UNIVERSITY OF CALIFORNIA
RIVERSIDE

Fault Discovery, Localization, and Recovery in Smartphone Apps

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Md Tanzirul Azim

August 2016

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Iulian Neamtiu
Dr. Zhijia Zhao
Dr. Zhiyun Qian

The Dissertation of Md Tanzirul Azim is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to show my greatest respect and thankfulness to my Ph.D. adviser Dr. Iulian Neamtiu. Without his guidance and inspiration, this journey would not have been possible. From the very beginning, he was tremendously supportive and helped me undertaking my research by his constructive direction and wisdom. His mentorship was not limited to the academic projects; rather he was proven as a true friend with his valuable suggestions in improving myself as a researcher as well. I graciously thank my committee members Dr. Rajiv Gupta, Dr. Zhiyun Qian, and Dr. Zhijia Zhao for their tremendous support. Dr. Gupta gave valuable feedback in various parts of my work, and I am ever grateful to him.

I would like to take this opportunity to thank my most precious treasures in this life, my parents Delwarul Azim and Kawsar A. Sultana. Not only they owe me my very existence, but also their painstaking labor, hardship, and sacrifices made the actual person in me.

I am thankful to my super awesome lab mates and colleagues: Amlan Kusum, Yongjian Hu, Vineet Singh, Bo Zhou, Yan Wang, Steve Su, Lorenzo Gomez, Arash Alavi for their backing in my successes and failures. Gavin Huang, Shashank Reddy Kothapalli, and Xuetao Wei contributed to some of my projects with valuable insight. I also show my gratitude to my mentor at Microsoft Research Dr. Oriana Riva. I express my regards to my co-authors Dr. Suman Nath, Dr. Todd Millstein, Dr. Zhiyung Shan. University staff members Ms. Amy Ricks, Ms. Amanda Wong, and Ms. Vanda Yamaguchi also deserve my respect for their greatest support and help.

Finally, I consider myself lucky to have so many friends and well-wishers around me with whom I shared my moments of happiness and frustrations.

To my dearest parents.

ABSTRACT OF THE DISSERTATION

Fault Discovery, Localization, and Recovery in Smartphone Apps

by

Md Tanzirul Azim

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2016
Dr. Rajiv Gupta, Chairperson

Applications for smartphones or other mobile devices ("apps") are used by billions of subscribers worldwide. Apps and their users are at risk, however: due to their unique and unprecedented properties, apps have a high potential for faults and errors, and due to their popularity, apps have become a lucrative target for attackers' malicious activities.

This dissertation introduces static and dynamic analysis techniques for fault discovery, localization, and recovery in apps. The existing approaches for fault discovery and localization either are not mature enough (i.e., lack of sound static and dynamic techniques and low coverage with high overhead) or have severe limitations (e.g., work only on emulators).

We facilitate the discovery of faults through Automatic Android App Explorer (A^3E). As the name suggests, A^3E generates test cases on-the-fly and exercises them. It achieves that through two distinctive strategies: an exploration in a depth-first manner and targeted exploration from a particular activity (page) inside the app. After fault discovery, fault localization is necessary to contain the error. To permit fault localization we have developed three techniques: app state recreation through user defined deep links, automatic GUI-input generation through AndroidArrow, and the program slicing framework AndroidSlicer. uLink, our next solution, is a demand-driven record-and-replay library that creates replayable links on-the-fly; these links can be directly re-executed to re-construct a particular application state. AndroidArrow is a toolset to generate UI element event sequences which can be injected into an app to trigger a target method or program point. Our next localization strategy is AndroidSlicer; a slicing framework for Android apps. While program slicing is not new, no substantial slicing work has been done so far for smartphone platforms. As smartphone apps have new, unprecedented characteristics, slicing techniques need to evolve accordingly. Apart from fault localization, slicing has many other applications, such as improving dynamic taint analysis, finding relevant inputs, and undo computing. Finally, we have developed an app recovery technique that uses automated patching while the app runs to seal off faulty code with specific recovery routines, so the app can recover from faults on-the-fly.

In summary, this dissertation explores and employs static and dynamic analysis and techniques, algorithms and, in some cases, improves existing approaches to creating high coverage test cases (fault discovery), identifying program faults (fault localization), and finally, sealing off erroneous code blocks (fault recovery) to render a smooth and uninterrupted app experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices running Android or iOS have flourished and have continued to expand their user base: in 2015 there were around 2.6 billion smartphone subscriptions worldwide, and by 2020 this number is expected to grow to 6.1 billion [14]. However, the very features that made these devices successful (e.g., rich sensor inputs, continuous connectivity, ample computational or storage capabilities), have made the devices a favorite target for attackers. Moreover, due to the apps' short update cycle, developers often rely on crowdsourcing for error discovery. However, crowdsourcing is not necessarily useful, as shown in Section 2.3.2. We have constructed automated exploration techniques that can generate high-coverage test cases by employing two key tactics: dept-first exploration and targeted exploration (Chapter 2).

While automatic exploration aids us in achieving higher coverage and discovery of errors and faults, additional techniques are required to locate the faults. For instance, automated exploration may reveal app errors on the user interface, while a log of app events can point to erroneous parts of a program; but we need to develop specific test cases to contain the error. This requires a combination of static and dynamic analyses to inspect program elements such as the data flow facts and input sequences that contribute to such flows. We created strategies to efficiently reproduce and constrain the fault inside a program. Reproducing faulty test cases required adequate record-and-replay capability. We introduced a novel mobile deep linking framework [9] that can recreate the app state by tainting the execution order. Next, to generate appropriate sets of inputs we produced GUI-objects-events mapping to trigger an erroneous target method. Finally, we provide dynamic slicing for the Android platform to further identify sets of inputs associated with particular runs of the program by exploiting the control and data dependences inside the app.

Offline fault discovery and localization are insufficient. However, some apps cannot be patched promptly, or users might be without connectivity (e.g., a soldier in a remote, hostile location). In such cases on-the-fly fault recovery is necessary. Therefore, we developed means to “seal off” faulty code blocks in case of errors to ensure that the application was able to operate without interruption.

Static and dynamic analyses to precisely uncover faults and perform recovery without crashing the app (or ruining the user experience) form the foundation of our work. In a broader sense, this thesis revolves around the following three particular goals: fault discovery using systematic exploration, fault localization, and fault recovery. Figure 1.1 shows the overview of the framework.

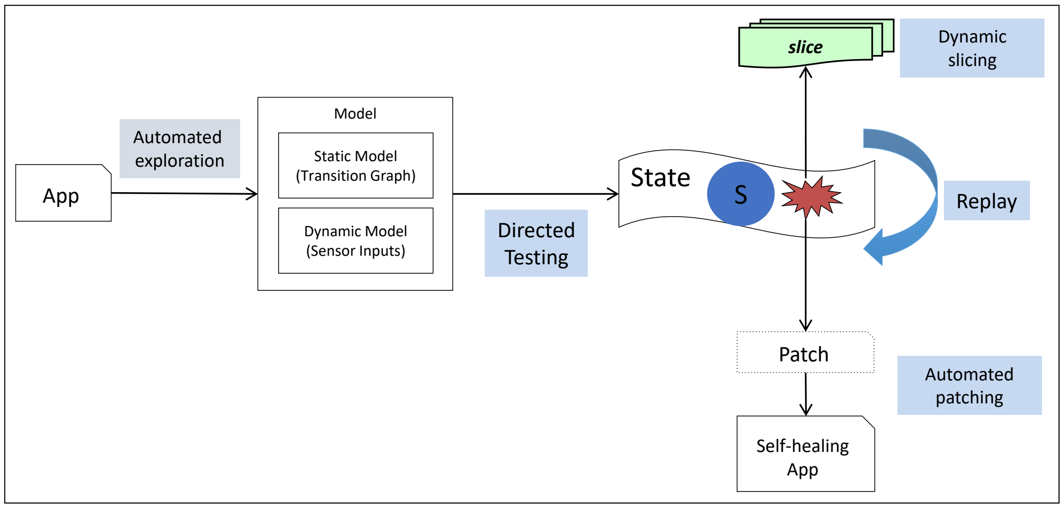


Figure 1.1: Framework overview.

1.0.1 Android architecture

Android platform architecture is designed as a stack of components. The stack layout comprises of applications, an operating system, run-time environment, middleware, services, and libraries layer. This design is outlined in the figure 1.2. High integration is maintained between the individual layers tuned to deliver an optimal environment for application development as well as execution.

Android callbacks Unlike traditional programs smartphone apps are mostly designed on event driven model. The flow of the program is regulated by events. An event can be user actions (touches to the screen, gestures), sensor data (microphone, accelerometer, gyroscope, ambient light sensor, pedometer), inter-process or inter-thread messages. Android SDK provides two major kinds of callbacks. Application life cycle callbacks, and event callbacks. Lifecycle callbacks are those that are triggered during various phases of application's presence on the memory before it gets killed either by the action of the user

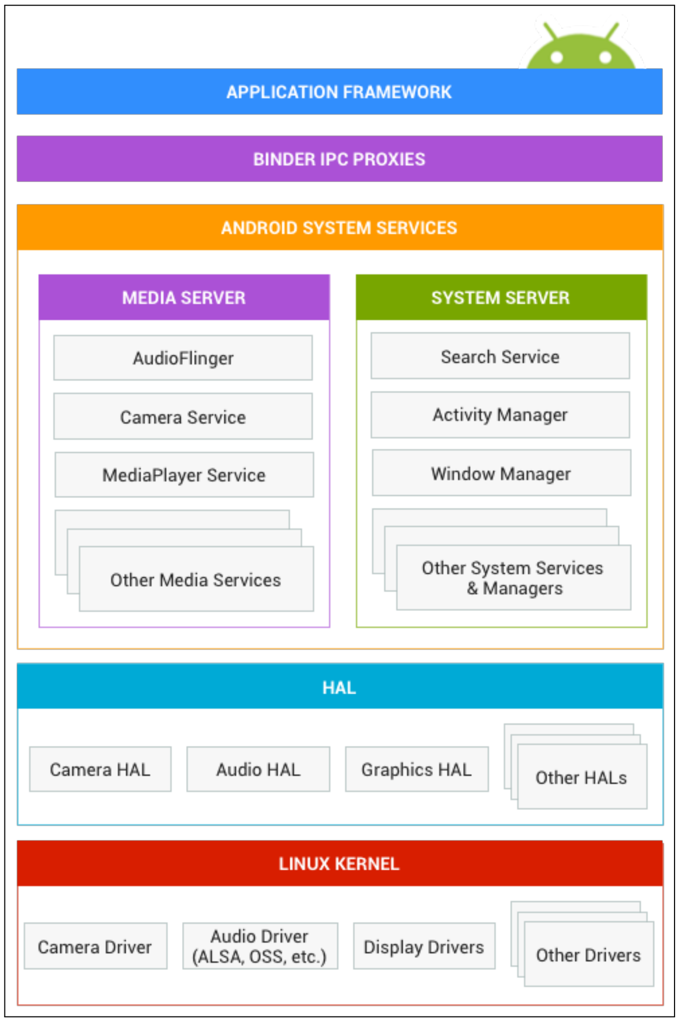


Figure 1.2: Android architecture.

or by the system. Event callbacks are asynchronous feedback and acknowledgment calls when some events are finished processing on the event queue. For instance, when a user touches the screen, the system adds it to the event queue, and upon completion triggers *onTouch* as a response. So, instead of traditional program control flow, Android control flow is distinctively manipulated by the callbacks. In the underlying architecture, there exists a main loop which constantly listens to events, processes them, and finally invokes a callback.

Android program entrypoint Android programs do not contain any particular "main" method. An app mainly presents an entry screen viewable to the user. The rendering of the screen are the results of a group of life cycle events triggered in a particular order. Moreover, depending on the design of the app, an app can have multiple entry screens. Several approaches exist to create pseudo entry points for data flow analysis [35, 29].

1.1 Fault Discovery

We aid fault discovery by developing exploration strategies that generate high-coverage test cases. In fact, 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. The practical challenges of a self-guided exploration mechanism can be put into two different categories: OS specific programming problems, and device or hardware related constraints.

To address these problems, we present $\mathsf{A}^3\mathsf{E}$, 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, data-flow 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. In addition, we developed a strategy named "Depth-first Exploration" that mimics user actions for exploring activities and their constituents in a slower, but a 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.

1.2 Fault Localization

The next step in the process is fault localization, which is essential to recreate the faulty state and to develop recovery tactics. The purpose of fault localization techniques is to compartmentalize error sources so that further investigations can be carried out efficiently. We have introduced three localization mechanisms, at different levels of granularity. First, we introduced a novel approach to generate input sequences responsible for a particular failure. Then, we developed a robust slicing framework tailored for smartphone apps to investigate errors from a much lower level. Finally, we have an app state recreation technique that can reproduce the fault condition keeping the app execution at a minimum level by triggering only those events necessary for reinstating the app state.

GUI input sequence generation for a reaching a target (program point) in the execution is undecidable in general, and challenging in practice. This requires comprehensive static analysis to discover data flow paths inside the app to the point of interest. Moreover, apps have complex GUI structures: for instance, in Android, GUI elements can be defined in layout files, or created programmatically. For finding a valid order of execution through distinct UI objects, we need to rebuild the explicit and implicit co-relation between the resources and UI elements and discover the underlying event handlers. To pursue this, we introduced the tool-set AndroidArrow. AndroidArrow proceeds in two steps: first, it uncovers the transitioning path to a specific target inside the program by conducting flow analysis, and second, it reproduces the inter-relation between external GUI resources and their associated event handlers by static object referencing. We have successfully analyzed 34 apps using AndroidArrow.

When evaluating errors and faults in Android apps, one technique that has proved to be valuable is dynamic analysis. Dynamic analysis critically relies on high-quality inputs that can ensure good coverage, i.e., drive program execution through a significant set of representative application states [50, 42]. Finding "high-quality inputs" for dynamic analysis on Android is a major challenge due to the sensor- and event-driven nature of apps: Android apps, whether benign or malicious, have rich, concurrent, high throughput inputs, e.g., from sensor and networks. This compounds the precision problem, as we now have to analyze not only data/control dependences, but also the sizable inputs. Finally, examining phone state to separate legitimate from malicious/buggy actions is a challenge, given the large extent of the state and the potential for system-wide corruption. As applications can be vulnerable from programming errors, or malicious code injections/tweaks, understanding the program behavior is essential to further localizing the fault or threat.

To address the aforementioned issues, we propose a dynamic slicing infrastructure for Android. The slice contains all instructions responsible for affecting a particular register value at a program point during a specific run of the app instead of aggregating all other statements that may change it in separate runs. This approach helps us better understand the role of inputs in application control flow, and aids us to identify fault-inducing input sets. Chapter 3 discusses our Android slicing implementation.

Our next contribution to fault localization is uLink, a user defined deep linking framework. Our primary motivation was to provide effective means to recreate a faulty state. The idea behind this strategy is rooted in the phenomenon of web deep links, which are instrumental to many fundamental user experiences such as navigating to one web page from another, bookmarking a page, or sharing it with others. Such experiences are not possible with individual pages inside mobile apps since historically mobile apps did not have links equivalent to web deep links. Furthermore, mobile deep links, introduced in recent years, still lack many important properties of web deep links. Unlike web links, mobile deep links need significant developer effort, cover a small number of predefined pages, and are defined statically to navigate to a page for a given link, but not to dynamically generate a link for a given page. We developed uLink, a novel deep linking mechanism for addressing these problems. uLink is implemented as an application library, which transparently tracks data- and UI-event-dependencies of app pages, and encodes the information in links to the pages; when a link is invoked, the information is utilized to recreate the target page quickly and accurately. uLink also employs static and dynamic analysis techniques that can provide feedback to users about whether a link may break in the future due to, e.g., modifications of external resources such as a file the link depends on. We have implemented uLink on Android. Our evaluation of 34 (of the 1,000 most downloaded) Android apps shows that compared to existing mobile deep links, uLink requires minimal developer effort, achieves significantly higher coverage, and can provide accurate user feedback on a broken link.

1.3 Fault Recovery

Having discovered, analyzed, and localized the fault in an Android application using the aforementioned techniques, we must now address fault recovery, a critical component of the user experience. Frequent app bugs and low tolerance for loss of functionality have created an impetus for self-healing smartphone software. We take a step towards this via on-the-fly error detection and automated patching. Specifically, we add failure detection and recovery to Android by detecting crashes and "sealing off" the crashing part of the app to avoid future crashes. In the detection stage, our system dynamically analyzes app execution to detect certain exceptional situations. In the recovery stage, we use bytecode rewriting to alter app behavior to avoid such cases in the future. When using our implementation, apps can resume operation (albeit with limited functionality) instead of repeatedly crashing. Our approach does not require access to app source code or any system (e.g., kernel-level) modification. Experiments on several real-world, modern Android apps and bugs show that our approach manages to recover the apps from crashes in an effective, timely manner and without introducing overhead.

1.4 Thesis Organization

The dissertation is organized as follows. In Chapter 2 we focus on our implementation of an automated exploration strategy. Chapter 3 discusses our slicing framework. We present our static analysis-based targeted GUI-event generation techniques in Chapter 4. User-defined deep linking is described in Chapter 5. In Chapter 6 we present our recovery techniques. Chapter 9 provides the conclusion and future research directions.

Chapter 2

Dynamic App Exploration

Effective fault discovery strategies require high coverage test cases. Due to their shorter update cycle smartphone, app developers highly rely on manual testing or crowd-sourcing based efforts to discover or reveal errors inside the apps. In this chapter, we will show why these techniques cannot always be trusted as a measure of yielding high-coverage test cases. We will also introduce $\mathsf{A}^3\mathsf{E}$, our novel approach, and tool to conduct automated exploration in Android apps. We will also show how $\mathsf{A}^3\mathsf{E}$ significantly achieves a higher percentage of code coverage than manual means.

2.1 Motivation

Users are increasingly relying on smartphones for computational tasks [56, 55], hence concerns such as app correctness, performance, and security become increasingly pressing [64, 67, 32, 49, 65]. 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 [65, 46] to profiling [121] and security [49]. 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 [50, 42].

To facilitate test case construction and exploration for smartphone apps, several approaches have emerged. The Monkey tool [24] can send random event streams to an app, but this limits exploration effectiveness. Frameworks such as Monkeyrunner [23], Robotium [59] and Troyd [73] support scripting and sending events, but scripting takes manual effort. Prior approaches for automated GUI exploration [20, 128, 93, 21, 99, 130] 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 2.6, since we connect to apps running naturally on the 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 [25] 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

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

their existing test suites with automatically-generated test cases aimed at systematic exploration. Since A^{3}E does not require access to source code, users other than the developers can execute substantial parts of the app automatically. A^{3}E 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^{3}E as a basis for dynamic analyses [42] (e.g., monitoring, profiling, information flow tracking), testing, debugging, etc.

In this dissertation, 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 2.2 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 5.1.

In Section 2.4 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 2.5 we provide an overview of $\mathsf{A}^3\mathsf{E}$'s implementation: hardware platform, tools and measurement procedures. In Section 2.6 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.

2.2 Android Activities, Graphs and Metrics

We have chosen Android as the target platform for our $\mathsf{A}^3\mathsf{E}$ implementation as it is currently the leading mobile platform in the US [41] and worldwide [70]. 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.

2.2.1 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 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.

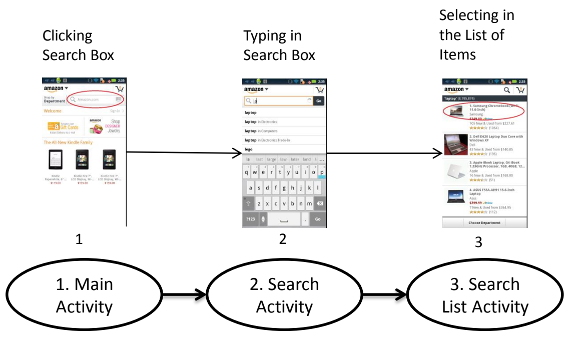


Figure 2.1: An example activity transition scenario from the popular Android app, Amazon Mobile.

Naturally, these activity transitions form a graph. In Figure 2.1 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.

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

2.2.2 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

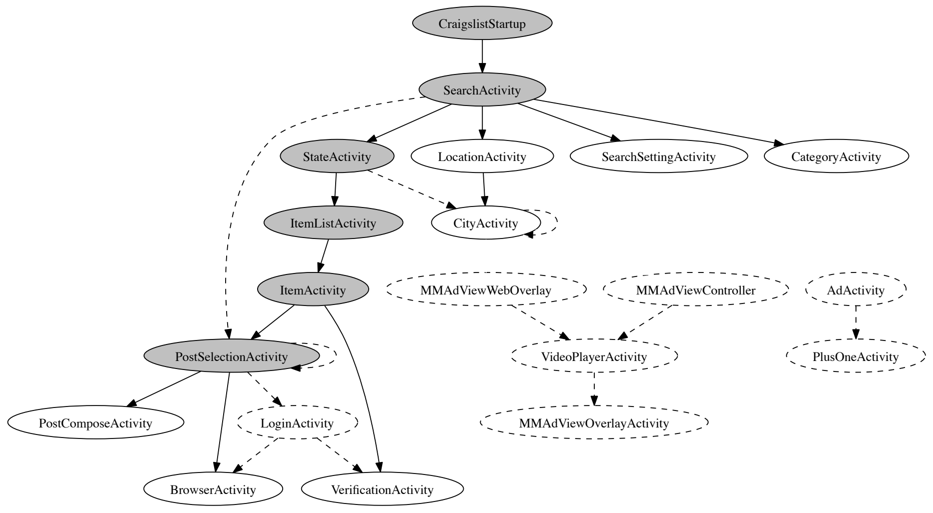


Figure 2.2: Static Activity Transition Graph extracted automatically by our approach from the Craigslist Mobile app. Grey nodes and associated edges have been explored by users. Solid-contour nodes (grey or white) and solid-line edges were traversed dynamically by our exploration. Dashed-contour nodes and dashed-line edges remained unexplored. Activity names are simplified for legibility.

described in Section 2.4.1.

Figure 2.2 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.

2.2.3 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

App	Type	Category	Size		# Down-loads
			Kinst.	KBytes	
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 2.1: Overview of our examined apps.

intuitive way and is a subgraph of the SATG. Figure 2.2 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 reproducing bugs.

2.2.4 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 2.5.2.

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 run-time profiling information and the MT via static analysis, as described in Section 2.5.2.

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 6.1. 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) [36]. 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.

2.3.1 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 2.2.3), we construct the union, i.e., a graph

[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.

$G = G_1 \cup G_2$ 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 2.2, 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 2.5.

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 2.2 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:

[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 2.2: The results of our user study.

- **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.

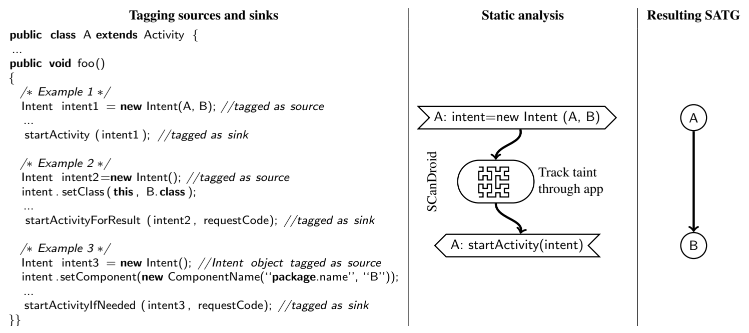
- **Ads.** Many free apps contain ad-related activities. For example, in Angry Birds, all the activities but one (play game) were ad-related. Therefore, in general, free apps contain more activities than their paid counterparts—see Angry Birds, Advanced Task Killer, Dictionary.com. When users do not click on ads, the ad-related activities are not explored.

2.3.2 Method Coverage

Since activity coverage was on average about 30%, we would expect method coverage to be low as well, as the methods associated with unexplored activities will not be invoked. The last group of columns in Table 2.2 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 2.6. In Section 2.6.1 we provide a detailed account of why method coverage is low.

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



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

To illustrate the main principles behind these strategies, let us get back to the flow of the Amazon Mobile app shown in Figure 2.1. 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 2.4.1), then show how it drives Targeted Exploration (Section 2.4.2); next we present Depth-first Exploration (Section 2.4.3) and finally how our approach supports test case generation and debugging (Section 2.4.4).

```
//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 2.4: Intent passing through superclasses in NPR News.

2.4.1 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 [25], 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 2.3, 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 2.3 (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 2.3 (right)).

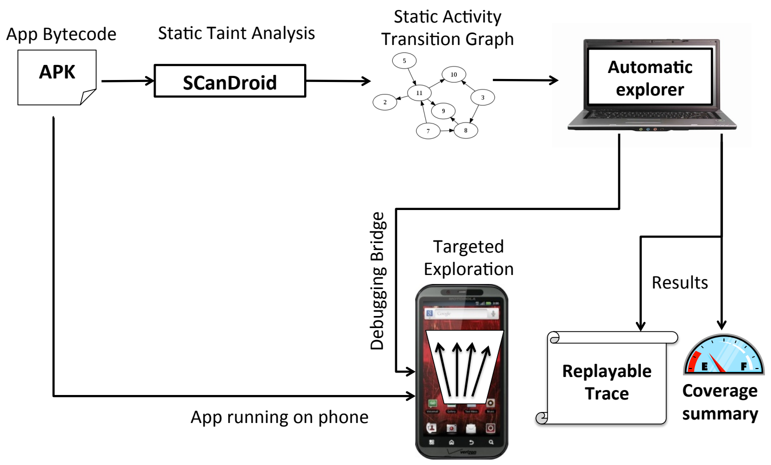


Figure 2.5: Overview of Targeted Exploration in A^{3}E.

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 2.4, 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.

2.4.2 Targeted Exploration

We now proceed to describing how we perform Targeted Exploration using the SATG as input. Figure 2.5 provides an overview. The automatic explorer, running on a desktop or laptop, orchestrates the exploration. The explorer first reads the SATG constructed by SCanDroid (a static dataflow analyzer that we customized to track intent taint-

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
```

ing, as described in Section 2.4.1) 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 [79]—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 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

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

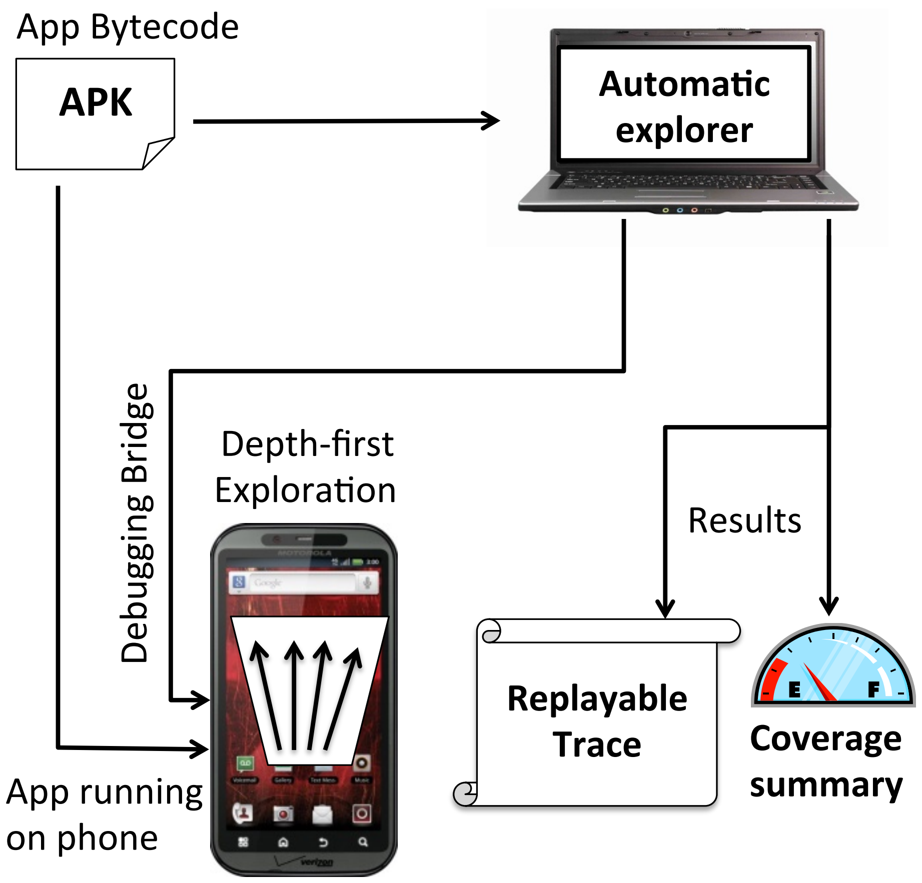


Figure 2.6: Overview of Depth-first Exploration in A^3E.

fast—we can switch to exportable activities without firing GUI events.

2.4.3 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.

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
```

Figure 2.5 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).

2.4.4 Replayable Test Cases and Debugging

During exploration, A^{3}E automatically records the event stream using RERAN, a low-overhead record-and-replay tool [79], 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.

2.5 Implementation

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

2.5.1 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 [79] 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 [110]. 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.

2.5.2 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 2.3.2).

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 [73] (which in turn is based on Robotium [59]). 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 2.3.1, our test users tended to skip features such as options, ads, settings, or sharing via social networks. To cover such activities and functionality, $\mathsf{A}^3\mathsf{E}$ employs several strategies. $\mathsf{A}^3\mathsf{E}$ 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 $\mathsf{A}^3\mathsf{E}$ 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 [79], synthesizing complex, multi-touch gestures is non-trivial.

In addition to the gesture library and log-on functionality, $\mathsf{A}^3\mathsf{E}$ 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, $\mathsf{A}^3\mathsf{E}$ uses the appropriate exploration algorithm depending on the kind of exploration we perform, as described next.

2.5.4 Targeted Exploration

Section 2.4.1 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.

2.5.5 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.

2.6 Evaluation

We now turn to presenting experimental results. From the 28 examined apps presented in Section 5.1, 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 2.6.2.

We first evaluate our automated exploration techniques on these apps in terms of effectiveness and efficiency (Section 2.6.1), then discuss app characteristics that make them more or less amenable to our techniques (Section 2.6.2).

2.6.1 Effectiveness and Efficiency

Activity coverage. We present the activity coverage results in Table 2.3. 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.

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 2.3: Evaluation results: activity and method coverage.

Method coverage. The method coverage results are shown in the last columns of Table 2.3. 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.

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

Exploration time. In Table 2.4 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 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 $\mathsf{A}^3\mathsf{E}$ have similar goals (automated exploration) they have several differences.

Targeted Exploration requires a preliminary static data-flow 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: 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 2.4 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.

2.6.2 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^3E. 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 2.5.3) 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^3E 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

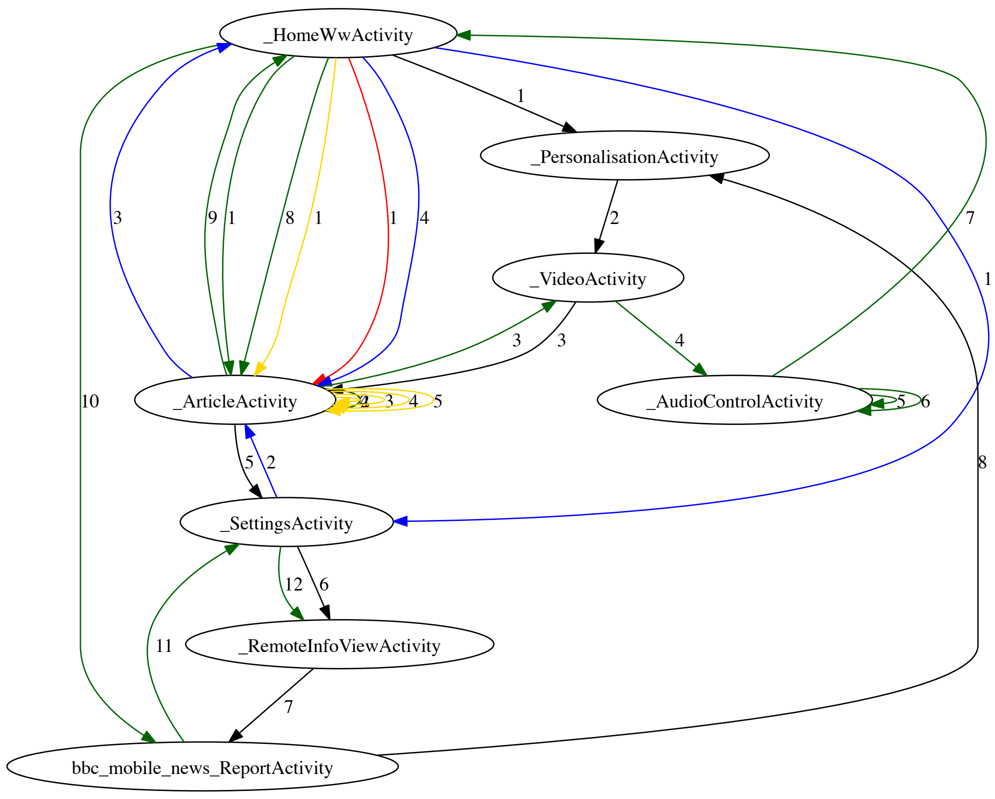


Figure 2.7: Dynamic Activity Transition Graph for the BBC News app, constructed based on runs from 5 different users: colors represent users and labels on the edges represent the sequence in which the edges are explored.

native GUI information. We could not explore the Facebook app for the same reason.

2.7 Conclusions

We have presented A^{3}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

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

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.

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 2.6: Method count and coverage.

Chapter 3

Android Application Slicing

In this chapter, we will introduce program slicing for Android platform. While the concept of program slicing is not new in the literature concerning program analysis, to the best of our knowledge there are no exclusive slicing methodologies have been implemented in event-driven smartphone platforms. Our approach to fault localization demands to identify error sources in a wide area of possible input scenarios. For example, apps can get user input from simple to complex gestures, or sensor feedbacks, or even from outside applications through inter-process messaging. To make things more complicated, the execution of an app is mostly determined by the non-deterministic nature of the callbacks. For this reason, fault localization approaches need to address these issues. Despite being an efficient tool for locating and identifying sets of failure-inducing inputs, traditional program slicing techniques will not work effectively in modern smartphone platforms. We will present our novel approach to program slicing in Android which is designed to deal with the above-mentioned smartphone specific challenges.

The Problem. The factors that have made the mobile platform popular and mobile OSes ubiquitous are also responsible for a host of problems. Constant connectivity means that a compromised phone can be turned into a "bot" and used to launch attacks 24/7. The data collected by physical sensors can be used for positive purposes, e.g., to track physical activity for well-being, but also nefarious ones — users' geographical position and physical activities can be subject to constant monitoring or revealed to attackers. The convenience introduced by rich processing and storage capabilities also make it more likely that phones and tablets store and process sensitive documents, e.g., tax returns or bank statements. Attacks can compromise the confidentiality, integrity, and availability of these sensitive documents. There is constant pressure on app developers to release new apps, or frequent app updates, to keep up with the competition which means apps can be released with little scrutiny. Similarly, app marketplaces, e.g., Google Play or Apple App Store are under pressure to accept new apps or app updates in a timely fashion to satisfy app users, which in turn means that limited time is being allocated to scrutinizing the apps that are about to be published.

Android attacks take many forms, and many studies have categorized these attacks. For example, Zhou and Jiang [135] have investigated the 1,260 apps in the Malware Genome project, with a focus on malware behavior and malware evolution. Felt et al. [51] have looked at malware, spyware, and grayware. Vidas and Christin [113] have categorized anti-detection techniques. Our prior work has quantified the risk associated with HTTP(S) communication [122]. We now summarize the most prevalent categories of Android attacks.

Exfiltrating user information. Malware can read private information such as device ID, phone number, list of contacts, and send this information to scammers or advertisers. Other malware steals user credentials or credit card numbers and sells this information so it can be used on the black market. "Personal spyware" apps collect personal information and then store it or exfiltrate it; such apps are up-front about their purpose, but the user might not be aware that the app is installed.

Financial attacks. In these attacks, malicious apps operate behind the scenes, e.g., make premium-rate calls, send premium-rate SMS messages, or sign up for premium-rate SMS services, and then filter the call or SMS logs to cover their tracks.

Botnets. Malware can turn a phone into a "bot" that takes commands from its peers or a Command-and-Control center; hence the phone becomes a base for launching attacks, sending spam, purchasing tickets for scalping, implementing pay-to-click schemes, etc.

Root-level exploits. Such exploits give the attacker root privileges, hence complete control of the phone.

Ransomware. Apps can employ various ways of blackmailing users — threatening to expose their information or locking the device — unless a ransom is paid.

Note how these attacks share certain traits:

1. Finding security issues reduces to answering information flow queries, e.g., whether information from a sensitive source (user credentials, location, etc.) flows to an untrusted sink; or, how commands flow from an untrusted remote source to a local execution engine on the device.

2. The relevant behavior is highly dynamic. For example, the destination for exfiltrating data (i.e., server addresses), or the premium phone/SMS numbers for financial attacks are learned at runtime [135]. Botnet models are dynamic: the Command-and-Control center's address, commands, and a list of peers are sent to the bots at runtime [107]. Our prior work has shown that mutations of the same virus might look very different to static analysis, but similarities are apparent when using dynamic slicing [52].
3. Large inputs have to be analyzed, to find that small input values that play a crucial role during execution, e.g., a botnet command in network input [107], or bitmap input data that triggers a `libSecMMCodec.so` vulnerability on Samsung devices [95].
4. Large extents of in-memory and persistent state have to be efficiently and precisely separated into tainted and untainted parts, e.g., to find how a root exploit or ransomware have altered system state.

Current approaches and their limitations. *Static information flow tracking* has been proposed to help find flows/leaks, e.g., using *taint* to track the flow from security-sensitive sources to untrusted sinks [40]. Static flow tracking [29] is sound and scales well, permitting analysis of a large number of apps, but is prone to false positives and cannot be used to reveal inherently runtime behavior/information, as mentioned above.

Dynamic analysis. in general is an attractive approach for tackling many Android concerns via profiling and monitoring, and has been used to study a wide range of properties, from energy usage [65, 46] to profiling [121]. In security contexts, dynamic techniques are well suited for exposing nefarious behavior and learning the operational model for attackers or botnets, because malware behavior is highly dynamic, and the data of interest/-operational model might not be known until well into the execution. Dynamic information flow tracking [49] has been used successfully in the past to find leaks — the TaintDroid dynamic taint tracker has exposed that the user's location, and phone information is routinely leaked to advertising and content servers. Similarly, constructing botnet operational models [107] or finding anti-detection techniques has relied on runtime monitoring and analysis [114, 113].

Problems with current dynamic approaches. Dynamic analysis critically relies on high-quality inputs that can ensure good coverage, i.e., drive program execution through a significant set of representative program states [50, 42]. Finding "high-quality inputs" for dynamic analysis on Android is a significant challenge due to the sensor-driven nature of Android apps. We and others [72, 16] have made inroads (e.g., our A^{3}E automatic app explorer achieves coverage that exceeds the coverage of several human users combined [31]) but achieving high coverage on real-world obfuscated apps running on real phones is still a challenge.

Even assuming high-quality inputs, dynamic taint trackers face several hurdles: (1) they tend to track data dependences, not control dependences, hence taint trackers are vulnerable to implicit flows; (2) they are imprecise, and even with a carefully crafted taint tracking rules, "taint explosion" is a problem — where large parts of program data appear to be tainted, which makes analysis difficult; (3) they point out a leak, but do not help the developer fix the leak. Android apps, benign or malicious, have rich, concurrent, high-throughput inputs, e.g., from sensor and networks. This compounds the precision problem, as we now have to analyze not only data/control dependences, but also the sizable inputs.

Finally, analyzing phone state after an intrusion or successful attack to separate legitimate from malicious actions is a challenge, given the large extent of the state and the potential for system-wide corruption.

Our solution. We propose developing a dynamic slicing infrastructure for Android that can track dependences between input, data, and control, in a precise and efficient way, and using this infrastructure in security applications. Our work will be focused mainly on an end-to-end infrastructure for dynamic slicing Android apps running on real phones and without requiring access to the app source code.

/

There have been a good number of works regarding program slicing. But they were mainly based on programs for desktop operating systems. While both the desktop and smartphone applications share much common structures, smartphone apps vary in different aspects. Hence there is a high need to shed some lights on those differences to develop efficient slicing algorithms for mobile apps.

3.1 Background

3.1.1 Dynamic Slicing

Dynamic analysis. Dynamic analyses are a class of program analyses where the program is executed and its behavior is monitored, to verify if a certain property continues to hold during the execution. Dynamic analysis is an effective "lens" for observing program behavior. As the program executes, instrumentation is used to collect a variety of data, from memory locations accessed, to values created and propagated, to energy used in each method. Then developers can use the collected data in various ways, e.g., perform runtime verification to ensure that the app fulfills functional requirements; or observe information flows (dynamic information flow tracking) to ensure sensitive information does not leak to unauthorized websites; or optimize energy-intensive methods to reduce their energy usage.

Program dependence graph. A program dependence graph (PDG) captures the data and control dependences relation inside a program. Each edge of the PDG represents a data or control dependence between the nodes. Depending on the purpose, a node can either represent an instruction or a basic block. A directed data dependence edge from node v_i to node v_j means any computation performed in v_i depends on the computed value at node v_j. A control dependence edge means that the execution decision of v_i is taken in v_j, that is, v_j contains an instruction which is a predicate statement. A static dependence graph consists of all possible data and control dependence relations. Figure 3.2 represents a static program dependence graph for the example program in 3.1 reproduced from [19]. A dynamic dependence graph is a subgraph that contains only the nodes executed during a particular run.

```
         begin
S1:          read(X);
S2:          if (X < 0)
             then
S3:              Y := $f_1$(X);
S4:              Z := $g_1$(X);
             else
S5:              if (X = 0)
                 then
S6:                  Y := $f_2$(X);
S7:                  Z := $g_2$(X);
                 else
S8:                  Y := $f_3$(X);
S9:                  Z := $g_3$(X);
                 end_if;
             end_if;
S10:         write(Y);
S11:         write(Z);
         end.
```

Figure 3.1: Example program.

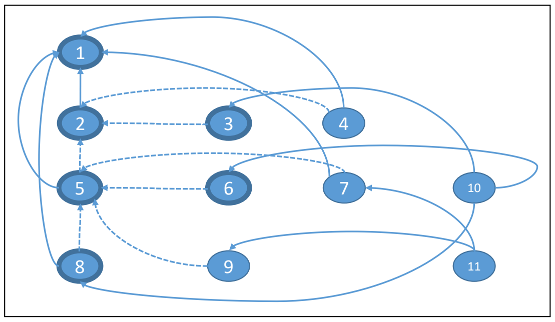


Figure 3.2: Static program dependence graph for figure 3.1.

Program slicing. Program slicing is a technique for isolating values, paths, or dependences of interest in the dynamically collected data, as the analysis demands verification, security or energy. Dynamic program slicing (a class of dynamic analysis) proposed by Korel and Laski [77], was first introduced to assist programmers in debugging. The dynamic slice of a value computed at a program point during an execution consists of all the executed statements that were directly or indirectly involved in the computation of the value; more precisely, a *dynamic slice* is the transitive closure of data and control dependences in the dynamic dependence graph. For multithreaded programs dynamic dependences between different threads (introduced via events, files, or shared memory) must also be considered.

3.2 Program Slicing on Android

Prior slicing work has targeted desktop and server platforms [77, 134, 104]. But traditional slicing techniques are not sufficient on smartphone platforms. That is why we were motivated to building a slicing infrastructure that will address several issues related to mobile platforms. We chose Android as target platform due to its popularity.

Slicing infrastructure for Android requires addressing several challenges, as outlined next.

Event-based programming model. Most Android apps are built around a GUI. The GUI control flow is however orchestrated by the Android Framework, rather than the app developers only define callbacks that are invoked when GUI elements are exercised. While this makes development easier, it complicates analysis, because reconstructing the control flow requires analyzing or modeling the control flow orchestration that takes places inside the Android Framework. Nevertheless, our recent positive experience with Gator [127], a tool that helps reconstruct app control flow by building a static model of an app's GUI, holds promise for constructing an app's control flow statically.

Sandboxing. Unlike desktop/server Linux, Android Linux uses application sandboxing (assigning unique user IDs to each app, so each app runs in its user space) for security reasons. We will have to determine the effect of isolation on tracking dependences in shared code, such as framework and system libraries. Addressing this might require "rooting" the phone or running the slicing process as root, which is acceptable in a development or research setting.

Inter-process communication (IPC). Android uses IPC (e.g., Intents) heavily to permit apps to access system resources in a controlled way. For example, the Facebook app can send an Intent to the Camera app asking it to take a picture. Apps can share data via structures named Bundles. Also, apps can have references to other services, called Binders, to initiate method calls from their side. These techniques are different from traditional IPC, hence we will need to implement a slicing algorithm to capture external influences due to, and data flow through, IPC. This challenge is related to sandboxing — we believe that we can track dependences in apps that are the source or destination of an Intent (besides the app under analysis) in a similar way to tracking dependences in system code.

Just-in-time and Ahead-of-time compilation. Dalvik VM, the VM used in Android before version 5.0, implements JIT compilation for efficiency reasons; hence the instruction stream that is executed will contain some differences from the app code, due to JITing. Similarly, for Android version 5.0 and above, where app bytecode is translated directly to native code using ahead-of-time compilation (AOT), ART performs optimizations hence static analysis assumptions might not hold. Our preliminary results (discussed next) indicate that the Android Pin infrastructure is promising for analyzing AOT code.

3.3 Our Approach: Android Slicer

In this section, we will represent our design of the slicing algorithm. We addressed several factors and needed to take various Android aspects into consideration. The key insight of our design is following:

1. The algorithm needs to detect complex gesture inputs by detecting appropriate framework callbacks.
2. It needs to provide a tracking mechanism for sensor inputs.
3. The design must maintain the order of the callbacks.
4. External communication elements such as inter-process communication (IPC) objects need to be taken into consideration,

Our framework conducts the analysis in three phases which we will describe in next three sections:

3.3.1 Offline Control Dependence

We statically analyze the app to produce control flow graph where each node is a basic block. This offline static analysis is later used to measure dynamic control dependence.

3.3.2 Block Summarization

We collect summaries of executed blocks during our static analysis. We calculate the definitions inside the blocks which are live at the end of the block. With this information, we can create the data dependency table during run-time application execution. Our block summarization records the following information:

Callbacks We enlist callback information inside a basic block. If a particular block is enclosed by a callback, we keep that information. We keep track of the use of the callback arguments. This information is required to determine if the source of error is due to any of the events associated with the callback. For example in *onClick(View v)* the argument v represents the view element that received the event.

Framework calls Framework calls are the Android SDK APIs invoked by the application. Currently, we do not analyze the framework. We track the API parameters and return values instead.

For example, consider the API calls *getCellLocation()* and *startActivity(Intent i)*. For the former, we summarize the return value from the API call and report it, and for the latter, we track down the Intent object *i*.

Live variable analysis We summarize the blocks by computing the reaching definitions at the block exit (OUTs).

3.3.3 Program Instrumentation

We took advantage of app instrumentation techniques to further improve the efficiency of our slicing algorithm. Instead of collecting the whole execution trace we only collect information of executed basic blocks. The blocks are cross referenced with our block summaries obtained above.

3.3.4 Dynamic Program Slicing

Dynamic control dependence. Effective online dynamic control dependence is important in achieving higher precision. While static definitions can be translated in the runtime, but such trivial measures can lead to inefficiency [125]. In our design, we followed the online detection algorithm described in the work of Xin et al. [125].

Dynamic slicing. We follow backward propagation technique to build run-time program dependence graph from app execution trace generated in phase 3.3.3. Instead of taking the whole execution trace we take another program slicing approach called LP processing [133]. We build dependence graph considering 200 instructions each time and summarize the results. This process involves dynamic control dependence calculation. We maintain two separate tables one for data dependency information for a variable x and the control dependency for the associated instruction to the predicate instruction that decides the execution of the instruction.

Algorithm 3 Dynamic program slicing

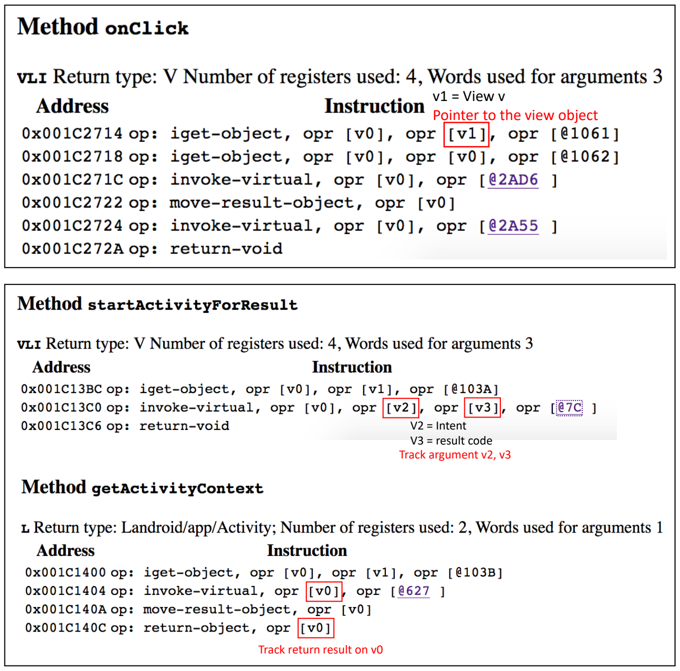
```
Input: slicing criterion $S_c = (a_v, v)$
 1: procedure SLICE($S_c$)
 2:     for all nodes $n$ that are in $D_v$ and $C_v$ do
 3:         calculate $v_d = D_v \cup C_v$
 4:         if $n$ is an occurrence of $v$ then
 5:             if $n_d \equiv v_d$ then
 6:                 point $v$ to $n$
 7:             else if
 8:                  then create a new node for $v$
 9:             end if
10:         end if
11:         if if $v_d \subset n_d$ then
12:             merge $n$ with $v$
13:         end if
14:         if if $n \epsilon D_v$ then
15:             SLICE($(a_n, n)$)
16:         end if
17:     end for
18: end procedure
```

To achieve the dynamic slicing for a particular slicing criterion with the format $(address, register)$ using our dependence table from phase 3.3.4, we first measure its immediate definition and calculate that node's reachable statements.

Algorithm 3 illustrates the approach. For a particular variable v, D_v and C_v denotes the data dependence and control dependence nodes respectively; a_v denotes the address of the instruction containing the variable.

We analyze the dynamic slices to address Android specific challenges,

Android callbacks If the slices include traces of the callbacks parameters as "uses" we analyze the parameters of the callbacks. Currently, we do not analyze the framework. The callbacks also represent the different input gestures and sensor event handlers as they are some callbacks. For example in figure 3.3 we show a code snippet for the


Method `onClick`

`VLI` Return type: V Number of registers used: 4, Words used for arguments 3

```
Address                      Instruction
0x001C2714 op: iget-object, opr [v0], opr [v1], opr [@1061]
0x001C2718 op: iget-object, opr [v0], opr [v0], opr [@1062]
0x001C271C op: invoke-virtual, opr [v0], opr [@2AD6 ]
0x001C2722 op: move-result-object, opr [v0]
0x001C2724 op: invoke-virtual, opr [v0], opr [@2A55 ]
0x001C272A op: return-void
```

Method `startActivityForResult`

`VLI` Return type: V Number of registers used: 4, Words used for arguments 3

```
Address                      Instruction
0x001C13BC op: iget-object, opr [v0], opr [v1], opr [@103A]
0x001C13C0 op: invoke-virtual, opr [v0], opr [v2], opr [v3], opr [@7C ]
0x001C13C6 op: return-void
```

Method `getActivityContext`

`L` Return type: Landroid/app/Activity; Number of registers used: 2, Words used for arguments 1

```
Address                      Instruction
0x001C1400 op: iget-object, opr [v0], opr [v1], opr [@103B]
0x001C1404 op: invoke-virtual, opr [v0], opr [@627 ]
0x001C140A op: move-result-object, opr [v0]
0x001C140C op: return-object, opr [v0]
```

Figure 3.4: Example API calls

callback onClick(View v) in Dex code format. As we do not inspect framework code, we only track the argument of the callback in this case the register *v1* which holds the pointer to the view element the callback was made upon.

API calls For framework API calls we analyze parameters to the call and the return value, but we do not analyze the framework itself. For instance, in figure 3.4 we demonstrate two code snippet from two different framework call startActivityForResult (Intent , resultCode) and getActivityContext (). The former takes two parameters into registers *v2*, *v3*, and the later returns result through register *v0*. We only track these registers in our analysis.

Event ordering. We precisely keep the ordering of the events regarding their execution. We yet to analyze the effect of event shuffling.

IPC objects. During instrumentation we collect information of the IPC objects and cross reference them with our block summarization. We examine the bundle object associated with intent callbacks to identify the source. We do not analyze the source if it is external. We just report it.

3.4 Conclusion

In this chapter, we have presented our approach to slicing Android applications. We tried to address distinct Android specific challenges. Moreover, we discussed three exciting applications of our proposed framework.

Chapter 4

Targeted GUI Input Generation

This chapter explores the problem of fault localization from a higher level. Unlike chapter 3 where we developed slicing techniques to reveal faulty inputs from a lower level, in this chapter we will identify the source of error in a much higher level. In this chapter, we present AndroidArrow a toolset designed to generate GUI-object-event ordering sets to trigger a particular point of interest, i.e. a target method inside the application. Our main purpose is to channel the flow of the program to the target, which is a particular method in our case. We achieve this by driving the execution in that direction. As Android apps are built upon GUI-object based driven models [84], we generate the event flows, and their $graphicaluserinterface$ or GUI counterparts. We took advantage to two specific program analysis techniques: Static object reference analysis and data flow analysis. We conducted experiments on a set of 324 apps [131].

Because of being event-driven and employing an extensive use of user interface elements, Android applications require non-conventional program analysis techniques.

Our design approach was made keeping this distinct model in mind. Existing reference and data flow analysis cannot be conducted explicitly on Android apps. Android follows a component-centric model. Control flow is determined by the underlying system which mostly manages the life-cycle of the components along with the data they communicate with. Being event driven an application primarily presents a user interface to the user, and she interacts with the UI elements through specific actions such as touches, swipes, etc. As a result, the program flow is largely ordered by the combination of the GUI-object interaction and associated event handlers' invocation. We statically build this mapping between the GUI elements and their corresponding event handlers inside the program.

To this point, our job is not complete. We still need to find the exact path transition to a particular target method. We employed directed path exploration using data-flow analysis that finds component transition paths towards the goal.

In this work, we made the following contributions

- GUI object-event mapping using static object referencing.
- Directed path transitioning using data-flow analysis.

In the following sections, we will be presenting our implementation of AndroidArrow, and demonstrate its usage. We will also show the results obtained from experiments.

4.1 Directed GUI Event Generation

In this section, we present our design and implementation of the toolset AndroidArrow. Before jumping into the implementation details, we would like to discuss different phases

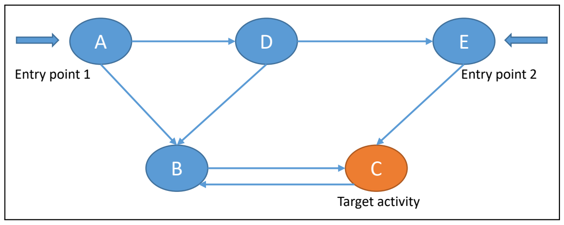


Figure 4.1: Sctivity transtion graph example for path pruning.

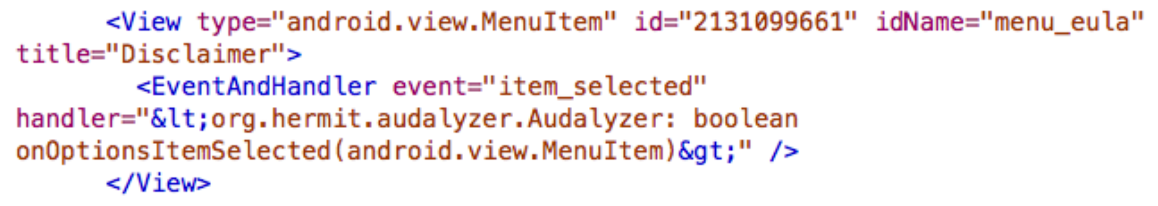
of our analysis.

Static pruning of path. We only take into account the part of the execution that is "useful" to us. For example, before reaching to a specific point in the event flow, there may be infinite ways to explore an app. So, just rely on the event flow may not give us the "optima" path. Lets us have a look on figure 4.1. In the figure, we show a static activity transition graph discussed in 2.4.1. Here each node represents an activity. Now let us assume our target method `foo()` is in activity `C`. We can see that there exist several ways to reach C if we start from `A`. Moreover, if there are multiple entry points, for example, `E`, we may have shorter or direct paths to `C`. So, before we dive into generating path transitioning we build the ascending order of the paths regarding their length by counting the number of the edges it requires to arrive at the target activity from any of the entry point. It is to be noted that not necessarily the shorter path will reveal a transitioning passage to our target activity `C`. But that we will find out later.

Directed path transitioning. Once we have the order of the paths in a higher level, we conduct data flow analysis techniques to generate the implicit flow of the execution. To achieve this, we utilized the path transition analysis support from Redexer [74]. As discussed in Bhoraskar et. al. [33] redexer builds the callgraph of the application and runs

```
====== path ======
        Lorg/hermit/audalyzer/Audalyzer;→
                                onCreate(Landroid/os/Bundle;)V
−##→  Lorg/hermit/audalyzer/Audalyzer;→
         onOptionsItemSelected(Landroid/view/MenuItem;)Z
−−> Lorg/hermit/audalyzer/Audalyzer;→ showAbout()V
−−> Lorg/hermit/android/core/MainActivity;→ showAbout()V
−−> Lorg/hermit/android/core/MainActivity;→ createMessageBox()V
−−> Lorg/hermit/android/core/AppUtils;→ getVersionString()
                                              Ljava/lang/String;
−−> Lorg/hermit/audalyzer/Audalyzer;→ showFirstEula()V
−−> Lorg/hermit/android/core/MainActivity;→ showFirstEula()V
−−> Lorg/hermit/android/core/OneTimeDialog;→ showFirst()V
−−> Lorg/hermit/android/core/OneTimeDialog;→ isAccepted()Z
```

Figure 4.2: Directed path transitioning to the method getVersionString(), as produced by Redexer.



```
       <View type="android.view.MenuItem" id="2131099661" idName="menu_eula"
 title="Disclaimer">
         <EventAndHandler event="item_selected"
 handler="&lt;org.hermit.audalyzer.Audalyzer: boolean
 onOptionsItemSelected(android.view.MenuItem)&gt;" />
       </View>
```

Figure 4.3: Our toolset, based on output from Redexer and Gator, reveals the Disclaimer menu item as the GUI element to invoke to reach this view.

constant propagation and hierarchy analysis to reveal the directed transition edges. Figure 4.2 shows a sample output from the app Audalyzer. We then analyze the revealed paths and order them in the ascending order of their length.

GUI-object-Event mapping. We conduct static object reference analysis using *Gator* [127] to create a GUI-object mapping with their associated event handlers. We cross reference the generated list the results found in the path transition phase to create the list of GUI-objects in their order of executing the target.

Example: Reproducing Audalyzer's KR Error

We start with an example, the Audalyzer app, to demonstrate how we verify the potential bug reports produced by the static analysis. Audalyzer is an audio analyzer. When the app is first opened, an End User License Agreement (EULA) is displayed; the user accepts it by pressing the 'Accept' button, and the app sets the isAccepted field to **true**. This ensures that the EULA confirmation will never pop up again. But if the app exits just after the user presses 'Accept', the isAccepted field value change might be lost, as it is not saved on all exit paths.

Further analysis indicates that isAccepted was changed in the isAccepted() method of the org/hermit/ android/core/OneTimeDialog class. Hence, to reproduce the error we need to trigger input events in such a way that we execute this method and then exit the app at the point after the change.

To do so, AndroidArrow uses the Redexer binary rewriting infrastructure [74] to generate a *directed path transition*. Given the Audalyzer.apk and the target method isAccepted(), Redexer produces a sequence of callbacks and associated method calls such that calling the sequence will lead to isAccepted() being invoked. Figure 4.2 shows the corresponding output of Redexer for this scenario: the types of callbacks that need to be generated to end up in the isAccepted() method.

Note, however, that just generating the sequence of callbacks is not enough. We need to identify the associated GUI elements that, when exercised by the user, trigger those callbacks. To do so, AndroidArrow uses another tool, Gator [127], which, given an APK file, produces the necessary GUI element-callback mapping utilized in the program. Coupling

App Name	Size (KB)	Installs	Time (sec.)	Events#
Facebook	23,112	1,000,000,000–5,000,000,000	73	4
UC Browser	13,429	100,000,000–500,000,000	47	1
Dr.WebAnti-virusLight	1,464	50,000,000–100,000,000	221	3
Yahoo weather	391	10,000,000–50,000,000	56	7
Alarm Clock Plus	2,245	5,000,000–10,000,000	88	3

Table 4.1: *Runtime statistics for top-5 most downloaded apps.*

Gator's and Redexer's outputs we can identify which GUI elements we need to exercise to reach the point of interest in the execution path for invoking the desired method. Figure 4.3 shows one of the outputs of Gator after we filter out results. From Figure 4.2 we see that the first GUI callback event was onOptionsItemSelected in the org/hermit/audalyzer/Audalyzer class. Figure 4.3 shows the output of our toolset, revealing that the associated GUI element is a menu item with the title 'Disclaimer'. That is, we need to first invoke the application menu and select the 'Disclaimer' item. We then automate GUI interactions based on these results, using the A^{3}E [31] app exploration tool: given the sequence of GUI element names, A^{3}E exercises the sequence and "lands" the app at the error point, with no user intervention.

After reaching the error point at runtime, we killed Audalyzer, i.e., forced the app onto an exit path. Finally, we restarted the app to verify whether the changes made in 'isAccepted' were persistent, i.e., the value of the isAccepted field was still **true** (it was not—this is precisely the problem). We now describe the general techniques for achieving directed transition and exiting.

4.2 Evaluation

We have conducted experiments on five most downloaded applications from Google play. Our target method was chosen from a kill-and-restart(KR error) dataset [131]. The results are shown in Table 4.1. Column 4 denotes the time taken by AndroidArrow, and column 5 represents number of events reported by AndroidArrow.

4.3 Conclusion

Effective fault localization requires generating optimum sets of inputs. In this chapter, we present AndroidArrow that exploits static object reference and data flow analysis to reveal GUI object-event sequences for a directed execution towards a target state, eg., a method invocation. AndroidArrow not only facilitates a directed transition but also it does in an optimal way, that is finding the shortest valid path.

Chapter 5

User Defined Deep Linking

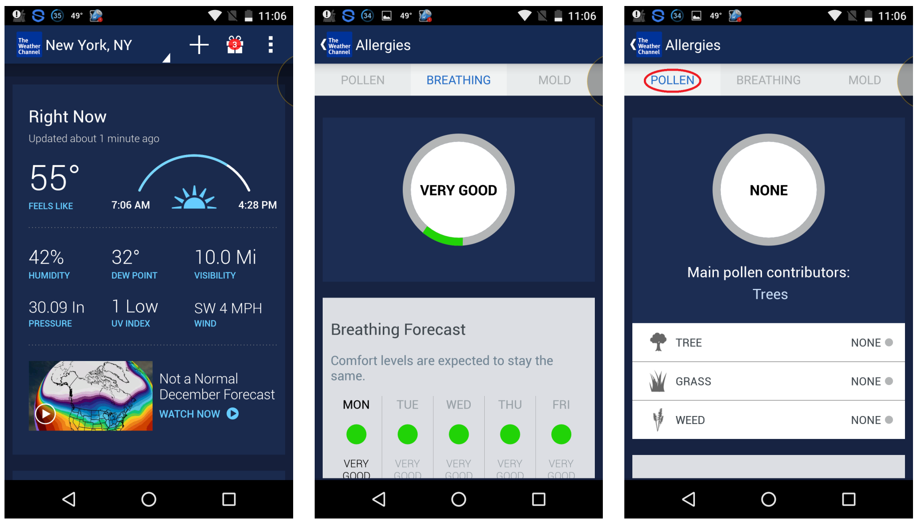
In the web, deep linking refers to the use of hyperlinks to a specific piece of web content (e.g., http://ulink.com/code/) on a website (e.g., http://ulink.com). Web deep links are instrumental to many fundamental user experiences: navigating to a web page from another, bookmarking it, and sharing it with others. They have also been crucial for many important services; for example, search engines use deep links to crawl web pages and to map search results to appropriate landing pages. Historically, mobile apps did not have any equivalent deep links, making the aforementioned tasks impossible for individual pages within the apps. As VentureBeat rightly put, "Imagine a web without URLs. That's what the mobile app world looks like now [July, 2014]" [112].

To address this, mobile deep links have been introduced in recent years. Mobile deep links are URIs that point to specific locations in a mobile app. A mobile deep link can launch an app that is already installed on a user's mobile device (similar to loading the home page of a website) or it can directly open a specific location within the app (similar to deep linking to an arbitrary web page). For example, the URI fandango://thelegomovie_159272/movieoverview directly navigates to the page with the details of the "The Lego Movie" in the Fandango app. Today, all major mobile platforms, including Android, iOS, and Windows, support mobile deep links.

Even though mobile deep linking is an important first step towards randomly accessing any arbitrary location within an app, it lacks many useful properties of web deep linking. First, unlike web deep links, mobile deep links require *nontrivial developer effort*–several lines of codes per unique deep link—resulting in a low adoption rate even within the top apps [108]. Second, unlike its web counterpart, mobile deep links have *poor coverage*—a small number of locations within an app, predefined by the developer, are directly accessible via deep links (details in 7.4). Finally, today's deep links are *defined statically* by developers to facilitate navigation to a target page given its link; the dual process of dynamically determining the link for a given page is not possible even if a deep link exists to that page.

We have developed *uLink*, a lightweight approach that addresses the above problems. uLink is compatible with existing mobile deep links (i.e., the underlying mobile OS handles them in the same way); but it requires very minimal developer effort, it supports dynamic link creation, and it achieves significantly higher coverage than existing mobile deep links. All this enables many novel user experiences that so far existed only in the web world.

One key challenge uLink addresses is improving coverage—creating links to any app location (referred to as *app view* or *view* hereafter), including to the ones that depend



(a) An independent view (b) A view dependent on previous view (c) A view dependent on UI action (on "POLLEN")

Figure 5.1: Examples of views uLink can support.

on previous views or on user interactions. uLink uses two key mechanisms. The first mechanism is *shortcut*. uLink continuously monitors for explicit data dependency between successive runtime views in an app. In Figure 5.1, view (a) launches view (b), by providing the location "New York, NY" selected by the user in (a). In some cases, e.g., if (a) and (b) are separate Android activities (i.e., pages), uLink can transparently capture the data transferred from (a) to (b) and encode it in the link to (b). This allows uLink to quickly invoke the link to go to (b), without first going to (a). More importantly, it improves coverage to views that depend on data from previous views (location in this example).

Shortcuts do not cover all app views. The view shown in Figure 5.1(c) is created by the user by tapping on the "POLLEN" tab, and there is no explicit data transfer between (b) and (c) for uLink to capture—both views are within the same Android activity. To create links to such views, uLink uses a limited form of record and replay. uLink continuously records UI actions in the current page and encodes them in the link (we call this a *shortcut-and-replay link*). When the link is invoked, uLink first directly navigates to the most recent shortcut-reachable view (e.g., (b) in Figure 5.1), and then replays the UI actions to navigate to the target view.

uLink's record and replay mechanism is very different from existing record and replay techniques that have been successfully used to repeatedly navigate to arbitrary locations of an application for desktop, server [48, 126, 100, 63, 88, 86], web [94, 82], and mobile platforms [57, 68]. Compared to a traditional record and replay systems, uLink's record and replay is (1) lightweight and universally deployable—this is because it records and replays only UI events, which, as we show later, is often sufficient to recreate a target view with high fidelity; (2) fast—this is because uLink does not replay a whole session; rather it replays UI events only after reaching the most recent shortcut-reachable view; and (3) user-friendly—during link creation, a user does not have to specify the starting point of recording (it is implicitly given by the most recent shortcut-reachable view) and during link invocation, replay happens in the background to give users a true click-and-go experience.

Another challenge uLink addresses is reducing developer effort. uLink is implemented as a library that developers can include in their apps. The library transparently interposes data dependency between views and UI events with very minimal developer effort. For the shortcut-only deep links, which are the only ones supported by existing mobile deep links, uLink needs no effort from the developer, except for including the library in their app (as opposed to several lines of code per deep links in existing mobile deep links). For shortcut-and-replay deep links, which are not supported by existing mobile deep links, developers need to write one line of code *per event handler*. Our evaluation with existing apps shows that the overall overhead is minimal.

The final challenge uLink addresses is identifying links that may not be correctly open in some later point in time. This is not surprising since even a full featured record and replay tool cannot guarantee reproducibility of the target view due to many nondeterministic factors. Broken links are common on the web as well. A ulink may not open correctly e.g., if the target view opens a file that is deleted after the link is created, if a user is not logged into the app, if some UI events cannot be correctly captured or replayed (e.g., Android does not provide APIs for applying long taps on list items), etc. However, it is important that the user gets a consistent experience—when she bookmarks a view, she should know if the bookmark is indeed valid—whether she will be able to open the link in the future on not, and if not, why it might fail. An important contribution of this dissertation is to develop efficient techniques to provide such feedback to users when a ulink is created or opened.

We have implemented uLink as an Android library and evaluated it with 34 (of 1000 most downloaded) Android apps. While existing mobile deep links require in the order of 20–30 LoC per deep link, uLink can support shortcut-only links (a superset of deep links) to *all* pages with an average of 8 LoC. Overall, uLink achieved 70% links coverage and provided accurate user feedback (especially for links with file system dependencies). We also found that ulinks remain reasonably stable over time with new app versions and updates in app contents.

In summary, we make the following contributions. (1) We develop uLink, a mobile app deep linking mechanism that requires much less developer effort, but provides significantly more coverage than existing mobile deep linking. (2) We develop techniques to predict if a ulink may become broken in future, and if so, under what conditions this might happen. (3) We evaluate uLink with 34 real apps.

5.1 Motivation and Goals

In this section, we motivate the need for uLink, set our goals and review related work.

5.1.1 uLink in Action

We motivate uLink with a few concrete scenarios. Our goal here is to demonstrate some of what uLink can enable, and we leave details of its design, challenges, solution, and implementation in next sections of the chapter.

Developer experience. For ease of deployment, uLink is implemented as a user-level library, similar to existing analytics libraries such as Localytics [8], Flurry [4], and Appsee [27]. The developer includes the uLink library in the app and this alone readily makes the app uLink-enabled, with shortcut-only links (Figure 5.1(a) and Figure 5.1(b)). To enable shortcut-and-replay links (Figure 5.1(c)), the developer adds one line of code in every UI event handler of the app.

uLink library and companion services. As the user uses a uLink-enabled app, the uLink library continuously tracks explicit data flow between app views (e.g., intents transferred between Android activities) and UI events. At any point of time, the user can request a ulink to the current app view by shaking the device. An external companion service, typically a first-party service, can request to be notified each time a user saves a new link or it can request uLink to automatically create links to all pages the user visits within specific types of apps (e.g., all 3rd party apps). We have implemented two such companion services: Bookmarks, which stores all links the user wishes to save and invoke later, and Stuff-I-Have-Seen, which indexes contents and links to all pages the user visits in all apps and, like a web search engine, allows the user to search the content and to directly navigate to any content of interest by using the associated uLink.

User experience. Here are few scenarios a user can experience with a uLink-enabled app and companion services.

(1) In a recipe app, the user can bookmark any page containing her favorite recipes. She can later invoke the ulink from the Bookmark service to directly open the page.[1]

(2) She can create macro-like ulink for frequent tasks in an app. For example, in a library app, she can go to the "renew book" page, select all books, extend the return date by one more month, and finally hit the "renew" button, and create a ulink to capture the whole sequence of actions. Later, she can invoke the ulink (from the Bookmark service) to renew all her books with one single click.

(3) She can use the Stuff-I-Have-Seen service, which runs in the background to transparently

[1]In theory, these bookmarks can also be shared with friends, in the same way we share web links.

index contents of all app pages the user visits such as recipes, news articles, etc. along with their ulinks. Later, the user can use the app to search for a recipe she read in some app in the past, and click on the ulink of the result to directly open the app page.

For implementations of these scenarios see §5.3.2.

5.1.2 uLink Goals

Our overall goal is to enable mobile deep links that are similar to web deep links in terms of functionalities and convenience. To provide useful and user-friendly links, uLink should satisfy the following requirements: **(1)** A user should be able to create a link *dynamically*, like web links, by specifying only the target app view. **(2)** uLink should have *good coverage*—a user should be able to create ulinks to most, if not all, views of an app. **(3)** Invoking a ulink should be *fast*, and it should produce a *consistent* app view, despite minor changes in the app or its contents.

To be practical and reach a large population of smartphone users, **(1)** uLink should require *minimal developer effort* to make an app uLink-enabled. **(2)** uLink should incur *minimal runtime overhead* and have no impact on the app's experience. For *ease of deployment*, uLink should not require changing the mobile OS or rooting the device. **(3)** uLink should be *compatible with existing mobile deep links* so that the underlying OS can offer the same user experience. For example, if a user generates a link for an app and later uninstalls the app and invokes the link, the OS will notice that the target app is missing and will redirect the user to the app store to install the app (this procedure is currently in place for deep links).

By definition, uLink must capture *correct* links. Our notion of correctness tries to be as close as possible to that of today's web URLs. As for web URLs, the correctness of a link depends on the backend providing the content (the app or the web site publisher). For instance, a link to a news article will work as long as the app or web publisher does not remove the corresponding data object from their database or does not change the object identifier. We cannot control the app (or web site) backend, so we aim for correctness under the assumption that the backend has not deleted a link's content. On the other hand, over time, links may break or point to different content also for other reasons. In general, the content of a link may vary across users, devices, time, and location (and sensors in general). We classify links into two broad categories:

- **Immutable** from client-side: links whose content remains the same, across users, devices, time and location. These links are unlikely to break over time. For instance, http://www.dictionary.com/browse/uri?s=t always points to the definition of the word "uri".

- **Mutable** from client-side: these are links whose content is dynamic and may change depending on the device (e.g., web sites using the device's file system), user (e.g., personalized web sites), time (e.g., news web sites), location (e.g., weather web sites), and sensors in general. For example, a link to a news web site's top stories will show different content during the day. The content of www.foreca.com/United_States/Washington/Seattle will change over time, but it will always show the weather for Seattle. Instead that of https://www.wunderground.com/ will automatically adapt to the current location. Mutable links can sometimes break. For instance, a Facebook URL saved on a device where the user is logged in, may fail when opened on a device where the user is currently logged out; same for a link to the content of the Amazon cart.

These examples show that links can be ambiguous; a priori, it is not always clear whether the content referenced by a link will change and whether a link may break. On the other hand, as the web shows, users are accustomed to what a URI can or cannot capture, and, to how different web sites behave. For instance, users expect the Foreca link above to always show the weather for Seattle, and the Wunderground one to adapt to the current location. We expect uLink will provide a similar, natural experience, and users will learn to deal with mutable app links.

uLink supports both immutable and mutable links. As for web URLs, uLink aims to provide *best effort* links, as consistent as possible with the app behavior. For instance, in a news app the page showing the "Daily Top Stories" will show different news stories every day. If a user saves a link to such page, *also* the link will show different content every day. A restaurant app page showing the "Nearby Restaurants" uses the GPS sensor to establish the user's current location. A link to such page will *also* adjust to the current location. On the other hand, we acknowledge that app links may arise new types of ambiguity which we discuss in §5.5.

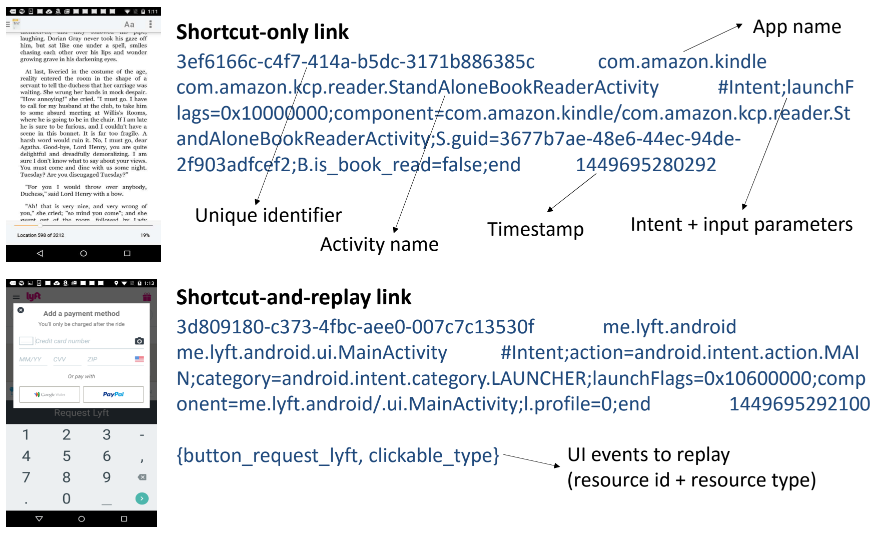
As in the web, app links may sometimes break. uLink promptly detects when a link cannot be safely saved or replayed, and provides detailed feedback to the user or to the application capturing such links on the user behalf. Fundamentally, uLink cannot guarantee 100% coverage of an app views because of the deployment constraints (minimal development effort and ease of deployment) it must satisfy. In §5.2.3, we enumerate uLink's possible failure cases and explain what the feedback contains.

5.1.3 uLink Approach

We now briefly describe how uLink achieves the goals listed above; the details will be described in the next section. uLink is implemented as a library that a developer includes in the app with *minimal effort*. The library continuously monitors various data dependencies and UI events of the current view so that, anytime, it can *dynamically* create a link with the dependencies encoded in it. Figure 5.2 shows two examples of ulinks: the first ulink points to page 598 in a Kindle book, and the second encodes the sequence of actions for requesting a lift in the Lyft app (the result is the dialog for entering payment). After being saved, a ulink can later be invoked to *quickly* access the view, by taking shortcuts to views that depend only on data encoded in the link (e.g., book page), and/or by replaying, in the background, the UI events encoded in the link (a clickable button in the second link in Figure 5.2). This approach gives uLink *high coverage* of dependent views, which are not supported by existing mobile deep links. The overhead of shortcut-and-replay links is reduced by recording and replaying only UI events (button clicks, checkbox selections, etc.), which, as we will show later, is sufficient to recreate the target page with high fidelity in many cases.

5.2 System Design

This section describes how uLink was designed to meet the aforementioned goals: high coverage and speed (§5.2.2) while ensuring minimal developer effort (§5.2.4).


Shortcut-only link

3ef6166c-c4f7-414a-b5dc-3171b886385c com.amazon.kindle com.amazon.kcp.reader.StandAloneBookReaderActivity #Intent;launchFlags=0x10000000;component=com.amazon.kindle/com.amazon.kcp.reader.StandAloneBookReaderActivity;S.guid=3677b7ae-48e6-44ec-94de-2f903adfcef2;B.is_book_read=false;end 1449695280292

Shortcut-and-replay link

3d809180-c373-4fbc-aee0-007c7c13530f me.lyft.android me.lyft.android.ui.MainActivity #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10600000;component=me.lyft.android/.ui.MainActivity;l.profile=0;end 1449695292100

{button_request_lyft, clickable_type}

Figure 5.2: Examples of shortcut-only and shortcut-and-replay links.

5.2.1 Overview

Mobile apps consist of a set of *pages*, where each page typically contains a set of *UI elements* such as buttons, checkboxes, lists, or menus. Each UI element can have an associated *event handler*, which is invoked when the element is interacted with. The whole of a page may not be viewable to the user at once; we call a part of the page shown in the current screen a *view*. A page may contain many views: the default view is what is shown on the screen when the user navigates to the page (Figure 5.1(b)), and the user can navigate to a different view within the same page by UI interactions such as selecting a tab (Figure 5.1(c)), choosing a date from a date picker control, filling out a search box and clicking on a search button, etc. A UI interaction can also lead from a view to the default view of a separate page.

Mobile apps are stateful and pages/views can have state dependencies. A page can use some data generated in a previous page. For example, the page in Figure 5.1(b) needs the location selected by the user in the previous page (in Figure 5.1(a)). A view can also depend on other views (e.g., one tab uses a flag set in another tab). Finally, a view can also depend on the sequence of UI actions starting from the default view. For example, the view in Figure 5.1(c) is obtained by tapping on the "POLLEN" tab in the default view.

Using this terminology, we identify three broad classes of links a user may capture in an app (listed in increasing order of complexity to support them). A user-defined link can link to the following states in an app:

1. *Stateless view*: A view whose state does *not* depend on states created in previous pages (e.g., a page showing weather, as in Figure 5.1(a)). It can be created without any input parameter, or a set of statically-defined parameters that do not depend on previous pages/views.

2. *Stateful view*: A view whose state *depends* on app states created in previous pages (e.g., showing breathing forecast at a location *selected in the previous page*, as in Figure 5.1(b).

3. *UI-driven view*: A view created by UI events generated on the *same* page (e.g., Figure 5.1(c), created by tapping on the "POLLEN" tab in Figure 5.1(b)).

Existing mobile deep links support stateless views only. They can pass static parameters to target views, but cannot observe the internal state of the app (i.e., they live *outside* the app). This is precisely the reason why they cannot cover stateful or UI-driven views that depend on states (e.g., location selected by the user) and UI events (e.g., tapping on a particular tab) *inside* the app.

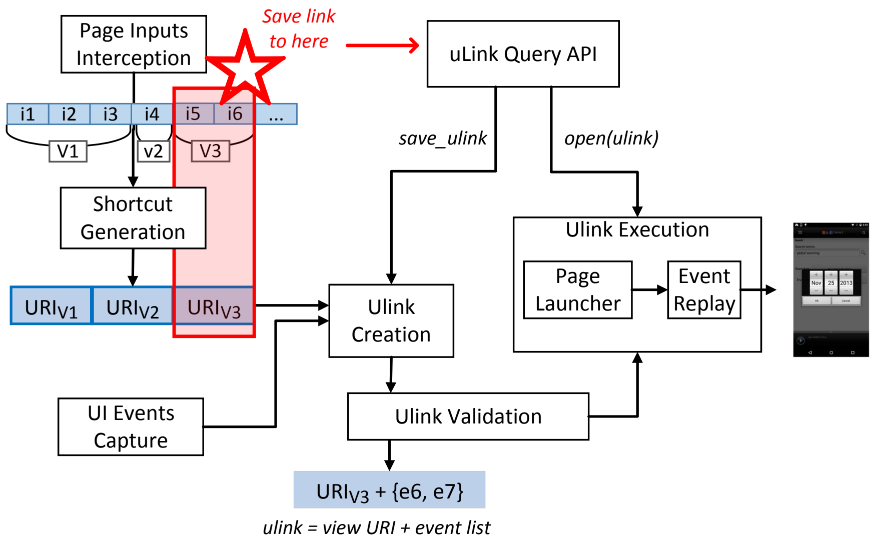


Figure 5.3: uLink system architecture.

In contrast, uLink supports links to all the three types of views and thus achieves its high coverage goal. Figure 5.3 shows the entire system architecture, including the Query API (§5.3). In the following, we discuss the techniques we propose for creating, executing and validating ulinks.

5.2.2 Improving View Coverage

We first describe how uLink supports links for stateful and UI-driven views (not supported by deep links).

Links for stateful views

Stateful views depend only on data from previous pages. By definition, a link to a stateful view points to the default view of its page.

uLink uses a novel technique called *shortcuts* to generate links to stateful views.

We observe that a page in an app is usually instantiated through a launcher method responsible for rendering the page in the foreground (startActivity(intent,options) in Android and prepareForSegue:(uiStoryboardSegue) in iOS). This method usually expects as input a description of the page to render and possibly other parameters, which are not known to processes external to the app. This is equivalent to the query string in a web URL (e.g., in https://uLink.com/index.php?title=uLink_details&action=edit the query string is the part of the URI after '?').

Our key insight is that uLink can *program user-defined links by demonstration*: by observing how views are assembled during user interaction (V1, V2 and V3 in Figure 5.3), uLink can learn how to re-construct them. Specifically, uLink continuously intercept all messages (i1, i2, etc.) sent to the page launcher method so to infer message structures and input parameters, necessary to render a view. uLink encodes the message structure and input parameters in a URI generated for the view (URI_{V3}). To open a saved ulink, the uLink library simply invokes the page launcher method with properly structured messages assembled using the parameters stored in the URI. In this way, uLink can *shortcut* to the default view of any page in the app.

Assuming the link is opened under the *same conditions* (e.g., file system, sensors, and so forth) as when it was created, this approach guarantees accurate and safe, stateful links. They are accurate because this approach is goal-oriented. When a user requests capturing a link to the current view, the link is derived directly based on what app state was provided to *that* view. They are safe because this process does not risk breaking the program logic. We discuss later what happens if the link is opened under conditions different from those at creation time.

The above idea is simple and can be implemented by overloading the launcher method of the framework page classes (e.g., startActivity method of Android Activity classes).

Links for UI-driven views

The above technique of intercepting data passed between pages does not capture UI events within a page and hence is not sufficient to recreate a UI-driven view. To support such views, uLink adopts a limited form of record and replay.

uLink continuously monitors UI events triggered during user interactions, and event handlers that are fired. To reduce overhead, uLink monitors UI events only in the current view; when the user moves to a different view, the UI events of the previous view are discarded. This approach does not compromise coverage since uLink can directly navigate to the default view of the current page by using a shortcut-only link to the (stateful) view. To create a link to a UI-driven view, uLink encodes two pieces of information in the link: (1) input parameters to launcher method of the current view (same as shortcut-only links), and (2) UI events that lead the user from the page's default view to the current view. When the link is invoked, uLink first launches the page's default view by using its input parameters, and then replays the UI events to navigate to the target view. The UI events are replayed in the background, and so the user sees the same click-and-go experience as shortcut-only links. We call such links shortcut-and-replay links.

Mobile app contents can be dynamic. For example, suppose a page's default view shows a list of restaurants. User clicks on the second item "Kabab Palace" to generate a view with the details and menu of the restaurant, and saves a link to the view. Now suppose, after a month, the app updates its contents and the same restaurant "Kabab Palace" appears as the fifth item in the list. To deal with such changes, unlike traditional record and replay systems, uLink prioritizes content over UI structure during replay—it will search the list to find "Kabab Palace"; if it exists, uLink will click on it irrespective of its position in the list. Only if "Kabab Palace" does not appear in the list anymore, uLink will fall back to structural replay and click on the second item of the list. Such fallback is useful to deal with highly dynamic contents, such as a link to the top news story, which may change frequently.

We also observe that compared to record and replay tools, this approach does not require any recording start point, and it is much faster. Imagine a task where the user searches through old news by first entering some keywords and then specifying a date range, thus landing on the page with the news matching the specified criteria. Assume the user wants to save a link to this page with the results. A traditional record and replay tool (i) would require the user to specify the start point of the task (i.e., the page where the keywords were entered) and (ii) would replay every single user action to re-create the search results, i.e., entering the search string, clicking on the Start date button, pressing the search button, etc. Instead, uLink intercepts the parameters needed to generate the search result page and, through *a single* function call, loads the page.

uLink does *not* need to record the time gaps between separate UI events. Instead uLink replays the sequence of recorded UI events such that each UI event is fired after the previous event has been dispatched, as notified by the device framework (e.g., on Android through the `onUserInteraction` callback). In this way, it is also possible to remove userinduced delays or idle periods.

Note that, on the other hand, uLink's record and replay is limited compared to existing record and replay systems: it captures only UI events (button clicks, checkbox selections, etc.), which, as we will show later, is sufficient to recreate the target page with high fidelity in many cases. Capturing IO and sensor access operations would bring us closer to the ideal world of deterministic replay, but monitoring these events would lead to unsupportable overheads in terms of annotations that developers would have to provide, in terms of OS modifications or in terms of runtime overhead. By capturing only UI events, uLink hits a sweet spot between existing lightweight but low-coverage deep links and heavyweight but high coverage full-blown record and replay.

Limitations

Since uLink captures only data passed through page's launcher methods and UI events within a page, there will be cases where uLink won't be able to correctly open a link at a later point in time. For example, consider an eReader app ulink that opens a book (stored in the local device) at a specific page (saved in a configuration file). The link will fail if the book is removed from the device or it may lead to a different page of the book if the page number in the configuration file is modified after the link is created. As mentioned before, taking a snapshot of all resources that a link might depend on, can be prohibitively expensive.

To address this, uLink incorporates a feedback mechanism. Intuitively, uLink tracks all dependencies that it may fail to capture in the link. This gives users (or applications on their behalf) an idea about whether the link can be faithfully invoked in the future, and if not, why. Using the feedback, users can rely on their own judgment to decide how to use the link. Such feedback can also be useful to deal with link ambiguity (as described in §5.1.2). For example, while creating a bookmark for later consumption, uLink may notify the user that the view depends on her location, which uLink is not capturing in the link. Knowing this, the user may or may not save the link in her favorites depending on whether she expects to invoke the link from a different location.

5.2.3 Link Validation

uLink provides feedback to users (or an application on their behalf) at the time of link creation and of link execution. To be more concrete, suppose a user navigates from Page1 to Page2 of an app, and wishes to create a link to Page2. Can the link be invoked later to correctly see the same content? The answer depends on whether the pages access any external resource such as a file (outside the parameters explicitly transferred from Page1 to Page2's launcher method). Let us consider the four cases in Figure 5.4, with different dependencies from external resources.

Case (a) No external dependencies: uLink can correctly open Page2, by providing the parameters stored in the link to the launcher method of Page2.

Case (b) Page2 reads an external resource: (1) uLink may not be able to correctly open Page2, or (2) it may be able to open the page, but with potentially different content. This may happen if the content of the external resource is modified after the link is created. For example, if Page2 plays a specific music file from the local device, a link to the page will fail if the music file is deleted from the device. Similarly, if Page2 shows content of a local file, a link to the page will show different content if the file is modified after the link is

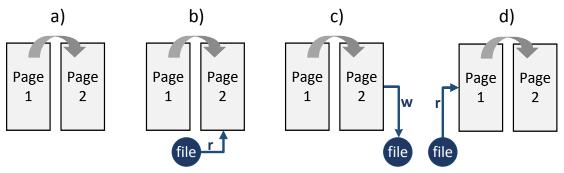


Figure 5.4: uLink can replay correctly a link to page 2 in case a), c) and d), and in case b) if the file doesn't change after link creation.

created.

Case (c) Page2 writes an external resource: uLink can correctly open Page2, since the content of the external resource does not affect the content of Page2. So Page2 remains unaffected even if the external resource is modified after the link is created.

Case (d) Page1 reads an external resource: uLink can correctly open Page2. If the external resource somehow affects the content of Page2, its value must propagate through the data passed from Page1 to Page2, which uLink correctly captures. Note that, in order to capture possible changes to the external resource Page 1 depends on (which may affect the value passed to Page2), the *reference* to the value (e.g., filename) and *not* the value itself should be passed from Page 1 to Page 2. Whether link arguments are passed by value or by reference is up to the application, and ulinks behave accordingly.

Of all the cases above, Case (b) is the only case where uLink *may* not be able to correctly open a link to Page2. Ideally, preventing such behavior would require recording contents of all the external resources as well, which can be prohibitively expensive. uLink therefore does not try to prevent such behavior, and rather notifies users or the companion services at link creation or execution time that the link may not be replayed correctly if a specific resource is modified. We call this process link validation.

Lightweight dependency tracking. The key challenge in supporting link validation is that it must happen during user interaction, at minimal overhead and with minimal changes to the app. A possible approach, which requires no changes to the app, is to monitor I/O and sensors by instrumenting generic OS-provided APIs (e.g., system calls). However, this is not ideal because it needs instrumentation of the framework and incurs runtime processing and storage overhead. Another approach is to inject the monitoring logic in the app and track information flow in the app (similar to taint tracking [49]); however, this requires nontrivial development effort and incurs high runtime overhead. Our solution is to rely on an offline automated analysis of the application to generate an app-specific *summary* of resource dependencies of each event handler. Once the summary is built offline, it is installed on the device (by the uLink library downloading it from the cloud), and consulted each time a link is saved or opened. By design, this approach cannot be as accurate as heavyweight API instrumentation, taint tracking or other approaches requiring OS modifications, but it provides a first, practical approximation of the problem while not compromising our goals of low overhead and minimal developer effort.

To frame the problem, we first define the following terms. We collectively call *resources* entities external to the app that can change arbitrarily, after a link is created. These include files, databases, and sensors. Our definition of "change" means any type of modification to the content or properties of the resource. For example, a file content can be "changed" by overwriting the file or by modifying one of the attributes such as read/write permissions. As discussed in §5.1.2, we do not monitor the network because, as with web URLs, we have no control on links that break due to the app (or web publisher) backend operations, such as deleting a data object or modifying an object identifier.

We call *source APIs* and *sink APIs*, the APIs the application framework provides to get or put data, respectively, into such resources. In particular, we consider the following categories: file system read and write, database read and write, and sensor read operations. The list of such APIs can be extracted using tools such as SuSi [12]. Finally, we collectively call *callbacks* event handlers triggered, synchronously or asynchronously, by the app (e.g., in response to interacted UI elements) or by the framework (e.g., app lifecycle events).

Static analysis. Our first effort on generating offline app summaries is to rely on static analysis of the app. We generate the call graph of the app, and then recursively traverse it to find out a connected path from a callback to a source or a sink API. If we find any connecting path, we add the mapping <callback,source> or <callback,sink> to the summary. Online, each time a link is saved, the uLink library logs which callbacks have been invoked to generate the app state. In this way, uLink can use the summary to look up whether any of such link-required callbacks has dependencies on external resources, and report that in the feedback. Note that once the uLink library is added to the app, the developer effort required for the feedback generation is zero, because all link-required callbacks are logged directly by the library.

This approach proved quite robust, meaning that we didn't encounter any case in which a resource that was accessed by a link was not caught by the summary. However, we found it too conservative because of its coarse-granularity. For example, knowing that a specific callback reads *some* file is not sufficient to infer whether this operation may or may not compromise the correctness of the link. If that file doesn't change after link creation, then the link will work correctly. If the file has changed, the link may or may not work. However, the problem is that online, with our restricted monitoring setup, we cannot capture the identifiers of the modified resources.

Static + dynamic analysis. To address this, we augment static analysis of the app with dynamic analysis. To exemplify, let us consider the file system resource. Our key insight is that while we do *not* know the file identifiers online, we can capture them offline and leverage them to establish *callback relationships*. We run each app using a Monkey (such as the Android [66] or Windows Phone [87] UI automation tools). We collect logs of all invoked callbacks as well as traces of file system accesses (using strace-like utilities). We then intersect read and write operations that share files with the same identifiers. The obtained file-relationships are added to the summary. For example, the dynamic analysis may produce a trace where callback c1 reads file f, and callback c4 writes to file f, so a c1 $\rightarrow$ c4 file-relationship is added to the summary.

Note that a perfect dynamic analysis would deem static analysis unnecessary. However, existing Monkeys have less than ideal coverage; we therefore primarily rely on static analysis to generate summaries for *all* callbacks and refine the summaries with more fine grained information whenever they are available from dynamic analysis (i.e., whenever the Monkey exercises the event handler).

Once we have generated the summaries, we use them online in the following way. As the user interacts with the apps, the uLink library continuously monitors all callbacks invoked due to user interaction, and keeps a log *only* of the callbacks that led to a file system or database write. When a link previously saved is opened, the link-required callbacks (recorded when the link was saved) are processed using the summary. Any link-required callbacks associated with file system (or database) writes can be safely ignored (as in Case (c) in Figure 5.4). Instead, any link-required callbacks associated with file system (or database) reads can be ignored only if they have no relationship with other write callbacks that were logged after the link was saved. In fact, this means that re-creating the link state requires reading a file (or a database) that was modified in the past, after the link was saved (as in Case (b) in Figure 5.4). If such file (or database) dependency is found, uLink generates a feedback report including details about the identified root cause, i.e., the callback information and the source/sink API. Using the example above, suppose that the link-required callbacks of the saved link include callback `c1`, and that the callback log contains `c4`. Because `c1` has a file dependency on `c4`, the link correctness may be compromised.

Note that, we cannot keep an indefinite logs of all write callbacks, for all apps. We approximate by keeping a runtime log for a maximum period of time (currently an hour). Our intuition is that writes that are older than that period are likely to have been absorbed by the system (e.g., changes in preferences) so that they can be safely forgotten.

Our implementation currently provides fine-grained feedback only for the file system, while detects database changes only at the granularity of read/write (through static analysis). However, the same approach can be applied to databases by intercepting database APIs in the app framework (offline, during dynamic analysis). For sensors, fine-grained analysis of read/write operations is less critical because with the exception of a few sensors (e.g., microphone) sensor operations are always reads. This also means that links cannot break because of changes in the sensor values. However, sensors can cause link ambiguity, which we discuss in §5.5. Finally, since it happens offline the dynamic analysis could also be improved by adopting heavyweight tools such as taint tracking [49].

Other failure types. A captured link may fail also due to technical limitations of our system or of the app framework on which it runs. For instance, on Android, it is possible to track long tap UI events for items in a list, but the framework doesn't provide an API for replaying such events (while it does for single tap events). A link may also fail because of developer errors. Record and replay requires the developers to add one line of code per each event handler. If the developer forgets to instrument them all, a link may fail. Such kind of problems are captured by the uLink library when re-creating the link state and feedback is provided.

5.2.4 Developer Effort

uLink is implemented as an application library. To make her app uLink-enabled, a developer includes the uLink library and extends the `uLinkPage` class provided by uLink, instead of the original `Page` class provided by the underlying framework (this is needed to overload the framework's page launcher method). Once the library is added, shortcut-only links are readily enabled.

To support shortcut-and-replay links, uLink requires app developers to add one line of code in each UI event handler of the app. This effort is larger but still small (see§5.4.1). Moreover, since this process is rather mechanical, the logging statements could also be injected automatically through a source-to-source transformation tool. Table 5.1

Change	**Dev effort (LoC)**	**Enabled links**
Add uLink library, extend from uLink-provided Activity	1 per main Activity class	shortcut-only links
Log event handlers	1 per UI event handler	shortcut-and-replay links

Table 5.1: Developer effort to add uLink in Android apps.

summarizes the developer effort for the two types of ulinks in Android.

5.3 Implementation and Use Cases

This section provides details on our Android implementation of uLink, and describes three companion services we have built leveraging ulinks.

5.3.1 uLink Library

We have implemented uLink on Android 5.1.1. On Android, pages, called *Activities*, are started by taking a direct Android *Intent*. Activity classes are created by extending one of 13 Activity classes provided by Android. An activity can be launched by supplying an Android *Intent*, which is essentially a passive data structure containing an abstract description of an action to be performed. To transparently capture the intents, so that a target activity can be directly launched by supplying the necessary intent, uLink uses the following tricks: (1) it provides one uLinkActivity class for each of the framework-provided Activity classes, with the same external interfaces, so that developers can extend the uLinkActivity class, instead of the framework-provided Activity classes, to create a new Activity class. (2) The launcher methods of the uLinkActivity classes are instrumented to dynamically capture the intents provided to them, and to encode them in a link generated for the views in the page.

The above tricks can be used in other platforms as well. For example, in iOS, Scenes extend NsObject.UIResponder.UIViewController, and, in Windows, Pages extend Systems.Windows.Controls.Page. So the uLink library for those platform could provide replacement classes to be extended by developers to create app pages.

Developer effort. To enable shortcut-only links, each *main* Activity must extend the corresponding uLink-provided Activity class. By main Activity, we mean activities that implements one of the 13 Android Activity classes. App developers often create one or a few customized main Activity classes that extend from the framework-provided classes and use them to instantiate all other activities, and hence the overhead is rather small (§5.4.1).

To support shortcut-and-replay links, uLink requires app developers to invoke the trackEventHandler(view, view_type) method of the uLink library, each time a UI event handler is fired. Developers provide the corresponding View object (i.e., UI Element) that raised the event and the its type. Due to the specific design of Android, a View object can be of many types, such as a button, a list item or a textbox. For some types of UI event handler, such as click event handlers, this information can be captured through the framework by probing the View objects inside an Activity (effectively its UI tree) when it is first loaded and assigned trackers. Since Click handlers are the most common type of handlers, this would be a significant reduction in the developer effort. We are currently exploring this approach. Table 5.1 summaries the developer effort needed for the two different types of link.

To support the generation of summaries for link validation, for the static and dynamic analysis of Android apps we generated the call graphs using the Soot[109, 98] analysis framework. Since Android applications do not have a traditional Main entry point, we created a dummy entry point leveraging the approach of Flowdroid [29]. We used the Android source-sink APIs listed by SuSi [12]. This list includes 26,322 source and sink APIs available from Android 4.2. We selected 2280 sources and sinks and grouped them into six categories: file system, database, resources, media, camera, and sensors.

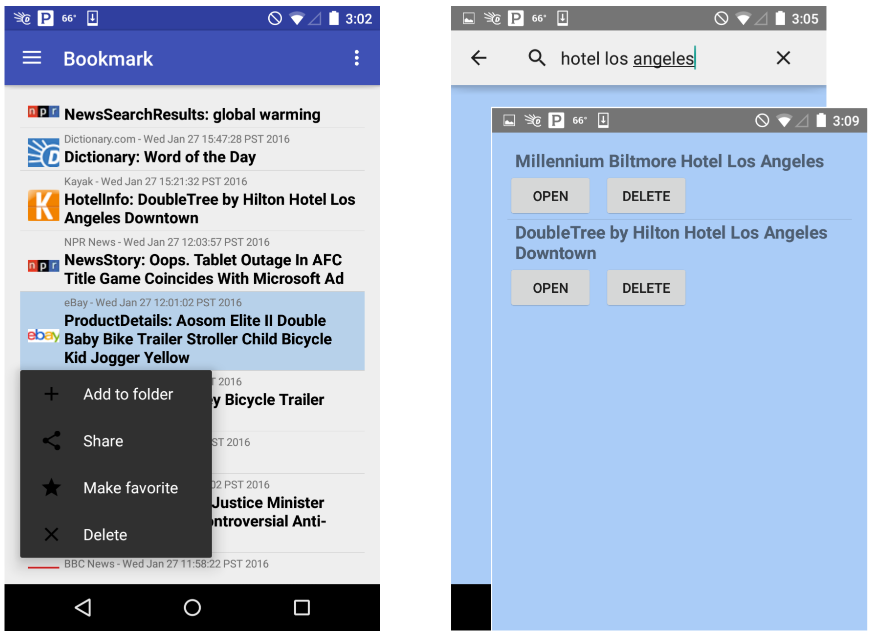
Query API. uLink provides the following API that an application or companion service can use to programmatically generate or invoke links. (1) getLinkToCurrentView(): Returns a link to the current view. (2) getLinkOnCondition(condition, callback): Registers a callback, which will be invoked with a link and content of the current view, when the current view matches a condition. For example, for the Stuff-I-Have-Seen service described below, the condition can be "when the view belongs to a 3rd party app and the user has spent more than 5 seconds on it". (3) openLink(link): Opens the specified link. If the app is not currently installed, it takes the user to the app store to install it. The ulink object ulink (pageURI, events, callbacks) contains the page URI, the list of events for replay (possibly empty), and the list of link-required callbacks (i.e., callbacks that were invoked when the link was first saved).

5.3.2 Companion Services using uLink

We implemented three Android services demonstrating different uses of ulinks.

Bookmark. Being able to bookmark links to an arbitrary state in an app is useful for various purposes. (i) When launched, mobile apps always start from the same entry page. Even if a user *always* only cares about the content appearing on a certain page, say the third, she must always go through the first and second page to reach her target. A user that uses such app every day, perhaps multiple times, would benefit from being able to create a shortcut to the target page. (ii) When interacting with an app, a user may find some content she wants to save for later. (iii) Most users have tasks that repeat identical every day or every so often, such as monitoring the price of an item to purchase, ordering a pizza or checking whether new interesting houses are on the market. These tasks are likely to always take the same user inputs (e.g., number and type of pizzas). Users would benefit from being able to record such repetitive tasks and automate their execution. (iv) Filling forms in mobile apps is notoriously painful. Having the ability to record user inputs (e.g., login information, search parameters, etc.) for specific app pages can improve user productivity. We built Bookmark (left-hand side of Figure 5.5) that collects links to content, actions, tasks a user wishes to capture for fast recall. Each time a user shakes her phone, a link to the current view is saved into the Bookmark.

Stuff-I-have-Seen. Users browse lots of content inside their apps (e.g., hotels to book, restaurants to visit, news article to read), and sometimes would like to be able to search through "all the stuff they have seen", and *not* through all the content those apps (or the web) offer. We built Stuff-I-Have-Seen (right-hand side of Figure 5.5), reminiscent of similar work for the web [47], for desktop applications [80], and for entire computers [45].


Bookmark

NewsSearchResults: global warming

Dictionary.com - Wed Jan 27 15:47:28 PST 2016
Dictionary: Word of the Day

Kayak - Wed Jan 27 15:21:32 PST 2016
HotelInfo: DoubleTree by Hilton Hotel Los Angeles Downtown

NPR News - Wed Jan 27 12:03:57 PST 2016
NewsStory: Oops. Tablet Outage In AFC Title Game Coincides With Microsoft Ad

eBay - Wed Jan 27 12:01:02 PST 2016
ProductDetails: Aosom Elite II Double Baby Bike Trailer Stroller Child Bicycle Kid Jogger Yellow

BBC News - Wed Jan 27 11:58:22 PST 2016

hotel los angeles

Millennium Biltmore Hotel Los Angeles

DoubleTree by Hilton Hotel Los Angeles Downtown

Figure 5.5: Bookmark (left) and Stuff-I-have-Seen (right) services we have built using uLink.

This service transparently logs contents the user sees in her apps, it indexes such content and provides basic search capability. The app contents (i.e., texts appearing in the UI elements in the app page) are obtained by processing the app's UI tree. For indexing it uses the Apache Lucene library[1]. Semantics analysis (currently not implemented) could also be performed locally [53] or using the cloud (as in Google's Now on Tap [58] and Bing Snapp [111]). The app currently tracks eBay product details, Kayak searches, NPRNews news, Spotify's songs and artists, and Kindle's books.

360-IFTTT. IFTTT [5] is a popular app that allows users to "program" <if-do> recipes such as "If it rains, remind me to take an umbrella". Currently these recipes are built using open APIs and web sites. With uLink, recipes can tap into app's content and actions. We

App	Category	Description	Downloads	Total	uLink LoC	
				LoC	shortcut-only	shortcut-and-replay
NPR News	News & Magazines	News reader	1M-5M	13,114	2	18
AnkiDroid	Education	Flash card manager	1M-5M	45,959	7	66
Book Catalogue	Productivity	Book list manager	100K-500K	41,587	8	228
Vanilla Music	Music & Audio	Music player	500K-1M	15,518	4	79
K9-Mail	Communication	Email client	5M - 10M	67,721	4	75
eBay	Shopping	e-Commerce app	100M-500M	500,251	6	368
Lyft	Transportation	Taxi service	1M-5M	356,894	1	30
Spotify	Music & Audio	Streaming music service	100M-500M	523,999	12	430
Amazon	Shopping	e-Commerce app	10M-50M	418,503	10	235
Amazon Kindle	Books & Reference	e-Book reader	100M-500M	432,040	13	346
BBC News	News & Magazines	News reader	10M-50M	398,835	12	170
Watch ESPN	Sports	Live sports	10M-50M	452,437	5	125
AccuWeather	Weather	Weather update	50M-100M	392,707	20	132
Aldiko Book Reader	Books & Reference	e-Book reader	10M-50M	239,967	6	198
ASTRO File Manager	Productivity	File manager	50M-100M	393,712	10	231
Photo Editor by Aviary	Photography	Photo editor	50M-100M	340,653	3	181
Booking.com Hotel Reservations	Travel & Local	Hotel reservations	10M-50M	317,012	5	491
APUS Booster+	Productivity	System utility	10M-50M	29,923	7	75
Compass PRO	Tools	Utility	5M-10M	167,415	3	68
Dictionary.com	Books & Reference	Dictionary software	10M-50M	388,963	32	229
Duolingo	Education	Language tutorial	10M-50M	253,975	2	190
Hulu Plus	Entertainment	Live streaming	100K-500K	411,272	11	204
KAYAK Flights, Hotels & Cars	Travel & Local	Hotel, flight, & car manager	10M-50M	380,609	6	390
MakeMyTrip-Flights Hotel	Travel & Local	Hotel, & flight	5M-10M	421,598	15	345
Dictionary-Merriam-Webster	Books & Reference	Dictionary software	10M-50M	217,013	3	71
Music Player for Android	Music & Audio	Music player	10M-50M	89,411	5	79
Retrica	Photography	Camera & photoeditor	100M-500M	266,428	8	172
SPB TV	Media & Video	Streaming TV	10M-50M	318,170	5	163
the Weather	Weather	Weather update	10M-50M	273,724	7	167
TuneIn Radio	Music & Audio	Streaming radio	100M-500M	340,260	19	276
Advanced Task Killer	Productivity	System utility	50M-100M	12,843	6	50
WebMD for Android	Health & Fitness	Health app	5M-10M	319,537	7	170
Yahoo!	News & Magazines	News reader	10M-50M	395,316	10	260
Zillow	Lifestyle	Apartment finder	10M-50M	404,068	11	344
Average				**283,572**	**8.4**	**195.8**

Table 5.2: The 34 Android apps to which we added the uLink library and the developer effort required. (First 5 apps are open source.)

built 360-IFTTT. Users can specify simple "if" conditions based on location and time. "Do" actions are specified by executing the desired task with the app (once), saving the ulink in the Bookmark (by shaking the phone), and copying the ulink into 360-IFTTT.

5.4 Evaluation

In this section, we evaluate uLink on five metrics: *developer effort*, *coverage*, *correctness of link validation*, *link consistency over time* (i.e., whether the links remain valid over time and across newer app versions), and *runtime overhead and performance*. We report the results for both shortcut-only and shortcut-and-replay links. Our evaluation is based on 34

Android apps, shown in Table 5.2. We first tested the library with three apps (NPRNews, Lyft, eBay) and later integrated it into 31 more apps, *without* any changes to the library, confirming the generality and applicability of our approach.

5.4.1 App Dataset and Developer Effort

The uLink library was integrated successfully in a total of 34 apps. Among the top 1000 Android apps, we selected apps based on popularity and compatibility with Android 5.0 from a variety of app categories with the exclusion of games (they are not in scope of our link-based scenarios) and native code apps. Five of the apps are open source, and hence we could modify their source code to include uLink. For the other apps, we used Soot [109, 98] for Dalvik bytecode instrumentation. The limiting factor in integrating uLink into closed source apps was not the complexity of the logic for injecting our changes, but the recurrence of bytecode obfuscation in apps. Once instrumented, we verified that they worked with 5 random links to 5 different pages.

The fact that we were able to integrate uLink through an automated instrumentation process is a first proof of how easy and mechanical the required changes are. In addition, we counted the lines of code (LoC) that were changed or added to integrate the shortcut-only or shortcut-and-replay variants of uLink. Table 5.2 shows the results. For comparison the table also reports the size of each app's codebase. To obtain an estimate for closed source apps, we counted the LoC after decompiling the app to Java source code using the dex2jar [15] and jd-gui [6] tools.The numbers do not give an exact count for LOC, but they provide a good approximation.

On average, shortcut-only required to change only 8.4 LoC in the app code. The

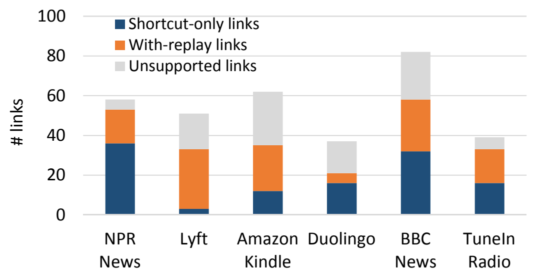


Figure 5.6: uLink link coverage in 6 apps (NPR News is open source, others are closed source).

smallest effort was 1 LoC (Lyft) and the largest 32 LoC (Dictionary.com). Recall that today's deep links can support only stateless links. Shortcut-only provides a superset of deep links, with a much smaller developer effort. To give some comparison points, we inspected the code of two open source apps that support deep links. Ankidroid exposes one deep link which is handled in 35 LoC. Wikipedia also exposes one deep link coded in 23 LoC. With less than a third of this developer effort, uLink enables deep links to most app page views (as we will show in the next experiment), and preserves application state.

The developer effort for shortcut-and-replay is higher (196 LoC in average) because it depends on the number of event handlers in the app, but the changes are still relatively few (on average 0.07% LoC of the entire codebase need to be changed) and are rather mechanical. As discussed in §5.3, in the Android framework, at least the very popular Click handlers could be handled automatically, without developer hints, with an expected reduction of the LoC by a third.

Needless to say, if the uLink library could be added to the Android framework, it would require zero developer effort.

App	Link description	Dependency	Same conditions			Altered conditions		
				Ground truth	uLink		Ground truth	uLink
Amazon Kindle	Open a book page	r db, r pref	same db & pref	Yes	Maybe (r db, r pref)	del db, w pref	**No** (r db, w pref)	**Maybe** (r db)
Amazon Kindle	Sync books with cloud	w db, w pref	same db & pref	Yes	Yes	del db, w pref	Yes	Yes
Photo Editor by Aviary	Open a photo	r file	same files	Yes	Yes	del files	No (r file)	No (r file)
Lyft	Share referral code	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Lyft	Request a lift	loc	same loc	Maybe (loc)	Maybe (loc)	different loc	Maybe (loc)	Maybe (loc)
NPR News	Open a story	r/w cache in db	same cache	Yes	Yes	del cache	Yes	Yes
NPR News	Add news to playlist	r/w db	same db	Yes	Maybe (r/w db)	del db	Yes	Maybe (r/w db)
NPR News	Locate nearest station	loc	same loc	Maybe (loc)	Maybe (loc)	different loc	Maybe (loc)	Maybe (loc)
eBay	View product	r db	same DB	Yes	Maybe (r db)	del db	Yes	Maybe (r db)
WebMD	Refill a prescription through Wallgreens	camera, r pref loc	same loc same pref	Maybe (camera, r pref, loc)	Maybe (camera, r pref)	different loc, del pref	Maybe (camera, r pref, loc)	Maybe (camera, r pref)
Vanilla Music	Play a song	r file, w pref	same files & pref	Yes	Yes (r file)	del files, w pref	No (r file)	No (r file)
AnkiDroid	View a card	r db, r pref	same db & pref	Yes	Maybe (r db, r pref)	del db, w pref	**No** (r db, r pref)	**Maybe** (r db, r pref)
Book Catalogue	Reset hint	w db, w pref	same db & pref	Yes	Yes	del db, w pref	Yes	Yes
Book catalogue	Manually add a book	w db	same db	Yes	Yes	del db	Yes	Yes
Merriam-Webster Dict.	View word of the day	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Dictionary Free	Search a word	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Summary	**Decision (Yes, No, Maybe):**		**Wrong: 2 (6%) Same as ground truth: 24 (75%) Conservative: 6 (19%)**					
	Accuracy No/Maybe feedback:		**Complete: 23 (88%) Incomplete: 3 (12%)**					

Table 5.3: uLink feedback for various links with dependencies on sensors, file system and database (r=read, w=write, db=database, pref=preferences).

5.4.2 Coverage

We evaluate whether uLink can provide high coverage of an app views. We pick 6 apps and manually enumerate all possible views in them. We are careful to report only unique views (e.g., if a menu for adjusting screen zoom appears in three different views, we count it as one link rather than three). Then, we manually save links to every such view, and open them to verify whether the result is correct. In these tests we do not vary the operating conditions (e.g., same file system) so if links fail it is because of technical limitations of uLink or of the app framework.

Figure 5.6 reports the results. Across the 6 apps we found that on average there are 55 views one may want to reference through a link. uLink provides coverage for 71% of them. Compared to the state-of-the-art where, if apps have deep links, it is no more than a handful of links, this is a significant improvement. In particular, shortcut-only alone (which comes with a tiny developer effort of 8 LoC, see Table 5.2) provides an average of 19 links per app and successfully enables links to almost all page's default views in the tested apps. Hence, with a much smaller developer effort, uLink provides much higher coverage.

The unsupported links are mainly due to failures in replaying UI events. Most failures are an artifact of binary instrumentation. In fact, for NPR News, the only open source app here, the coverage is 91%. In closed source apps, instrumentation fails to log custom event handlers (in the real world, the developer would provide the correct annotations), so some UI events (although captured) cannot be replayed. Other reasons for link failures were the following: i) there are UI elements to which the developer did not assign a resource identifier so they cannot be replayed (this was the reason for the 9% failed links in NPR News), ii) there are some special UI events (e.g., list long click) for which the Android framework does not provide a replay API, and iii) there are page views that are displayed in a browser (in Kindle) which cannot be reached by uLink. Although not 100%, uLink provides a good coverage. With the framework's support we are optimistic this coverage can become close to 100%.

5.4.3 Correctness of Link Validation

To evaluate whether uLink can discover external dependencies of links and correctly report when the link may fail, we conduct a controlled experiment with 16 links in 11 apps with dependencies on file system, sensors and databases.[2] The links emulate what a real user would like to save in such apps, such as shortcuts to relevant pages (e.g., hourly news, product pages) or to recurrent actions with saved inputs (e.g., sync the book list, refill a prescription, locate nearest radio stations). To verify the dependencies reported

[2]For clarity of analysis, we further distinguish read/write operations on preferences and cache. These resources can be easily recognized as their names remain constant across apps.

by uLink, we change the resources after links are created, and examine if the link can open the original app view. For instance, in the Vanilla Music app, we save a link for playing a song stored on the SD card, delete the song (from the app) and open the link. Each link is opened twice: (1) in the same conditions: after creating the link, we interact with the app for a while and then open the saved link, and (ii) in altered conditions: after creating the link, we force a change in the resource(s) the link depends on and then open the link.

Table 5.3 shows the results. In analyzing them, recall that currently uLink monitors file system dependencies at fine-granularity, but database and sensor dependencies at coarse granularity. The table reports the ground truth and the uLink's output for both conditions. "Yes" means that the link can be safely opened, "No" means that the link won't work, and "Maybe" means that the link may not work (if the resource it depends on is modified). For No and Maybe, uLink provides a feedback on the root cause. Overall, uLink is wrong in only 2 out of 26 test cases (marked in bold in the table). In 75% of the cases it agrees with the ground truth, and in the remaining 19% of the cases it takes a conservative decision, mainly due to the lack of details on database read/writes. For instance, the "open a Kindle book" link requires reading the database. As uLink cannot yet track whether those reads have been affected by previous writes, it fires a warning. For links requiring reading the file system, uLink can be more accurate. For instance, the open photo link in Aviatar requires reading a file (the photo). In the same conditions, uLink correctly detects the link will work. In the altered conditions, uLink captures that the photo has been deleted and understands the link-required callbacks have a dependency on that operation, thus correctly concluding the link won't work. In No/Maybe situations, the

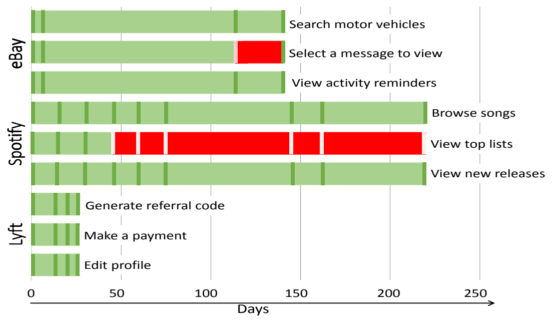


Figure 5.7: uLink across different app versions over time (green = link opened correctly, red = link failed).

feedback is complete in most cases (88%).

Overall, uLink is able to provide accurate feedback. With the addition of fine-grained database analysis we expect the accuracy to be close to 100%. The services consuming ulinks can further decide how to interpret this feedback, especially in Maybe situations. For instance, "Stuff-I-have-seen" may be less conservative and treat Maybe verdicts as Yes. Another service for automating recurrent user tasks (e.g., refill prescriptions) may be more conservative and treat them as No.

5.4.4 Consistency over Time

We now explore whether ulinks are reliable over time, in the face of i) app updates, and ii) app content changes. To evaluate consistency across different app versions, we downloaded several older versions of some of our apps and tested whether ulinks saved in the oldest version could be opened correctly in the latest versions. We tested 3 different links in Spotify (9 versions covering 7 months), in eBay (4 versions covering 4.5 months) and in Lyft (4 versions covering 1 month). Figure 5.7 reports our findings. For

Spotify, 2 out of 3 links worked for the entire period; one link broke after the 3rd version update. Upon investigating further, we found that the cause was the removal of a UI element in the page layout. The second failure we noticed was for an eBay ulink which did not work for an intermediate version. In this case, uLink could not load the UI element and the app showed a dialog instead, but there was no crash. This study shows that links are relatively stable over a short period of time—there was no single failures for 50 days after the link creation.

Then we investigated how uLink copes with app contents updates. We created a set of 10 links from 5 different apps (NPR News, Vanilla Music, Spotify, Book Catalogue and eBay). We selected common links such as viewing an eBay item, viewing the top NPR news, playing a song in Vanilla Music or viewing the top releases in Spotify. Then for 4 weeks (8 weeks for NPR News), at least once day, we ran a script that would open each of the links, and save a screenshot of the resulting page and debug logs. By inspecting the saved screenshots and logs, we observed that all 10 links, except 1, provided correct results for the entire duration of the experiment. Note that *some of the content were no longer available in the app itself; but the app backend still maintained the content and hence uLink could retrieve it*, highlighting the benefit of uLink. The failed link was for viewing a specific top news in NPR News. After 4 weeks, the link returned an empty page: the news item was not available anymore in the app backend with the same news ID.

App and link description	Replay time *(ms)* (& UI events)		Replay log size (*bytes*)	
	RERAN	uLink	RERAN	uLink
TuneInRadio: Stream a radio station	5,351 (4)	1,105 (1)	7,480	651
VanillaMusic: Go to a song and play it	7,762 (4)	982 (1)	6,880	693
NPRNews: Search news for some dates	17,345 (5)	1,392 (0)	11,296	984

Table 5.4: Comparison of uLink and the record and replay RERAN tool in terms of replay time, number of replayed UI events and log size.

5.4.5 System Overhead and Performance

The uLink library is 245 kB bytes big so it adds a small storage overhead to existing apps. The computation overhead is also minimal and processing occurs asynchronously to the app execution with no app slowdown. ulinks are typically 100–150 bytes (the average size of the links used in Table 5.3 is 136 bytes).[3] For shortcut-and-replay links, uLink needs to log UI events. We executed AnkiDroid, a relatively I/O intensive app, for 10 minutes, and measured a callback log size of only 432 kB. With an hour time windows the cost is a few MBytes. We conclude uLink is a lightweight system.

When opening a ulink, if the app targeted by the link is already running (in foreground or background), the delay is the page rendering time. If the app is not running, uLink needs to first start the app and then render the page, so the delay includes the app loading time (typically 1–2 seconds depending on the app) and the page rendering time. We executed a simple experiment to verify that uLink's overhead on an app page rendering time is negligible. We took five apps and created three links for each. We measured the uLink page rendering overhead as the difference between the 1) the average time necessary to open each link and achieve a stable page with the uLink-enabled app, and 2) the

[3]The size of ulinks mainly depends on the size of the intent objects captured by input interception. According to Android guidelines, this is maximum 1 MB. However, to keep links small, rather than saving large objects in the URI, the objects can be saved in storage and their reference be included.

time the unmodified app takes to load the same page. As we are interested in comparing the page rendering times, we kept the app running in both conditions, but ensured the cache was cleared after each test. Moreover, when measuring the uLink rendering delay, we ensured the Activity targeted by the ulink was cleared on the stack (i.e., when the ulink is opened, we force a call to OnCreate). To measure the rendering time, we measured the time between the first call of OnCreate and the time when the root view of the Activity was inflated (i.e., all UI elements in the page are rendered). Across the 15 tests, the mean delay was 28 ms.

In the previous test, we considered shortcut-only links. To evaluate the delay of shortcut-and-replay links we use RERAN [57], a record and replay tool for Android, as a baseline. Table 5.4 shows three example tasks with three different apps. We recorded the tasks with both systems, stored the log (a link in the case of uLink) and then replayed each task (open the link in the case of uLink). We report execution time, number of replayed UI events and size of the logs. The tasks are ordered by increasing number of clicks. uLink is from 5 up to 13 times faster than RERAN. More importantly, while in RERAN the replay time increases as the task's complexity increases (number of visited pages and clicks), with uLink it remains fairly constant. Likewise, the RERAN log sizes are at least 10 times larger—for uLink, replay log is essentially the link itself, which is quite small (last column). Finally, the average replay time with uLink is 1.2 seconds, which is what users would expect as page loading time.

Overall, these experiments show that uLink is a lightweight system and provides an acceptable user experience.

5.5 Limitations

We discuss key limitations of our work.

Link ambiguity. It is not always possible to correctly understand a user's expectations when saving a link. For example, after saving a link to the "Nearby restaurants" view, when the user invokes this link later on, does she expect to see the restaurants near the current location or near the location the link was recorded at? Or let us consider the NPR News app: when clicking on the *first* news item in the list of "Hot Daily News", a view showing the selected news story will be displayed. If a user saves a link to this view, does she want to save a link to that specific news article or to the top daily news (i.e., the first item in the list)? As discussed in §5.1.2, such link ambiguity is not unique to uLink, but we expect more of such cases with mobile apps. As for web URLs, practice will help users understand what links can or cannot capture, on a per app basis. However, the system can help identify ambiguous situations and prompt the user for clarification. Our system-generated feedback is a first step towards this goal.

Unwanted replay. How can we prevent users from buying the same Amazon item twice by mistakenly clicking twice the same app link? We think such problems cannot be resolved without help from the app developer, in the form of annotations for pages or UI elements which should be excluded from link capture (or request user confirmation) or by providing some app semantics so the system can generated link descriptions and promptly warn the user.

Security and ulink sharing. Users can share web URLs. From a technical point of view, users can share share also ulinks. However, to make this viable, uLink needs to address at least two problems. First, it needs to handle cases in which a user may receive a link for a not-installed app or from an app with a different version than the installed one. Second, security and privacy issues must be addressed: What if the shared link is malicious? What if the shared link makes changes to the app's preference? What if the share link contains personal information such as login information or home address? Link encryption, cloud-aided link security analysis, user/developer opt-out policies can help alleviate these issues.

App pages revisitation. By default, uLink always provides the most updated content that a ulink references (which is also the case for the majority of web URLs). Revisiting app content is currently not supported, but it could be enabled by relying on a caching infrastructure and diffing tools similar to what proposed for the web [17, 18, 105].

Technical limitations. We currently only support capturing of UI elements that have a unique resource identifier associated with them. Moreover, we rely on the Android framework APIs to programmatically replay UI events. For UI events for which Android provides only the screen coordinates (e.g., random touches on screen), uLink can only mimic that action across devices with similar resolutions. Finally, uLink cannot handle certain types of gesture such as swipes, pinch and zooms due to the lack of support from the Android framework. These limitations have little impact on most apps and for our use cases. However, they can be a problem for use cases involving games.

uLink largely relies on the parameters passed between app pages through intents. If the syntactic structure of the program changes, mostly due to new app versions, uLink will be unable to reflect those changes. This means that old ulinks captured may not work. Changes in app versions can be tracked with cloud support so to automatically detect broken links (this is also important for link sharing).

Fine-grained app summaries. uLink currently provides fine-grained summaries only for file system operations. For the file system we were able to leverage OS utilities (strace). For databases it is necessary to instrument the Android framework and app APIs to track the information flow. Taint-tracking can also improve the precision of our approach.

5.6 Conclusions

uLink is a novel approach to enable deep links in mobile apps. uLink is distributed as a small library that developers include in their apps with tiny changes. Compared to deep links, uLink provides higher coverage of an app views with less developer effort. uLink goes beyond the state-of-the-art: it provides links that are stateful and that can be specified by a user on demand (unlike deep links), and it achieves these benefits without incurring large resource overheads or modifying the OS (unlike record and replay systems). Although usability is not a goal of our system, uLink provides the first elements towards that goal: fast experience, no specification of a session start point, feedback for links that may not work properly. We implemented uLink on Android and used it with 30+ apps with very promising results.

Chapter 6

Application Recovery from Faults

As smartphones and tablets continue to increase in popularity [91, 90], more and more critical software (e.g., financial, military, medical apps) shifts to these new platforms. Unfortunately, smartphone software (from the OS to libraries to apps) has a high defect rate [78] due to many factors, including the novelty and rapid evolution pace of smartphone software, the low barrier to entry for publishing software via app marketplaces, as well as the myriad devices and user-specific configurations on which it is running. Maintaining certain functionality (such as the ability to place phone calls) is critical on smartphones, and unlike on desktop systems, we cannot always rely on network connectivity for downloading and applying a patch to fix the bug. Hence there is a strong impetus for self-healing smartphone software. In this chapter, We take a step towards this by presenting an approach for automating patch construction to recover from and prevent future crashes in Android apps.

Our approach is more suitable for smartphone apps than, say, desktop or server programs, due to the compartmentalized nature of smartphone apps: many pieces of functionality, e.g., GUI elements, can be turned off without affecting user experience [31]. Our implementation first employs dynamic analysis to detect when an app has entered an error state and to identify the offending part of the app; then it implements recovery, either by eliminating transient faults and continuing to run at full functionality, or rolling back to a safe state followed by sealing-off the offending part and operating in a limited mode while avoiding further crashes. An example would be a bug in the auto-completion code that crashes the smartphone's Dialer app whenever the user tries to dial a number: when we detect the crash, rather than rendering the whole app inoperable and unable to place emergency (911) calls, we create and apply a patch to turn off auto-completion, hence containing the damage and allowing the Dialer app to continue to run (albeit with some limitations). While this is an incipient form of self healing in smartphone software, it is compelling nevertheless.

Exceptional conditions or bugs have many causes, and manifest in a variety of ways: unhandled exceptions, assertion failures, system overload; our framework detects several such exceptional conditions and reacts accordingly, depending on whether the fault is transient or persistent.

As the context of the error may be different in different situations, it is unreasonable/infeasible to expect the smartphone user to know how to circumvent the error and keep the app functional. Hence our system automatically detects the error and changes the app's behavior to respond to these circumstances so that most of the normal app workflow is not hampered. The approach is centered around several techniques which facilitate self-healing: (1) extracting a high-level model of app operation which captures legal app transitions as a graph; (2) continuous monitoring to detect crashes; (3) identifying and sealing-off the offending app component through app bytecode rewriting.

Although crash prevention has value and is preferable, smartphone software bugs are a fact of life, so in this work we will outline our ideas regarding recovering (potentially limited) functionality, as follows. Using the statically-constructed model and the crash point, the app goes to a "rollback" point and depending on the nature of the fault (transient or persistent) it creates appropriate conditions, potentially via seal-off, to avoid future crashes.

We have implemented a prototype that equips Android apps with the aforementioned self-healing functionality; we have chosen Android as a target due to its leader status [71]. Our framework is robust, running on actual phones and supporting widely popular apps such as K-9 Mail and Facebook Mobile. Moreover, our approach does not require access to the app source code or modifications to the kernel or libraries; rather we rely on static analysis and rewriting at the bytecode level.

We have evaluated our implementation on a set of bugs in real-world, popular Android apps running on Motorola Droid Bionic phones. Experiments show that our implementation manages to successfully perform self-healing without prohibitive overhead, and the self-healing process is accomplished very efficiently, in less than one minute.

We expect that the fault information revealed by our system could provide feedback to the app developers to help them develop bug-fixing patches.

6.1 Approach

We first present discuss self-healing in the context of the Android platform and then present our approach in detail.

Self-healing in the context of Android apps Self-healing computing systems' capabilities include inferring (1) ways of detecting failures (e.g., due to system malfunctions such as exceptions, violations of operational constraints), and (2) strategies for applying corrections to restore (some or all) system functionality. The key concept behind our self-healing mechanism in Android is that an app must resume to a normal GUI state after the app experiences a failure. Hence, it is key that we discover a model (set of GUI states) beforehand, so during recovery the app can be driven to an appropriate state and avoid future crashes. For the purpose of this work, we focus on two types of Android app faults: *transient* and *persistent*. Transient faults occur when operations fail as a result of resource unavailability and will disappear if the operation executes again if the resource becomes available; a simple and effective recovery strategy for these is re-execution. Persistent faults do not go away via re-execution, e.g., because they are due to errors in application logic; in such cases, sealing-off the offending operation is an effective recovery strategy.

Our approach consists of the following phases:

- **Model construction and rollback point identification.** In this phase we identify discrete, safe and unsafe points in the app (which form the basis for our approach), as well as transitions between them, using static analysis.
- **Detection.** In this phase, our framework performs dynamic analysis on systems's behavior and app output (e.g., system-wide resource usage, app method calls, GUI

elements, privileged actions) to detect crashes and identify faulty components.

- **Recovery.** Our recovery mechanism works in two phases. First, after a crash point is detected, we identify a safe rollback point and if needed (depending on the nature of the fault), we seal off the bytecode associated with the crash point by using the model to identify the faulty part of the binary and then rewrite the bytecode to avoid future execution of code associated with the crash point. Second, we restart the app to a nearby safe point so that users can continue their work and interaction with the app.

6.1.1 Architecture

Figure 6.1 shows our system architecture, centered around detecting crashes and in response applying seal-off patches. A *model* is constructed first, via static analysis on the *app* (bytecode); the model includes rollback (resume) points where the app will be driven when recovering from a fault. Next, the app is executed, either manually by users or automatically via systematic exploration tools and its execution *log* is monitored via dynamic analysis. When a failure is detected, we employ bytecode rewriting (code generation and instrumentation) to create a *patch*. We apply the patch via bytecode rewriting; the patch seals off the functionality responsible for the crash, yielding a *self-healing capable app*. We now describe our system's operation in detail.

6.1.2 Model Construction and Rollback Point Identification

The app model forms the basis for identifying safe and unsafe points in the app. Safe points will be used for rollback and unsafe points will be sealed off. The model,

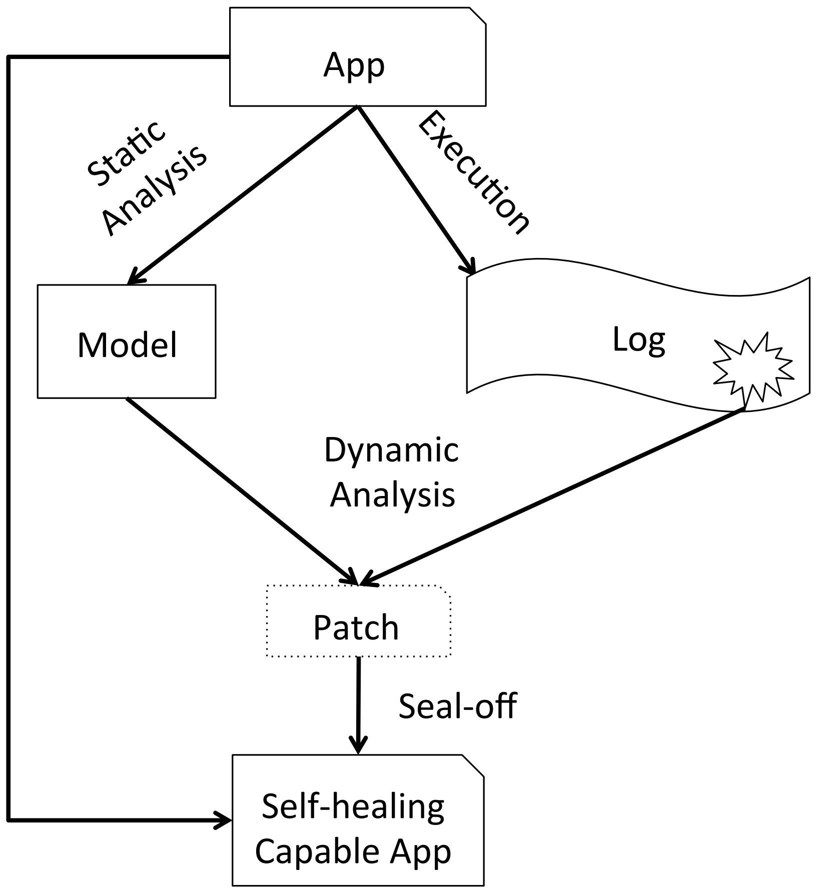


Figure 6.1: System architecture.

named *Static Activity Transition Graph* (SATG [31]), is a transition graph where nodes are app screens ("activity" = GUI screen in Android parlance) and edges represent possible transitions between screens (which will take place, e.g., as the user navigates around using the GUI). For example, a directed edge from activity A to activity B points to a valid transition from A to B as a result of the user exercising a GUI object associated with A.

Note that SATG construction is not a contribution of this work, but the SATG is nevertheless essential to identifying rollback points and taking recovery actions: in the case of a fault, we calculate the nearest "safe" activity that can be used as a rollback point.

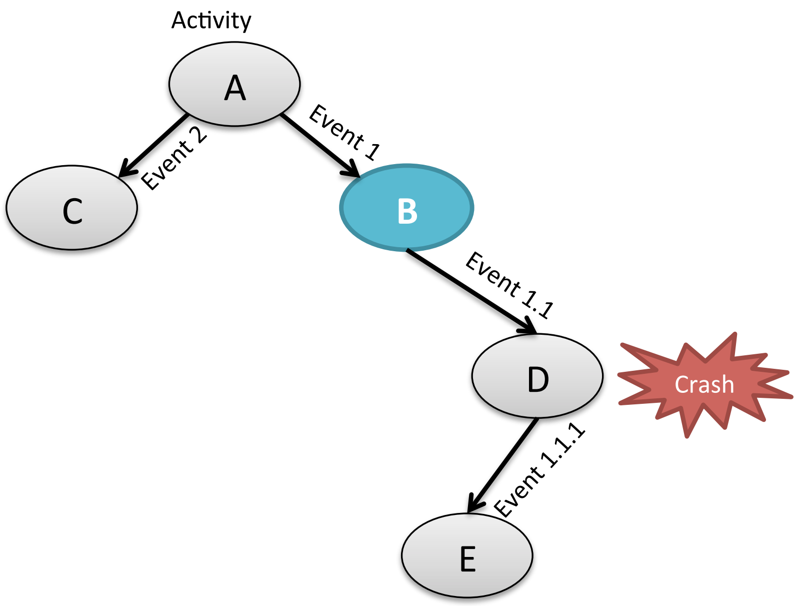


Figure 6.2: Our static analysis infers B as the rollback point when the app crashes at point D.

Static analysis is important because it reveals the sequence of callbacks associated with activity transitions: invoking these callbacks (which in normal user interaction corresponds to exercising a sequence of GUI elements) allows us to reach the rollback point. Figure 6.2 shows a SATG constructed with our A^{3}E tool [31]. Here the root node A is the initial activity. Each edge to the next node is labeled with the callbacks or events triggering that transition. For example Event 1 is responsible for the transition from activity A to activity B. Suppose the app crashes at activity D (marked in the figure). From the SATG we can see that activity D was reached via activity B, so activity B is the nearest safe point to restart the app. More generally, rollback points can be obtained via a backward traversal from a crash point.

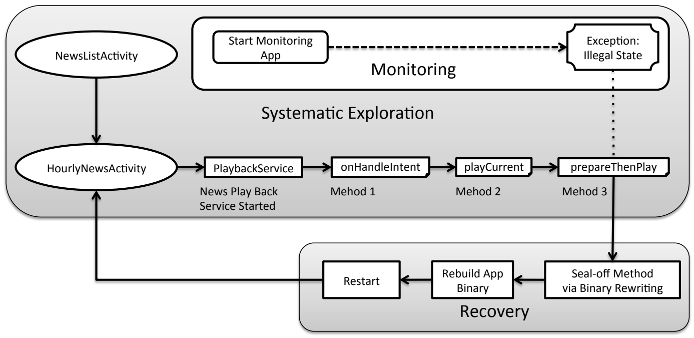


Figure 6.3: Example: fault point detection and rollback in the NPR News app.

6.1.3 Detection

We now discuss *what we detect* —classes of faults in Android app—and *how we detect* them via monitoring.

What we detect We begin by presenting several common classes of Android apps faults, along with app names that contain these faults (in certain versions). Note that these faults are not particular to Android, as they affect other smartphone platforms as well.

(a) **Resource Shortage/Unavailability.** Unlike the desktop or server platforms, resource availability cannot be taken for granted. For example, smartphone multitasking is much more restrictive: when an app is not in focus it is placed on a stack and essentially put to sleep, its resources taken away and assigned to apps in focus. Apps not properly designed to work with this kind of behavior may experience failure because of resource shortage.

(b) **Unhandled Exceptions.** These failures are mostly due to poor programming practices and inadequate testing that result in failure to anticipate and handle the potential exceptions raised by the app or the system (e.g., NPR News, SoundCloud, K-9

Mail).

(c) **Crashes due to Semantic Errors.** This is a broad class of errors; for example the app fails to accepts certain types of input that otherwise should be accepted and dealt with by the program. For example, an app crashes because the input file is not in the correct format or broken, hence the app crashes instead of generating appropriate warnings (APV PDF Viewer).

(d) **Crashes due to Loss of Network Connectivity.** Most Android apps communicate with remote servers. Even the apps which do not require a network to carry on their functionality may still require network access for loading advertisements. However, Internet connectivity might be intermittent, hence apps must deal with situations where network access is temporarily unavailable (e.g., Facebook Mobile).

(e) **Permission Violations.** In Android, access to sensitive resources is protected by a set of permissions. When the app tries to access resources or functionality it does not have permission for, the OS will terminate the app.

(f) **IPC errors.** Inter Process Communication (IPC) is heavily used in Android for isolation and security. Apps must abide by the IPC communication protocol; failure to do so may lead to apps being terminated.

How we detect Currently our detection strategy relies on Android's system-wide logging facility (`logcat`). In Android, the Dalvik VM constantly monitors the app and when a fault is encountered, the VM reports the potential cause of error and the associated methods or callback into the `logcat`. To implement monitoring, we add a listener in the Dalvik VM's logging system and in the event of a fault, we isolate the exact method and activity (screen) responsible for the fault.

6.1.4 Recovery

Example: recovering from a bug in NPR News We first illustrate how our system recovers from an actual bug in the NPR News app (Figure 6.3). App execution starts from the root activity, NewsListActivity. The A^3E systematic explorer clicks a menu button to get the hourly news update, which takes the app to HourlyNewsActivity. Then A^3E plays the radio stream repeatedly. This initiates a service component, PlaybackService. At this time the program enters an illegal state and crashes; the crash is captured in the log. Analyzing the log, our system finds that the closest method associated with the crash is prepareThenPlay in the service class PlaybackService. This concludes the online fault detection phase. Next, in the recovery phase, we apply a seal-off patch to prepareThenPlay, as described next.

Constructing seal-off patches We sketch the construction of the bytecode patch (inserting code in the app via binary rewriting, achieving seal-off) in Figure 6.4. Suppose erroneousMethod is the method associated with the fault. First, we surround the original method code with a generic exception handler. Upon failure, the handler will just return (because the original method's return type is **void**), thus preventing executing the erroneous code. Custom code can be added at this point, e.g., to perform more extensive checkpointing. In general, though, the returned value will have to be of the same type as the original mehtod's return type, hence we create a return object of the appropriate type.

Next, for methods containing activities (recall that an activity roughly corresponds to a screen in Android), we apply a similar technique, inserting a try/catch block around the onCreate virtual method from the Activity class. The onCreate method is called when

```
//inside the faulty method
void erroneousMethod(T param)
{
   //surround method with try/catch block
   try
   {
      // original method code
   }
   catch(Exception e) //generic exception handler
   {
      // write custom exception handling code
      return;
   }
}

//inside the activity
@Override
protected void onCreate(Bundle savedInstanceState)
{
   //Surround with generic try-catch block
   try
   {
      // initialize activity and load GUI components
   }
   catch(Exception e)
   {
      // write custom error handling code
      //refresh the activity
      startActivity (getIntent ());
   }
}
```

Figure 6.4: Code sketch of patch construction.

the activity is loaded (i.e., the screen is displayed). If the sealed-off method execution generates further exceptions, the handler will catch the exception and refresh the activity. Thus the activity will remain operational. With the above technique, we gain two advantages: first, by sealing off just the actual problematic method we are ensuring the least amount of functionality loss; second, we are limiting the functionality seal-off only in the time of an actual fault—the rest of the time the app will behave normally. We thus implement a demand-driven approach, with self-healing taking over only when necessary, and minimizing operational limitations.

The general technique Our failure detection is dynamic hence it takes an actual execution to find and recover from a crash. When the system is used "in the wild," users interact with the app as they normally do, and if the app crashes, users will experience a small delay due to recovery. For this dissertation, however, we used an automated exploration tool we develop in prior work, A^{3}E [31] to drive the execution, so we could reliably drive the app into a state where it crashes. In the background, we constantly monitor the VM log for events that may indicate failure (Section 6.1.3).

When a failure does occur, we determine the finest granularity level for inserting our fault-avoiding code. Note that we have several options here. First, we could mark the entire current activity as the potential fault point and deny access to the activity; but this is not realistic, as an activity contains many other GUI features that may be completely unrelated to the fault observed. Second, we could limit the functionality of the associated GUI object. For example, if the crashing GUI object is a button we can disable it. But this may be also unrealistic. For example, for some inputs the button code may fail, but it will work on other inputs. Third, we can operate (seal off) at the method level. Therefore, the method is the finest seal-off granularity; we employ this granularity level in our approach by assigning the fault to the crashing method. For example, in Figure 6.3 we will seal-off the third method, prepareThenPlay (using the patching procedure explained above) because it is on the lowest level of the exception trace.

Once a crash point is reached, we rollback and resume the app. The rollback point depends on whether the crash is transient or persistent.

For *transient errors* (generated in response to external events such as illegal sensor data, unexpected shared memory deletion by the Android OS, background services shutdown due to low energy, network unavailability, resource shortage, etc.) the rollback point is the point of the crash. The idea is that after rollback and restart these transient environmental exceptions may not be raised and the app can resume its functionality normally.

For *persistent errors* (e.g., unhandled exceptions, semantic errors, IPC communication errors, unauthorized access), we rollback to an earlier point (previous node in the SATG) and use bytecode rewriting to seal off the faulty method in the faulty SATG node. While this limits functionality, it ensures that the app will not call the offending method again.

6.2 Implementation

Platform We implemented our approach and conducted experiments on a Motorola Droid Bionic phone, running Android 2.3.4 (note, however, that the test results can also be achieved by running the app in the emulator).

Tools For model construction we used the SATG extraction component (static analysis-based) of A^3E. To drive exploration, we used the systematic exploration component of A^3E. Bytecode rewriting was done using the smali Dalvik assembler/disassembler [11]. We wrote the main instrumentation code in Java.

In our current setup the phone was "tethered" to a laptop; this was necessary for running A^3E, smali, and initiating rollback/restart. However, we expect that in the future the approach will run solely on the phone, as we envision it should run "in the wild," with no tethering required.

App	Version	Bug Type	Size	
			Kinst.	KBytes
Facebook Mobile	1.6.0	Network Unavailability	173	3,000
NPR News	2.1b	Semantic Error	21	70
K-9 Mail	4.0.0.3	Unhandled Exception	157	2,300
SoundCloud	1.2.2	Unhandled Exception	48	250
APV PDF Viewer	0.2.7	Semantic Error	3	1,100

Table 6.1: Examined apps.

6.3 Evaluation

Examined apps. For evaluation we chose several sizable, popular Android apps that contained known bugs. In Table 6.1 we present the apps: version, type of bug, and app size. According to Google Play, each app was highly popular, with more than 1 million downloads. We have evaluated our approach in terms of *effectiveness*, i.e., can the system recover from actual bugs in popular sizable apps? and *efficiency*, i.e., is the overhead of our approach acceptable?

Effectiveness Our approach was effective at performing self-healing in response to three categories of bugs encountered in five popular apps.

Efficiency Our approach incurs a one-time overhead for model extraction, via static analysis, to enable rollback point detection. The second column of Table 6.2 shows the static analysis time for each app. Model extraction time is solely depending on the app's binary size and code complexity, and as it is a one-time cost incurred *before running the app*, we believe that the 34–94 seconds figure is acceptable.

We drove the apps to crash points via systematic exploration. Depending on the bug, exploration time will vary, though techniques such as targeted exploration [31] or fast-forwarding record-and-replay [79] can significantly accelerate the procedure. The time-to-crash is presented in the third column of Table 6.2. For example, for Facebook Mobile, the actual fault was in the initial login screen, hence the systematic exploration time (4 seconds) was much lower than for the other apps.

As mentioned in Section 6.1.4, self-healing might require bytecode rewriting (if seal-off is involved) and always requires rollback and restart. The bytecode rewriting time (performed only once after the crash, for non-transient bugs) depends on the size of the app. This time is shown in the fourth column of Table 6.2: 13–44 seconds. Facebook Mobile required no rewriting because it experiences a transient bug, hence the '0' figure for rewriting time. Finally, the time required for rollback and restart is shown in the last column of Table 6.2. The rollback time involves uninstalling the current version, installing the modified app, and rolling back to the nearest safe point within the app. While just rolling back requires very little time (in our case not more than 1 second), uninstalling the current faulty app and reinstalling the modified app takes longer, 3–8 seconds. However this is much shorter than any manual rollback and restart because not only a human would require longer time to uninstall and reinstall but also a human would restart the app from the home screen and therefore would take longer to reach the former point (the point where the app was before the crash). As we rollback to the nearest safe point, we can ensure faster exploration to the safe state. As shown in the last column of Table 6.2, our automated rollback required at most 9 seconds for the apps. Hence the total self-healing time is 9–50 seconds, which we believe is acceptable.

App	**Static analysis** (seconds)	**Systematic exploration (time-to-crash)** (seconds)	**Self-healing**	
			Bytecode rewriting (seconds)	**Rollback and restart** (seconds)
Facebook Mobile	86	4	0	9
NPR News	53	22	22	9
K-9 Mail	94	52	44	6
SoundCloud	51	16	17	4
APV PDF Viewer	34	7	13	4

Table 6.2: Efficiency measurements results.

For transient faults, recovery is faster because we do not rewrite the app: a simple rollback is usually enough to resume normal behavior. For example, our examined version of Facebook Mobile failed when there was no network connectivity. A rollback restored the app and as the connectivity was reestablished, the app resumed its normal operation. Hence recovery was faster than for the other apps, as no bytecode rewriting was performed. Note that app performance is not affected by seal-off, since only a specific part of a method's bytecode (i.e., the prologue) is rewritten.

6.3.1 Limitations

Our prototype is subject to several limitations that we intend to address in future work.

First, mobile apps tend to be GUI-centric, so upon rollback and restart we only lose GUI state such as previously-entered data, or selected items. For more stateful scenarios, we will have to perform more sophisticated healing operations (e.g., more sophisticated fault detection analyses and more extensive checkpoint and rollback).

Second, our approach is *reactive* (responds to bugs after they manifest), rather than *proactive*. We expect that, using techniques such as consistency constraints or invariant checking, we can detect and fix errors before they develop into full-fledged crashes.

Third, in our current implementation, the phone was tethered to a laptop. There is however no fundamental hurdle to running the approach entirely on the phone. We used tethering for systematic exploration (which will not be necessary when apps crash "in the wild"); and to benefit from existing app rewriting support offered by desktop tools.

6.4 Conclusions

We have presented an approach that uses automatic error detection and patch construction towards providing a certain degree of self-healing capabilities to Android apps. We use dynamic analysis to identify crash points, static analysis to identify rollback points, and binary rewriting to seal off methods associated with crash points so that apps can continue to function even after a crash, albeit with limited functionality. Through experiments on actual bugs in several popular apps, we show that our approach is effective and reasonably efficient.

Chapter 7

Related work

Over the years an extensive amount of research has been conducted focusing on fault localization and program recovery on the runtime. We compare our approaches with other research works regarding the differences we have found in the design, implementation, and application.

7.1 Program Exploration

The work of Rastogi et. al. [93] 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 [25], 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 [130, 129, 84] 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 2.5.3) 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 [30] is a GUI testing framework for Java and Windows applications based on EIG. Android GUITAR [99] 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. [128] 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. [21] 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. [72] 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 [24] 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 [23] 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 [59] 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 [73] 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 [103] 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 [20] 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* [34] can record and run the tests on multiple devices. *Ranorex* [92] is a test automation framework. *Eggplant* [106] facilitates writing automated scripts for testing Android apps. *Framework for Automated Software Testing (FAST)* [115] can automate the testing process of Android apps over multiple devices.

Finally, there exist a variety of static [110, 61, 97, 60] and dynamic [62, 49] 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.

7.2 Dynamic Program Slicing

Zhou et. al. [136] and Zeng et al. [132] mentioned the use of program slicing in their analysis of Android applications. Although they used traditional bytecode slicing to achieve entirely different goals. Zhou et. al. used slicing for mining sensitive credentials inside the application, and Zeng et. al. used slicing to generate low level **C** equivalent code. In their paper, they created slices at the bytecode level and only considered slicing to reveal data dependence at the byte-code level, which is imprecise as it does not count for the features Android represents such as inputs from the callbacks, and sensors.

Although we find several works of program slicing on the Java bytecode level targeted for single entry sequential Java programs [120, 116, 7, 102] none of those addresses event-based models that we find in Android programs.

More recently Wang et.al. [117, 119] showed several applications of program slicing for native programs such as delta debugging, deterministic replay, relevant input analysis.

7.3 GUI Event Generation

Automated input generation for Android platforms through static and dynamic analysis are getting focus in recent years. Machiry et. al. presented a system called Dynodroid [28] where they examine the app on top of a VM. They also instrument the whole Android framework to monitor the apps. But their approach differs from ours as we can directly drive the execution to a certain part of the program by extracting inputs only applicable to that particular run.

Linares-Vasquez et. al. [83] showed another approach to generating actions for GUI-based executions. They record app runs and from the traces, they mine GUI-object input sequences to produce the inputs. While this method is more precise due to the involvement of Dynamic analysis, they require actual applications runs and cannot exploit the static analysis to prune paths that are not useful.

7.4 User Defined Deep Linking

We compare uLink with other related approaches and explain why they are not sufficient to achieve our goals. Table 7.1 summarizes the discussion below.

Mobile deep links. As mentioned in 5, existing mobile deep links require nontrivial developer effort, have poor coverage, and are statically defined; therefore they do not satisfy many of our goals. We now elaborate on these limitations.

Mobile deep links require *nontrivial developer effort*. As an example, the open source Wikipedia app for Android has one deep link and it requires 23 LoC to handle the associated intent. As a consequence, a small number of mobile apps, even among the top ones, expose deep links. An estimate by URX from 2014 says that 19% of the top 100 Android apps expose deep links (and only 11% have deep links for Android, iOS and iPad [108]). To confirm, we analyzed 13,848 Android apps downloaded in the month of May 2015 and covering all app categories.[1]

Existing mobile deep links have *poor coverage*—a small number of locations, predefined by developers, within an app are directly accessible via deep links.This is due to two key reasons. Developer's effort to support deep linking increases almost linearly with the number of unique deep links, and hence apps tend to expose very few deep links. It is unlikely that developers will open up all possible deep links in the app. Another more fundamental reason is that while deep links are stateless, mobile apps are stateful—an app's current view may depend on data from a previous view (e.g., location selected in a previous page) or as a result of specific user interactions (e.g., doing a search, selecting an item) on the current page. Thus, reaching the view may not be possible without transferring states from previous views or without applying the UI interactions. The stateful nature of apps also implies that even if deep links are free and a developer includes deep links to each and every page in an app and with a large number of supported input parameters, existing deep links, due to their statelessness, would not be able to capture and preserve user data generated in an app during interaction.

[1]We counted deep links by looking in app manifest files for declarations of intent filters complying with the Android specifications for deep links [22]. Our counts can be over-estimates, since we did not verify if the app actually supports the deep links.

System	**Dynamic links**	**Convenience in user exper**	**Coverage**	**Dev effort**	**Ease of deployment**
URLs	Yes	High	Good	None	Yes
Web Macros	Yes	Low	High	None	Yes
Deep links	No	-	Low	Some	Yes
Record& Replay	Yes	Low	High	No	No
uLink	**Yes**	**High**	**Good**	**Little**	**Yes**

Table 7.1: uLink goals and comparison with the state-of-the-art.

Record and replay. Record and replay techniques can record macros that can later be replayed to navigate to an arbitrary location of an application for desktop, server [48, 126, 100, 63, 88, 86], web [26, 94, 69, 81, 82], and mobile platforms [57, 68]. One might consider using such macros as links. A full blown record and replay mechanism can have very good coverage, but it is not suitable to be augmented to support ulinks for several reasons. First, these systems are too heavyweight to be used in the wild. Recording and replaying all sources of nondeterminism has prohibitive costs on mobile devices [54] (e.g., special hardware support or virtual machine instrumentation). There are tools that successfully record and replay smartphone apps by relying only on the sensor and UI event streams [57, 68], but they are still heavyweight and they require either a rooted phone or changes to the mobile OS, thus limiting their applicability to consumers' phones, at scale. Second, record and replay can be slow, especially when the user uses the app for a long period of time before arriving to the current view, and hence the replay phase needs to replay many user interactions. Finally, existing record and replay tools require the user to explicitly specify when a recording starts and ends—an unacceptable user experience when a user dynamically wants to generate a link to the current app page (e.g., for bookmarking it).

Commercial tools. Several startups today offering mobile deep linking. AppLink [2], mobile.deep.linking [10], and deeplink.me [3] allow app developers to define deep links to specific pages in an app, such as the homepage, product pages, and shopping cart, exactly like with a web site. When the app link is clicked, if the app is not installed, the user is directed to the app store or to the equivalent web page. In addition to enabling app deep linking, URX [13] also crawls web pages and constructs deep linking metadata to be leveraged by app developers. For example, if the site contains any information or news about a particular music, it can readily generate a deep link to a supported music player to play that song. Although these approaches all contribute to building effective deep linking inside app content, all of them require significant developer effort, they are statically defined, and have the same limitations of deep links (low coverage and no support for stateful and UI-driven views).

7.5 Recovery from Faults

Self-healing and automated patch construction have been studied in many contexts, from clusters of Internet servers [101, 37] to web browsers [89]. Demsky et al. [44] use formal specifications for data structures that allow integrity properties in data structures to be monitored and data structures to be repaired in case the specification is violated. Perkins et al. [89] introduced a system named ClearView that monitors an application's execution to learn application invariants, detect failures, and in case of failure automatically constructs and applies a patch to heal the application. ClearView has been applied to Firefox with a high degree of success and resilience to attacks. Sidiroglou et al. [96] developed an approach named ASSURE that employs rescue points to recover from unanticipated failures in desktop/server Linux applications. ASSURE utilizes online code injection and restores program execution to a rescue point where existing error handling mechanism is used to inject fault recovery code. Candea et al. [38] have proposed "microreboots" (rebooting small components instead of entire applications) as a recovery technique for Internet services. Sultan et al. [101] and Bohra et al. [37] use remote DMA to perform peer monitoring and take-over in a cluster. to provide seamless service to clients. However, to the best of our knowledge, we are the first to study self-healing on the smartphone platform.

Wei et al. [123] proposed an automated patch generation technique based on contracts. Their approach is limited to systems built using the design-by-contract pattern. Although their strategy has shown promising results, smartphone apps are not developed using design-by-contract.

Weimer et al. [124] demonstrated a fully automated, genetic programming approach for finding and fixing bugs. Their tool, *GenProg*, identifies legal program variants for positive test cases and they generate fixes with the means of structural differences and delta debugging upon the correct program variant for the faulty input. Michail et al. [85] proposed a scheme to use user-generated bug reports to predict future bugs in a software execution path to warn the users to avoid that path. Their scheme is based on predicting the presence of faults in a particular execution based on previous reports from the users. The work of Kim et al. [75] generates automatic patches from already existing patches written by human developers. They manually inspected the human written patches and automatically develop the repair code by identifying common fix patterns. Their approach requires manual effort and might not be always practical to employ in a quick succession which is required in mobile platforms with shorter update cycle. In contrast to these three efforts, our approach does not rely on test cases or bug reports, but rather reacts dynamically to a set of predefined errors.

Dallmeier et al.'s approach [43] automatically extracts anomalies in object behavior and generate patches accordingly. This idea may be useful on smartphone apps, but it does not guarantee sealing-off faulty code in a deterministic manner, e.g., random events can occur in a particular executions and the same set of faults can manifest differently.

Carzaniga et al. [39] employ code rewriting to work around API-related faults in web applications. They have showed their approach in popular web APIs such as Google maps and YouTube. While the are similarities (e.g., event-driven) between smartphone and web apps, there are significant differences: smartphone apps are centered around rich gestures and sensors, so it is unclear how their approach would translate to smartphones.

Chapter 8

Future Work

We have shown that there is a practical need for conducting static and dynamic analysis for fault localization and application recovery. We also demonstrated some applications that can directly benefit from our work. Nevertheless, a wide range of possibilities exists to continue further the research works presented in this dissertations. In this chapter, we outline some future directions.

8.1 Improvements

Dynamic slicing Several opportunities exist to improve further the slicing mechanism presented in this dissertation. For example, analyzing the Android Framework will yield more precise slicing. Furthermore, generating slices for particular slicing criteria might not reveal an exact set of inputs, since Android is heavily gesture-centric. Different types of gestures are handled in a variety of ways and precise slicing cannot be carried out possibly without discovering their patterns.

App self-healing Application recovery strategies can be informed by mining bug repositories. Learning the most common error pattern will aid the researchers to design more efficient test cases and error routines. Such tactics, once applied can render self-healing more effective.

8.2 Program Analysis Applications

Application profiling Automatic exploration techniques presented in this dissertation can be further extended by adding functionality to measure various application, system, and user specific information. For instance, monitoring information leakage can be useful to identify app vulnerabilities. Automatic exploration can also be an excellent tool for test automation.

Undo computing Undo computing [76] has been proposed to recover from such attacks and state corruption bugs by identifying both the benign user actions and the malicious/buggy actions, and then restoring the system to a “clean” state where only the legal changes are kept, and the malicious/buggy actions are undone. So far undo computing has only been studied on server systems; however, those techniques cannot be easily applied to Android due to differences in platform, app construction, and overhead performance tolerance. Android application slicing techniques can be proven fruitful in this regard.

Relevant input analysis Understanding the role that input values play during execution is critical for understanding program behavior, including for finding and reproducing vulnerabilities. For example, Samsung's security patch list [95] includes vulnerabilities triggered by certain GIF, bitmap, JPEG, or file-path inputs. Furthermore, on mobile platforms, relating app inputs to app behavior is particularly complicated, because mobile apps are event-oriented, that is, they revolve around processing event streams. Dynamic slicing strategies can aid to better understanding of the input patterns.

Improving dynamic taint analysis Dynamic taint analysis has managed to find leaks in desktop, server, and Android applications, but it is prone to several issues: it can be rendered ineffective, e.g., via control dependence attacks; it can be imprecise, a condition know as "taint explosion"; and it might not be very effective in helping developers/users fix a leak, once a leak has been found. These shortcomings can be addressed using our results in control influence and value propagation chains [118].

Proactive error checking In Chapter 6 we mainly discussed reactive approaches to recovery. However, proactive measures can also be taken into consideration. This may require both static and dynamic methods to learn application behavior hence identify the most common patterns of faults prior to app runs (or at least prior to crashes).

Chapter 9

Conclusions

Smartphone apps are getting into people's lives in every possible way. For this reason, any crash or security vulnerability in apps has a large effect. We believe that traditional (or manual) techniques for application analysis and testing for faults, bugs, errors, and leaks inside an app are wither insufficient or unreliable. Other specific challenges imposed by the smartphone platform are technical difficulties such as unique sets of inputs, dependency on sensors, distinct IPC mechanism, shorter update cycle, etc. For all of these reasons, we argue that application analysis approaches need to be automated and must employ software engineering tactics such as static and dynamic analysis.

In this context, the thesis makes the following contributions:

- We present individual strategies that aid to the discovery of unseen faults by generating high coverage test cases, which cannot be obtained by manual means. As apps get updated frequently, developers need to rely on the manual techniques that render poor coverage. Our novel breadth-fast and targeted exploration strategy not only yields a higher percentage of coverage but also explores them efficiently by exploiting the constructed app model.

- Upon the discovery of faults, we move further by implementing a toolset (AndroidArrow), that produces UI events to trigger target methods. AndroidArrow attempts to trigger the target by finding the optimum path transition and generates the necessary event sequences along with their associated GUI objects.
- uLink, a user-defined deep-linking framework was designed to recreate stateful application links. uLink can reinstate the full graphical state by directly rendering the corresponding app page and performing the exact sequence of actions.
- AndroidSlicer, an approach to implementing program slicing on Android. Our slicing algorithm adapts distinct Android characteristics such as event-based callbacks, gesture-based inputs, and IPC message passing mechanism through intent objects.
- Finally, we have shown a conceptual model of self-healing Android apps. We generate recovery routines to throttle app failure on the fly to ensure the app does not get interrupted on the run.

We expect this thesis will provide a solid foundation for future research. We also believe it will continue to aid researchers and developers with their smartphone application analysis tasks.

Bibliography

[1] Apache lucene. `http://lucene.apache.org/`.

[2] Applinks. `http://www.applinks.org`.

[3] Deeplink. `https://www.deeplink.me`.

[4] Flurry. `http://www.flurry.com`.

[5] Ifttt. `https://ifttt.com/recipes`.

[6] JD-GUI. `http://jd.benow.ca/`.

[7] jSlice. `http://jslice.sourceforge.net/`.

[8] Localytics. `http://www.localytics.com/`.

[9] Mobile deep linking. `https://en.wikipedia.org/wiki/Mobile_deep_linking`.

[10] mobile.deep.linking. `http://www.mobiledeeplinking.org`.

[11] Smali: An assembler/disassembler for Android's dex format. `http://code.google.com/p/smali/`.

[12] susi. `http://sseblog.ec-spride.de/tools/susi/`.

[13] URX. `http://www.urx.com`.

[14] 6.1B Smartphone Users Globally By 2020, Overtaking Basic Fixed Phone Subscriptions, June 2015. `http://techcrunch.com/2015/06/02/6-1b-smartphone-users-globally-by-2020-overtaking-basic-fixed-phone-subscriptions/`

[15] dex2jar. Retrieved on 08/09/2016.

[16] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In FSE '13, 2013.

[17] Eytan Adar, Jaime Teevan, and Susan T. Dumais. Large scale analysis of web revisitation patterns. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1197–1206. ACM, 2008.

[18] Eytan Adar, Jaime Teevan, Susan T. Dumais, and Jonathan L. Elsas. The web changes everything: Understanding the dynamics of web content. In *Proc. of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 282–291. ACM, 2009.

[19] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[20] Domenico Amalfitano, Anna Rita Fasolino, Salvatore De Carmine, Atif Memon, and Porfirio Tramontana. Using gui ripping for automated testing of android applications. In *ASE'12*, pages 258–261.

[21] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *FSE '12*, pages 1–11.

[22] Android Developers. Enabling Deep Links for App Content. `http://developer.android.com/training/app-indexing/deep-linking.html`.

[23] Android Developers. monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

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

[25] Android Developers. Android Emulator Limitations, March 2013. `http://developer.android.com/reference/android/content/Intent.html`.

[26] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. Automating Web Navigation with the WebVCR. In *Proc. of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pages 503–517, 2000.

[27] Appsee. `https://www.appsee.com/`.

[28] Aravind MacHiry, and Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *FSE '13*, 2013.

[29] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[30] Atif Memon. GUITAR, August 2012. `guitar.sourceforge.net/`.

[31] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[32] 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*.

[33] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, 2014.

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

[35] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 19–25, 2015.

[36] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47–56, New York, NY, USA, 2011. ACM.

[37] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *ICAC'04*, pages 256–263, 2004.

[38] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A technique for cheap recovery. In *OSDI'04*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[39] Antonio Carzaniga, Alessandra Gorla, and Mauro Perino, Nicolòand Pezzè. Automatic workarounds for web applications. In *FSE '10*, FSE '10, pages 237–246, 2010.

[40] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

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

[42] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, pages 684–702, 2009.

[43] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *ASE '09*, ASE '09, pages 550–554, 2009.

[44] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *ICSE'05*, pages 176–185, 2005.

[45] David Devecsery, MIchael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[46] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. MobiSys '11, pages 335–348, 2011.

[47] Susan Dumais, Edward Cutrell, JJ Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. Stuff i've seen: A system for personal information retrieval and re-use. In *Proc. of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 72–79. ACM, 2003.

[48] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, volume 36, pages 211–224, 2002.

[49] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–407. USENIX Association, October 2010.

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

[51] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[52] Min Feng and R. Gupta. Detecting virus mutations via dynamic matching. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 105–114, Sept 2009.

[53] Earlence Fernandes, Oriana Riva, and Suman Nath. My OS ought to know me better: In-app behavioural analytics as an OS service. In *Proc. of HotOS XV*, 2015.

[54] Jason Flinn and Z. Morley Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proc. of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 84–89. ACM, 2011.

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

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

[57] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81. IEEE Press, 2013.

[58] Google. Now on Tap. `https://support.google.com/websearch/answer/6304517?hl=en`.

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

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

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

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

[63] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An Application-level Kernel for Record and Replay. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208. USENIX Association, 2008.

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

[65] S. Hao, Ding Li, W.G.J. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 1–7, 2012.

[66] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Proc. of MobiSys*, pages 204–217. ACM, June 2014.

[67] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *AST '11*, pages 77–83, 2011.

[68] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366. ACM, 2015.

[69] Darris Hupp and Robert C. Miller. Smart bookmarks: Automatic retroactive macro recording on the web. In *Proc. of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 81–90, New York, NY, USA, 2007. ACM.

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

[71] IDC. Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, Novemeber 2013. `http://www.idc.com/getdoc.jsp?containerId=prUS24442013`.

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

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

[74] Jinseong Jeon and Kristopher Micinski and Jeffrey S. Foster. Redexer, September 2013. `http://www.cs.umd.edu/projects/PL/redexer/index.html`.

[75] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, ICSE '13, pages 802–811, 2013.

[76] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *9th USENIXSymposium on Operating Systems Design and Implementation, OSDI2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 89–104, 2010.

[77] Bogden Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[78] A. Kumar Maji, Kangli Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *ISSRE'10*, pages 249–258, 2010.

[79] L. Gomez, I. Neamtiu, T.Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE '13*, 2013.

[80] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. Dejaview: A personal virtual computer recorder. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 279–292, 2007.

[81] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[82] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 723–732, 2010.

[83] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122, 2015.

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

[85] Amir Michail and Tao Xie. Helping users avoid bugs in gui applications. In *ICSE '05*, ICSE '05, pages 107–116, 2005.

[86] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*, pages 284–295.

[87] Suman Nath, Felix Xiaozhu Lin, Lenin Ravindranath, and Jitendra Padhye. SmartAds: bringing contextual ads to mobile apps. In *Proc. of MobiSys*, pages 111–124, 2013.

[88] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *SIGPLAN Not.*, 43(3):308–318, March 2008.

[89] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP '09*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[90] Pew Research Center. Report: Mobile Tablet Ownership 2013. `http://pewinternet.org/Reports/2013/Tablet-Ownership-2013.aspx`.

[91] Pew Research Center. Report: Smartphone Ownership 2013. `http://pewinternet.org/Reports/2013/Smartphone-Ownership-2013.aspx`.

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

[93] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODASPY*, pages 209–220, 2013.

[94] Alex Safonov, Joseph A. Konstan, and John V. Carlis. End-user web automation: Challenges, experiences, recommendations. In *Proc. of WebNet 2001 - World Conference on the WWW and Internet, Orlando, Florida, October 23-27, 2001*, pages 1077–1085, 2001.

[95] Samsung Mobile Security Blog. Samsung Android Security Updates, Nov 2015. `security.samsungmobile.com/smrupdate.html`.

[96] Stelios Sidiroglou, Oren Laadan, Carlos R. Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS'09*.

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

[98] Soot. Soot: A Framework for Analyzing and transforming Java and Android applications. `http://sable.github.io/soot/`.

[99] SourceForge. Android GUITAR, August 2012. `http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR`.

[100] Sudarshan M. Srinivasan, Srikanth Kandula, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[101] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering internet service sessions from operating system failures. *Internet Computing, IEEE*, 9(2):17–27, 2005.

[102] Attila Szegedi and Tibor Gyimothy. Dynamic slicing of java bytecode programs. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 35–44, 2005.

[103] 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.

[104] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 207–218, New York, NY, USA, 2007. ACM.

[105] Jaime Teevan, Edward Cutrell, Danyel Fisher, Steven M. Drucker, Gonzalo Ramos, Paul André, and Chang Hu. Visual snippets: Summarizing web pages for search and revisitation. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 2023–2032. ACM, 2009.

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

[107] Tim Strazzere. The New NotCompatible: Sophisticated and evasive threat harbors the potential to compromise enterprise networks, November 2014. `https://blog.lookout.com/blog/2014/11/19/notcompatible/`.

[108] URX Blog. How Many of the Top 200 Mobile Apps Use Deeplinks? `http://blog.urx.com/urx-blog/how-many-of-the-top-200-mobile-apps-use-deeplinks`.

[109] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[110] Various. SCanDroid, January 2013. `https://github.com/scandroid/scandroid`.

[111] VB. Microsoft beats Google to the punch: Bing for Android update does what Now on Tap will do. `http://venturebeat.com/2015/08/20/microsoft-beats-google-to-the-punch-bing-for-android-update-does-what-now-on-tap-w` 2015.

[112] VentureBeat. Imagine a web without URLs. That's what the mobile app world looks like now. `http://venturebeat.com/2014/07/08/imagine-a-web-without-urls-thats-what-the-mobile-app-world-looks-like-now/`.

[113] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

[114] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '14, pages 39–50, New York, NY, USA, 2014. ACM.

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

[116] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, pages 10:1–10:49, 2008.

[117] Yan Wang, R. Gupta, and I. Neamtiu. Relevant inputs analysis and its applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 268–277, Nov 2013.

[118] Yan Wang, Iulian Neamtiu, and Rajiv Gupta. Generating sound and effective memory debuggers. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 51–62, New York, NY, USA, 2013. ACM.

[119] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[120] Wang, T. and Roychoudhury, A. Using compressed bytecode traces for slicing java programs. 2004.

[121] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: multi-layer profiling of android applications. Mobicom '12, pages 137–148, 2012.

[122] Xuetao Wei, Iulian Neamtiu, and Michalis Faloutsos. Whom does your android app talk to? In *GLOBECOM'15*.

[123] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA '10*, ISSTA '10, pages 61–72, 2010.

[124] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, ICSE '09, pages 364–374, 2009.

[125] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *ISSTA'07*, ISSTA '07, pages 185–195, New York, NY, USA, 2007. ACM.

[126] Min Xu, Rastislav Bodik, and Mark D Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135.

[127] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[128] 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.

[129] Xun Yuan and Atif M. Memon. Using GUIrun-time state as feedback to generate test cases. In *ICSE '07*, pages 396–405, 2007.

[130] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using gui run-time state feedback. *IEEE Transactions on Software Engineering*, pages 81–95, 2010.

[131] I. Neamtiu Z. Shan, T. Azim. Finding resume and restart errors in android applications. In *In Proc. of OOPSLA '16*.

[132] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 487–498, 2013.

[133] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *ICSE'03*, pages 319–329, May 2003.

[134] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 81–91, New York, NY, USA, 2006. ACM.

[135] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.

[136] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 23:1–23:12, 2015.