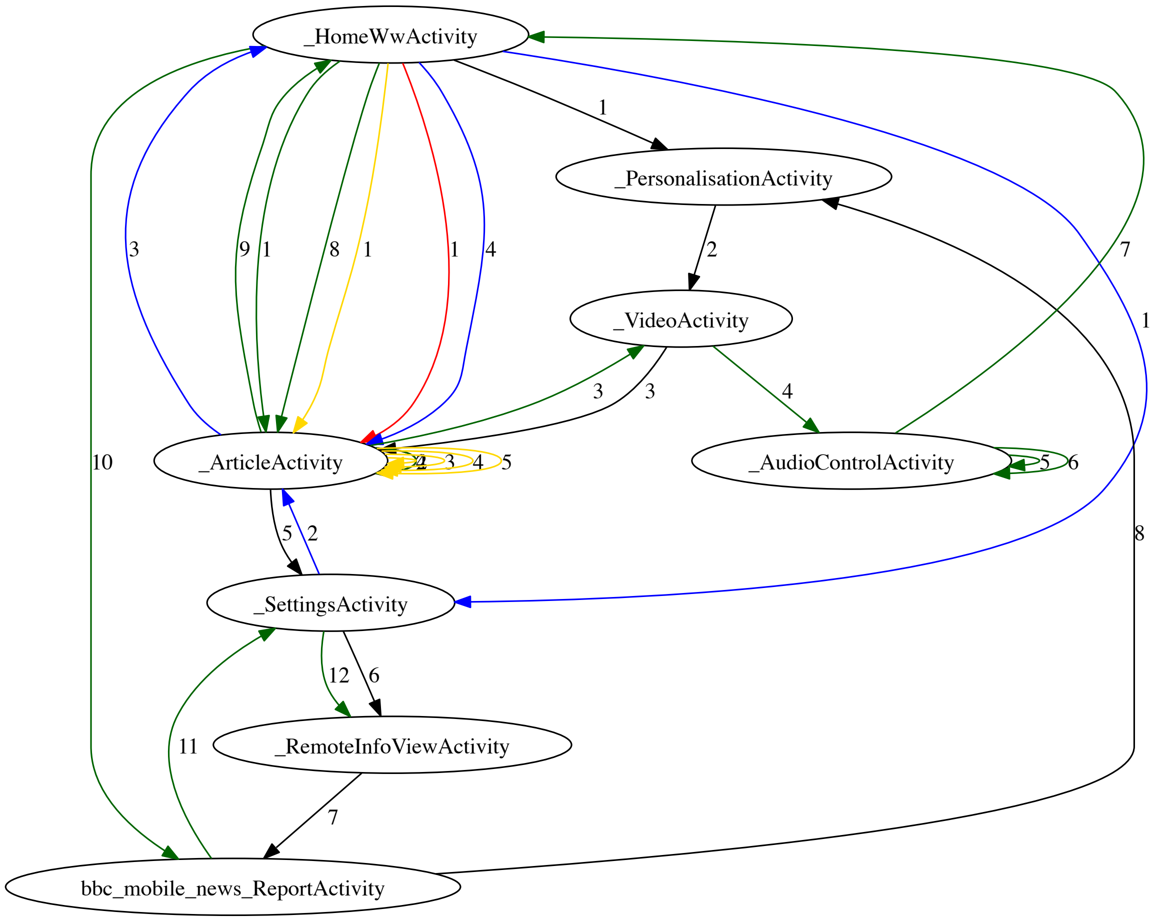

Figure 7. Dynamic Activity Transition Graph for the BBC News app, constructed based on runs from 5 different users: colors represent users and labels on the edges represent the sequence in which the edges are explored.

App	**Activities**		**Activity coverage (%)**						
	total #	excluding ads	User 1	User 2	User 3	User 4	User 5	User 6	User 7
Amazon Mobile	39	36	18	13	15.4	10.26	13	25.64	13
Angry Birds	8	1	100	100	100	100	100	100	100
Angry Birds Space Premium	1	1	100	100	100	100	100	100	100
BBC News	10	10	70	20	30	-	-	70	20
Advanced Task Killer	7	6	28.6	43	43	28.6	28.6	28.6	57
Advanced Task Killer Pro	6	6	50	33.33	50	16.66	16.66	16.66	33.33
CNN	42	39	19.05	9.5	14.29	12	12	19	14.29
Craigslist Mobile	17	15	23.5	35.3	41.2	23.5	29.4	29.4	35.3
Dictionary.com	22	18	41	41	59	32	41	59	41
Dictionary.com Ad-free	15	15	33.33	60	53.33	20	20	73.33	33.33
Dolphin Browser	56	56	12.5	8.9	1.78	1.78	1.78	1.78	1.78
ESPN ScoreCenter	5	5	60	40	20	20	20	20	20
Facebook	107	107	5.60	2.8	4.67	4.67	6.54	3.73	3.73
Tiny Flashlight + LED	6	4	50	50	50	50	50	50	66.67
Movies by Flixster	68	67	8.8	14.7	5.9	13.2	8.8	20.6	7.3
Gas Buddy	38	33	29	29	23.6	21	29	26	15.8
IMDb Movies & TV	39	37	25.64	15.4	15.4	-	-	20.5	12.8
Instant Heart Rate	17	14	23.5	29.4	29.4	23.5	23.5	23.5	23.5
Instant Heart Rate - Pro	17	16	11.8	17.65	11.8	11.8	11.8	11.8	11.8
Pandora internet radio	32	30	9.4	9.4	12.5	12.5	12.5	9.4	12.5
PicSay - Photo Editor	10	10	10	10	10	10	10	10	10
PicSay Pro - Photo Editor	10	10	10	30	10	10	10	10	10
Shazam	38	37	5.3	15.8	5.3	5.3	5.3	5.3	8
Shazam Encore	38	37	15.8	21	21	8	8	10.5	10.5
WeatherBug	29	24	27.6	24.13	20.7	20.7	20.7	20.7	27.6
WeatherBug Elite	28	28	10.71	14.3	14.28	7.14	7.14	3.57	3.57
YouTube	18	18	11.11	11.11	11.11	11.11	11.11	11.11	27.77
ZEDGE	34	34	35.29	29.41	32.35	29.41	20.58	11.76	23.52

Table 5. Activity count and coverage. User 1 has explicitly tried to achieve high coverage, while 2–7 are "regular" users.

App	Method count	External packages	App specific method count	Method coverage (%)						
				User 1	User 2	User 3	User 4	User 5	User 6	User 7
Amazon Mobile	13151	5	7154	4.21	1.36	3.93	1.64	4.93	3.99	2.92
Angry Birds	12245	4	6176	10.81	10.27	10.37	10.98	10.94	10.43	10.17
Angry Birds Space Premium	12953	4	7402	0.68	0.33	0.37	0.63	0.42	0.31	0.19
BBC News	4918	1	427	11.46	6.52	4.92	9.37	10.30	11.24	11.00
Advanced Task Killer	525	0	257	19.26	17.51	16.73	14.01	16.34	18.29	16.54
Advanced Task Killer Pro	257	4	3836	3.96	3.18	2.77	-	-	3.47	7.69
CNN	13029	11	7725	4.86	4.72	4.12	4.44	1.89	4.44	4.44
Craigslist Mobile	2765	4	2095	6.88	5.78	8.78	3.10	9.27	10.07	10.69
Dictionary.com	4664	6	2784	0.97	0.97	5.96	10.86	12.31	4.64	11.02
Dictionary.com Ad-free	2199	4	1272	18.64	17.55	15.33	15.65	17.37	18.08	15.80
Dolphin Browser	23701	6	13800	13.26	9.98	10.06	7.54	3.95	4.17	11.15
ESPN ScoreCenter	5511	5	4398	0.55	0.45	0.23	0.28	0.28	1.04	1.20
Facebook	34883	12	21896	1.67	1.61	1.64	1.48	1.59	1.56	1.53
Tiny Flashlight + LED	2121	3	1578	15.59	15.85	4.81	13.68	0.70	15.85	15.05
Movies by Flixster	12476	8	7490	3.53	3.30	4.60	4.66	2.86	3.41	4.68
Gas Buddy	7841	4	5792	7.38	5.51	3.82	7.84	5.52	3.53	5.31
IMDb Movies & TV	19781	9	8463	4.60	1.78	0.98	-	-	0.89	0.47
Instant Heart Rate	3044	5	2002	2.69	8.44	1.35	3.30	2.30	3.60	4.90
Instant Heart Rate - Pro	3044	5	1927	6.49	7.79	6.43	2.07	1.14	6.54	7.84
Pandora internet radio	13704	7	7620	2.88	2.01	1.44	2.07	3.24	2.75	2.18
PicSay - Photo Editor	1580	0	1580	2.97	3.04	2.66	2.59	4.37	1.39	2.97
Shazam	22071	13	9884	9.40	7.61	5.23	8.46	7.93	8.89	6.25
Shazam Encore	22071	9	9914	6.92	6.52	6.72	6.66	6.99	7.03	9.24
WeatherBug	9581	10	7948	3.70	8.02	3.11	3.82	4.52	3.93	5.75
WeatherBug Elite	9688	8	8194	5.06	4.24	6.36	3.41	6.12	5.89	3.83
YouTube	19902	10	11125	4.85	5.01	2.04	3.08	3.86	2.83	5.12
ZEDGE	8308	11	6287	6.96	3.00	4.82	6.44	7.32	8.44	5.75

Table 6. Method count and coverage.