Goal-driven Exploration for Android Applications

by

Duling Lai

B.A.Sc, Simon Fraser University, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

July 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, a thesis entitled:

GOAL-DRIVEN EXPLORATION FOR ANDROID APPLICATIONS

submitted by DULING LAI in partial fulfillment of the requirements for

the degree of MASTER OF APPLIED SCIENCE

in ELECTRICAL AND COMPUTER ENGINEERING

Examining Committee:

JULIA RUBIN, ELECTRICAL AND COMPUTER ENGINEERING
Supervisor

KARTHIK PATTABIRAMAN, ELECTRICAL AND COMPUTER ENGINEERING
Supervisory Committee Member

PHILIPPE KRUCHTEN, ELECTRICAL AND COMPUTER ENGINEERING
Supervisory Committee Member

Abstract

This thesis proposes a solution for automated goal-driven exploration of Android applications – a scenario in which a user, e.g., security auditor, needs to dynamically trigger the functionality of interest in an application, e.g., to check whether user-sensitive info is only sent to recognized third-party servers. As the auditor might need to check hundreds or even thousands of apps, manually exploring each app to trigger the desired behavior is too time-consuming to be feasible. Existing automated application exploration and testing techniques are of limited help in this scenario as well, as their goal is mostly to identify faults by systematically exploring different app paths, rather than swiftly navigating to the target functionality.

The goal-driven application exploration approach proposed in this thesis, called GOALEXPLORER, automatically generates an executable test script that directly triggers the functionality of interest. The core idea behind GOALEXPLORER is to first statically model the application UI screens and transitions between these screens, producing a Screen Transition Graph (STG). Then, GOALEXPLORER uses the STG to guide the dynamic exploration of the application to the particular target of interest: an Android activity, API call, or a program statement. The results of our empirical evaluation on 93 benchmark applications and 95 most popular GooglePlay applications show that the STG is substantially more accurate than other Android UI models and that GOALEXPLORER is able to trigger a target functionality much faster than existing application exploration.

Lay Summary

Goal-driven exploration of mobile applications is a scenario in which a user needs to promptly trigger a specific functionality of an application, to explore its behavior. Goal-driven exploration is useful in a variety of tasks, e.g. when a security auditor needs to inspect an application for compliance with security protocols. Existing application testing techniques are not suitable for goal-driven exploration because they cannot promptly navigate to the functionality of interest.

In this thesis, we present our approach to goal-driven exploration, called GOAL-EXPLORER. Our main insight is to first analyze the application without running it and construct a model of its user interface. Then, we run the application and use the constructed model for identifying a set of user gestures that navigate the execution towards the desired, user-selected functionality.

The evaluation result shows that GOALEXPLORER performs substantially better than other existing approaches, providing an efficient and effective solution to goal-driven exploration.

Preface

The research project included in this thesis is original intellectual product of its author, Duling Lai, done under the supervision of Prof. Julia Rubin (supervisor).

Table of Contents

List of Tables

List of Figures

List of Acronyms

ADB Android Debug Bridge. 29

API Application Programming Interface. 2, 9, 13

APK Android Package. 19

ATG Activity Transition Graph. 12–14, 20

FN False Negative. 43

FP False Positive. 43

ICC Inter-Component Communication. 27

IETF Internet Engineering Task Force. 5

MAC Message Authentication Code. 5

NLP Natural Language Processing. 50, 51

OAuth Open Authorization. 5

RQ Research Question. 32

SDK Software Development Kit. 2, 13

STG Screen Transition Graph. 3, 4, 22, 23, 28, 29, 48

UI User Interface. 4, 5, 7, 9–15, 17, 18, 29, 32, 45

WTG Window Transition Graph. 12, 13, 20

Acknowledgments

Throughout years of my graduate study, I have received a great deal of support and assistance from my supervisor, Dr. Julia Rubin, whose expertise and patient guidance was invaluable in formulating the thesis topic and methodology in particular. This thesis would not have been completed without her continuous support. She introduced me to the research world, yet had more profound influence on my life beyond research.

I also would like to thank my examination committees, Dr. Philippe Kruchten and Dr. Karthik Pattabiraman, for reviewing this thesis and providing constructive comments.

I offer my enduring gratitude to my colleagues and supportive friends at the UBC, who have inspired and supported me to work in this field. I would like to thank Junbin Zhang, in particular, for his insightful comments and generous help on the evaluation of the research project. I owe my gratitudes to Khaled Ahmed, for the inspiring discussions that help devise the methodology. I also want to thank Yingying Wang for help me formulate this thesis and provide me canny advice on my study.

Special thanks are owed to my beloved parents, who have supported me throughout my years of education, both morally and financially. Their love and encouragement made me never feel alone in a country far from home.

Dedication

To my parents

Chapter 1

Introduction

1.1 Problem Statement

Mobile applications (apps) have grown from a niche field to the forefront of modern technology. Mobile phone users have now access to more than 5.5 million apps in the major online stores [64] and at least as many more in alternative stores [12]. As our daily life becomes more dependent on mobile apps, various stakeholders, including app developers, store owners, and security auditors, require efficient techniques for validating the correctness, performance, and security of the apps.

Dynamic execution is a popular technique used for auditing [34, 37, 66, 72] and validating [18, 45, 60, 33] mobile apps. For example, a security auditor often monitors the network traffic generated by an app to ensure that user-sensitive info is sent encrypted over the network [45, 60] and that it is sent to recognized third-party servers only [18, 33]. It is common for the auditor to know which part of the app is responsible for the functionality of interest, but dynamically triggering that functionality is still a challenging task.

Consider, for example, a popular mobile personalization app, Zedge [74], which provides access to a large collection of wallpapers, ringtones, etc. and has more than a hundred million installs on Google Play. A security auditor exploring this app can quickly determine that it uses the Facebook SDK [16] and allows the users

to log in with their Facebook accounts. Such determination can be done by simply browsing the app code: the Facebook login is triggered by a particular API from the Facebook SDK [15]. Now, the auditor wishes to check what information is transmitted during the login process and what information is granted by Facebook when the user logs into their account [59].

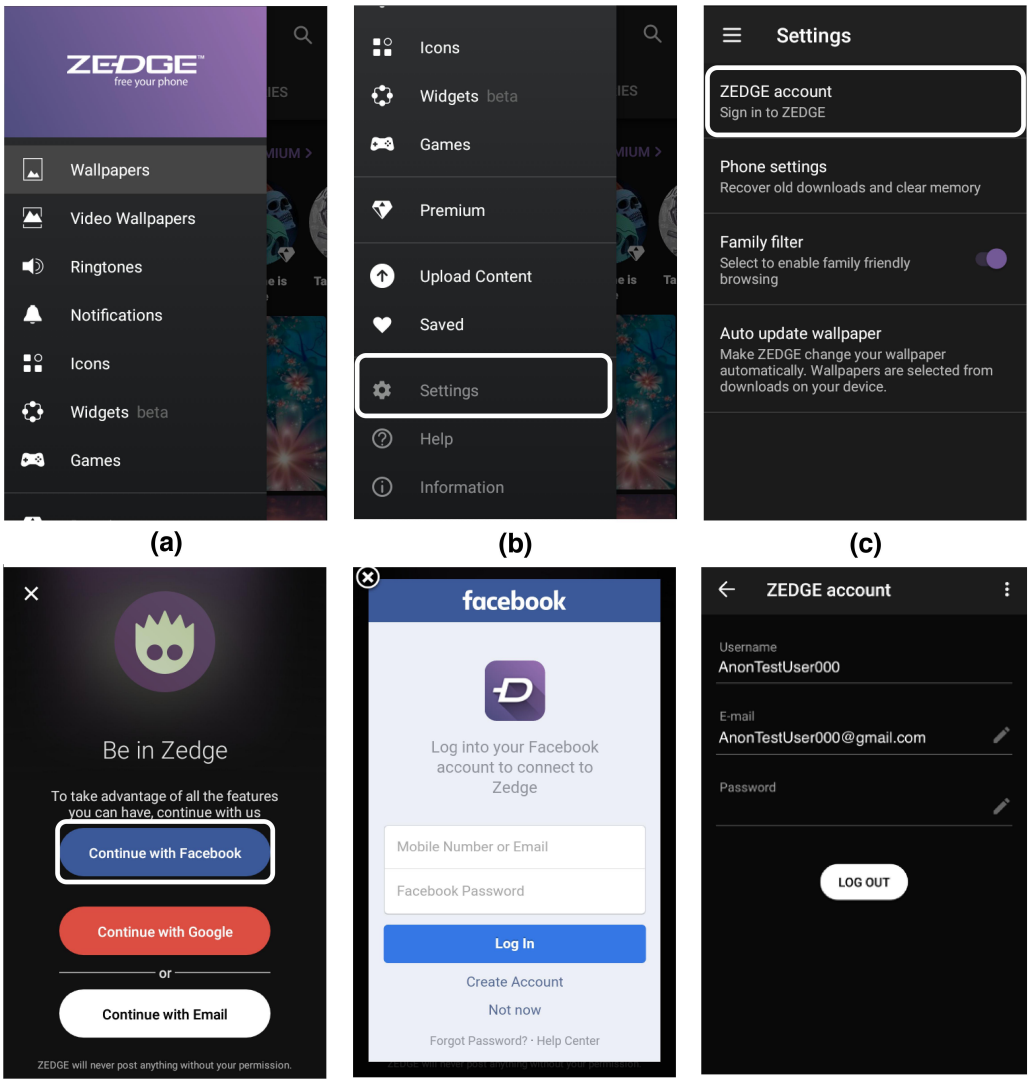


Figure 1.1: Example App - ZEDGE.

To perform this investigation, the auditor needs to generate a set of user gestures that navigate the app to the Facebook login screen. For Zedge, that entails a nontrivial sequence of steps shown in Figure 1.1: first, scroll all the way down in the menu of all possible operations in Figures. 1.1(a-b) to the 11th menu item, then click the “Settings” button in Figure 1.1(b) to open the “Settings" view, then click “ZEDGE account” in Figure 1.1(c), and only then click on “Continue with Facebook" in Figure 1.1(d).

Such a manual exploration, especially when one needs to analyze hundreds or thousands of different apps, is time-consuming. Existing testing frameworks, such as Espresso [24], UIAutomator [28], and Robotium [61], are not helpful in this scenario as they are designed to provide the user a way to *manually* script test cases and later repeatedly execute these test cases.

Automated app exploration techniques that exercise the app by generating user gestures and system events [1, 5, 10, 32, 35, 46, 47, 48, 54, 65] are of limited help as well. The goal of these techniques is to detect faults through a thorough exploration of different app behaviors [42]. However, when there is a large number of feasible paths to explore, these techniques will have difficulties to quickly navigate to the functionality of interest. For the Zedge example in Figure 1.1, both the most prominent automated app exploration and testing tools, Sapienz [48] and Stoat [65], failed to reach the Facebook login screen after two hours of execution, even when configured to prioritize previously unexplored paths. That is because these techniques lack knowledge of which of the unexplored paths is most likely to lead to the functionality of interest and thus end up exploring a large number of possible unrelated paths.

1.2 Research Objective

In this thesis, we introduce a *goal-driven app exploration approach* and the associated tool, called GOALEXPLORER, which directly triggers a particular functionality of interest. GOALEXPLORER combines a novel approach for modeling an app User Interface (UI) as a *Screen Transition Graph* (STG) with a dynamic STG-driven app

exploration technique. Given a particular target of interest – an Android activity, API call, or a code statement, GOALEXPLORER automatically builds an executable script that swiftly triggers the target.

We believe that our work would benefit stakeholders who are interested in validating the behaviors of Android apps on a large scale, e.g., app store owners and security auditors. The static UI model proposed in our work, STG, can also be used by the research community to perform UI-related analyses, such as detecting spurious UI transitions [75], e.g., when clicking on the “Settings" button directs the user to the “Shopping" page.

1.3 Research Contributions

The idea of building a static UI model of an Android app is not new by itself. To the best of our knowledge, A3E [5] was the first to build a static model of the Android app UI, which only focused on Android activities. Gator [73] further extended this model to capture information about the menu items and the widgets that facilitate transitions between the modeled UI components, e.g. activities and menus.

However, the main building blocks of both these models are UI components rather than their composition into UI screens. As such, they do not accurately model the current Android UI framework [8] that allows composing screens from multiple components, such as fragments, menus, drawers, etc. Thus, they will misrepresent transitions originating in fragments, e.g., the transitions between screens in Figures. 1.1(b) and (c). Moreover, they do not model states and transitions triggered by background tasks and events (*broadcast receivers*, in Android terminology). For example, an app relying on the Facebook login mechanism can use a broadcast receiver to deliver login information to the interested app components, as done in Zedge example in Figure 1.1: after the login is triggered in Figure 1.1(e), the app broadcasts the login status and, in the case of a successful login, additional meta-information such as the user email address, to the *Controller* activity in Figure 1.1(f). Without modeling broadcast receivers, a goal-driven exploration tool will fail to reach that screen.

Our analysis shows that 63% of the top 100 apps from Google Play have at least one screen transition originating from a fragment and 34% have at least one transition originating from a background task or broadcast receiver. Modeling these behaviors is thus critical for the goal-driven exploration task and we do so in our work.

In addition, we introduce the concept of modeling UI screens as composition of UI components rather than individual UI components. We empirically show that, for our task, such approach is superior to the existing concepts of modeling app UI.

More specifically, our empirical evaluation of GOALEXPLORER on a large number of Android apps from existing benchmarks and the Google Play store shows that it is able to explore a substantially higher portion of each app when compared with the static model generated by Gator, even if we extend Gator's model to include additional elements, such as fragments, services, and broadcast receivers. When compared to state-of-the-art dynamic exploration and testing techniques, i.e., Sapienz [48] and Stoat [65], GOALEXPLORER is able to explore a similar portion of each app, which attests to the accuracy of its statically-built model of screens and transitions. However, GOALEXPLORER is able to reach the functionality of interest substantially faster than the dynamic exploration tools: it takes GOALEXPLORER 47 seconds on average to trigger a target statement, compared with 710 seconds and 861 seconds for the baseline tools, respectively. As such, GOALEXPLORER provides an efficient solution to the goal-driven exploration problem in mobile apps.

Interestingly, our analysis of network traffic associated with Facebook login in Zedge helped revealing a potential security vulnerability in this app: after receiving the OAuth [58] authentication/bearer token from Facebook, the app openly sends it to its server. Such behavior is discouraged by the Internet Engineering Task Force (IETF) [30] because any party holding the token can use it to access the user private information, without any further proof of possession. According to IETF, when sent over the network, bearer tokens have to be secured using a digital signature or a Message Authentication Code (MAC) [30]. Zedge is not following

these guidelines; thus, a man-in-the-middle can steal the token and access private information of users logged into the app.

To summarize, this thesis makes the following major contributions:

- It defines the problem of goal-driven exploration for mobile apps: efficiently triggering a functionality specified by the user.
- It shows that existing approaches are insufficient for performing goal-driven exploration.
- It presents a technique for constructing a static model of an app UI called *Screen Transition Graph* (STG) and using STG to generate an executable script that triggers a specific functionality of interest defined by the user.
- It implements the proposed technique in a tool named GOALEXPLORER and empirically shows its effectiveness on the 93 benchmark apps used in related work and 95 top Google Play apps. Our implementation of the GOALEXPLORER and its experimental evaluation package are available online [2].

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

- Chapter 2 provides the background on Android app component structure, lifecycle, and UI.
- Chapter 3 reviews the related work.
- Chapter 4 presents our approach and its implementation.
- Chapter 5 outlines our evaluation methodology and presents the evaluation results.
- Chapter 6 concludes the thesis, discusses the limitations of our approach and threats to its validity, and presents the possible future research directions.

Chapter 2

Background

In this chapter, we give the background on app component structure, lifecycle, and UI.

2.1 Android Applications

2.1.1 App Components

Android apps are built from four main types of components — activities, services, broadcast receivers, and content providers [23]. *Activities* are the main UI building blocks that facilitate the interactions between the user and the app. *Services* perform tasks that do not require UI, e.g., prefetching files for faster access or playing music in the background after the user has switched to other activities. *Broadcast receivers* are components that respond to notifications. A broadcast can originate from

(i) the system, e.g., to announce that the battery is low,
(ii) another app, e.g., to notify that a file download is completed, or
(iii) from other app components, e.g., to notify that the login was completed successfully.

Content providers manage access to data, e.g., for storing and retrieving contacts.

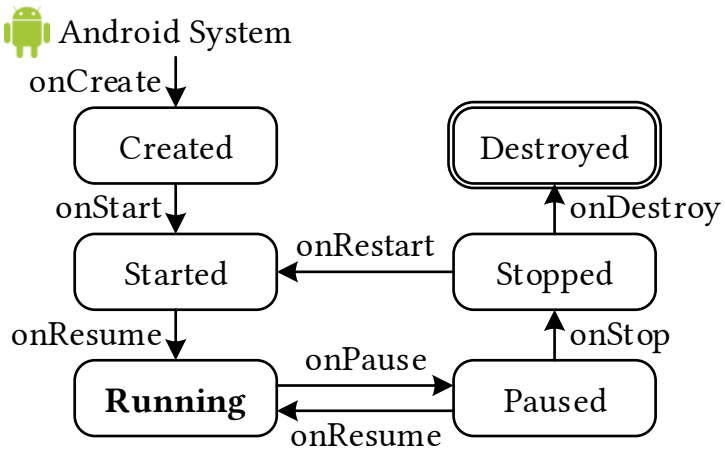


Figure 2.1: Activity Lifecycle Events.

Except broadcast receivers, all app components must be declared in the manifest file; the declarations include components' basic properties, such as Java handler classes. Broadcast receivers can be declared either in the manifest file or dynamically in code by calling system API `Context.registerReceiver`.

2.1.2 Component Lifecycle

Unlike Java programs, an Android app does not have a main method; instead, each app component implements its own lifecycle events. The activity lifecycle [20] is shown in Figure 2.1. Once an activity is launched, the system calls its `onCreate` method, followed by `onStart` and `onResume`. Developers specify the behavior of these methods, e.g., the UI setup is typically done in `onCreate`. The activity completes the initialization after `onResume`. It then enters a running state, and begins handling callbacks, e.g., those triggered by user inputs or by system changes. The activity is paused when it loses focus, and can eventually be stopped and destroyed by the operating system.

The lifecycle of services is similar to that of activities, except additional methods that handle service bindings, i.e., cases when other app components maintain a persistent connection to a service. Binding is achieved by calling

`Context.bindService`, followed by invocation of the `onBind` lifecycle method of the service.

Broadcast receivers only have one lifecycle event, `onReceive`, to handle the received events. Content providers do not expose any lifecycle events.

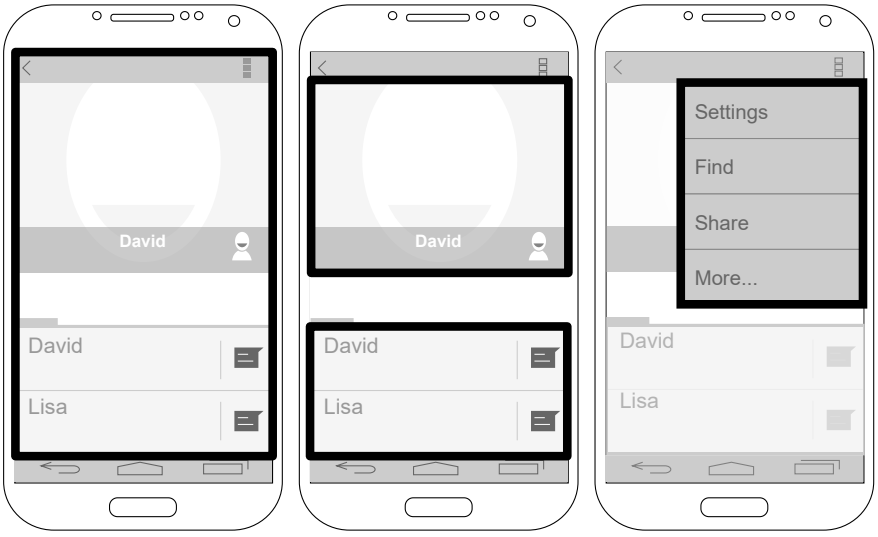
The transition between Android components relies on `Intents`, which explicitly or implicitly specify the target component and the data to be passed to that component. An activity, service, or broadcast receiver can launch another activity or service by calling corresponding Android API functions, such as `startActivity` and `startService`, and passing the `Intent` as a parameter.

Broadcast receivers are registered with intent-filters, which specify the types of events that the broadcast receiver handles. Once an event occurs, the system will trigger the `onReceive` callback methods of all broadcast receivers registered for this event. For example, an app can register a broadcast receiver listening to changes of the network status and, in the `onReceive` callback, start a certain activity or service once the device has acquired access to the Internet.

2.1.3 App UI Structure

An application screen is represented by an activity "decorated" by additional UI components, such as fragments, menus, dialogs, and navigation drawers [23], which correspond to modular sections of the screen. Figure 2.2 depicts such modular composition of screens. The solid box in Figure 2.2a highlights the container activity; its two hosted fragments are highlighted in Figure 2.2b.

A fragment is a re-usable UI component with its own lifecycle and event handling. A fragment can be instantiated in multiple activities and must always be hosted by at least one activity. Its lifecycle is directly affected by the owner activity, e.g., when the owner activity is destroyed, all of its hosted fragment instances are also destroyed. Fragment instances can be added, replaced, or removed from the host activities through the `FragmentTransaction` APIs, forming different screens of the same activity. Multiple fragments can exist in a single screen, as in Figure 2.2b; a screen can have any number of fragments or no fragments at all.



(a) Activity. **(b)** Fragments. **(c)** Menu.

Figure 2.2: Application Screens.

Menus and navigation drawers are UI components used primarily to provide additional options to the user. The former are typically used for system-wide events, such as adjusting settings and the latter – to ease access to other app components, e.g. launching new activities. Figure 2.2c shows the activity with a menu opened, which allows the user to access system-wide actions, such as settings and search. Figure 1.1(a) shows a drawer that occupies most of the screen and contains buttons for navigating to other parts of the app. The user activates and deactivates menus and drawers by interacting with their action buttons, such as the one in the top left corner of Figure 1.1(c) and the top right corner of Figure 2.2c.

Dialogs are small windows that prompt the user to take an action before the app can proceed with its execution, e.g., to agree to granting access to the device location. Menus, navigation drawers, and dialogs must be hosted by either an activity or a fragment.

UI components are composed of widgets, such as `Buttons` and `TextViews`, which are called `Views` in the Android terminology. `ViewGroups` are invisible containers used for grouping the widgets and specifying their layout before placing

them inside the UI components. User mostly interacts with UI widgets and are typically unaware of the app and UI components and their lifecycle.

Chapter 3

Related Work

In this chapter, we discuss related work on testing Android apps (Section 3.1). We then review desktop and web application testing approaches related to goal-driven exploration (Section 3.2).

3.1 Android Testing

Our goal-driven exploration technique consists of two parts – a statically constructed UI model and a dynamic app exploration technique based on the model. Thus, Section 3.1.1 focuses on existing Android UI models and Section 3.1.2 discusses existing automated app exploration techniques.

3.1.1 Existing Android UI Models

Approaches that statically construct an Android UI model, such as A3E [5] and Gator [73], are the closest to our work. Activity Transition Graph (ATG) introduced by Azim and Neamtiu in A3E [5] captures app activities and transitions between these activities. It does not model other app components and does not model actions that trigger the transitions, i.e., which user input triggers the transition between activities. Window Transition Graph (WTG) introduced by Yang et al. in Gator [73] extends ATG by addressing some of these limitations: it models

the event handler for each transition and also incorporates menus and dialogs as distinct nodes in the model. However, it does not consider fragments, drawers, services, and broadcast receivers.

Due to these reasons, for the Zedge example in Figure 1.1, both ATG and WTG will only include three nodes that correspond to the app activities: “Controller”, which corresponds to the activity in Figures 1.1(a-c) and (f), “Authenticator”, for the activity in Figure 1.1(d), and “Facebook”, for the activity in Figure 1.1(e), which is implemented inside the Facebook SDK. ATG and WTG models will include no transitions, as all transitions in this app originate from fragments. As such, none of these models will contain the path that reaches the Facebook activity. Moreover, the models do not account for broadcast receivers and thus no activity that follows the Facebook login, e.g., the one in Figure 1.1(f), is reachable.

An even more severe drawback of these existing UI models is that they treat UI components, such as activities and menus, as separate nodes. These models do not accurately represent the composition of UI components into screens, which hinders their applicability for the goal-driven exploration scenario: as the representations do not include enough details to extract an executable path. In Chapter 4, we will discuss this limitation in more detail, outline alternative ways to model Android UI components and their relationships, and then present our proposed solution, Screen Transition Graph.

A number of approaches build up on or extend ATG and WTG [9, 69, 71, 79]. For example, StoryDroid [9] statically renders the UI of each activity in ATG. The goal of this visualization of an app UI is to assist app development process, e.g. to provide recommendations for UI design and layout code. Wang and Rountev [69] statically estimate the execution time of each WTG path based on the compute-intensive API invocations along the path. Wu et al. [71] adopt a similar approach to detect if any WTG path is energy-inefficient by statically estimating the energy usage based on the energy-intensive API invocations along a WTG path. CHIME [79] considers information about activity launch modes, which determine how activities are placed in stack: e.g., an activity launched with “singleTask"

mode is only allowed to have one instance and the latest instance will kill all other instances of the same activity in the stack. By extends the modeling of transitions in ATG with launch modes, CHIME produces a more accurate graph than A3E. Yet, these approaches do not change the underlying design decisions related to the representation of UI elements in a model, thus sharing all the limitations of these models.

3.1.2 Automated App Exploration

The primary goal of automated app exploration techniques is to detect faults in apps. These techniques are typically used by app developers to discover issues before deployment [11]. Automated app exploration can also be used as the starting point of analyses performed by app store maintainers and auditors, e.g. to verify security of apps published to the store. As discussed earlier, existing app exploration techniques are usually designed to test the whole app and not optimized to test only the functionality of interest.

In the remaining of this section, we categorize the existing automated app exploration techniques based on their main test methodologies, further dividing this section into random testing, model-based testing, and symbolic execution.

Random Testing. Random testing techniques generate random, independent input events sequences during the test session. Monkey [27] is the most famous of such approaches, which is part of the official Android SDK Tools [22]. Monkey randomly generates various input events, including user gestures and system events, such as adjusting volume and capturing screenshots. Interestingly, up to date, Monkey achieves largely the same app-level coverage for real-world apps as the most advanced state-of-the-art exploration techniques [68].

Dynodroid [46] is another pseudo-random test generation tool that extends Monkey with feedbacks that continuously adjust the weights of possible input events. DiagDroid [41] randomly exercises apps and tracks the asynchronous executions to detect the performance issues related to these asynchronous executions. All such tools explore the app in a random manner, and are unable to rapidly trigger

the exploration target for the goal-driven exploration scenario.

Model-based Testing. Model-based testing techniques automatically generate test cases based on a model of the app under test. According to Kong et al. [42], model-based testing is the most common Android testing methodology and 63% of the techniques involve model-based testing steps. Most model-based app testing techniques rely on a dynamically constructed model to facilitate the testing process.

For example, Stoat [65] builds a model at runtime that captures the application screen states and input events that trigger the changes in screen states, e.g., button clicks and signal strength change. Although Stoat also statically analyzes the apps, it only uses static analysis to identify all possible input events for the app, especially the system-level events that cannot be inferred during dynamic exploration. Stoat's dynamic model contains probabilities to transitions, which are mutated during the model refinement stage, to prioritize generation of test that optimize for higher test coverage and better fault detection.

Sapienz [48] is another state-of-the-art tool that seeks to maximize code coverage and minimize the length of the generated test sequences in order to reveal more faults in shorter time. It combines random exploration with pre-defined event sequence patterns that capture complex interactions with the app. The dynamic UI model in Sapienz contains widgets in the app UI; the set of model states is continuously extended with states collected when interacting with the app widgets.

APE [31] represents the dynamic model as a decision tree and continuously tunes it with the feedback obtained during testing to construct a finer-grained model than Stoat [65] and Sapienz [48]. APE's model contains attributes of all widgets, e.g. to indicate whether the widget is clickable or not, whereas such attributes are ignored in Stoat and Sapienz. For example, Stoat assumes that all items in a list (`ListView` in Android terminology) behave equivalently, which is inaccurate when the attributes of these items are completely different, e.g. one is clickable and others are not.

Although the models used in these model-based testing techniques are different, the basic idea is similar: dynamically build a model of the app and mutate it so

that the generated tests achieve higher coverage and faults detection. Since these tools focus on systematically executing the whole app to detect faults rather than swiftly navigating to a particular functionality of interest, they are inefficient for goal-driven exploration. Specifically, our experiments demonstrate that while Stoat and Sapienz are able to achieve coverage similar to GOALEXPLORER, the time it takes these tools to reach the target is substantially higher. That is because without building an accurate static model and identifying the target upfront, it is difficult to determine which part of the app to ignore/explore first.

Symbolic Execution. Another line of work explores approaches for generating semantically meaningful inputs through symbolic execution to facilitate app exploration. To perform symbolic execution on Android apps, several approaches [53, 39, 70] build a customized Dalvik virtual machine so that the app can be exercised in a controlled environment. For example, IntelliDroid [70] statically resolves the callgraph path constraints leading to the malicious behaviors of Android apps and then executes the subject app in a customized Dalvik virtual machine with inputs that force the execution to a specific callgraph path. Such approaches can only apply to a specific version of Android, and the customized virtual machine has to be rebuilt if an updated version of Android is available. In contrast, our approach does not require such modifications.

Instead of building a customized virtual machine, a number of approaches [38, 52, 62, 70, 19] instrument the application to force executing a given callgraph path. For example, Jensen et al. [38] proposed an approach that uses concolic execution to generate input event sequences that force the execution towards the target line of code in the callgraph. However, this approach only resolves the path constraints within the component. It requires an UI model of the app to handle the transitions between different components of the application, and is complementary to our work on modeling UI screens and transition between these screens.

Moreover, approaches based on symbolic execution usually face the path explosion problem [6]. The main sources of path explosion are loops and function calls, and the event-driven nature of Android apps make it even more challenging.

The existing techniques make simplifying assumptions to overcome path explosion, which introduce inaccuracy in extracting the path constraints. In addition, the complexity of symbolic execution grows exponentially with the code size. Most of the symbolic execution approaches are evaluated on small open-source apps or malware samples. However, our observation of running those techniques on real applications reveals that the majority of the constraints that they manage to resolve are related to if-conditions instead of UI-related operations. Due to these limitations, the existing symbolic execution approaches are not applicable to the goal-driven exploration.

3.2 Desktop and Web Testing

UI testing for desktop apps have been extensively studied mainly for executing manually scripted test cases [49]. Techniques that generate UI models for desktop apps [63, 3, 78] are closest to our work. For example, Palus [78] used static analysis to refine dynamically-constructed model, generating test cases that detect more bugs in Java applications. Gazoo [3] statically constructs a model of a Java application, which is used to generate test cases that cover larger portion of application and detect faults in the application. These UI testing approaches are not applicable to automated UI tests in mobile apps as desktop apps normally possess a single entrypoint and mobile apps have multiple entrypoints due to their event-driven nature. However, these approaches have demonstrated the benefit of using a statically construct model to guide the exploration.

As for web apps, app UI models often cannot be generated through static analysis. Since the majority of the web apps are JavaScript-based, static techniques, while work well on statically typed language, are error-prone in analyzing highly dynamic JavaScript apps [56].

Given the difficulty in statically analyzing JavaScript applications, most of the existing work focus on pure dynamic approaches [7, 13, 50, 51, 67, 17]. Guided test generation techniques produce test suites that achieve a specific goal [67, 17] are closest to our work in the area of web testing. For example, the goal of WATEG [67]

is to achieve a higher coverage of specific business rules, such as calculating interest rates, loan eligibility, and customer status in a banking application. WATEG relies on the user for specifying business rules as predicates over the UI state of the app, e.g. a credit card number is entered and `Pay` button is pressed. It then performs a two-phase exploration: it first crawls the web app to construct a UI model of the transitions while collecting the predicates and then executes the app to follow the paths in the generated model that satisfy all required predicates.

FEEDEX [17] uses the coverage metrics measured during exploration session to guide the exploration towards unexplored parts of the application. Yet, its goal is different from ours, which is to trigger a functionality of interest. Moreover, all these approaches generate app models solely from the data received during the dynamic exploration. They are similar to the model-based testing techniques for Android apps, which lack the knowledge of which unexplored path can lead to the functionality of interest, hence not applicable to the goal-driven exploration problem defined in this thesis.

Chapter 4

GOALEXPLORER

This chapter introduces our approach for performing goal-driven exploration, which is implemented in a tool called GOALEXPLORER [2]. The high-level overview of GOALEXPLORER is presented in Figure 4.1. The shaded boxes are three major steps of GOALEXPLORER; the main components of each step are depicted in white boxes. The arrows indicate data or control flows between steps and components.

GOALEXPLORER obtains two inputs: the Android Package (APK) file of the app and the target functionality of interest, which can be provided as an app activity, an API call, or a code statement. For example, when monitoring the network traffic related to the Facebook login scenario in Zedge, we set as target a Facebook API call, as discussed in Chapter 1.

In the first step, GOALEXPLORER statically constructs the Screen Transition Graph (STG) of the app (the *STG Extractor* component in Figure 4.1). It then maps the target(s) to (possibly multiple) nodes of the graph (the *Target Detector* component). Finally, it uses the graph to guide the dynamic exploration to a reachable target node, starting with the one that has the shortest path from the initial screen (the *Dynamic Explorer* component). Due to the conservative nature of our static analysis, dynamic exploration can encounter infeasible paths; it then backtracks and try the next shortest path to the same target or chooses the next target, if multiple targets are specified.

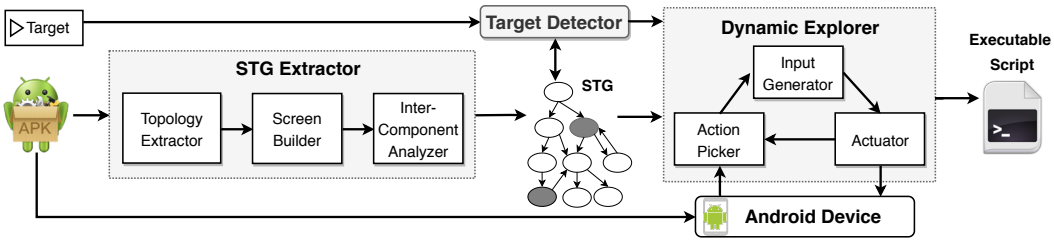


Figure 4.1: GOALEXPLORER Overview.

In what follows, we first introduce the STG and discuss our design choices. We then describe the *STG Extractor* (Section 4.2), *Target Detector* (Section 4.3), and *Dynamic Explorer* components (Section 4.4).

4.1 Screen Transition Graph

As discussed in Chapter 2, both Window Transition Graph (WTG) and its predecessor, Activity Transition Graph (ATG), model UI components as distinct nodes in the graph [5, 73]. Moreover, they do not model fragments, drawers, services and broadcast receivers. As such, for the example app in Figure 1.1, these graphs will only contain three activity nodes: elements #2, #3, and #4 presented with solid lines in Figure 4.2.

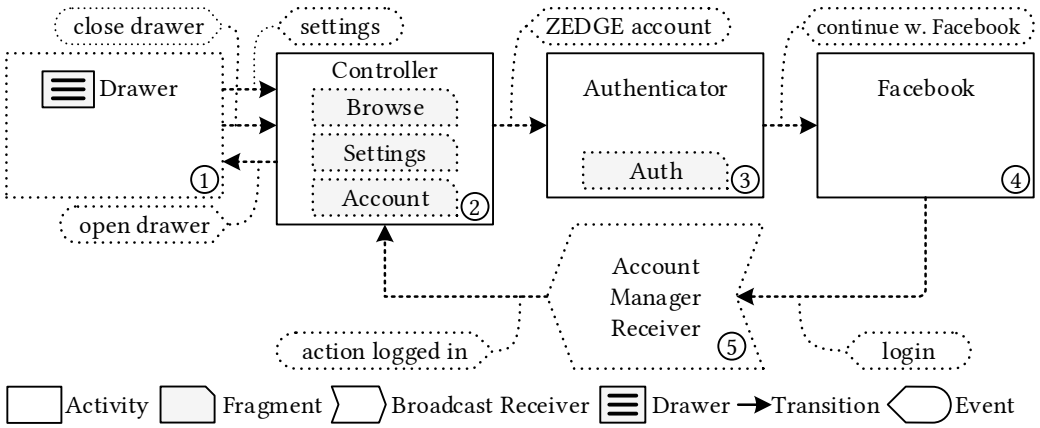


Figure 4.2: WTG (solid lines) and WTG-E (solid and dotted lines) of Zedge.

Our "direct" extensions to this model, which we refer to as WTG-E, introduces

a number of additional elements, presented with dotted lines in Figure 4.2. For conciseness of the discussion, we only include in this snippet parts of the Zedge shown in Figure 1.1:

- Assuming drawers are handled similar to menus, as they correspond to similar UI concepts, element #1 in Figure 4.2 would represent the drawer in Figures 1.1(a,b). Adding drawers would also lead to creating transitions labeled "open drawer" and "close drawer" between elements #1 and #2.
- The transition labeled "settings" from element #1 to #2 would be contributed by an analysis of the drawer behavior that launches a new activity.
- The "Account Manager" broadcast receiver, responsible for notifying other app components when Facebook authentication is completed, would be represented by element #5, and its corresponding incoming and outgoing transitions.
- The fragments contributing to each activity could be represented by inner boxes inside elements #2 and #3, as only these two activities have fragments in this example. Representing the fragments would allow the model to capture transitions from element #2 to #3 and #3 to #4, as these transitions are triggered from fragments embedded in each corresponding activity.

However, even the extended WTG-E model is sub-optimal for producing the execution scenario that leads to the Facebook login screen in the Zedge example. That is because WTG-E's main nodes are UI components rather than their composition to screens. Thus, the model cannot represent the actual screens, omitting important information used for exploration.

For example, from the WTG-E graph in Figure 4.2, one can conclude that "close drawer" action for element #1 that leads to element #2 – the "Controller" activity, can be followed by the "ZEDGE account" action to reach element #3 – the "Authenticator" activity. Such execution is not possible in practice as "ZEDGE account" action is only enabled when the "Controller" activity is opened with

the “settings“ fragment. However, when the app execution starts, the “Controller” activity is rather opened with the “browse“ fragment, thus there is no option to press the “ZEDGE account” button after closing the drawer. Without distinguishing between these different states of the “Controller” activity (element #2), one cannot successfully explore the application.

The Screen Transition Graph (STG) proposed in this work addresses this limitation. It models application screens by representing the composition of the container activity, hosted fragments, menus, navigation drawers, and dialogs on each screen. Figure 4.3 shows the STG representation of the Zedge snippet in Figure 1.1. Unlike in the WTG-E model, the “Controller” activity here is represented by five different elements: #1-4 and #8. These elements correspond to different compositions of fragments and drawers hosted by the activity. This representation clarifies that the only possible path to the Facebook login activity (element #6 in Figure 4.3) is to start from the “Controller” activity (element #1), open the drawer (arriving at element #2), then press the ‘settings” option (arriving at element #3). Only after that one can transition to elements #4, #5, and #6.

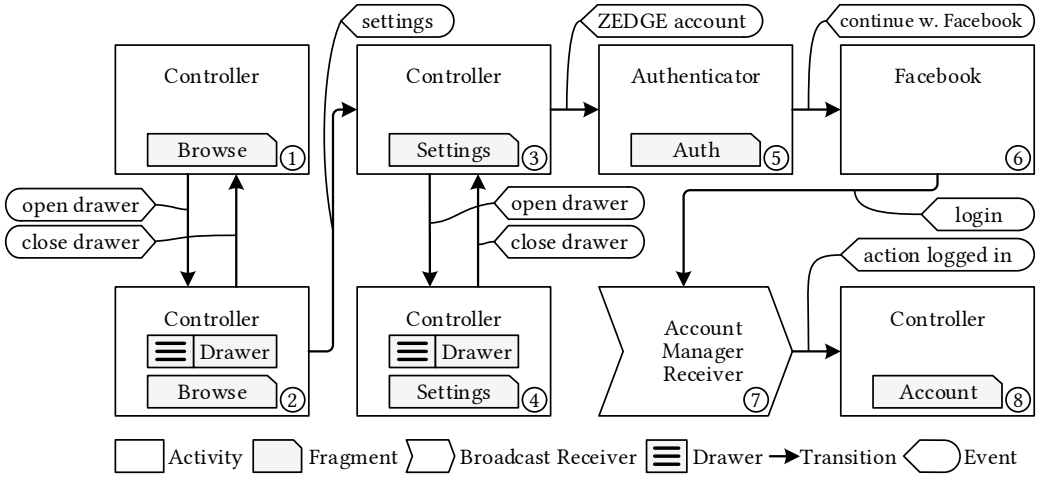


Figure 4.3: STG of Zedge.

Moreover, after the Facebook login is completed, the execution arrives at element #8, which is another instance of the “Controller” activity, but with the “Account” fragment. It is clear from this representation that the user cannot press

"ZEDGE Account" again, as can mistakenly be concluded from the WTG-E representation in Figure 4.2.

The example demonstrates that the STG representation is more accurate than the original WTG, and even WTG-E, for producing execution scripts in the goal-driven exploration scenario. We empirically evaluate the differences between these representations in Chapter 5. Below, we give a more formal definition of STG.

In STG, the main nodes V_s represent app screens and the supporting nodes V_r and V_b represent services and broadcast receivers, respectively. Each screen node in V_s consists of one container activity V_s^A, zero or more fragments V_s^F, zero or one menu/drawer V_s^M, and zero or one dialog V_s^D. The collection of all screen nodes corresponds to possible screens allowed in the app. Intuitively, an activity with n fragments and one drawer could correspond to 2×2^n different screens (all combination of fragments, with and without the drawer). Of course, not all combinations are possible in a particular app, and we describe our screen extraction algorithm in Section 4.2.

Supporting nodes are required as transitions between screens can pass through these elements. The edges E in STG correspond to transitions between nodes $V = V_s \cup V_r \cup V_b$. Each edge $e \in E$ has a label e_τ which represents the event that triggers the transition: either a user action, e.g. pressing the "ZEDGE account" button to move from element #3 to #5, or an intent that triggers the broadcast receiver event, e.g. the notification received by the system for transitioning from element #7 to #8. Transitions that are triggered automatically, e.g., when a service opens a UI dialog without any notifications or clicks, have no event triggers in the model: $e_\tau = \emptyset$.

4.2 STG Extractor

We now describe our approach for constructing STG. The *STG Extractor* consists of three main components outlined in Figure 4.1: *Topology Extractor*, *Screen Builder*, and *Inter-Component Analyzer*. These components contribute to collecting and refining STG elements.

4.2.1 *Topology Extractor*

Topology Extractor identifies the main app components, i.e., activities, services, and broadcast receivers, and their call graphs. It relies on FlowDroid's implementation [4] to extract the components from both the app source code and xml files, to associate each components with its entry points, i.e., lifecycle events and application-specific callback methods, and to build a call graph for the components that comprise of the call graphs of the entry points. For example, an activity may register callbacks that get invoked when a button is pressed; or a location update is received, and the respective callback handler would then being linked to the activity's call graph and being analyzed between the onResume() and onPause() lifecycle events of the activity.

App methods annotated with "@JavascriptInterface" can also be triggered by JavaScript-enabled WebViews [29] . As the invocation of these methods depends on the dynamic WebViews content delivered at runtime and cannot be determined statically, *Topology Extractor* conservatively extends the call graph by assuming that any JavaScript-enabled method in a component can be called by any JavaScript-enabled component's WebView.

Services and broadcast receivers collected by *Topology Extractor* are supporting nodes in STG and are added directly to V_r and V_b, respectively. App screens in V_s are built in the next step.

4.2.2 *Screen Builder*

A key insight of our approach is modeling the UI screens and the transitions between the screens. To this end, *Screen Builder* analyzes the call graph of each activity, collecting the hosted fragments, menus, dialogs, and navigation drawers.

Fragments can be defined statically in the activity layout file or dynamically in code. *Screen Builder* starts by extracting the layout files using AXMLPrinter [55] and then identifies fragments they declare. To collect fragment declarations in code, *Screen Builder* scans the call graph of each activity, identifying fragment transaction methods that add, remove, or replace fragments. The full list of such

methods, collected from the Android Developer website [25], is available in our online appendix [2].

The activity call graphs are scanned in multiple iterations. First, *Screen Builder* analyzes the activity lifecycle events triggered when an activity is initialized: from onCreate() to onResume(). Fragments collected from these lifecycle events will appear in the 'base' screen of the activity, which is displayed when the activity is launched.

When collecting fragments, *Screen Builder* performs an inter-procedural flow-sensitive analysis to identify fragment transaction methods s that add, remove, or replace fragments. It then performs a context-sensitive point-to analysis on the calling path of s, starting from the lifecycle event, to identify the Java type of the fragment(s) handled in s. Once all fragments from the onCreate() to onResume() lifecycle events of an activity A are processed, *Screen Builder* creates the "base" screen node V_s^A in V_s for the activity, with V_s^F containing a (possibly empty) list of the identified fragments.

Listing 4.1: Zedge code snippet.

```
1  public class AuthActivity extends BaseActivity {
2    protected void onCreate(Bundle bundle) {
3      initMainFragment();
4    }
5    protected void initMainFragment() {
6      FragmentManager manager = getSupportFragmentManager();
7      Fragment mFragment = new AuthFragment();
8      addFragment(manager, mFragment, R.id.output);
9    }
10   static addFragment(FragmentManager manager, Fragment fragment, int id
         ↪ ) {
11     FragmentTransaction transaction = manager.beginTransaction();
12     transaction.add(id, fragment);
13     transaction.commit();
14   }
15 }
```

This process can be demonstrated on the simplified code snippet from Zedge in Listing 4.1. In lines 12-13, a fragment `fragment` is added to the activity

`AuthActivity`. The variable `fragment` can be traced back to the variable *mFragement* in line 8, which is defined in line 7 and has the type `AuthFragment`. This fragment addition is triggered from the `onCreate` lifecycle event in line 3. Thus, *Screen Builder* will create a screen node for `AuthActivity` with one fragment of type `AuthFragment`.

Lifecycle methods issued when the activity is paused, stopped, or resumed, as well as callback methods, can further add, remove, or replace fragments in the "base" screen. *Screen Builder* thus analyzes possible lifecycle method invocation chains: "onPause $\rightarrow$ onResume" and "onPause $\rightarrow$ onStop $\rightarrow$ onRestart $\rightarrow$ onResume" (see Chapter 2) to extract the fragments from each of these chains. If the fragments in a chain differ from the fragments in the "base" screen node V_s^A, it adds a new screen node $V_{s'}^A$ for the activity A, which contains a union of fragments in V_s^A and those identified in the chain. It also adds a transition between V_s^A and $V_{s'}^A$, with an empty label, as this transition is triggered automatically by the system.

Fragments in callback methods are handled similarly. Since the order of callbacks cannot be predicted, *Screen Builder* assumes the callbacks can occur in any order; it thus creates a new screen $V_{s''}^A$ for any possible order of callback methods which modify fragments of the "base" screen. For example, given two callbacks c_1 and c_2, *Screen Builder* will create four additional screens augmenting the "base" screen and the screens collected after analyzing the lifecycle events. That is, each of the already created screens s will "cloned" to produce four new screen: after applying c_1 only, c_2 only, c_1 and c_2, and c_2 and c_1.

Screen Builder also creates transition edges between these newly created screens and sets the transition label e_τ to be the action triggering the corresponding callback. That is, for callbacks defined for UI widgets, e.g., of `Button.onClick()`, *Screen Builder* sets as the transition label the label associated with the widget. For the system event callbacks, `LocationManager.onLocationChanged()`, the transition label is empty.

After fragments are handled, *Screen Builder* analyzes activity call graphs, as well as its associated fragment call graphs, to identify methods initializing menus,

drawers, and dialogs. The full list of such methods is obtained from the Android Developer [26] and is available online [2]. Once an initialization of a menu, drawer, and dialog is found, *Screen Builder* copies each relevant screen s for the activity A into s', adds the found menu, drawer, or dialog to s', adds s' to the set of all screen nodes V_s, and creates a transition edge between s and s'. The transition label e_τ is set to be the action that triggers the callback method.

4.2.3 *Inter-Component Analyzer*

After the previous step, we have collected all screen nodes and the transitions between screen nodes of the same activity but with different fragments, menus, drawers, and dialogs. The *Inter-Component Analyzer* collects the inter-component transitions between nodes corresponding to different Android components, e.g. screens of different activities, services, and broadcast receivers. These transitions are performed via Inter-Component Communication (ICC) methods that launch new app components, e.g. `startActivity` and `bindService`. The target of the transition is specified in the Intent parameter of these methods, either explicitly or implicitly. We rely on FlowDroid's integration with IccTA [43] to identify ICC links between app components.

For each link where the source and target are services and/or broadcast receivers, *Inter-Component Analyzer* simply creates the corresponding transition edge e in STG, If the target of the transition is a broadcast receiver, the transition label e_τ is set to be the broadcast which triggers the event, as specified in the broadcast receiver intent-filter. If the target of the transition is a service, $e_\tau = \emptyset$, as this transition is triggered "automatically", without any user or system event.

Activities are represented by a number of screens and thus require a more nuanced treatment. If a source of an ICC link is an activity A, *Inter-Component Analyzer* identifies the activity entry point p from which the communication is issued. It then finds all screen nodes in STG that correspond to A, which can be reached from the "base" screen node of A via transitions associated with the action that triggers p. It adds a transition from all these nodes to the nodes that represent

the targets, labeling them with the action that triggers p. When the target is an activity as well, *Inter-Component Analyzer* finds only the "base" screen node of that activity and creates a transition to that node. That is because when a new activity is launched, it starts in the initial screen; transitions between different screens of the target activity are already handled by *Screen Builder*.

4.3 Target Detector

When the exploration target is specified in a form of an activity, *Target Detector* simply traverses and marks all screen nodes that correspond to that activity. Reaching any of these nodes will be considered reaching the target. For targets given as an API, *Target Detector* scans the app call graph to find all statements that invokes the given API. For targets given as a single statement, *Target Detector* simply finds the statement in the app call graph.

Next, it maps the identified statements to STG nodes as follows: if a statement is identified in a lifecycle event of a component, it marks all nodes that correspond to that components as targets. If a statement is in a callback method, simply reaching the component is insufficient as one also needs to invoke the callback method itself. Therefore, *Target Detector* identifies the STG transitions labeled by the action that triggers the callback and sets the destination of these transitions as the targets.

4.4 Dynamic Explorer

Given an STG of the app and a set of targets, *Dynamic Explorer* performs goal-driven exploration guided by the STG, to trigger at least one of these targets. It has three main components outlined in Figure 4.1: *Action Picker*, *Input Generator*, and *Actuator*, executing in a loop until a target node is reached.

4.4.1 *Action Picker*

In each iteration, *Action Picker* first evaluates all possible actions enabled from the current screen, such as clicking a button or filling a textual input field. For each screen, the number of possible actions is determined by the number of widgets and the type of each widget. We refer to all possible actions of a screen collectively as the state of this screen. *Action Picker* retrieves the current activity and fragments using an Android Debug Bridge (ADB) [21] `shell dumpsys` command and reads the screen UI hierarchy using UIAutomator [28]. This information is used to map the current screen to the corresponding node in STG and verify that the exploration arrived at the intended STG node.

If *Action Picker* is already "locked" on a particular target, it performs a breadth-first search on STG to find all paths from the current screen node S to that target and sorts the paths from the shortest to the longest. It picks the next action from the shortest path that leads to the new screen node S' and checks if the action is available on the screen. If so, it proceeds to the next step: *Input Generator*. Otherwise, it attempts to change the state of the current screen S by dynamically exploring the available actions until the transition to S' is available. That is done because some user actions do not trigger transitions between screens that are thus absent from STG; yet, they rather toggle the state of the screen, which might enable future transitions. For example, when adding an item to the shopping cart, an application stays in the same screen. This action is still required to move to the next screen – the checkout, which is only enabled when there is at least one item added to the shopping cart.

To activate the actions that can enable future transitions, *Action Picker* adopts a weighted UI exploration strategy, which was shown to efficiently discover new states of a screen [46, 65]. The exploration strategy assigns each action e an execution weight $W(e)$, which is defined as:

$$W(e) = \frac{\alpha * T_e + \beta * C_e}{\gamma * F_e}$$

where T_e is the event's type, e.g., click, scroll; F_e is the event's history execution frequencies that measure how many times an event has been executed during this exploration session. To avoid division by zero, F_e is initially set as 1 and it increments each time the event has been executed. C_e is the number of unvisited children widgets indicating how many of the widgets with $F_e = 1$ are in the screen after the event e has been executed. α, β, γ are the weight parameters that Stoat [65] carefully tuned based on their observation, hence we adopt the same value in *Action Picker*, and set α, β, and γ to 0.4, 0.2, and 0.4 respectively.

If S' cannot be reached within a certain number of attempts (currently set to 50 iterations), *Action Picker* backtracks and proceeds to the next path. If no action is available on any of the paths reaching to the current target or if there is no "working" target set yet, *Action Picker* performs a breadth-first search to find the next available target and repeats the search for that target. It returns "unreachable" if no action leading to any of the targets is found.

4.4.2 *Input Generator*

If an action towards a target is picked, *Input Generator* checks whether specific textual inputs are necessary to successfully trigger the transition.

First, as many Google Play apps also require the user to log in for accessing some of app features, an exploration tool cannot perform well without handling login screens. We equip GOALEXPLORER with the ability to handle logins, assuming that the login credentials are supplied as an optional input.

To this end, *Input Generator* analyzes the UI hierarchy of the current screen to search for textual input fields, such as `EditText`. It further evaluates whether the textual inputs requires login credentials. Similar to prior work [36], this is done by checking whether the widget ID, text, hint, label, and content description match the regular expressions associated with login and password strings, e.g., "username", "user id", etc. The exact list of the patterns is listed in Table 4.1.

If a match is found, *Input Generator* enters the credentials into the corresponding text fields. Otherwise, it feeds random strings to all textual input fields in the

Table 4.1: Regular Expressions for Login Detection

Concept	Regular Expression
username	(?i) (user | account | client | phone | card) [\s_\-]*(name | id | number) (?i) (log | sign) [\s_\-]* in[\s_\-]* (name | id |) (?i) [e] mail
password	(?i) (pass | pin) [\s_\-]* (word | code |)

current screen.

4.4.3 *Actuator*

Actuator fires the action selected by the *Action Picker*. Its implementation extends the one of Stoat, which provides an automated framework that supports *click*, *touch*, *edit* (enter random texts), *navigation* (e.g., *back*, *scroll*, *menu*). The original Stoat only supports Android API level 19 (i.e., Android 4.4), and we extend it to support newer Android platforms. *Actuator* also records all triggered actions, allowing us to produce an executable test scripts for reaching the target.

Chapter 5

Evaluation

We now describe our experimental setup and discuss evaluation results. Our aim is to answer the following Research Questions (RQs):

- **RQ1 (Coverage):** What is the fraction of targets GOALEXPLORER can reach and how does that compare with the baseline approaches?
- **RQ2 (Performance):** How fast can GOALEXPLORER reach the target and how does that compare with the baseline approaches?
- **RQ3 (Accuracy):** What is the accuracy of the model constructed by GOALEXPLORER and the ability of dynamic UI exploration to mitigate model inaccuracy?
  - **RQ3.1 (FN):** What is the fraction of cases when the model does not contain transitions that exist in practice?
  - **RQ3.2 (FP-Dynamic):** What is the fraction of cases when the model contains transitions that exist but cannot be triggered dynamically?
  - **RQ3.3 (FP):** What is the fraction of cases when the model contains transitions that do not exist in practice?

5.1 Experimental Setup

5.1.1 Methodology

To answer our research questions, we perform two main experiments. First, we compare exploration that is based on STG with the exploration based on the WTG and WTG-E models described in Section 4.1. At the time of writing, the WTG model has been used as the foundation for practically all techniques that rely on statically constructed UI model of a mobile app, e.g., [40, 76, 77, 69, 71]. As the original WTG model introduced by Gator [73] cannot be directly used for goal-driven app exploration, we extracted the model produced by Gator and plugged it into our *Dynamic Explorer*. Moreover, since Gator's implementation is not compatible with the current Android API level that we use in our work (API level 23), Gator failed to run for most of the Google Play apps. We thus re-implemented the WTG extraction process for apps with API level 23. We then extended WTG to handle fragments, services, broadcast receivers, etc., producing the WTG-E representation, as described in Section 4.1[1]. That is, we compare STG with three static models:

- WTG (Gator): the original implementation from Gator;
- WTG: our re-implementation of Gator's model to handle modern apps;
- WTG-E: our extension of WTG to handle fragments, drawers, services and broadcast receivers.

As a second line of experiments, we compare GOALEXPLORER with the state-of-the-art automated exploration techniques: Sapienz [48] and Stoat [65]. We run both tools with one hour allocated execution time for each target. We choose that timeout following the tools' own evaluation methodology: both Sapienz and Stoat have used one hour execution time when comparing test coverage to other

[1]Concurrent to our work, similar extension was proposed in StoryDroid [9]; but its implementation is not publicly available.

existing testing techniques. We used configuration parameters specified by the papers describing these tools and kept default values for parameters not described in the papers. We also experimented with modifying the configuration parameters, to ensure the baseline is most favorable for the goal-driven exploration task, e.g., by increasing the sequence length and population size for Sapienz and prioritizing coverage to test diversity for Stoat. As these adjustments did not affect the results, we kept the default configurations.

Moreover, we extended both tools to handle login screens, as we did for GOALEXPLORER (see Section 4.4). Specifically, we added the same login logic that we used for our tool to that of Stoat. As the relevant source code of Sapienz is not open-sourced (the motif part is only available as binary), we implemented a login detection component which runs in parallel to Sapienz and constantly checks if the execution arrived at the login screen. When it does, we pause Sapienz and handle the login flow as we do for GOALEXPLORER and Stoat. The execution time of this operation is negligible and is comparable to the handling of logins in GOALEXPLORER and Stoat.

5.1.2 Exploration Targets

We experiment with two scenarios. In the first one, we set each activity of the subject apps as the target, one at a time, and repeat the experiment for all activities in an app. That allows us to align our results with those commonly reported in related work. Then, we experiment with setting a code statement as a target, investigating a more nuanced, goal-driven exploration scenario which motivates our work. To this end, we pick a particular API, namely *URL.openConnection*, identify all occurrences of this API in the subject apps, and set them as targets, one at a time.

5.1.3 Subject Apps

A survey of automated testing tools by Choudhary et al. [11] used 68 open-source F-Droid [14] apps in their survey, which de facto became a standard benchmark in

analyzing automated testing tools. Both Sapienz [48] and Stoat [65] were evaluated on that benchmark and the Stoat authors further extended the benchmark with additional 25 F-Droid apps, producing a benchmark with 93 F-Droid apps in total. We evaluate our approach on the same 93 benchmark apps.

In addition, since F-Droid apps are easier targets, we downloaded the 100 most popular apps on the Google Play store as of July 31, 2018. This dataset allows us to evaluate our approach on real applications commonly used in practice. Of the 100 Google Play apps, five failed to run on the emulator (the decision to use emulators for the experiments is discussed in the *Environment* sub-section below). We thus excluded these five applications from further analysis, arriving at the set of 93 benchmark and 95 Google Play apps. A detailed description of these apps is available online [2].

5.1.4 Metrics

To answer **RQ1** and **RQ2**, we measure the fraction of targets reachable by each tool per app and the average time it takes to reach a target, respectively. The goal of these experiments is to detect whether design choices implemented in GOALEXPLORER allow us to cover a large fraction of app behaviors in a rapid manner.

For **RQ3**, we manually establish the ground truth of all paths leading to all reachable `URL.openConnection` targets in a subset of apps. Establishing ground truth for closed-source Google Play apps is unrealistic because these apps are largely obfuscated and manually tracking reachability of statements would not be reliable. We thus selected 36 open-source apps from our benchmark, which are also available on Google Play, to ensure we arrive at a representative and practically valuable subset. For these 36 apps, the author of this thesis manually collected the set of all reachable statements that invoke `URL.openConnection` API and cross-validated the findings with another member of the research group. The set of the analyzed apps and the constructed ground truth is available online [2].

We then analyze the main reasons preventing GOALEXPLORER from reaching

some of these targets, i.e., cases when the correct path is missing in STG. We also analyze the cases GOALEXPLORER encounters spurious STG paths that cannot be followed due to the over-approximation made by our model construction, i.e., cases when the tool has explored a wrong path and needs to backtrack. The goal of these experiments is to evaluate the accuracy of the model produced by our tool and the ability of our dynamic explorer to recover from failures.

5.1.5 Environment

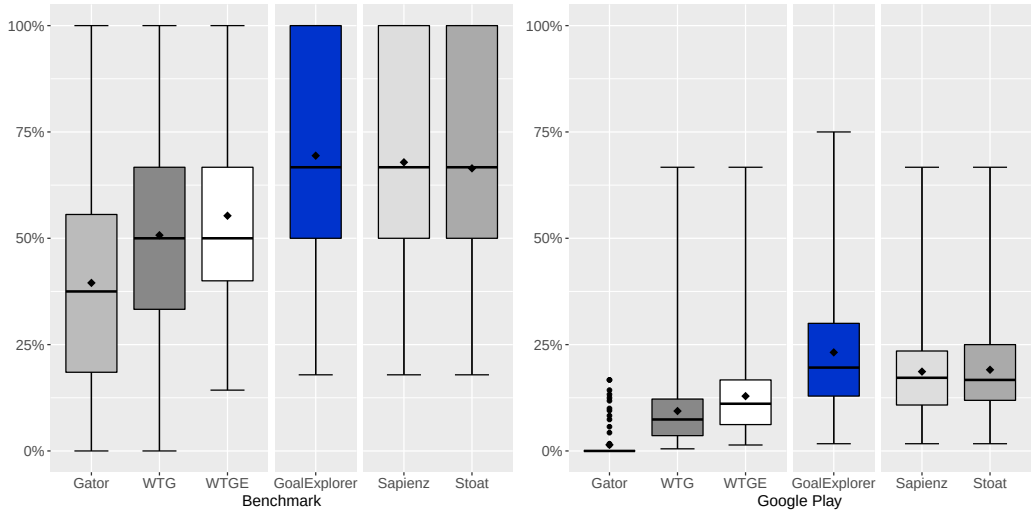
We run all tools on a 64-bit Ubuntu 16.04.4 physical machine with a 20-core CPU (Intel Xeon), allocating 64GB of RAM for each tool. We run each app for one hour with each of the compared tools; the total evaluation time thus exceeds 500 hours only to answer RQ1 and RQ2. Therefore, we run the experiments on Android emulators, to allow running multiple executions in parallel.

5.2 Results

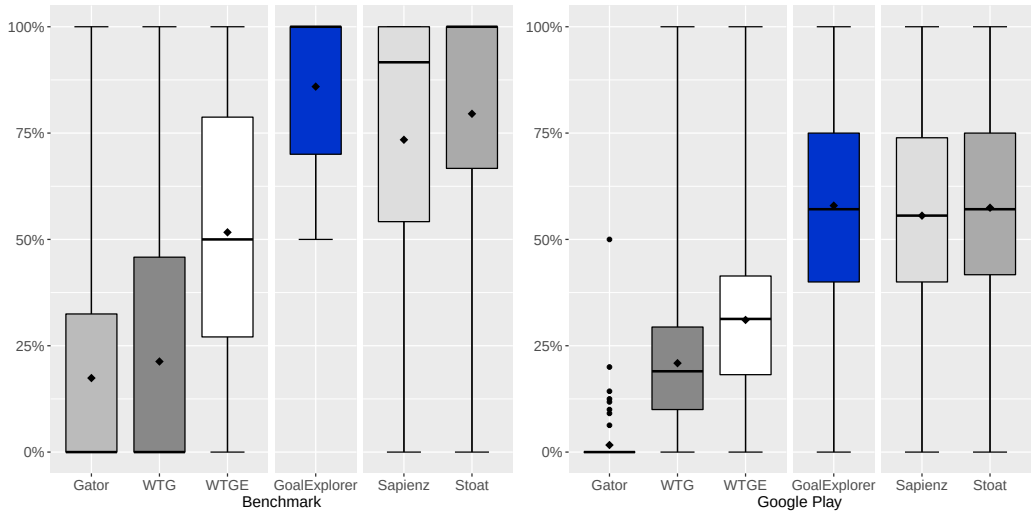
5.2.1 RQ1 (Coverage)

Figure 5.1a shows the box and whisker plot representing the minimum, median, mean, and maximum percentage of activities reached by goal-driven exploration under four different models: WTG (Gator), WTG, WTG-E, and STG (GOALEXPLORER). It also shows the fraction of activities reachable by the state-of-the-art dynamic exploration tools: Sapienz and Stoat. The results are plotted separately for the 93 benchmark F-Droid apps and the 95 Google Play apps.

Table 5.1a summarizes Figure 5.1a, reporting min, mean, median, and max percentage of target activities triggered. For the benchmark apps, GOALEXPLORER reaches between 17.9% and 100% of activities, with an average of 69.4% and a median of 66.7%. For comparison, WTG and WTG-E reach 50.7% and 55.3% of activities receptively, on average per app. The divergence between the results of WTG (Gator) and WTG is because Gator cannot process 5 of the benchmark



(a) Fraction of reachable activities in an app (higher is better).



(b) Fraction of reachable *openConnection* statements in an app (higher is better).

Figure 5.1: Coverage results for the benchmark and GooglePlay apps.

applications, resulting in crashes. These apps are excluded from the statistics reported in the figure.

For Google Play apps, the differences between the UI models is much more substantial. WTG (Gator) can only process nine out of 95 Google Play apps, thus

Table 5.1: Coverage results for the benchmark and GooglePlay apps.

(a) Fraction of reachable activities in an app (higher is better).

Category	Metrics	Gator	WTG	WTG-E	GoalExplorer	Sapienz	Stoat
Benchmark	*Min*	0	0	14.3	**17.9**	17.9	17.9
	Mean	39.5	50.7	55.3	**69.4**	67.9	66.5
	Median	37.5	50	50	**66.7**	66.7	66.7
	Max	100	100	100	**100**	100	100
Google Play	*Min*	0	0.5	1.4	**1.7**	1.7	1.7
	Mean	1.4	9.4	12.9	**23.2**	18.7	19.1
	Median	0	7.4	11.1	**19.6**	17.2	16.7
	Max	16.7	66.7	66.7	**75**	66.7	66.7

(b) Fraction of reachable *openConnection* statements in an app (higher is better).

Category	Metrics	Gator	WTG	WTGE	GoalExplorer	Sapienz	Stoat
Benchmark	*Min*	0	0	0	**50**	0	0
	Mean	17.4	21.3	51.7	**85.9**	73.4	79.5
	Median	0	0	50	**100**	91.7	100
	Max	100	100	100	**100**	100	100
Google Play	*Min*	0	0	0	**0**	0	0
	Mean	1.7	20.9	31.1	**57.9**	55.6	57.5
	Median	0	19	31.3	**57.1**	55.6	57.1
	Max	50	100	100	**100**	100	100

its overall activity-level coverage is very low: around 1%. Our re-implementation of Gator, WTG, can process all apps but covers around 9% of each app activity on average. WTG-E increases the activity-level coverage to 12.9% on average, as it is able to deal with fragments, services, and broadcast receivers. Finally, GOALEXPLORER reaches the highest activity-level coverage of 23.2% on average, due to its accurate representation of app screens. This result demonstrates that only extending existing static models with handling of fragments, services, and broadcast receivers, as we did in WTG-E, is less beneficial than the full set of solutions GOALEXPLORER applies.

Notably, GOALEXPLORER achieves similar (and even 3% higher) activity-level coverage than the state-of-the-art dynamic exploration tools, for both benchmark and the Google Play apps: for the benchmark apps, Sapienz and Stoat cover 67.9% and 66.5% of activities per app on average, respectively, while GOALEXPLORER covers 69.4%. For the Google Play apps, both tools reach around 19% of activities, on average, while GOALEXPLORER covers 23.2%. That is an encouraging result for GOALEXPLORER, showing that the STG model it builds statically is accurate and comparable with the models constructed at runtime during the dynamic exploration.

Unlike for the benchmarks, none of the tools is able to reach 100% activity level coverage for the Google Play apps. This is not a surprising result as most of the benchmark apps are simple utility apps with limited functionality and a relatively small number of activities (9 on average). In contrast, Google Play apps contain complex logic and large number of activities (60 on average), some of which might not be reachable at all [68].

Figure 5.1b and Table 5.1b show similar data for the the percentage of reachable `URL.openConnection` statements in an app. Here, GOALEXPLORER largely outperforms other static UI models, triggering 85.9% target API statements on benchmark apps, compared with 17.4%, 21.3%, and 51.7% for WTG (Gator), WTG, and WTG-E, respectively. It again performs better than dynamic tools, triggering 12.5% more target statements on average than Sapienz (85.9% v.s. 73.4% covered by Sapienz) and 6.4% more than Stoat (which covers 79.5%).

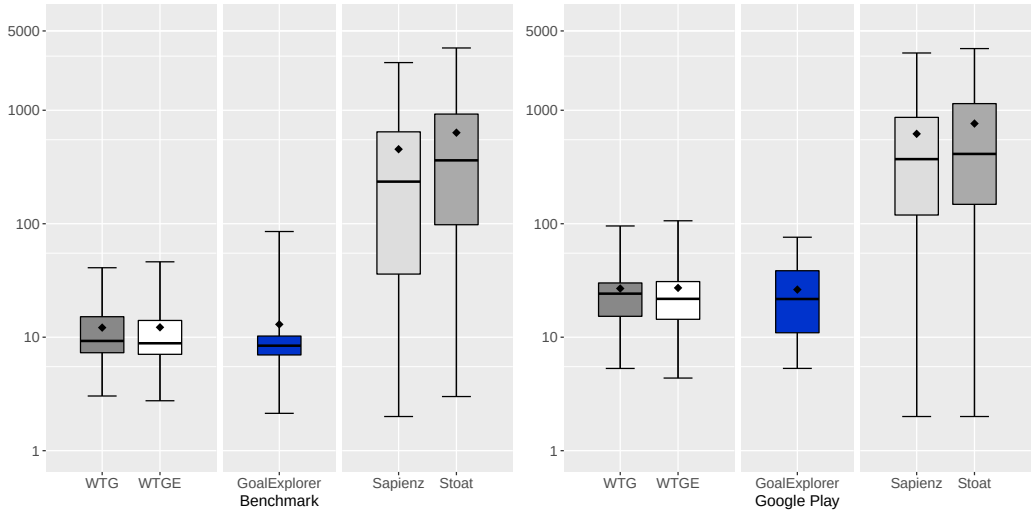
For GooglePlay apps, GOALEXPLORER also substantially outperforms other UI models, triggering 57.9% of target statements, while other models trigger only 21% and 31% of the targets. GOALEXPLORER performs comparable with dynamic tools: Sapienz triggers 55.6% of the target statements on average and Stoat – 57.4%.

To answer RQ1: Our experiments show that using STG model allows GOALEXPLORER to trigger substantially more targets than for other statically-built Android UI models, namely, WTG (Gator), WTG and WTG-E. Moreover, GOALEXPLORER is as effective as the dynamic tools in the ability to reach a certain target.

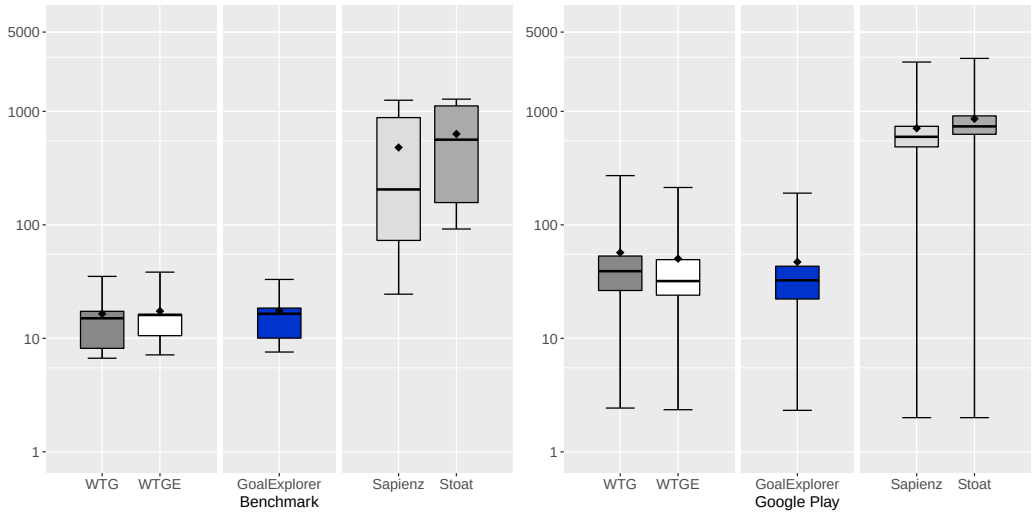
5.2.2 RQ2 (Performance)

To evaluate performance, we measure the minimum, mean, median, and maximum time, in seconds, it takes GOALEXPLORER to reach a target activity or statement. For fair comparison with dynamic tools, we run these tool for one hour each and measure the time it takes a tool to reach each new target: an activity or a statement. Such a comparison is more favorable for these tools, as we do not restart the tool from scratch for any new target. We only present times of reaching targets that are reachable by all of the compared tools, excluding the original Gator implementation from our analysis as it runs only on 9 out of 188 apps.

Table 5.2a and Table 5.2b present the results of the comparison for a target activity and statement, respectively. The results are also plotted in Figure 5.2, on a logarithmic scale. The results show that GOALEXPLORER can reach a target much faster than dynamic tools: for the benchmark apps, it takes GOALEXPLORER 13 seconds on average to reach a target activity and 17.6 seconds to reach a target statement, compared with 453.7 and 480.7 seconds for Sapienz and 643.3 and 632.4 seconds for Stoat, respectively. For the Google Play apps, GOALEXPLORER reaches the target activities in 26.3 seconds and target statements in 47.1 seconds, on average, compared with 619.5 and 701.3 seconds for Sapienz and 763.5 and 860.9 seconds for Stoat, respectively. Not surprisingly, GOALEXPLORER performs better than other static models – WTG (26.9 seconds for target activity and 57.0 seconds for statement) and WTG-E (27.2 seconds for target activity and 50.9 seconds for statement). Due to its more accurate representation of Android UI, GOALEXPLORER encounters less backtracking, thus spends less time to trigger



(a) Average time, in seconds, to reach an activity (lower is better).



(b) Average time, in seconds, to reach a *openConnection* (lower is better).

Figure 5.2: Performance results for the benchmark and GooglePlay apps.

Table 5.2: Performance results for the benchmark and GooglePlay apps.

(a) Average time, in seconds, to reach an activity (lower is better).

Category	Metrics	WTG	WTG-E	GoalExplorer	Sapienz	Stoat
Benchmark	*Min*	3	2.8	**2.1**	2	3
	Mean	12.1	12.2	**13**	453.7	634.3
	Median	9.3	8.9	**8.4**	235	362
	Max	40	46.2	**85.4**	2631	3539
Google Play	*Min*	5.3	4.4	**5.3**	2	2
	Mean	26.9	27.2	**26.3**	619.5	763.5
	Median	24.2	21.8	**21.7**	371	412.2
	Max	95.6	106.2	**76**	3196.8	3495.8

(b) Average time, in seconds, to reach a *openConnection* (lower is better).

Category	Metrics	WTG	WTGE	GoalExplorer	Sapienz	Stoat
Benchmark	*Min*	6.7	7.2	**7.6**	24.5	91.9
	Mean	16.5	17.4	**17.6**	480.7	632.4
	Median	15	16.1	**16.5**	205.2	563.4
	Max	35.1	38.3	**33**	1254.2	1282.2
Google Play	*Min*	2.4	2.4	**2.3**	2	2
	Mean	56	50.4	**47.1**	710.3	860.9
	Median	39.1	31	**32.5**	596.9	737.6
	Max	271.6	213.5	**190.4**	2720.9	2931.7

the targets.

Our experiments also show that it takes GOALEXPLORER around 93 seconds per app to build the static model. Even given this number, it outperforms dynamic app exploration tools: since our exploration is guided by a static model, GOALEXPLORER only performs the actions that lead to the target activity while Sapienz and Stoat, by design, aim to trigger various functionalities, not necessarily those leading to the target.

To answer RQ2: GOALEXPLORER can trigger exploration targets substantially faster than dynamic exploration tools, which shows its efficiency for the goal-driven exploration scenario.

5.2.3 RQ3 (Accuracy)

To evaluate the accuracy, the author of this thesis manually identified all paths leading to `URL.openConnection` statement in 36 apps, as discussed in Section 5.1. The analyzed apps contain 127 such statements in total (3.53 per app). Out of those, 102 are reachable (2.83 per app), 21 are in library methods that apps do not use (which occurs in 8 apps), and the remaining 4 are in unused app methods (all in the same app).

We analyze the accuracy of GOALEXPLORER along three directions:

- **RQ3.1 (FN):** What is the fraction of cases when the model does not contain transitions that exist in practice? Such cases are False Negatives (FNs) of our STG model.
- **RQ3.2 (FP-Dynamic)**: What is the fraction of cases when the model contains transitions that exist but cannot be triggered dynamically? Such cases are False Positives (FPs) not handled by dynamic exploration.
- **RQ3.3 (FP):** What is the fraction of cases when the model contains transitions that do not exist in practice? Such cases are False Positives (FPs) of the model.

In what follows, we discuss these cases one by one and also list their implications on the ability of GOALEXPLORER to reach the desired target.

RQ3.1 (FN): Model Does Not Contain the Correct Transitions

When a correct transition does not exist in the model, GOALEXPLORER cannot construct a complete path to the target and thus will not attempt to follow that path.

In our sample, that happen in eight cases in total, in five apps. The reasons for missing a transition (STG false negatives) are discussed below:

(I) Reflection: two transitions (in one app) are missing due to the limitation in handling complex reflective calls. GOALEXPLORER relies on call graph construction implemented by FlowDroid, which only handles reflective call targets for constant string objects.

(II) Unmodeled Component: two transitions (in two apps) are missing due to unmodeled components, such as `App Widgets`, which are miniature app views that can be embedded in other apps, e.g. in the home screen. GOALEXPLORER does not model these components as they are placed outside of the main app.

(III) Intent Resolution: four transitions (in two apps) are missing due to failures in resolving the targets of intents, e.g. which activity to be launch by the intent. GOALEXPLORER relies on a constant propagation technique from IC3 [57] to resolve the intents and identify the targets. When the constant propagation failed, the correct transitions are missing in STG.

GOALEXPLORER finds alternative paths to avoid missing the targets for four out of the eight missing transitions in STG; the remaining ones lead to four missed targets – two are due to reflection and two are consequences of unmodeled components. It should be noted that Sapienz and Stoat cannot trigger the latter two missed targets.

RQ3.2 (FP-Dynamic): Model Contains Correct Transitions That Cannot be Triggered

GOALEXPLORER encounters 44 correct transitions in 21 apps that it fails to trigger dynamically (2.1 transitions per app). The main causes for such failures (false positives of dynamic exploration) are discussed below:

(I) Event Order: in nine cases (in five apps, 1.8 transitions per app), a transition is only possible under a certain order of events in a screen. For example, in

the BookCatalogue app, the target can be reached only after the user marks certain electronic books as an anthology. While dynamic explorer is able to resolve many cases that require such interactions (17 cases got successfully resolved in our data set), the correct resolution is not always guaranteed.

(II) Semantic Inputs: in 17 cases (in 11 apps, 1.54 transitions per app), meaningful inputs (other than login credentials) are required explore the path to target. For example, a valid zip code is required to start the search in the Mileage app and an .mp3 file has to be selected for upload in AnyMemo. Neither GOALEXPLORER nor the contemporary dynamic exploration techniques can generate such semantic inputs.

(III) Remote Triggers: in 13 cases (in eight apps, 1.63 transitions per app), the transitions can only be triggered given a certain response from the remote server. For example, in SyncToPix, a target can only be triggered when the app receives a certain reply from the server, to syncing data. During our testing period, such reply was never received and none of the tools were able to trigger these transitions.

(IV) Widget Identification: in 5 cases (all from the same app), transitions cannot be triggered due to the failure in widget identification. GOALEXPLORER relies on UIAutomator [28] to identify widgets where the input events, e.g. clicks, should be applied. UIAutomator fails to locate the widget that does not have a resource identification assigned. When that happens, GOALEXPLORER delegates to the dynamic UI exploration, which randomly interacts with the screen to hit the correct widget without locating it by the resource identification.

Overall, the 44 transitions that GOALEXPLORER could not follow resulted in missing 11 targets in nine apps; GOALEXPLORER finds alternative paths to the remaining 33 targets. Sapienz and Stoat cannot trigger 10 of the missing targets either.

RQ3.3 (FP): Model Contains Transitions That Do Not Exist in Reality

GOALEXPLORER identifies spurious 42 transitions, in 27 apps (1.55 transitions per app), that do not exist in reality (STG false positives). Below, we analyze the causes for these false positives:

(I) Callback Sequence: in 28 cases (in 17 apps, 1.64 transitions per app) incorrect transitions in STG are caused by over-approximation in modeling the sequence of the callbacks. Since the order of callbacks cannot be predicted statically, GOALEXPLORER assumes that the callbacks can happen in any order (see *Screen Builder* in Section 4.2.2), producing screens in STG that are not feasible in practice.

(II) Fragment Properties: in 14 cases (in seven apps, 2.33 transitions per app) incorrect transitions are due to not analyzing detailed fragment properties. Apps can hide the fragments by setting its properties, i.e. changing the visibility level to `View.GONE`. We only model the set of fragments but do not track the properties of each fragment in our model, over-approximating the set of fragments on a screen, some of which are invisible in practice.

While GOALEXPLORER over-approximates the set of possible transitions when constructing STG, resulting in transitions that do not exist in reality, all these transitions were "ignored" during exploration by backtracking and selecting another path. In fact, none of them impacted the GOALEXPLORER's ability to triggering the targets. However, such spurious transitions, as well as transitions that cannot be triggered (RQ3.2: false positives of dynamic exploration), prolong the exploration as it takes time to resolve the false positives dynamically. Our results demonstrate that the execution time of the tool is not substantially affected by that and GOALEXPLORER still able to reach the target substantially faster than dynamic tools.

Overall, GOALEXPLORER fails to reach 16 targets in 11 apps due to false negatives (missing correct transitions) and false positives of dynamic exploration (being unable to follow the correct transition). The baseline dynamic app exploration tools

can reach two and four of these targets, for Sapienz and Stoat, respectively. At the same time, GOALEXPLORER can reach 14 and 6 additional targets that these tools cannot reach, respectively. We thus believe the results demonstrate that that the statically constricted model is accurate enough to facilitate the goal-exploration task.

5.2.4 Evaluation Summary

Overall, our experiments demonstrate that the combination of the static STG model and the run-time exploration techniques applied by GOALEXPLORER is valuable for the goal-driven exploration task. By using this techniques, tool is able to reach substantially more targets than existing techniques based on static models of an app UI. It is able to reach a comparable number of targets as the dynamic app exploration techniques, only substantially faster, which further attest to the quality and value of the statically-built model.

GOALEXPLORER is able to trigger 84.3% of all reachable targets, better than the state-of-the-art dynamic app exploration techniques (72.5% for Sapienz and 78.4% for Stoat). The main reasons for GOALEXPLORER's missed targets are the absence of a certain event sequence and the lack of meaningful inputs, e.g., files of a certain type. GOALEXPLORER encounters transitions that cannot be followed and backtracks 2.39 times per app, which does not substantially impact the exploration time; it still triggers the target substantially faster than the baseline dynamic tools.

Chapter 6

Discussion

6.1 Summary

In this thesis, we first defined the problem of goal-driven exploration for Android applications – directly triggering the functionality of interest in an application. We then discussed existing approaches, demonstrating that most of the existing approaches focus on systematically exercising the whole application rather than swiftly navigating to the target functionality. As these approaches often lack the knowledge of unexplored paths that are likely to lead to the functionality of interest, they are not applicable to the goal-driven exploration scenario.

We proposed a goal-driven exploration approach that statically constructs a UI model of the application to guide the dynamic exploration towards a particular target of interest: an Android activity, API call, or a program statement. We analyzed the existing UI models of Android applications to find that they do not provide an accurate representation for the goal-driven exploration, thus we presented a new UI model – the Screen Transition Graph (STG). We implemented a tool, called GOALEXPLORER, that constructs STG and uses it for goal-driven exploration. We empirically evaluated GOALEXPLORER on 93 benchmark applications and 95 the most popular GooglePlay applications. The evaluation result showed that the STG is substantially more accurate than other Android UI models and that

GOALEXPLORER is able to trigger a target functionality much faster than existing application exploration.

6.2 Limitations and Threats to Validity

6.2.1 Limitations

The main limitation of our approach is that it does not accurately model combinations of events on the same screen required to issue a transition. Our static UI model over-approximates the set of possible transitions, thus some collected transition paths might not be feasible in practice, due to such constraints, e.g., transitions requiring a certain combination of events or a specific system state. However, GOALEXPLORER incorporates the dynamic exploration logic to handle such cases at runtime, which has proven to be effective in our evaluation of the tool.

Although we extended our implementation to handle login screens, GOALEXPLORER cannot handle screens that require other types of semantic inputs, such as a zip code or a specific type of file. This limitation is common to many other automated exploration approaches, and is left for future work.

Another limitation of our approach is tied with the weakness of static analysis: GOALEXPLORER is not able to generate accurate UI models for apps that use native code, dependency injection, and complex reflective calls. As GOALEXPLORER relies on existing static analysis tools, FlowDroid and Soot, for constructing the static UI model of the app, the generated model might be incomplete when these tools produce inaccurate results. Such limitation is common for any static analysis tool. Although the model is not perfect, GOALEXPLORER demonstrates its usefulness by achieving even slightly higher coverage than existing dynamic automated testing techniques.

6.2.2 Threats to Validity

The **external validity** of our results might be affected by the selection of subject apps that we used and our results may not necessarily generalize beyond our subjects. Moreover, the benchmark F-Droid apps we used are from several years ago, thus they may not represent the latest apps in functionality and app design. We attempted to mitigate this threat by using “standardized" set of benchmark apps and by extending this set to include the 95 top Google Play apps. As we used most-popular real-world apps of considerable size, we believe our results will generalize for other apps.

For **internal validity**, our static analysis relies on FlowDroid to construct the callgraph and collect the callbacks. The validity of our results thus directly depends on the accuracy of that tool.

6.3 Future Research Directions

We envision two possible directions for future work:

(1) Tackling transitions that require specific semantic inputs, e.g. zip code, is an important challenge for future work. Recent work [44] proposed a Natural Language Processing (NLP) approach to effectively generates semantic textual inputs based on the textual contents of the screen. Integrating such learning algorithms could potentially improve GOALEXPLORER.

(2) Another possible direction for future work is to improve the accuracy of the statically-constructed STG. For example, GOALEXPLORER currently analyzes the set of fragments in each screen without modeling the properties of these fragments, leading to the incorrect transitions discussed in Section 5.2.3. Tracking the properties of fragments can help identify the fragments that are set invisible and exclude them from the model, hence producing a more accurate STG.

6.4 Conclusion

This thesis introduced GOALEXPLORER – a tool for goal-driven exploration of Android applications. Such a tool is useful for scenarios when a security auditor needs to automatically trigger a functionality of interest in an app. The main contributions of GOALEXPLORER are

(a) the static technique for constructing STG – a model that represents UI screens and transition between the screens, and
(b) an engine that uses STG to build an executable script that triggers the functionality of interest.

Our empirical evaluation shows that the STG model introduced in GOALEXPLORER is more accurate than other existing static UI models of Android applications. That is because it models advanced UI elements used in contemporary Android applications, including fragments, services, and broadcast receivers. Moreover, unlike earlier work that models UI components, STG represents the composition of these components to UI screens, which makes it more appropriate for targeted-exploration executions. Compared to dynamic application exploration and testing tools, GOALEXPLORER relies on the statically-build model of an application, which allows it to reach the target substantially faster.

As future work, GOALEXPLORER can be extended to support semantic textual inputs, which can be done using NLP. Improving the accuracy of statically-produced STG, e.g., by tracking the properties of fragments, is another possible extension of this work.

Bibliography

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI Ripping for automated testing of Android applications. In *Proc. of ASE'2012*, pages 258–261, 2012.

[2] Anonymous. Supplementary materials. anonymized for the review. https://anonymous-review-submission.github.io/goalexplorersite/, 2019. (Last accessed: Jan 2019).

[3] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight static analysis for GUI testing. In *Proc. of ISSRE'2012*, pages 301–310, 2012.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. of PLDI'2014*, pages 259–269, 2014.

[5] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. of OOPSLA'2013*, pages 641–660, 2013.

[6] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, pages 50:1–50:39, 2018.

[7] K. Benjamin, G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut. A strategy for efficient crawling of rich internet applications. In *Proc. of ICWE'2011*, pages 74–89, 2011.

[8] T. Bray. Fragments for all. https://android-developers.googleblog.com/2011/03/fragments-for-all.html, 2011. (Last accessed: Jan 2019).

[9] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu. StoryDroid: Automated generation of storyboard for Android apps. In *Proc. of ICSE'2019*, pages 596–607, 2019.

[10] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proc. of OOPSLA'2013*, pages 623–640, 2013.

[11] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *Proc. of ASE'2015*, pages 429–440, 2015.

[12] A. Dogtiev. App stores list 2018. http://www.businessofapps.com/guide/app-stores-list/, 2018. (Last accessed: Jan 2019).

[13] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. AJAX crawl: Making AJAX applications searchable. In *Proc. of ICDE'2009*, pages 78–89, 2009.

[14] F-Droid Limited. F-Droid. https://f-droid.org/en/, 2019. (Last accessed: Jan 2019).

[15] Facebook. Facebook login API. https://developers.facebook.com/docs/facebook-login/android/, 2019. (Last accessed: Jan 2019).

[16] Facebook. Facebook SDK for Android. https://developers.facebook.com/docs/android/, 2019. (Last accessed: Jan 2019).

[17] A. M. Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proc. of ISSRE'2013*, pages 278–287, 2013.

[18] D. Ferreira, V. Kostakos, A. R. Beresford, J. Lindqvist, and A. K. Dey. Securacy: An empirical investigation of Android applications' network usage, privacy and security. In *Proc. of WiSec'2015*, pages 11:1–11:11, 2015.

[19] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury. Android testing via synthetic symbolic execution. In *Proc. of ASE'2018*, pages 419–429, 2018.

[20] Google. Activity-lifecycle concepts. https://developer.android.com/guide/components/activities/activity-lifecycle, 2019. (Last accessed: Jan 2019).

[21] Google. Android debug bridge (ADB). https://developer.android.com/studio/command-line/adb, 2019. (Last accessed: Apr 2019).

[22] Google. Android SDK tools release note. https://developer.android.com/studio/releases/sdk-tools, 2019. (Last accessed: Jan 2019).

[23] Google. App components. https://developer.android.com/guide/components/fundamentals, 2019. (Last accessed: Jan 2019).

[24] Google. Espresso. https://developer.android.com/training/testing/espresso/, 2019. (Last accessed: Jan 2019).

[25] Google. Fragment. https://developer.android.com/reference/android/app/Fragment, 2019. (Last accessed: Jan 2019).

[26] Google. Menus. https://developer.android.com/guide/topics/ui/menus, 2019. (Last accessed: Jan 2019).

[27] Google. Monkey. https://developer.android.com/studio/test/monkey, 2019. (Last accessed: Jan 2019).

[28] Google. UI Automator. https://developer.android.com/training/testing/ui-automator, 2019. (Last accessed: Jan 2019).

[29] Google. Webview. https://developer.android.com/reference/android/webkit/WebView, 2019. (Last accessed: May 2019).

[30] O. W. Group. The OAuth 2.0 authorization framework: Bearer token usage. https://tools.ietf.org/id/draft-ietf-oauth-v2-bearer-23.xml, 2019. (Last accessed: Apr 2019).

[31] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su. Practical GUI testing of Android applications via model abstraction and refinement. In *Proc. of ICSE'2019*, pages 269–280, 2019.

[32] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proc. of MobiSys'2014*, pages 204–217, 2014.

[33] M. Henze, J. Pennekamp, D. Hellmanns, E. Mühmer, J. H. Ziegeldorf, A. Drichel, and K. Wehrle. CloudAnalyzer: Uncovering the cloud usage of mobile apps. In *Proc. of the MobiQuitous'2017*, pages 262–271, 2017.

[34] O. Hou. A look at Google bouncer. https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/, 2012. (Last accessed: Jan 2019).

[35] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proc. of EuroSys'2014*, pages 18:1–18:15, 2014.

[36] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. SUPOR: Precise and scalable sensitive user input detection for Android apps. In *Proc. of USENIX Security 2015*, pages 977–992, 2015.

[37] IBM. Identify and remediate application security vulnerabilities with IBM application security. https://www.ibm.com/security/application-security/appscan, 2019. (Last accessed: Jan 2019).

[38] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. of ISSTA'2013*, pages 67–77, 2013.

[39] R. Johnson and A. Stavrou. Forced-path execution for Android applications on x86 platforms. In *Proc. of SERE-C'2013*, pages 188–197, 2013.

[40] M. Junaid, D. Liu, and D. Kung. Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered lifecycle models. *Computers & Security*, 59:92 – 117, 2016.

[41] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In *Proc. of FSE'2016*, pages 410–421, 2016.

[42] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2019.

[43] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proc. of ICSE'15*, pages 280–291, 2015.

[44] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *Proc. of ICSE'17*, pages 643–653, 2017.

[45] Y. Liu, H. H. Song, I. Bermudez, A. Mislove, M. Baldi, and A. Tongaonkar. Identifying personal information in Internet traffic. In *Proc. of COSN'2015*, pages 59–70, 2015.

[46] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proc. of ESEC/FSE'2013*, pages 224–234, 2013.

[47] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *Proc. of FSE'2014*, pages 599–609, 2014.

[48] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *Proc. of ISSTA'2016*, pages 94–105, 2016.

[49] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins. The first decade of GUI ripping: Extensions, applications, and broader impacts. In *Proc. of WCRE'2013*, pages 11–20, 2013.

[50] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Trans. Internet Technol.*, pages 378–419, 2004.

[51] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, pages 3:1–3:30, 2012.

[52] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. SIG-Droid: Automated system input generation for Android applications. In *Proc. of ISSRE'2015*, pages 461–471, 2015.

[53] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.

[54] K. Moran, L.-V. Mario, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *Proc. of ICST'2016*, pages 33–44, 2016.

[55] R. Naga. AXML Printer. https://github.com/rednaga/axmlprinter, 2018. (Last accessed: Jan 2019).

[56] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. Autoflox: An automatic fault localizer for client-side JavaScript. In *Proc. of ICST'2012*, pages 31–40, 2012.

[57] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *Proc. of ICSE'2015*, pages 77–88, 2015.

[58] A. Parecki. An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. https://oauth.net/, 2019. (Last accessed: Apr 2019).

[59] Privacy International. How apps on Android share data with facebook. https://privacyinternational.org/report/2647/how-apps-android-share-data-facebook-report, 2018. (Last accessed: Jan 2019).

[60] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and controlling PII leaks in mobile network traffic. In *Proc. of MobiSys'2016*, pages 361–374, 2016.

[61] RobotiumTech. Robotium. http://www.robotium.org, 2016. (Last accessed: Jan 2019).

[62] J. Schütte, R. Fedler, and D. Titze. ConDroid: Targeted dynamic analysis of Android applications. In *Proc. of AINA'2015*, pages 571–578, 2015.

[63] S. Staiger. Reverse engineering of graphical user interfaces using static analyses. In *Proc. of WCRE '2007*, pages 189–198, 2007.

[64] Statista. Number of apps available in leading app stores. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2018. (Last accessed: Jan 2019).

[65] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based GUI testing of android apps. In *Proc. of FSE'2017*, pages 245–256, 2017.

[66] Tencent. Tencent Kingkong app scan. https://service.security.tencent.com/kingkong, 2019. (Last accessed: Jan 2019).

[67] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proc. of ICSE'2013*, pages 162–171, 2013.

[68] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie. An empirical study of Android test generation tools in industrial cases. In *Proc. of ASE'2018*, pages 738–748, 2018.

[69] Y. Wang and A. Rountev. Profiling the responsiveness of Android applications via automated resource amplification. In *Proc. of MOBILESoft'2016*, pages 48–58, 2016.

[70] M. Wong and D. Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proc. of NDSS'2016*, pages 21–24, 2016.

[71] H. Wu, S. Yang, and A. Rountev. Static detection of energy defect patterns in Android applications. In *Proc. of CC'2016*, pages 185–195, 2016.

[72] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time Android application auditing. In *Proc. of SP'2015*, pages 899–914, 2015.

[73] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *Proc. of ASE'2015*, pages 658–668, 2015.

[74] Zedge. ZEDGE. https://play.google.com/store/apps/details?id=net.zedge.android, 2019. (Last accessed: Jan 2019).

[75] A. Zeller. Mining apps for anomalies. In *Proc. of SoftwareMining'2016*, ASE'2016 Workshop, pages 1–1, 2016.

[76] H. Zhang, H. Wu, and A. Rountev. Automated test generation for detection of leaks in Android applications. In *Proc. of AST'2016*, pages 64–70, 2016.

[77] L. L. Zhang, C.-J. M. Liang, Z. L. Li, Y. Liu, F. Zhao, and E. Chen. Characterizing privacy risks of mobile apps with sensitivity analysis. *IEEE Transactions on Mobile Computing*, 17(2):279–292, 2017.

[78] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. of ISSTA'2011*, pages 353–363, 2011.

[79] Y. Zhang, Y. Sui, and J. Xue. Launch-mode-aware context-sensitive activity transition analysis. In *Proc. of ICSE'2018*, pages 598–608, 2018.