UNIVERSITY OF CALIFORNIA,
IRVINE

Automated Assistive-Service Driven Accessibility Testing for Mobile Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Navid Salehnamadi

Dissertation Committee:
Professor Sam Malek, Chair
Associate Professor Jim Jones
Assistant Professor Stacy Branham

2022

DEDICATION

To my dear aunt, Sima Joun– With extraordinary abilities to love and be kind, unconditionally

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I express my gratitude to my advisor, Professor Sam Malek, for his guidance and support. He believed in me when I was at my lowest and inspired me when I needed it the most.

I want to thank the other committee members, Professors James Jones, and Stacy Branham, for their valuable feedback on my dissertation. I appreciate professors Joshua Garcia, Cristina Lopes, and Iftekhar Ahmed for their help and guidance during this journey. I also enjoyed collaborating with other members of the SEAL lab at UCI, especially Forough Mehralian, Abdulaziz Alshayban, Dr. Jun-wei Lin, Negar Ghorbani, Syed Fatiul Huq, and Ziyao He.

This dissertation was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation. Also, I would like to thank Sigma Xi, Noyce Initiative, Bob and Barbara Kleist, and UCI Grad Division for partly supporting my research financially.

I am grateful for my friends who were beside me and supported me in tough times, from the beginning of my journey as an international Ph.D. student to the day I defended my dissertation. In particular, I would like to thank my dear friend, Dr. Maryam Asghari, who was a great source of inspiration and help.

I am thankful for my lovely family, who always support me. Being far from Maman Ozra, Sima Joun, Baba Reza, and Maman over the past five years was difficult, but they constantly showed their love and support, which was a relief in helping me to stay motivated and strong. I am grateful for having Saeed in my life, who is more than a brother to me.

Finally, I do not know how to express my deepest gratitude and love toward Yasaman, my beautiful wife. It is hard to imagine how I could get through all obstacles and difficulties without her love, support, and wisdom. I was fortunate to have her by my side in past challenges, and I am happy to be with her in unforeseeable stages of our life.

VITA

Navid Salehnamadi

EDUCATION

Doctor of Philosophy in Software Engineering — **2022**
University of California, Irvine — *Irvine, California*

Master of Science in Software Engineering — **2017**
Sharif University of Technology — *Tehran, Iran*

Bachelor of Science in Computer Engineering — **2015**
University of Tehran — *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2017-2022**
University of California, Irvine — *Irvine, California*

Research Intern — **Summer 2021**
Microsoft — *Seattle, Washington*

Graduate Research Assistant — **2015-2017**
Sharif University of Technology — *Tehran, Iran*

TEACHING EXPERIENCE

Teaching Assistant — **2018-2019**
University of California, Irvine — *Irvine, California*

Teaching Assistant — **2017**
Sharif University of Technology — *Tehran, Iran*

Teaching Assistant — **2013-2015**
University of Tehran — *Tehran, Iran*

REFEREED JOURNAL PUBLICATIONS

ROUTE: Roads Not Taken in UI Testing **2022**
Transactions on Software Engineering and Methodology (TOSEM)

DELTADROID: Dynamic Delivery Testing in Android **2022**
Transactions on Software Engineering and Methodology (TOSEM)

REFEREED CONFERENCE PUBLICATIONS

Assistive-Technology Aided Manual Accessibility Testing in Mobile Apps, Powered by Record-and-Replay **April 2023**
(Under Review) 2023 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)

Groundhog: An Automated Accessibility Crawler for Mobile Apps **October 2022**
37th International Conference on Automated Software Engineering (ASE)

Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps **October 2022**
37th International Conference on Automated Software Engineering (ASE)

Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps **Aug 2021**
2021, The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android **May 2021**
2021, ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)

Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study **September 2020**
35th International Conference on Automated Software Engineering (ASE)

ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android **September 2020**
35th International Conference on Automated Software Engineering (ASE)

A Benchmark for Event-Race Analysis in Android Apps (Poster) **June 2020**
18th International Conference on Mobile Systems, Aplications, and Services (MobiSys)

ABSTRACT OF THE DISSERTATION

Automated Assistive-Service Driven Accessibility Testing for Mobile Applications

By

Navid Salehnamadi

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2022

Professor Sam Malek, Chair

For 15% of the world population with disabilities, accessibility is arguably the most critical software quality attribute. The ever-growing reliance of users with disability on mobile apps further underscores the need for accessible software in this domain. Manual accessibility testing with assistive services is a high-fidelity form of testing; however, it is time-consuming and requires deep knowledge of various aspects of accessibility. Existing automated accessibility assessment techniques primarily aim to detect violations of predefined guidelines, thereby often overlook the way software is actually used by users with disability, i.e., with assistive services. Since disabled users, especially the ones with motor or visual impairments, are heavily reliant on assistive services in interacting with apps, many important cues are missed when these services are not considered in the evaluation of an app's accessibility.

This dissertation proposes a three-pronged approach to advance accessibility testing for mobile applications by including assistive services in the evaluation process. In the first prong, I introduce a new technique to extract main use cases of the software from the existing software tests, then re-execute them from the standpoint of users with disabilities with various assistive services. In the second prong, I introduce a completely automated way to crawl applications using assistive services to detect accessibility issues at runtime. Finally, in the third prong, I introduce a semi-automated technique to aid manual accessibility testers to efficiently evaluate applications with various assistive

services. To show the feasibility of these ideas, this dissertation particularly presents and proposes automated and semi-automated tools implemented for Android platform, namely for each prong (1) LATTE, (2) GROUNDHOG, OVERSIGHT, and (3) A11YPUPPETRY.

All conducted experiments on real-world subject apps corroborate the effectiveness and efficiency of the proposed approaches, and their ability to detect various types of accessibility issues in mobile applications.

Chapter 1

Introduction

Mobile applications (apps) are permeating every aspect of the daily life of billions of people around the world, from personal banking to communication, transportation, and more. The ability to access and use these apps with ease is vital for everyone, especially for approximately 15% of the world population with some form of disability [124]. For visually-related disabilities alone, there are 217 million people worldwide with moderate to severe visual impairment and 36 million who are completely blind [2]. However, recent studies have shown accessibility issues are prevalent in mobile apps, hindering their use by users with disability [106, 7, 36].

Technology institutes such as World Wide Web Consortium and companies such as Apple and Google encouraged developers to test the accessibility of their software applications either manually or automatically. Since there is a large range of disabilities with varying severity, it may not be possible for a development team to manually test an app through user evaluation. Moreover, due to time and budget constraints, such manual approaches often result in insufficient evaluation [115]. Relying on manual evaluation also makes it challenging to re-evaluate new releases of apps, which may frequently occur due to short release cycles, changing requirements, and rapidly evolving technologies [33].

To date, various automated accessibility analysis techniques have been proposed to deal with the widespread prevalence of accessibility issues [13, 9, 27, 28]. Common across all these tools is the way they aim to identify accessibility issues in terms of predefined rules derived from accessibility guidelines. For instance, whether a label for an icon is missing, whether there is sufficient contrast between text and background, whether the actionable elements are too close to each other, etc. While it is important for developers to follow these guidelines in the construction of their apps, the rules by themselves are not able to precisely determine the difficulties a user with disability may experience. For example, from a disabled user's standpoint, there is a significant difference between accessibility issues affecting the main functionalities of an app versus those affecting its incidental functionalities (e.g., advertisement banners, copyright disclaimers), yet the existing techniques provide no effective means of distinguishing between the two. Prior studies [7] have shown the developers tend to either not utilize or simply ignore the results of existing accessibility analysis tools, because they produce a massive amount of accessibility warnings, many of which are minor, or simply wrong.

Another limitation of the existing automated accessibility analysis techniques is that none consider the assistive services such as *TalkBack* (a screen reader for Android users with blindness or visual impairment) or *SwitchAccess* (an Android service for navigating app for users with motor impairment) in their analysis. Since disabled users are heavily reliant on assistive services in interacting with apps, many important cues are missed when these services are not considered in the evaluation of an app's accessibility. For instance, a screen with a dynamic user interface (UI) may have no apparent accessibility issue in the implementation of its individual elements, yet be completely unusable by a disabled user due to the assistive service's inability to detect the changes in UI.

The key insight that guides our research is that a high-fidelity form of evaluating accessibility needs to reflect the way disabled users actually interact with apps, i.e., using the assistive services. Another key insight comes from the huge body of work on automated Graphical User Interface (GUI) test

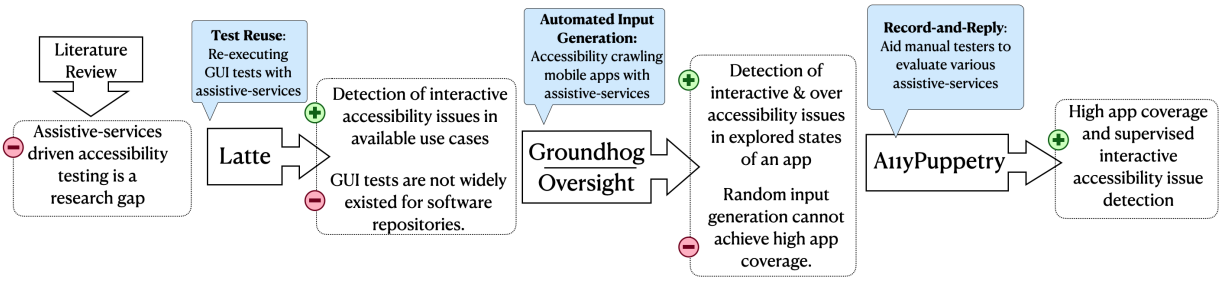


Figure 1.1: Overview of this dissertation

input generation techniques in the past decade, including test reuse [78, 30, 85], automated input generation [55, 82, 83, 61, 76, 123, 116, 32, 8, 29, 94, 77], and record-and-reply techniques [101, 60, 107, 74, 50, 59, 63, 75]. Although these techniques aim to evaluate the functionality of apps' GUI, they only consider the way users without disabilities interact with an app, e.g., performing touch gestures on a screen.

Figure 1.1 depicts an overview of the contribution of this dissertation. Informed by the above-mentioned insights, the first prong of this dissertation introduces a new form of automated accessibility analysis, called LATTE, that builds on the way developers already validate their apps for functional correctness. A widely adopted practice in software development is for developers to write system tests, often in the form of GUI test scripts, to validate the important use cases (functionalities) of an app for correctness. These use cases are the important functionalities of an app that should also be accessible. Given an app under test and a set of regular GUI tests (written by developers) as input, LATTE first extracts a *Use-Case Specification* corresponding to each test. A Use-Case Specification defines the human-perceivable steps a test takes to exercise a particular functionality in an app. LATTE then executes the Use-Case Specification using an assistive service, i.e., TalkBack and SwitchAccess.

The effectiveness of LATTE heavily depends on the availability of GUI tests for validating the functionalities of the app under test. Therefore, LATTE may not be suitable for software without GUI tests. Studies show that more than 92% of Android app developers do not have any GUI test for their apps [79]. Even if GUI tests are available for proprietary apps, the test cases are rarely

available to the public or app store operators that may want to assess the accessibility of apps for users. Furthermore, GUI tests may fail to achieve good coverage, making their approach ineffective at finding accessibility issues in uncovered parts of the app under test.

To address the aforementioned limitations of LATTE, we have developed a fully automated approach, a prototype called GROUNDHOG in Android, for validating the accessibility of Android apps that replicates the manner in which disabled users actually interact with apps, i.e., using assistive services. GROUNDHOG gets the app in a binary form, i.e., APK, and installs it on a Virtual Machine (VM). It utilizes an app crawler to explore a diverse set of screens to be assessed. For each screen, GROUNDHOG extracts all the possible actions and executes the same action with different interaction models, including different assistive services, to validate if the app is accessible. Moreover, powered by GROUNDHOG we studied over-accessibility and its threats, enabling an assistive-service user to get access to app content or functionality that is not available otherwise. We also studied the characteristics of overly accessible elements and proposed OVERSIGHT to automatically detect them in mobile apps with high accuracy.

Nevertheless, the most reliable method of validating an app's accessibility is through user evaluation [115]. Besides the fact that GUI tests may not be available or the lack of app coverage for automated app crawlers, many of accessibility issues require human judgement. For example, automated tools like Accessibility Scanner can report issues like missing speakable text; however, they do not detect any issue when the provided alternative text is not descriptive or relevant to the intended functionality of corresponding GUI elements. However, finding users with different types of disability and conducting such evaluations can be prohibitively difficult, especially for small development teams with limited resources. On the other hand, exploring and testing an app with conventional interaction method, e.g., use touch screen in a mobile device, is quite an easy task for most software developers and testers. For the last piece of this dissertation, I designed and built a semi-automated tool to assist manual testers, even the ones with virtually no knowledge on accessibility, to asses the accessibility different functionalities of an app from the standpoint of users

Table 1.1: Potential stakeholders for each part of the dissertation

Chapter	Content	Stakeholders
5	LATTE	app developers, app testers, researchers
6	GROUNDHOG	app developers, app testers, researchers, app stores, users
7	OVERSIGHT	app developers, app testers, researchers, app stores, security analyst
8	A11YPUPPETRY	app developers, app testers, researchers, users

with different disabilities. A prototype has been implemented for Android, called A11YPUPPETRY, which utilizes record-and-reply techniques. Given a human tester and a device, A11YPUPPETRY records all interactions of the tester on the device and replies it in another device with an assistive service, like screen reader. When the exploration is finished, the human tester can review a full accessibility report of the executed use case, including the captured video of the execution of the same use case for each assistive service.

1.1 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the prior related research and identifies the position of this work in the research landscape. Chapter 3 presents the research problem and the scope of this thesis. Chapter 4 examines different ways of interaction with a mobile device and introduces Proxy Users, a software app that can automatically perform an action with an assistive service. Chapter 5 will present my solution for re-executing GUI test with assistive services (LATTE), then Chapter 6 will introduce our completely automated accessible crawler (GROUNDHOG). Next, Chapter 7 will discuss over accessibility issues and our automated solution for detecting them (OVERSIGHT). Chapter 8 will introduce our record-and-reply technique for assisting human testers to conveniently verify the explored functionalities with assistive services (A11YPUPPETRY). Finally, Chapter 9 concludes the dissertation with future work.

Each part of this dissertation may be of interest to different groups of readers. Table 1.1 lists the potential stakeholders for each chapter of this dissertation.

The research presented in this dissertation has been published in or submitted to the following venues:

- Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek, Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android, 2021 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), Yokohama, Japan, May 2021 [111].

- Navid Salehnamadi*, Forough Mehralian*, and Sam Malek, Groundhog: An Automated Accessibility Crawler for Mobile Apps, 2022 37th IEEE/ACM, International Conference on Automated Software Engineering (ASE) [113].

- Forough Mehralian*, Navid Salehnamadi*, Syed Fatiul Huq, and Sam Malek, Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps, 2022 37th IEEE/ACM, International Conference on Automated Software Engineering (ASE) [88].

- Navid Salehnamadi, Ziyao He, and Sam Malek, Assistive-Technology Aided Manual Accessibility Testing in Mobile Apps, Powered by Record-and-Replay, Submitted to review in 2023 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), Hamburg, Germany, April 2023.

In addition, the following publications are not included in the dissertation but are related:

- Jun-Wei Lin, Navid Salehnamadi, and Sam Malek, ROUTE: Roads Not Taken in UI Testing, ACM Transactions on Software Engineering and Methodology (TOSEM) [80].

- Negar Ghorbani, Reyhaneh Jabbarvand, Navid Salehnamadi, Joshua Garcia, and Sam Malek, DeltaDroid: Dynamic Delivery Testing in Android , ACM Transactions on Software Engineering and Methodology (TOSEM) [49].

- Forough Mehralian, Navid Salehnamadi, and Sam Malek, Data-driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps, ESEC/FSE 2021, the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, August 2021 [89].

- Jun-Wei Lin, Navid Salehnamadi, and Sam Malek, Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study, 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual Event, Australia, September 2020 [79].

- Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed and Sam Malek, ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android, 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual Event, Australia, September 2020 [109].

- Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed and Sam Malek, A benchmark for event-race analysis in android apps, Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys), Virtual Event, June 2020 [108].

Chapter 2

Related Work and Research Gap

The purpose of this chapter is to provide an overview of the related work which forms the basis of the proposed research. In particular, it discusses previous studies on accessibility and GUI testing techniques. Furthermore, it identifies the research gap and situates the proposed research within the literature.

2.1 Related Work

This section discusses prior work on accessibility evaluation in software development, particularly for the Android platform. It first reviews the accessibility guidelines and standards that are the basis of existing accessibility evaluation techniques, followed by previous studies and tools on automated accessibility testing. Finally, it discusses existing GUI test generation techniques since they are the building blocks of the proposed tools in this dissertation.

2.1.1 Accessibility Standards and Guidelines

Accessibility can be defined as the degree to which people with disabilities easily use an app or a website. The more people with different types of disabilities can reach and interact with the app, the more accessible it is.

There are several accessibility standards used across the world to help promote and implement accessibility, including Web Content Accessibility Guidelines (WCAG 2.0 and 2.1), a well-recognized and general accessibility standard published by the Web Accessibility Initiative (WAI). Other accessibility standards, e.g., U.S. Revised Section 508 standard [56], provide a variety of recommendations to provide better support for individuals with different kinds of disabilities, including motor, hearing, and visual impairment.

Developers of popular mobile platforms such as Google and Apple also provide their own set of developers accessibility guidelines, which aim to provide better support for developers in making their apps more accessible [15, 26]. These guidelines are based on WCGA but are platform-specific and provide more technical details on how developers can remove common accessibility barriers in the platform.

2.1.2 Accessibility Testing

In general, automated accessibility testing techniques evaluate app compliance with accessibility guidelines [121] using static or dynamic analysis approaches [115]. Static analysis approaches such as Lint [20] identify accessibility violations in the source code upon compilation. Thus, they are not able to detect issues that can be detected at runtime. To mitigate their limitations, dynamic analysis techniques are proposed to analyze the runtime attributes of rendered UI components on the screen. Google accessibility Scanner [9] and other tools that are built on top of Accessibility Testing Framework [61, 65, 45] take a single app screen from the developers to run their tests and

report issues such as small touch target size or duplicate name issues. The capabilities of these tools are limited to a small number of issues that were supported by accessibility guidelines that are found to only cover around 50% of the issues [100]. Thereby, they are not able to detect issues that manifest themselves in interactions with apps. This limitation, similarly, exists for enhanced dynamic techniques that evaluate the same accessibility rules but replace the developers' effort in exploring an app with a crawler [7, 45] or provide the ability to write app exploration scenarios in the form of GUI tests [17, 104].

After the publication of my first work on using assistive services to detect accessibility issues, Latte [111], a few other works consider assistive services for accessibility testing [37, 5]. The most relevant to this thesis is Alotaibi et al. [5], which utilizes TalkBack to identify accessibility issues, such as unfocusable elements. However, this tool is limited to a single app screen and fails to detect accessibility issues related to actions such as clicking and typing.

2.1.3 Automated GUI Test Generation

The most common way to create GUI test cases is for a software developer to manually write the test scripts using frameworks like Espresso [17] or Appium [25]. Because of developers' expensive effort to write GUI tests, automated GUI test generation has become an active research area.

Most automated test generators use three main strategies: random, model-based, and targeted. Monkey [55], the state of the practice on app crawling, utilizes the random strategy to send random events regardless of the state of the app under test. DynoDroid [82] has a more sophisticated approach than Monkey by filtering out unacceptable events. Sapienz [83]'s goal is to increase coverage by searching for test sequences with a genetic algorithm. The model-based strategy is adapted in many testing tools [61, 76, 123], which analyze the app and build a GUI model to create test cases. The model-based techniques can be optimized in run-time to optimize the testing approach, like Stoat [116]. Alternatively, they can use the knowledge embedded in other apps, like

DroidMate [32], to better understand different UI states. Some parts of the app cannot be accessed except with unique and consistent input in long sequences of events. The targeted strategy tries to mitigate this problem. These testing tools [8, 29, 94] usually use data flow analysis or symbolic execution to generate a sequence of events with proper inputs to reach the target states. Other types of GUI test generation techniques like Humanoid [77] or DeepGUI [42] utilize a deep learning technique and associate GUI events to the GUI visual information.

Besides the manual or complete GUI test generation techniques, there are other techniques that are automated but require some forms of manually produced inputs to generate GUI test cases. For example, test reusing techniques like [78, 30, 85] analyze the existing GUI test cases from similar apps or other platforms and generate test cases for the source app.

Another area of semi-automated GUI test generation is record-and-reply techniques that assist software developers or testers to generate GUI test scripts by recording the user interactions with the app. RERAN [50], appetizer [24], Mosaic [60], and Orangutan[73] rely on the Linux Kernel for recording and replaying events. For example, RERAN requires a rooted device and employs the ADB command **getevent** and **sendevent** to record and replay events. For tools that rely on Linux Kernel, the captured events are low-level and hard to translate to high-level gestures that are understandable by assistive services. VALERA [62] has a high accuracy for recording and replying and can capture various events, such as network inputs. However, VALERA replies on a customized OS as it requires a modified Android system image, which imposes threats to its application.

Mobiplay [101], Espresso[17], Barista [46], Robotium[105], Culebra[92], Ranorex[102], SARA [59], RANDR [107], Sugilite [75] rely on the application layer to capture inputs. Mobiplay utilizes client-server architecture. The client and target apps run on an Android device and a remote server, respectively. Mobiplay identifies the targeted node based on the screen coordinates during the replay stage. Nevertheless, Mobiplay is not publicly available to researchers. Espresso can record motion events via an attached debugger but requires the recorded app's source code. Barista is a cross-platform record-and-replay tool. However, Barista fails to record and replay on non-open-

source apps as it highly relies on the Espresso framework. Robotium can only capture widgets that are rendered by the app's main process, but usually, the apps will run several processes [59]. Culebra provides a desktop GUI for user recordings, and the widget that interacts with users is identified via the view hierarchy. The drawback of Culebra is that it causes a significant overhead while identifying the view hierarchy of the interacted widget. Ranorex can record interactions via instrumentation, but the instrumentation fails when it encounters apps that have a large size. SARA can record and replay several input sources via dynamic instrumentation, and the interaction can be recorded in the form of coordinates and widgets. Specifically, SARA will record the interaction coordinate at first and identify the corresponding widget information via the self-replay technique. Then, SARA employs an adaptive replay method to replay captured interactions on different devices. The drawbacks of SARA are the lack of a graphical user interface and the high reliance on the third-party dynamic instrumentation tool called Frida. Frida can not instrument classes that implement the Android Interface **android.text.Editable**, which will cause SARA to lose essential interactions during the recording. RANDR utilizes static and dynamic instrumentation to record and replay multiple input sources, including external non-deterministic sources such as random numbers. While RANDR can record and replay abundant input sources, it does not require administrative device privileges or access to the app source code. However, RANDR is not publicly available to researchers. In addition, as RANDR and SARA both utilize instrumentation to capture the events and interactions, the non-standard widgets such as **android.webkit.WebView** will be ignored. Current popular android apps implement WebView to display web contents as a part of an activity layout, so failing to identify WebView will make the recorder lose essential interactions during the recording. Sugilite is a publicly available android application for recording and replay that utilizes an overlay to intercept interactions, such as click and typing. Using an overlay enables Sugilite to capture various events and widgets, even non-standard widgets such as WebView. Users must confirm whether the identified interaction is correct for each being recorded. After confirmation, Sugilite will perform that interaction on behalf of users. However, it fails to recognize the node that is not clickable and will get stuck in the current window if the clicked node has accessibility issues.

2.2 Research Gap

The key research gap that we will address in this dissertation is the lack of considering assistive services for evaluating the accessibility of software applications. One end of the spectrum of accessibility testing is manual testing which uses assistive services to explore a software application. However, manual accessibility testing is expensive; moreover, the testers should have deep knowledge and expertise in using assistive services and understanding accessibility requirements (if they are not disabled). The other end of the spectrum is automated accessibility testing that can be done without any manual effort. However, they cannot detect accessibility issues manifested by interaction with assistive services and only check the compliance of a limited set of accessibility guidelines in fixed GUI states of the app under test. More specifically, we are interested in addressing three specific research challenges regarding assistive-service-driven accessibility testing in the related literature:

- **Reusing Existing GUI Tests.** Software developers write GUI tests to evaluate their applications' main functionalities automatically. However, this evaluation is limited to conventional interaction, e.g., touch-based interaction in mobile devices. It does not consider other alternative ways, i.e., assistive services like screen readers and switch access. The use cases embedded in GUI test cases may be executed with assistive services to assess the accessibility of the use cases.

- **Automated GUI Input Generation (App Crawling).** There are many automated GUI test generation techniques; however, they all use conventional interaction methods to crawl software applications. No prior work has studied the idea of crawling applications with assistive services to validate the accessibility of the apps under test.

- **Assisting Manual Accessibility Testing.** There are various record-and-replay techniques to assist manual testers in testing the functionality of mobile apps under different settings, like different display sizes or platforms. However, none of them have considered replaying the

recorded actions using assistive services to evaluate the accessibility of apps' functionalities.

Chapter 3

Research Problem

3.1 Problem Statement

Developers are obliged by law and expected by ethical principles to build accessible apps for users regardless of their abilities. However, prior studies reveal that many popular apps are shipped with some form of accessibility issues, hindering disabled users ability to interact with them [36, 106, 7]. Technology institutes such as World Wide Web Consortium and companies such as Apple and Google have published accessibility guidelines [121, 26, 15], and encouraged developers to use assistive services to evaluate the accessibility of their software applications. Beside the fact that manual testing is an expensive process in terms of time and human resources [115, 33], developers and testers usually do not have enough knowledge and expertise to use various assistive services to evaluate the accessibility [7]. There are accessibility analysis tools [13, 9, 27, 28] that automatically analyze the GUI of a fixed state of an app and detect accessibility issues. However, static assessment of UI specifications cannot reveal many critical accessibility issues that only manifest themselves in interacting with an app using assistive services, such as a screen reader. In short, the problem statement of this dissertation can be summarized as follows:

The accessibility of software application can be evaluated either manually or automatically. Although manual accessibility testing with assistive services is a high-fidelity form of evaluation, it is expensive and requires accessibility knowledge which developers/testers usually do not have. On the other hand, the existing automated accessibility testing techniques are fast and convenient. They generally rely on accessibility guidelines to detect accessibility issues in fixed states of software applications; however, they do not consider the actual way users with disabilities interact with software applications, i.e., assistive services, and fail to detect accessibility issues that are manifested at runtime by interacting with assistive services. Thereby, there is a demand by software developers for fast, intuitive, and high-fidelity accessibility testing tools.

3.2 Thesis Statement

Insight 0: A common theme among almost all GUI test generation techniques is that they target users without disabilities, i.e., users who can see the screen and can perform complex touch gestures.

Insight 0 is the guiding insight to motivate this dissertation. Inspired by automated and semi-automated GUI test generation techniques, the existing research gap in accessibility testing can be addressed by incorporating assistive services in the evaluation process to represent the interactions of users with various disabilities, e.g., with motor or visual impairments. In this context:

The goal of my research is to provide a set of automated and semi-automated techniques inspired by GUI test generation techniques that incorporate assistive services in the accessibility testing (1) to detect complicated accessibility issues that manifest themselves under particular modes of interaction, and (2) to assist and enlighten developers in finding and understanding accessibility issues efficiently.

3.3 Research Hypotheses

This section describes the insights and their corresponding hypotheses that guided the three prongs in this dissertation.

Insight 1.1: Software developers write GUI tests to evaluate the essential functionalities of the application.

Insight 1.2: The main functionalities of an app should be accessible using an assistive service.

Hypothesis 1: GUI tests can be automatically reused for detecting accessibility issues of software applications at runtime using assistive services.

To demonstrate the feasibility of this hypothesis, I will use Android as the platform and develop an automated tool, namely LATTE, that is able to analyze existing GUI tests written by developers, generate corresponding human-readable use cases, and execute them from the standpoint of users with visual or motor impairment who uses screen reader (TalkBack), and physical switches (SwitchAccess) respectively.

Insight 2: Many software repositories have zero or limited GUI tests that may not cover all functionalities of an app.

Hypothesis 2: It is possible to devise an automated GUI input generation technique targeting assistive services for validating the accessibility of software applications.

To verify the correctness of this hypothesis, I will present two completely automated tool, called GROUNDHOG and OVERSIGHT, that interacts with Android apps with and without assistive services by performing all actions in an app. While GROUNDHOG is mainly concerned about the functionalities that users with disabilities cannot perform, OVERSIGHT focuses on the dual of this problem, i.e., the functionalities provided to users with assistive services that cannot be accessed by users without assistive services.

Insight 3.1: Manual or exploratory testing is a reliable method of evaluating different aspects of an app.

Insight 3.2: Most software developers and testers lack knowledge of accessibility principles and challenges faced by disabled users.

Hypothesis 3: It is possible to devise a record-and-replay testing technique to guide the developers in both the construction of accessibility tests and their execution using assistive services.

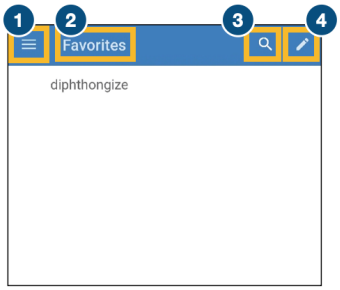
For the last hypothesis of this dissertation, I present a semi-automated tool, called A11YPUPPETRY, that observes and records the interaction of a human tester, who may not know how to work with assistive services, with an app, then replies the equivalent of those interactions with various assistant services on the same app. Finally, the human tester will receive a report of the execution of her exploration from the standpoint of users who use assistive services.

Chapter 4

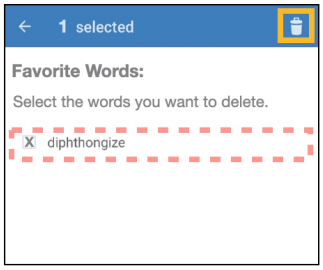
Proxy Users

This thesis's main idea is to automatically utilize assistive services to evaluate the accessibility of mobile applications. Therefore, knowing how users with disabilities use these services and how we can operate them programmatically is essential. This chapter first explains how users with different abilities may interact with a mobile device, what the actions are, and introduces *Proxy User*. A Proxy User is a software application that performs actions on behalf of users with or without disabilities who may use assistive services. For example, *TalkBack Proxy User* can click on an element by interacting with a device using TalkBack (the official screen reader in Android) to focus an element, then perform a double tap on the screen to click on the focused element.

Proxy Users are not designed to model actual users and how they think; instead, they model how users interact with a mobile device. Proxy Users are programs that communicate with a device and assistive services to perform a given action without understanding the app's semantics. In this thesis, Proxy Users work as the primary building block of the proposed automated approaches in the following chapters. All examples and implementation details of Proxy Users in this chapter are provided for the Android platform; however, the concept and idea of Proxy User can be implemented in other platforms like iOS if access to the related API is given.



(a)



(b)

```
<node class="android.widget.FrameLayout" ...>
    <node class="android.widget.LinearLayout" ...>
        <node class="android.widget.FrameLayout" ...>
            ...
        </node>
        <node class="android.widget.FrameLayout" ...>
            ...
            <node bounds="[980,10][1080,110]" class="android.widget.Button"
                clickable="true" clickableSpan="false"content-desc="Delete"
                enabled="true" focusable="true" focused="false" index="0"
                importantForAccessibility="true" long-clickable="false"
                package="com.dictionary" password="false"
                resource-id="com.dictionary:id/btn_delete"
                scrollable="false" text="" visible="true"/>
            ...
        </node>
        ...
    </node>
</node>
```

(c)

Figure 4.1: (a) The page showing all the words favorite by users, (b) The page after users tap on the edit button, (c) A part of the excerpted XML representation of UI structure in the dictionary app shown in subfigure (a)

The following section illustrates how users with or without disabilities utilize assistive services to interact with an app. Then, Section 4.2 briefly explains the essential backgrounds of Android and Accessibility API used for implementing Proxy Users. Section 4.3 formalizes different ways of performing actions with and without assistive services. Next, Section 4.4 defines four Proxy Users and explains how to use Accessibility API to control them. Finally, this chapter is concluded in Section 4.5.

4.1 Illustrative Example

This section illustrates how users with different abilities interact with apps in various ways.

Figure 4.1(a) shows a screen of the Dictionary.com app [43], which lists the favorite words and

lets the user edit the words. Assume a user wants to remove a word from favorites. A user without a disability who can see all elements on the screen and perform any touch gestures can perform this use case easily. First, she taps on the edit button, yellow-solid box 4 in Figure 4.1(a), then the navbar changes to depict the number of selected words, and the delete button, Figure 4.1(b). Next, the user selects the checkbox next to the word, and taps on the delete button, the yellow-solid box in Figure 4.1(b).

To perform the same use case, users with visual impairments, particularly blind users, have a completely different experience. They rely on screen readers, e.g., TalkBack for Android [18], to interact with the app. Users can perceive the screen's content by navigating through elements and listening to the textual description of the focused element by TalkBack. A common accessibility issue among mobile apps is the lack of content description for visual icons [36, 7]. For example, if the delete button in Figure 4.1(b) does not have a content description, a blind user cannot understand the functionality of this button. For the sake of this example, assume that this app does not have such issues and that all elements have an accurate textual description.

There are several ways of navigating the elements of an app with TalkBack. Using *Linear Navigation*, the user can navigate to the next and previous element of the currently focused element by swiping right and left on the screen. For example, to reach the edit button in Figure 4.1(a), the user can start from box 1 (top left icon) and navigate to the next elements until it reaches box 4.

Note that TalkBack may group elements for a more fluent announcement; for example, the red-dashed box in Figure 4.1(b) consists of a textual element and one checkbox. Secondly, the user can utilize *Jump Navigation* to focus on elements with specific types, e.g., buttons or edit-text boxes. For example, by jumping in button elements, the user can focus on boxes 1, 3, and 4 in Figure 4.1(a).

The third way is *Touch Navigation*, where the user touches different parts of the screen, and TalkBack focuses on the elements behind the user's finger. For example, if the user touches the top right of the screen in Figure 4.1(b), it focuses on the delete button, and TalkBack announces "Delete

Button".

Another way to locate an element is by searching the name. TalkBack user can enter the name of the element she is looking for, either by text entry or voice command, and TalkBack focuses on the element with the same text. For example, by searching "Edit" TalkBack focuses on box 4 in Figure 4.1(a).

To click on an element, the user should perform a double-tap gesture on the screen when the target element is focused. TalkBack perceives this gesture and sends a click accessibility event, *ACTION_CLICK*, to the focused button, which is the equivalent of tapping on the button by touch.

To sum up, a blind user needs to use TalkBack to first locate the edit button (with any navigation mode), then perform a double-tap, then locate the favorite word (red-dashed box in Figure 4.1(b)), double tap, and finally locate the delete button and press double tap. If she uses LinearNavigation, it requires at least 12 actions (9 swipes and 3 double-tap) compared to a user without a disability who only needs to perform 3 single taps.

On the other hand, a user with motor disabilities uses SwitchAccess [16] to navigate the app. SwitchAccess is an assistive service that enables users to interact with the device using a special keyboard with a limited set of buttons such as *Next* and *Select*. SwitchAccess highlights the focused element on the screen. The user uses the two buttons to change the focus to the next element or select the currently focused element. For example, to get to the edit button in Figure 4.1(a), the user needs to press the Next button several times, focusing on boxes 1, 3, and 4. Note that since the target user of SwitchAccess is assumed to be able to see the whole screen, SwitchAccess does not focus on textual elements, like box 2.

4.2 Background

This section provides a brief overview of User Interface (UI) components and accessibility support in Android to help the reader understand the material presented later.

4.2.1 Android UI

Android provides a variety of pre-built UI components, such as structured layouts and widgets, that allow developers to build the GUI of their app. The UI of an Android app is a single-root hierarchical tree where the leaf nodes are called *Views* or *Widgets* that users can see and interact with, e.g., buttons, text fields, and checkboxes. The non-leaf nodes, on the other hand, are invisible to the user. These non-leaf nodes are called *ViewGroups* or *Layouts* and are used for arranging or positioning the widgets.

Both *Widgets* and *Layouts* have variety of attributes. For example, the *content-desc* attribute is used by accessibility services to describe widgets without textual representation, or the *clickable* attribute shows if the widget is clickable. The UI hierarchy of a screen in an Android device can be retrieved as an XML file. Figure 4.1(c) shows part of the UI structure in the dictionary app. Lines 8-14 represent the delete button in Figure 4.1(b), where its attributes, such as clickable or content description, are represented.

XPath [122] (XML Path Language) is an expression language that supports various queries in XML documents. In particular, XPath can be used to identify nodes accurately using structural information. For example, the delete button in Figure 4.1(b) can be identified by its absolute path in XPath created by the class attribute as "/Framelayout/LinearLayout/FrameLayout[2]/Button" (the "android.widget" part is omitted from classes). Since the class of android widgets cannot be changed at runtime, the absolute path in XPath, or in short *apath*, is a reliable identifier of widgets in Android.

4.2.2 Accessibility in Android

Android provides an accessibility API for alternative modes of interacting with a device. It also offers several assistive services, including TalkBack, which is the official screen reader in Android and is built on top of the accessibility API. We briefly describe accessibility API in Android and how an assistive service like TalkBack or SwitchAccess can leverage this API.

The Android framework provides an abstract service, called *AccessibilityService*, to assist users with disabilities. The official assistive tools in Android, such as TalkBack, also implement this abstract service [11]. *AccessibilityService* works as a wrapper around an Android device interacting with it (performing actions on and receiving feedback from it).

Since these abilities may introduce security and privacy issues, an implementation of this service must specify the types of feedback it can receive and the actions it can perform. For example, TalkBack can receive feedback about all elements on the screen since it has *android:canRetrieveWindowContent* attribute in its specification. Moreover, it can perform actions, such as clicking, on elements; however, it cannot perform gestures, such as swiping on the screen, since the attribute *android:canPerformGestures* does not exist in TalkBack's specification.

The feedback is delivered to accessibility services through *AccessibilityEvent* objects. Accessibility services should implement the method *onAccessibilityEvent* to receive feedbacks in form of *AccessibilityEvent* objects. *AccessibilityManager* is a system-level service that monitors any changes in the device and manages accessibility services. When anything important happens on the device, *AccessibilityManager* creates an *AccessibilityEvent* object that describes the changes and passes it to *onAccessibilityEvent* method of accessibility services. The accessibility services can analyze the delivered event and may provide feedback to the user. For example, TalkBack announces the textual description of an element to the user when it is focused. An *AccessibilityEvent* object is associated with an *AccessibilityNodeInfo* object that contains the element's attributes. For instance, when a user clicks on delete button (Figure 4.1(b)), the system creates an *AccessibilityEvent* of type

TYPE_VIEW_CLICKED, which is associated with the *AccessibilityNodeInfo* object corresponding to the element shown in lines 8-14 in Figure 4.1(c).

Moreover, an *AccessibilityService* can access all GUI elements on the screen in the form of an *AccessibilityNodeInfo* object. An `AccessibilityNodeInfo` object not only represents the attributes of a GUI element on the screen, but also provides the ability to perform actions on the corresponding element. For example, an *AccessibilityService* can perform a click action on an *AccessibilityNodeInfo* by sending *ACTION_CLICK* event to it.

4.3 Actions

This section explains and formalizes how to interact with systems with and without assistive services. In particular, three ways of interaction for users with different abilities will be examined: touch gestures, TalkBack, and SwitchAccess.

4.3.1 Touch Gestures

Touch gestures are the primary interaction with most smartphone devices for users without visual or motor impairments. To have a complete and sound understanding of different ways of interaction, we used the official documentation of user interactions and touch gestures in Android [86, 22]. We categorized the standard touch gestures into several categories; however, the first and second categories are the most frequently used gestures.

- **PointGesture.** This is the most common way of interacting with a touch-based mobile device. To perform this type of gesture, the user uses one finger at a specific point on the screen without moving her finger to other parts of the screen. This type of touch gesture is identified as $\mathbb{PG}(t, p)$ where t is the type of the gesture, e.g., single-tap or long-press, and p is the coordinates of a point

on the screen.

- **LineGesture.** In this type of touch gesture, the user puts her finger on the screen and draws a line. The movement's velocity and starting point may lead to different behaviors. For example, suppose the user draws the line from the edge of the display. In that case, it is considered an edge swipe usually associated with system actions, e.g., going to the home screen or navigating back. This type of touch gesture is identified as $\mathbb{LG}(l, v)$ where l is a straight line on the screen and v is the velocity of the gesture, i.e., fast, regular, and slow.
- **TwoFingersLines.** The user uses her two fingers in these gestures to draw two separate lines. The most famous example of this category is pinching in and out (drawing two lines in toward each other or in opposite directions), which is usually associated with zooming in and out actions. This touch gesture is defined as $\mathbb{TFL}(l_1, l_2)$ where l_1 and l_2 are two straight lines on the screen which is swept by two fingers.
- **Miscellaneous.** These gestures are not commonly used in apps; however, app developers can use them to create alternative ways or shortcuts. For example, pinching in with three fingers to copy the selected text on iOS devices or double tapping with two fingers is equivalent to pinch-out for zooming out.

4.3.2 TalkBack Actions

We studied and examined the TalkBack documentation to understand TalkBack and its actions. Also, we followed official tutorials on TalkBack on Android devices and interacted with at least five popular Android apps.

TalkBack, when it is enabled, creates a virtual layer between the app and the user to enable users to perceive the UI without performing unintended actions. TalkBack draws an overlay on the screen, receives touch gestures, and translates these gestures into different actions. We categorized different ways of interactions into the following three categories:

- **ElementBased.** This type of interaction is mainly used to perceive the content of an element or perform a click or long-press on the focused element. TalkBack focuses on an element and announces its textual description. Given that the element is e and the type of the action is t, an ElementBased action can be defined as $\mathbb{EB}(t,e)$, meaning that the element e should be focused by TalkBack and action t, e.g., click, should be performed on the focused element. There are various ways to focus on an element that previously were mentioned in Section 4.1.
  - **LinearNavigation.** Users can change the focus to the next and previous element of the currently focused element. The actions associated with linear navigation are swiping right and left. The order of the next and previous elements is determined based on their position in the UI hierarchy. TalkBack may also perform a scroll action while navigating to the next or previous element if they are (partly) out of the screen.
  - **JumpNavigation.** Users can jump through elements of specific types for faster navigation by swiping up and down. For example, users can go to the next heading, paragraph, control, or link instead of navigating element by element. Moreover, users can adjust the announcements' granularity to make the content more accessible. For example, users can move to other lines, words, or characters instead of focusing on elements.
  - **SearchNavigation.** Users can search for a specific element on the screen with text or voice interface enabled by a three-finger long-press. It is similar to finding a specific word on a page in a text viewer/editor.
  - **TouchNavigation.** Users touch a spot on the screen, and TalkBack focuses on the underlying element in the same coordinates. This navigation method is usually used when the user has an estimation of the coordinates of the element she is looking for, e.g., the top or bottom menu. Another use case is when the other navigation methods cannot detect the element and the user has to conduct an exhaustive search to find all elements on the screen.
- **TouchGestureReplication.** Besides the click and long-press actions that ElementBased can do, users can replicate several other touch gestures, in particular, *LineGesture*s, by bypassing the

TalkBack overlay. A user can replicate scrolling, dragging, or edge swiping by swiping with two fingers when TalkBack is enabled. A TouchGestureReplication can be defined as $\mathbb{TGR}(lg)$ where lg is a LineGesture.

- **PredefinedActions.** Various actions that TalkBack can perform are not dependent on the app that the user is interacting with. For example, global actions, e.g., Home, Recent Apps, or Back, are not dependent on an app and can be performed with special gestures in TalkBack, e.g., swiping up then left will go to the home screen of the device. A PredefinedAction is $\mathbb{PA}(t)$ where t determines the action(s) to be performed, e.g., scroll forward or volume up.

4.3.3 SwitchAccess

Like TalkBack, we examined and studied SwitchAccess documentation and tutorials to understand its actions comprehensively. SwitchAccess is an assistive service in Android phone that utilize physical switches to interact with devices. One switch is assigned to "Next" and another switch is assigned to "Select". If the user can use more than two switches, other switches can be used for a better user experience, e.g., a switch can be assigned to change the focus to the previous element. We categorized the SwitchAccess actions into the following categories:

- **ElementBased.** Like TalkBack, the primary way to interact with an app is to locate an element and perform an action on it. By pressing the Next switch, SwitchAccess focuses on the next control element, e.g., button or edit textbox, and highlights the element. Note that since the target users of SwitchAccess are assumed not to have visual impairments, SwitchAccess does not focus on non-actionable elements, like TextView. When the element is selected, the user can select the element by pressing the Select switch. If an element is scrollable, a menu appears letting the user scroll in four directions by selecting the element.

- **PointScan.** In this mode, the user can click on a specific coordinate of the screen by sweeping a horizontal and vertical line from the edges of the screen. In this mode, the user can select

```
Path swipePath = new Path();
swipePath.moveTo(x_1, y_1);
swipePath.lineTo(x_2, y_2);
gestureBuilder.addStroke(new GestureDescription.StrokeDescription(swipePath, 0, 200));
GestureDescription gestureDescription = gestureBuilder.build();
accessibilityService.dispatchGesture(gestureDescription, callback, null);
```

Listing 4.1: A code snippet that performs a LineGesture on the screen using `AccessibilityService` API

the coordinate of the point by pressing the Select button when the horizontal/vertical line reaches their desired coordinates.

- **Menu.** Regardless of the app, SwitchAccess shows a floating Menu button to the user, which the user can select. The menu consists of predefined actions (similar to TalkBack), like Home, Back, or changing volumes.

4.4 Proxy User

A Proxy User is a software application, an implementation of *AccessibilityService*, that performs actions on behalf of users with or without disabilities who may use assistive services. Although implementations of *AccessibilityService*, such as TalkBack and SwitchAccess, are typically used for assisting users in interacting with the mobile device, these services can also be designed to cooperate with one another, as we have done here. This section explains four Proxy Users: Touch, TalkBack, SwitchAccess, and Abstract Proxy Users.

4.4.1 Touch Proxy User

This Proxy User interacts with the system from the standpoint of users without disabilities. These users do not use any assistive service and see the elements depicted on the screen to locate them. Recall that from Section 4.3, a touch gesture can be a PointGesture, LineGesture, TwoFingersLines, and Miscellaneous. Given a touch gesture, the Touch Proxy User performs the gesture using APIs

provided by Accessibility API. For example, to perform a Line Gesture $\mathbb{LG}(l, v)$, where l (line) is (x_1, y_1, x_2, y_2) and v (velocity) is fast (the actual value is different based on displays), Listing 4.1 shows a code snippet performing this gesture on the screen.

4.4.2 TalkBack Proxy User

TalkBack Proxy User interacts with the device using TalkBack, a screen reader, from the standpoint of blind users or users with visual impairments. As mentioned before, Proxy Users communicate with other components in an Android device to interact with an app. Figure 4.2 depicts the six steps of such communication between TalkBack Proxy User and other components. We explain this process for TalkBack; however, keep in mind that this process is almost identical for SwitchAccess Proxy User.

1. TalkBack Proxy User performs a touch gesture using APIs provided by *AccessibilityService*, similar to Touch Proxy User. For example, Listing 4.1 with a line from the middle to the right side of the screen performs swipe right. The performed gesture is received by the *AccessibilityManager* service, *A11yManager* in short, and generates accessibility events corresponding to the action, e.g., `TYPE_GESTURE_DETECTION_START` and `TYPE_GESTURE_DETECTION_END` events for the swipe.

2. All implementations of *AccessibilityService*, including TalkBack, receive the generated accessibility events. TalkBack may suppress delivering the incoming events to the app and possibly translates them to something else. For example, while swiping right on the screen may result in a menu option being shown, TalkBack may translate that gesture to changing the focus to the next element when TalkBack is enabled on the device.

3. TalkBack analyzes the incoming event and initiates another action accordingly. For example, in the case of swipe right, TalkBack changes the focus to the next element, and in the case of

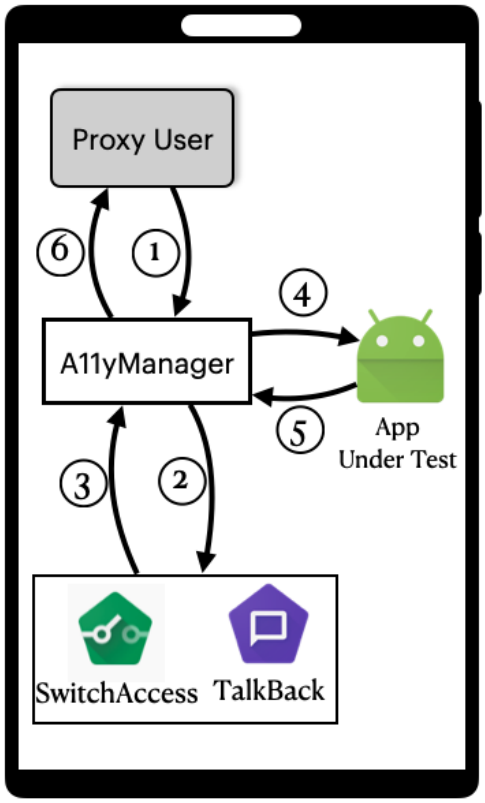


Figure 4.2: The process of communication of TalkBack or SwitchAccess Proxy User with their corresponding assistive services.

a double tap, the currently focused element is clicked. Note that TalkBack is unaware of the sender of these events, in this case, the TalkBack Proxy User. As a result, TalkBack behaves the same as it would if a human user had performed the action.

4. A11yManager receives the new action from TalkBack and sends it to the app under test. For example, if the TalkBack's action clicks on the focused element, A11yManager sends an event to the `onClickListener` class associated with the focused element in the app. The app receives the action and updates its internal state accordingly, e.g., executing `onClick` method of the clicked element.

5. The app informs A11yManager regarding the changes in the GUI elements. For example, when edit button in Figure 4.1(a) is clicked, the screen changes to Figure 4.1(b).

6. A11yManager receives the changes in the layout and dispatches feedback events accordingly,

e.g., a TYPE_VIEW_FOCUSED accessibility event associated with the focused element's *AccessibilityNodeInfo* object. As a result, the TalkBack Proxy User is informed of the latest changes on the device. For instance, it becomes aware of the element that is currently focused on. Note that there is a possibility that because of the changes caused by step 5, i.e., showing a new screen, another interaction is initiated between A11yManager and TalkBack, similar to steps 2 and 3.

Recall that from Section 4.3 a TalkBack Action can be ElementBased ($\mathbb{EB}$), TouchGestureReplication ($\mathbb{TGR}$), or PredefinedAction ($\mathbb{PA}$). To perform $\mathbb{TGR}(lg)$, TalkBack Proxy User makes a copy of the LineGesture lg, called lg', and moves its coordinate 2cm toward the top or right of the display, then combine the two LineGestures (lg and lg') and perform them when TalkBack is enabled. Performing a $PA(t)$ is easier since it is predefined and not dependent on the app. TalkBack Proxy User has a database of PredefinedActions and can perform the actions accordingly, e.g., perform swipe right then left when t is "Scroll Forward".

However, performing an ElementBased action is relatively challenging since it requires finding and focusing on the element first. Moreover, there are various ways of navigating to locate an element, i.e., Linear, Jump, Search, and Touch. To that end, we introduce TENG (**T**alkBack **E**lement **N**avigation **G**raph) to model the different ways of navigating an app with TalkBack.

Simply put, TENG is a graph modeling the different states of TalkBack when enabled. TENG is defined over the UI hierarchy of an app screen, where the nodes include GUI elements that can be focused by TalkBack and the edges represent actions that can be done by the user (or TalkBack Proxy User) to change the focus from one node to another. For example, Figure 4.3(a) represents a part of the TENG of the app screen in Figure 4.1(b). For now, please ignore the Start and End red boxes; we will define and explain them shortly. The blue ovals represent control elements, e.g., buttons or checkboxes, and green-round boxes represent the textual elements. Also, the gray boxes are a View element containing a set of elements that are grouped by TalkBack to announce. Recall

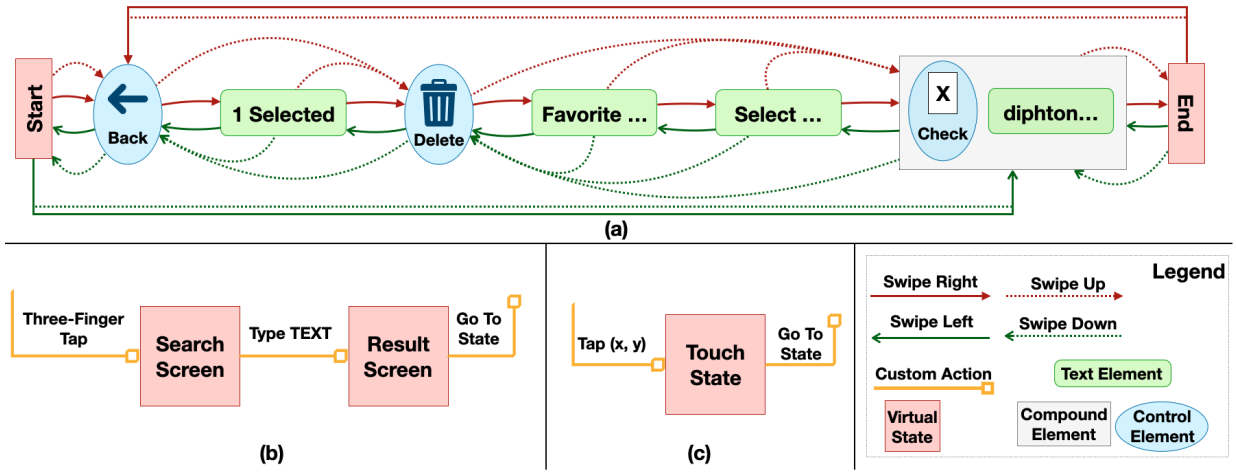


Figure 4.3: (a) TENG representing Linear Navigation of Figure 4.1(b), (b) TENG representing Search Navigation, (c) TENG representing Touch Navigation

that in Section 4.1, we discussed TalkBack grouped related elements and associated the group with an action for a better user experience. In runtime, when Talkback is in any of these nodes (states), i.e., focused on their corresponding element, we call it an *active node*. The solid arrows in Figure 4.3(a) represent Linear Navigation between elements, e.g., red arrows are associated with swiping right or moving to the next element. The dotted arrows represent Jump Navigation which changes the active node to the next control element. For example, if the Delete node is active, by swiping right, TalkBack focuses on the text element that starts with "Favorite" and by swiping down, TalkBack jumps on the previous control element, which is "Back".

Besides the UI elements, TENG has some other nodes, which we call *Virtual State*s. These states do not correspond to an element on the screen; however, they represent some internal states of TalkBack. For example, the virtual states *Start* and *End* in Figure 4.3(a) represent the states where TalkBack reaches the first or last element on the screen and notifies the user there is no element left to visit. Note that the user can still change the focus to other elements by Linear or Jump Navigation, even if TalkBack is in a virtual state, e.g., swiping left from Start changes the focus to the compound element in the end.

Recall that TalkBack supports two other navigation modes, i.e., Search and Touch. We model these

navigations in TENG using virtual states. Figure 4.3(b) shows the part of TENG related to the search navigation. The entry edge is a representative edge that comes from all nodes in TENG and is associated with a three-finger tap. We did not draw all edges not to make the figure complicated and messy. Once the Search Screen is activated, the user can type the text she is looking for, then the result appears in a list (Result Screen). Once the user selects a search entry, TalkBack focuses on the selected element. Finally, the Touch Navigation is modeled and depicted in Figure 4.3(c). Whenever the user taps on the screen, TalkBack finds and focuses on the underlying element. Similar to Search Navigation in Figure 4.3(b), the entry edge of the Touch State comes from all nodes of TENG.

Given a target element, we can use TENG to plan a sequence of interactions with the device to focus on the element. For example, similar to the last step of our motivating example in Section 4.1, assume we want to click on the checkbox, and at the beginning, TalkBack is focused on the Back button. Therefore, the TENG's active node is the Back button in Figure 4.3(a), and the goal is focusing on the TENG's node containing the target element (which is the compound element denoted by the grey box), and then performing double-tap. There are various ways to reach the target node; for instance, by performing two swipe-up actions, TalkBack first jumps to the Delete button and then to the target node.

However, traversing with TalkBack is not as easy as it sounds. There are three reasons that TENG may be modified during the interaction with TalkBack. First, the app may dynamically update the visible elements on the screen. For example, a slide show constantly changes the visible content after showing it for a specific time. Secondly, TalkBack may change the app state by performing extra gestures for navigating. Lastly, the app may change the focused element in runtime. For example, suppose developers do not want users to access certain elements, regardless of the rationale behind this decision. In that case, they can focus on another element as soon as TalkBack focuses on that element. Therefore, we cannot rely solely on the TENG created UI hierarchy before navigation.

To that end, once the TalkBack Proxy User performs an action associated with an edge, e.g., swiping right to focus on the next element, the service listens to any changes in the UI to determine if

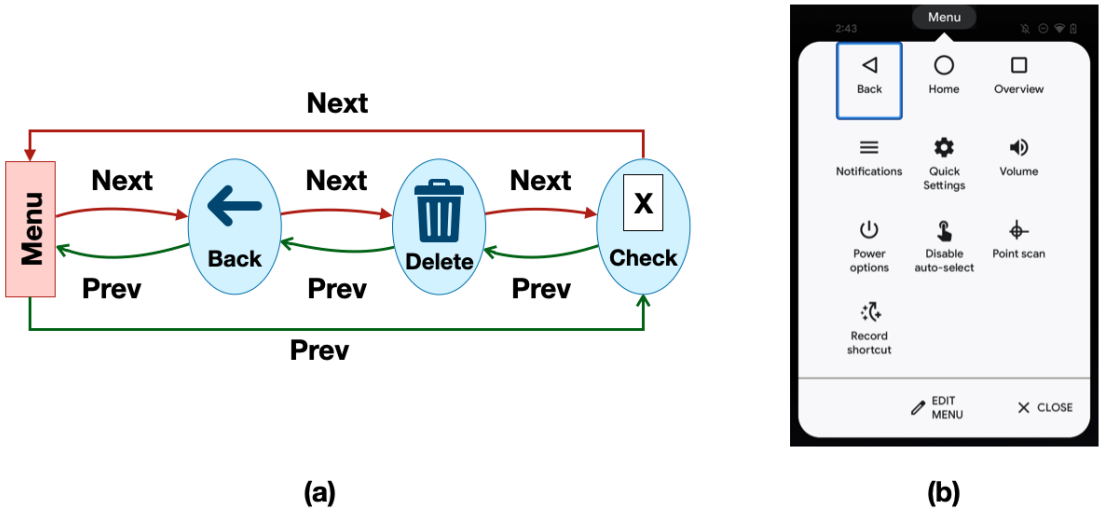


Figure 4.4: (a) The corresponding SENG of Figure 4.1(b), (b) the actions in the SwitchAccess menu

the UI hierarchy is changed. If anything changes, the TalkBack Proxy User recreates the TENG and continues the navigation. Otherwise, the service verifies if the currently active node in TENG is focused by TalkBack. If it was not, we mark the performed edge as *ineffective* and replan the locating path again.

4.4.3 SwitchAccess Proxy User

This Proxy User represents how users with motor impairment interact with a device using SwitchAccess. In terms of implementation detail, it is similar to TalkBack Proxy User. For example, the communication process (depicted in Figure 4.2) is identical for SwitchAccess, except instead of dispatching touch gesture events, SwitchAccess Proxy User emulates clicking the switches.

Recall that there were three types of actions in SwitchAccess: ElementBased, PointScan, and Menu. SwitchAccess performs ElementBased action using a graph called *SENG* (**S**witchAccess **E**lement **N**avigation **G**raph). Basically, SENG is similar to an induced subgraph of TENG in JumpNavigation, except instead of the Start node, it has a Menu node. Figure 4.4(a) represent the corresponding SENG of the app screen in Figure 4.1(b), where red and green arrows represent pressing the

Next and Previous switches, respectively. Since SwitchAccess does not need to announce textual elements, it does not focus on them or combine them with control elements, like the checkbox in Figure 4.3(b). Similar to the TalkBack Proxy User, the SwitchAccess Proxy User locates the target element using SENG, then presses the Select switch.

For PointScan actions, given the coordinate of the target point, the SwitchAccess Proxy User initiates scanning and observes the SwitchAccess state until it reaches the target point. Then it presses the Select switch. Also, for actions inside the Menu, it first selects the Menu element (the red box in Figure 4.4(a)), then locate the action and selects it. For example, Figure 4.4(b) shows some actions in the SwitchAccess menu.

4.4.4 Abstract Proxy User

As mentioned before, the assistive services above (TalkBack and SwitchAcess) are built on top of Accessibility API, and any other assistive services helping users with a disability has to use this API to provide alternative ways of interacting with devices. Abstract Proxy User represents an abstraction of any assistive services built on Accessibility API. We are interested in this API here because since all assistive services must use this API if there is an accessibility issue with the Accessibility API itself, it possibly means there is an accessibility issue with all assistive services.

The Accessibility API enables assistive services to receive information about the current state of the app (by using AccessibilityNodeInfo), be notified about the changes in the app state (by listening to AccessibilityEvent), and send actions (by dispatching AccessibilityAction). For example, an assistive service can retrieve the information about the delete button in Figure 4.1(b) as an AccessibilityNodeInfo. Then it can click on this element by dispatching AccessibilityAction.ACTION_CLICK. Finally, it can be notified if anything in the app has changed by listening to AccessibilityEvents, e.g. if any element on the screen has changed or a new element is inserted.

The list of possible actions that can be performed by Accessibility API can be found in [12]. For example, ACTION_CLICK, ACTION_SCROLL_UP, and ACTION_SET_TEXT are responsible for clicking, scrolling up (inside and element), and typing a word in an edit text box, respectively. Given an action, the Abstract Proxy User first identifies the target element and retrieves its corresponding AccessibilityNodeInfo. Then it sends the action event to this node.

4.5 Conclusion

This chapter examined different ways of interacting with a device with or without assistive services and introduced Proxy Users. Proxy Users are software applications that input and perform the action in various ways. In the following chapters, we utilize Proxy Users to evaluate the accessibility of applications automatically.

Chapter 5

Assistive-Service Testing Through Reusing GUI Tests

Existing automated accessibility assessment techniques primarily aim to detect violations of predefined guidelines, thereby produce a massive amount of accessibility warnings that often overlook the way software is actually used by users with disability. This chapter presents a novel, high-fidelity form of accessibility testing for Android apps, called LATTE, that builds on the way developers already validate their apps for functional correctness.

A widely adopted practice in software development is for developers to write system tests, often in the form of Graphical User Interface (GUI) tests, to validate the important use cases (functionalities) of an app for correctness. These use cases are the important functionalities of an app that should also be accessible. Given an app under test and a set of regular GUI tests (written by developers) as input, LATTE first extracts a *Use-Case Specification* corresponding to each test. A Use-Case Specification defines the human-perceivable steps a test takes to exercise a particular functionality in an app. LATTE then executes the Use-Case Specification using an assistive service, i.e., TalkBack and SwitchAccess. If a use case cannot be completed using an assistive service, it naturally means

the corresponding use case has an accessibility problem, which is reported to the developer.

LATTE mitigates the limitations of existing automated accessibility analysis techniques by evaluating the accessibility issues in a more realistic setting, i.e., using assistive services. In more than half of the subjects apps in our experiments, LATTE detected accessibility issues that were not detected by Google's Accessibility Scanner, the most widely used accessibility analyzer for Android. Moreover, unlike prior solutions that produce a massive number of accessibility warnings by simply scanning an app's screens irrespective of its purpose, our approach produces a small number of actionable accessibility defects that are guaranteed to affect a disabled user's proper usage of the app's main functionalities. LATTE produces a detailed report for each failed use case that provides the developer with the exact cause of inaccessibility and steps to replicate it.

Although the most reliable method of validating an app's accessibility is through user evaluation, finding users with different types of disability and conducting such evaluations can be prohibitively difficult, especially for small development teams with limited resources. Using LATTE, developers are able to gain useful insights into how their apps behave when engaged through an assistive service, allowing them to fix the issues prior to their release. Our approach can also complement user evaluation by allowing the development teams to hone in on a subset of problematic use cases that are flagged by our tool.

The remainder of this chapter is organized as follows. Section 5.1 illustrates an accessibility issue that cannot be detected using existing automated techniques, while Section 5.2 describes the details of our approach. Section 5.3 presents our experimental evaluation. Finally, Section 5.4 concludes this chapter. The tool and experimental artifacts can be found on the companion website, `https://github.com/seal-hub/Latte`.

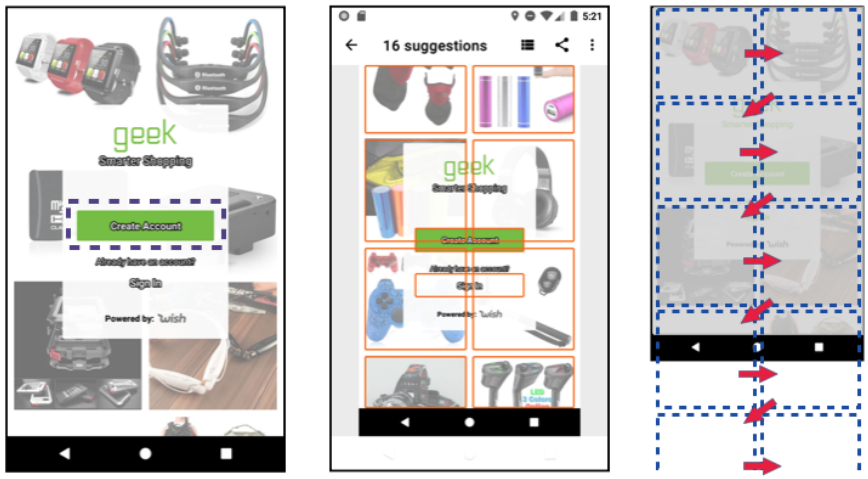


Figure 5.1: a) The very first step of creating account in "geek" shopping app (the dotted box) b) The accessibility issues reported by Google Accessibility Scanner c) Navigating the app using assistive services (TalkBack and SwitchAccess)

5.1 Illustrative Example

Figure 5.1(a) shows the launch screen of *Geek - Smarter Shopping* app (version '2.3.7') with more than 10 millions users [66]. The foreground layout contains register and login buttons, while the background is a layout of rolling decorative images. One of the most important use cases in this app is registration, since it is the prerequisite for accessing all other functionalities. This use case starts by clicking on the *Create Account* button (the dashed box in Figure 5.1(a)) followed by filling a form with user information (not depicted in the figure). A developer can create a GUI test to automatically verify this use case is working. For example, Listing 5.1 shows a GUI test in Appium [25] testing framework written in Python. It is basically a sequence of steps performing actions on specific elements on the screen, e.g., clicking on an element with resource-id `com.contextlogic.geek:id/login_fragment_create_account_button`.

While the developer of an app like this is likely to write a test to evaluate the functional correctness of registration, given its important to the overall functionality of the app, the conventional execution of such test does not reveal anything about the app's accessibility issues. To test the accessibility

```
1 find_element_by_id("com.contextlogic.geek:id/login_fragment_create_account_button").click()
2 find_element_by_xpath("/android.widget.FrameLayout/.../android.widget.EditText[1]").send_keys("
      John Doe")
3 find_element_by_id("fragment_email_text").send_keys("john.doe@example.com")
4 find_element_by_id("fragment_password_text").send_keys("StR0nGp@ss")
5 find_element_by_xpath("/android.widget.FrameLayout/.../android.widget.TextView[3]").click()
```

Listing 5.1: The test script corresponding to the registration use case

of this app, a conscientious developer would also run the Google Accessibility Scanner [9]—a de facto standard tool for analysis of accessibility in Android—on the launch screen and review the identified issues, as shown in Figure 5.1(b). In total, 16 accessibility issues are detected by the Scanner, denoted by orange borders placed around the elements with a problem. Out of these, there are 8 *"missing speakable text"* and 6 *"low image contrast"* issues for the decorative images in the background, and 2 *"small touch target size"* issues for the buttons in the foreground. As can be seen, there are many issues with the very first screen, and no particular hint as to the severity of these issues is provided to help the developer prioritize the effort involved in fixing the reported issues. The only accessibility issue reported for *Create Account* button is the *"small touch target size"*, which in fact does not affect users who rely on assistive tools for their interactions. Once the reported issues are fixed, this screen becomes supposedly accessibility-issue free, according to the automated accessibility scanner.

In practice, however, when TalkBack and SwitchAccess are used to operate this app, the first decorative image in the background receives the focus (top left dotted box in Figure 5.1(c)). To reach the *Create Account* button, users have to navigate through the elements. But here the decorative background layout refills dynamically, i.e., it is a revolving list. As a result, the focus never reaches to the foreground layout. The navigation path taken through the use of assistive tools is depicted in Figure 5.1(c) as arrows. This makes it difficult, if not impossible, for both TalkBack and SwitchAccess users to reach the *Create Account* button. In some cases, it may be possible for the user to touch random spots on the screen and find the button by chance; nevertheless, it would be far from perfect and frustrating at the very least.

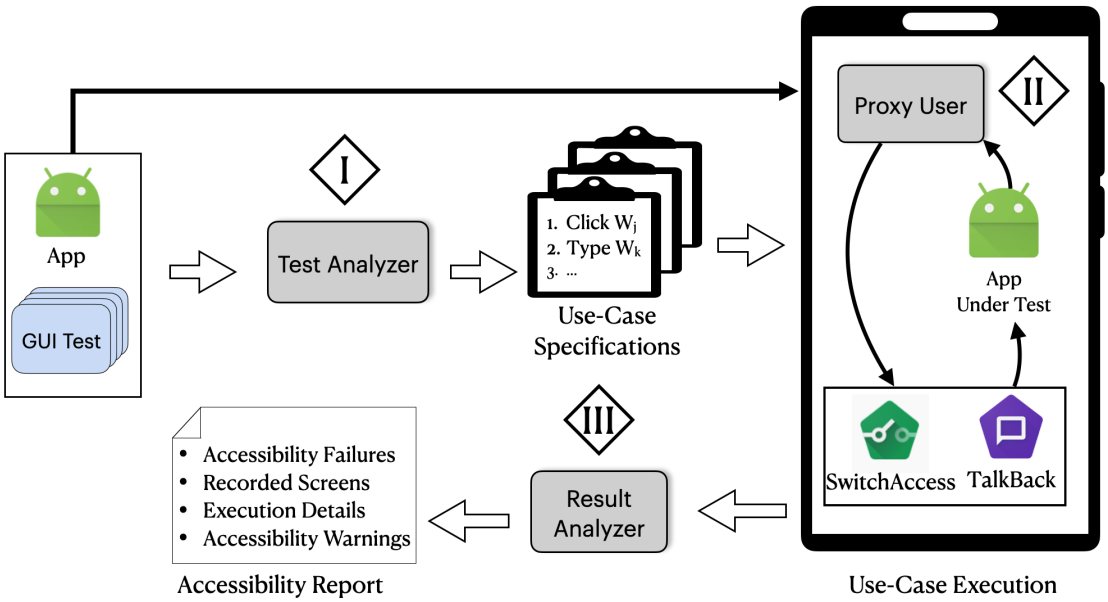


Figure 5.2: Overview of LATTE

5.2 Approach

Our objective is to develop an automated accessibility analyzer that is use-case and assistive-service driven. Figure 5.2 shows an overview of our approach, LATTE, consisting of three phases: (I) analysis of the provided GUI test suite of an Android app to determine the corresponding use cases, (II) execution of each use case on the app using an assistive service to evaluate the accessibility of the use case, (III) collection and analysis of the results to produce an accessibility report. In this section, we describe these phases in detail.

5.2.1 Test Analyzer

A use case is a sequence of interactions between a user and a software system for achieving an objective. In the case of a shopping app, for instance, creating an account, searching for a product, and purchasing a product, are examples of use case. As a common development practice, developers write GUI tests to automatically evaluate the correctness of a software system's use cases. A GUI

test is a sequence of steps, where in each step, the test (1) locates a GUI element, and (2) performs an action on that element. For example, the first step (line 1) in Listing 5.1 locates an element with *resource-id* equal to `com.contextlogic.geek:id/login_fragment_create_account_button` and then clicks on it. GUI tests need to uniquely identify elements on the screen. They leverage the implementation details of an app, such as *resource-id*, to interact with the GUI elements of the app. A GUI test thus follows a *white-box approach*, i.e., uses the implementation details of an app to drive the execution. Although this format is quite effective for machine execution, it differs vastly from how users interact with an app. A user may exercise the same use case as a test, but follows a *black-box approach*, i.e., interacts directly with the UI elements of an app to drive the execution.

Since our objective is to evaluate the accessibility of use cases exercised by tests, we first have to extract a description of the use case in terms of constructs available to a user. For instance, while the test script is able to access a button through its programmatic identifier (i.e, *resource-id* attribute), a blind user would access it through its textual content description. The Test Analyzer component takes a GUI test as input and transforms it into a *Use-Case Specification*, consisting of a set of equivalent steps as those performed by the test at the level of abstraction understood by users. In other words, Use-Case Specification of a test represents the steps a user would need to perform to exercise the same functionality as that of the test.

To extract the use cases from GUI tests, we have developed a novel, dynamic program analysis technique that, given a test script and an app, determines (1) the various GUI elements involved in the test and their attributes, and (2) the actions performed on those elements. Dynamic program analysis entails evaluating a program by executing it. In fact, software testing is the most common form of dynamic program analysis. By dynamically analyzing a test script (i.e, running the test on the app), we are able to identify the `AccessibilityNodeInfo` object corresponding to each GUI element. `AccessibilityNodeInfo` class is provided by the Android framework and represents the attributes of a GUI element on the screen. For example, the `AccessibilityNodeInfo` of the element in the first step in Listing 5.1, “Create Account” button, can be found in Listing 5.2.

```
// Resource Id
viewIdResName: com.contextlogic.geek:id/login_fragment_create_account_button;
// Textual information
className: android.widget.TextView;
text: Create Account;
contentDescription: null;
// Other fields
boundsInParent: Rect(0, 0 - 498, 110);
boundsInScreen: Rect(291, 856 - 789, 966);
clickable: true;
focusable: true;
focused: false;
selected: false;
longClickable: false;
enabled: true;
importantForAccessibility: true;
...
```

Listing 5.2: The `AccessibilityNodeInfo` object corresponding to "Create Account" button

The first field is `viewIdResName` (or *resrouce-id*) that is the identifier of the element. The textual attributes are `className`, `text`, and `contentDescription`. There are also other types of attributes such as coordinates and supported behaviors, e.g., this element is clickable, focusable, etc. We extract the textual attributes (*text*, *contentDescription*, and *className*) for each element, since these are the attributes perceived by users in locating GUI elements. Note that the *className* attribute is perceivable by users, since a sighted or blind user can recognize it visually or textually, i.e., `EditText` element has its distinguishable shape, and TalkBack announces it as *Edit Text Box*. We further extract actions (e.g., click, type) performed on the GUI elements from the test script itself. From Listing 5.1, we are able to determine that the use case consists of five steps, where the first and last steps click on GUI elements and the other steps enter textual information in GUI elements.

We finally combine the information obtained through the above-mentioned analysis of the GUI tests to arrive at the equivalent Use-Case Specifications. For example, Listing 5.3 is the Use-Case Specification generated from the GUI test shown in Listing 5.1. The first step shows the user clicking on a `TextView` element with the text "Create Account" and the last step is clicking on an `ImageButton` element with content description equal to "Submit". Intuitively, we have transformed a white-box description of a use case (i.e., GUI test) to a black-box description of that use case (i.e., Use-Case Specification).

```
Click on element with Text: "Create Account", ContentDescription: null, Class: TextView
Type "John Doe" on element with Text: "Name", ContentDescription: null, Class: EditText
Type "john.doe@example.com" on element with Text: "Email", ContentDescription: null, Class:
    EditText
Type "StR0nGp@ss" on element with Text: "Password", ContentDescription: null, Class: EditText
Click on element with Text: null, ContentDescription: "Submit, Class: ImageButton
```

Listing 5.3: The use case corresponding to the registration test case in the illustrative example

The Test Analyzer component is written in Python programming language on top of the Appium testing framework [25].

5.2.2 Use-Case Executor

The existing testing frameworks can access all GUI elements and perform any actions on them, even if the target element is not visible to the user. For example, the first step of the test shown in Listing 5.1 is able to locate the "Create Account" button and click on it, no matter where the button is located on the screen. However, users with disability may not be able to perform such actions smoothly. Blind users need to explore the app using a screen reader to locate the element. Although recognizing elements is comparatively easier for users with motor disability, they may have difficulty reaching and initiating action on the element, as we saw in the illustrative example of Section 5.1.

To improve the fidelity of evaluating accessibility issues for users with disability, LATTE is designed to automatically execute a use case using assistive services. To that end, we utilize TalkBack and SwitchAccess Proxy Users (defined in Section 4.4). Recall that, a Proxy User inputs an action, and perform the action the way the target user interact with the device. For example, to click on "Create Account" button, a TalkBack Proxy User use Linear Navigation to locate the button, then perform a double-tap. Basically, the Use-Case Execution phase iterates through the Use-Case Specifications, and perform them with TalkBack or SwitchAcces Proxy Users.

There are two scenarios during the use-case execution where the scanning process may not finish; in

other words, none of the focused elements match the description of the target element in the use-case step. First, the textual description of the element is not sufficient to uniquely recognize the element, because either there are multiple elements with the same description (duplicate labels issue) or the target element does not have any textual description (unlabeled element issue). This scenario occurs only in the case of TalkBack. The other scenario occurs when the target element could not be focused (or reached) by TalkBack or SwitchAccess, e.g., illustrative example of Section 5.1 in which "Create Account" button could not be reached.

LATTE defines two termination conditions for Proxy Users to prevent getting stuck in such cases: (1) if an element is visited more than a predefined number of times, or (2) if a step takes more than a predefined number of interactions to complete. These thresholds are configurable. Once either one of these conditions is satisfied, LATTE marks the step as inaccessible. However, since we would like to identify all accessibility issues in a use case, and not just the first encountered issue, when an inaccessible step is encountered, LATTE executes it using the corresponding instruction in the original test script, i.e. using Touch Proxy User. This allows LATTE to continue the analysis and report all accessibility issues within a use case.

5.2.3 Result Analyzer

To retrieve the information generated during use case execution automatically, we implemented a Command Line Interface (CLI) on top of the Android Debug Bridge (ADB) [14]. Using the CLI, the Result Analyzer component communicates with the Use-Case Executor to receive and record details of the execution for each step of a use case (recall Figure 5.2). Moreover, it automatically records the screen during the use-case execution and stores the video clip. Once all use cases are executed, the Result Analyzer aggregates the results and generates an *Accessibility Report*, consisting of the following four components.

Accessibility Failures. For each use case, LATTE reports if it encountered an accessibility failure

during its execution using assistive services. A use case has an accessibility failure if the GUI element of one of its steps cannot be located (focused).

Recorded Screens. While LATTE executes a use case, it records the screens to help developers (1) localize the accessibility issues, and (2) obtain insights into how users with disability may interact with their apps using assistive services.

Execution Details. LATTE reports other information extracted from the execution of each use case, including the execution time and the number of interactions to complete the use case. This information can be used as a source of insight for developers.

Accessibility Warnings. If a specific use case takes an exorbitant number of interactions to complete, it indicates a usability concern for disabled users. We report this category of issues as accessibility warnings, since in practice they can adversely affect users with disability. The threshold of what constitutes an exorbitant number of interactions is configurable in LATTE. For the purpose of experiments reported in the next section, we empirically observed that on average 1 touch interaction with an app requires approximately 5 times more interactions using TalkBack. We thus set the threshold to 15 times the number of direct interactions, or 3 times the average number of TalkBack interactions.

5.3 Evaluation

In this section, we evaluate LATTE on real-world apps to investigate the following research questions:

- **RQ1.** How accurately does LATTE execute use cases using assistive services?
- **RQ2.** How does LATTE compare to Google Accessibility Scanner (the official tool for detecting accessibility issues in Android)?

- **RQ3.** How do the detected accessibility failures and warnings impact the usage of apps?

5.3.1 Experimental Setup

We evaluated our proposed technique using 20 apps, 5 of which have known accessibility issues, as confirmed through user studies with disabled users in prior work [125]. The rest have been randomly selected from 13 different categories on Google Play (e.g., entertainment, productivity, finance), where 12 of them have more than 1 million installs.

We constructed a set of 2 to 4 test cases per app using Appium [25], which is an open-source testing framework. In total, we ended up with 50 test cases for 20 apps. The test cases reflect a sample of the apps' main use cases, as provided in their descriptions (e.g., register an account, search for products, place products in a shopping cart). For the apps with confirmed issues (first 5 apps highlighted in Table 5.1), one of the test cases corresponds to the previously reported use case that users with disability could not perform. Our experiments were conducted on a MacBook Pro with 2.8 GHz Core i7 CPU and 16 GB memory (a typical computer setup for development) using an Android emulator (SDK 27).

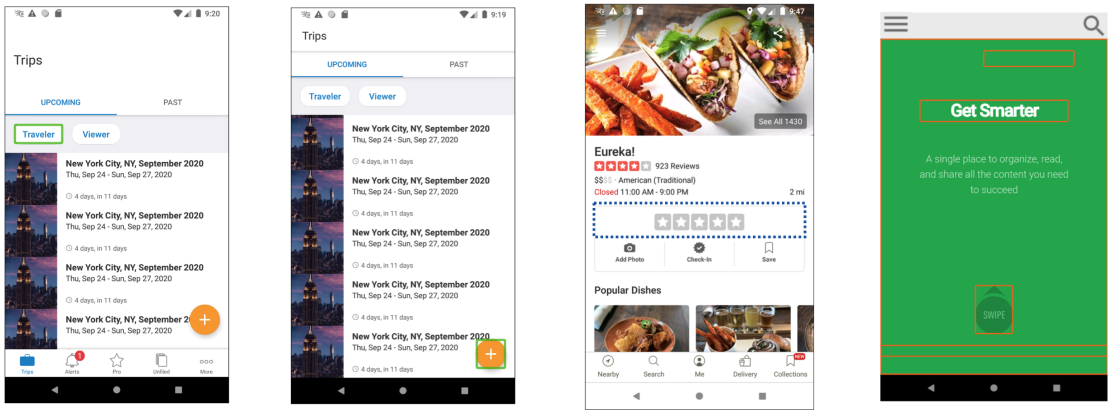
5.3.2 RQ1. Accuracy of LATTE

We first executed the 50 GUI test cases to ensure they are constructed correctly. We then generated the Use-Case Specifications from the tests and executed them using both SwitchAccess with two physical switches (Next and Select) and TalkBack with directional navigation (swiping).

Table 5.1 summarizes the presence of accessibility failures in different settings. In a cell, 'x' indicates a use case of an app (row header) that could not be executed using an assistive service (column header) due to an accessibility failure, and '✓' indicates a use case that could be fully executed without any failure. As shown under column heading "None", all original test cases

Table 5.1: The summary of detected accessibility failures. 'x' shows a failure was found in an app (row) while executing under a setting (column). Red bold '**x**' is a failure that was detected using LATTE but not using Google Accessibility Scanner. '✓' means the test or use case could be executed completely under a setting. The first five highlighted apps have confirmed accessibility issues per prior user study [125] .

	None (Test Cases)	SwitchAccess (Use Cases)	TalkBack (Use Cases)
iPlayRadio	✓✓	**xx**	**xx**
Feedly	✓✓	**xx**	**xx**
Checkout51	✓✓	✓✓	**x**✓
Yelp.	✓✓	✓✓	**xx**
Astro	✓✓	✓✓	x✓
BillManager	✓✓	✓✓	x✓
Budget	✓✓✓	**xx**✓	**xxx**
CalorieCounter	✓✓✓	**x**✓✓	**xx**x
Clock	✓✓	✓✓	xx
Cookpad	✓✓	✓✓	**x**x
Dictionary	✓✓✓	✓✓✓	**xx**x
Fuelio	✓✓✓	✓✓✓	x✓✓
Geek	✓✓	**xx**	**xx**
SchoolPlanner	✓✓✓	✓✓✓	xxx
SoundCloud	✓✓	✓✓	**x**x
TodoList	✓✓✓✓	**xx**✓✓	**xx**x✓
TripIt	✓✓✓	✓✓✓	**x**x✓
Vimeo	✓✓✓	✓✓✓	xx✓
Walmart	✓✓	✓✓	✓✓
ZipRecruiter	✓✓✓	✓✓✓	xx✓



(a) TripIt - The initial screen

(b) TripIt - After navigation the bottom menu is disappeared

(c) Yelp - the dotted box could not be focused by TalkBack

(d) Reported accessibility issues by Scanner for Feedly (Orange solid boxes)

Figure 5.3: The screenshots of some apps with accessibility failures

passed, since they do not check the accessibility of apps, but rather evaluate the correctness of corresponding use cases. All accessibility results were manually examined and the failures were verified by the authors (the video clips of the failures can be found on the companion website [110]). LATTE achieves 100% precision (no false positives) in determining accessibility failures in the use cases; in other words, all of the failed use cases in our experiments manifest a real accessibility issue. As can be seen, 11 use cases in 6 apps and 39 use cases in 19 apps have accessibility failures with SwitchAccess and TalkBack, respectively. Additionally, LATTE detected 17 and 25 accessibility warnings using SwitchAccess and TalkBack, respectively. The warnings are not reported in Table 5.1, but discussed in more detail later.

We also analyzed the number of interactions for executing a use case with different assistive services. On average, LATTE requires 11, 51, and 43 interactions to finish each use case under None, SwitchAccess, and TalkBack settings, respectively. Additionally, the ratios of the number of interactions required for SwitchAccess and TalkBack over those required for None were 5 and 4, respectively. This means LATTE requires more than 4 interactions using assistive services to fulfill a single interaction without such services, giving us a glimpse into the practical challenges disabled users face in their usage of mobile apps.

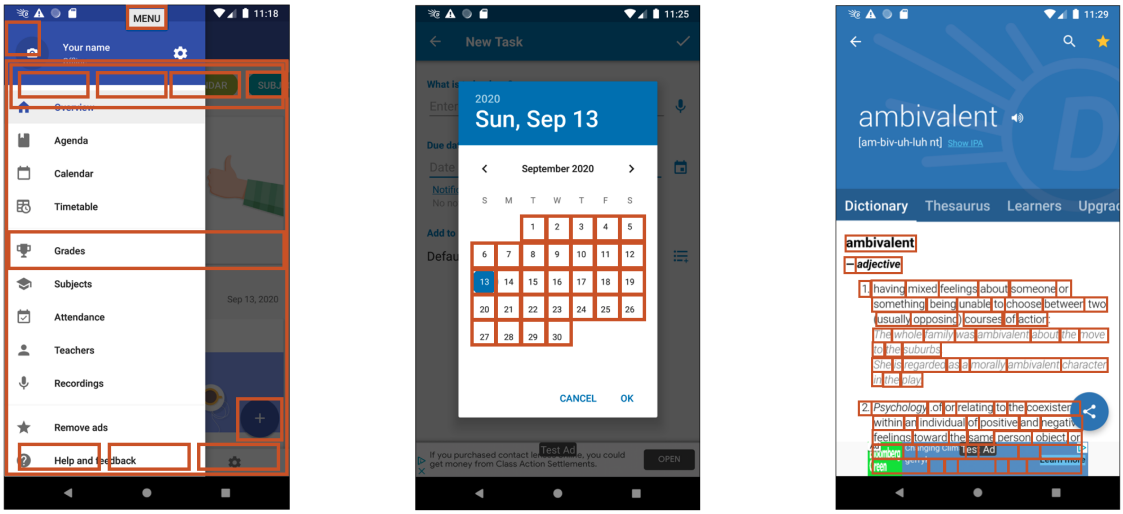
5.3.3 RQ2. LATTE vs. Google Accessibility Scanner

We ran Google Accessibility Scanner at each step of all use cases. We then compared the failures detected by LATTE against the issues reported by Scanner. Red bold '**x**' in Table 5.1 represents the corresponding use case has an accessibility failure detected by LATTE that Scanner could not detect.

Scanner was able to detect only 18 of the 50 accessibility failures detected by LATTE in the evaluated use cases. For each failure detected by LATTE, we examined all of the issues reported by Scanner. If any of those issues were found to be related to the actual fault, we assumed the Scanner can help to find the failure, e.g., Scanner can detect missing labels. Scanner could not detect any of the 11 accessibility failures detected by LATTE using SwitchAccess, and 21 of the 39 failures detected by LATTE using TalkBack. While LATTE was able to detect all of the 5 issues confirmed by actual users with disability in the first 5 apps of Table 5.1, Scanner was only able to detect 1 of the issues (in Astro app). In addition, Scanner was not able to find the accessibility failures in 8 of our randomly selected subject apps.

Scanner reports an exorbitant number of issues that would overwhelm a typical developer. It reports on average 34 issues per use case for a total of 1,716 issues in the 50 use cases in our experiments. Out of the 1,716 reported issues by Scanner, only 18 were relevant to the accessibility failures reported by LATTE. In comparison, LATTE produces at most one accessibility failure per use case. For example, in Figure 5.3(d), Scanner detected a number of issues, e.g., "Get Smarter" has low text contrast. The Scanner did not report any problem regarding the top two buttons (menu and search icons) that cannot be reached using TalkBack and SwitchAccess, making the app totally inaccessible.

On the other hand, the current prototype of LATTE could not detect issues that are not related to assistive-services, like low-text contrast. We believe LATTE can be improved by the predefined checks of Scanner to cover a more comprehensive set of accessibility issues.



(a) School Planner (b) Todo List (c) Dictionary

Figure 5.4: Screens of few apps with accessibility issues

5.3.4 RQ3. Qualitative Study of Detected Accessibility Failures and Warnings

Accessibility Failures

We manually examined all use-case failures and categorized them into the following three groups:

Dynamic Layout Some apps change the visibility of elements on the screen dynamically. For example, Figure 5.3(a) shows the initial screen of *TripIt* app. If a user wants to reach the bottom menu, e.g., clicking on the Alert icon, she needs to explore the elements to locate the target widget; however, during the directional navigation with TalkBack, the bottom menu disappears (Figure 5.3(b)). The reason behind hiding the menu is to improve the user experience by providing more space in the middle list (where a sighted user is looking for an item). However, this change in the layout makes the bottom menu inaccessible for a blind user, since she does not know the menu has disappeared. The accessibility failures in *TripIt* and *Dictionary* apps belong to this category. This observation is consistent with the findings in a prior work [98] that showed usability and

accessibility concerns are not a subset of each other. Furthermore, this example suggests improving the usability of a use case for some users may in fact degrade the accessibility of that use case for others.

Navigation Loop Assistive services may not reach a GUI element in some apps because of a static or dynamic loop in directional navigation. Developers can create a static loop by defining custom traversal order over elements using `accessibilityTraversalAfter` attribute. While Accessibility Scanner can detect static loops, none of the apps in our experiments had this issue. On the other hand, a dynamic loop is caused by inserting elements while the user interacts with an app. For example, as shown in Section 5.1, the images in the background are inserted as the user navigates through them, making the navigation list virtually infinite. This issue is usually caused by `RecyclerView` widget where its adapter indefinitely inserts items into the container. The accessibility failures of this type could be found in *Yelp*, *CalorieCounter*, *CookPad*, *Geek*, and *SoundCloud* apps.

Non-Standard Implementation Developers use customized GUI widgets in their apps that may have different behavior when users interact with them using an assistive service. For example, Figure 5.3(c) is the page of a restaurant in Yelp where users can rate the restaurant (the dotted blue box). However, TalkBack cannot focus on the rating widget since it is a customized `TextView` without any text. Therefore, even a sighted user using TalkBack cannot select this widget to rate the restaurant. Another source of these failures is using `WebView` widgets carelessly. `WebView` allows Android apps to load UI elements using web technologies, e.g., HTML, JavaScript. For example, in *Feedly* app (Figure 5.3(d)), the search icon at the top right is a `WebView` icon where its `clickable` attribute is false, meaning it cannot be invoked using assistive services. This attribute, however, does not prevent a user without disability from directly tapping the icon, which results in the corresponding JavaScript event handler to be invoked. LATTE detected these types of failures in *iPlayRadio*, *Feedly*, *Checkout51*, *Yelp*, *Budget*, and *TodoList* apps.

Accessibility Warnings

We also studied the use cases with *Accessibility Warnings* and categorized them into four categories. Recall that LATTE reports an accessibility warning when an step in the use-case execution takes more than a specific number of interactions (15 interactions in our experiments).

Overlapping Layouts Most of the apps have multiple layouts that overlay on top of each other, i.e., Activity, menu, and dialogue layout. A user who directly interacts with the screen only considers the elements on the top layout. However, TalkBack and SwitchAccess visit all focusable elements regardless of the layout hierarchy. Therefore, users who use assistive services often navigate through elements even if they are not on the top layout. For example, Figure 5.4(a) shows the main screen of the School Planner app. As can be seen, the side menu is the active window (it is fully visible). However, it takes at least 12 interactions for a user to even reach the first item in the menu. Developers can fix this issue by making the elements in the non-top layouts unfocusable.

Far-Off Widget All screen elements can be accessed virtually in no time for a user who directly interacts with the device. However, users relying on assistive services access the elements sequentially. Therefore, it takes longer for them to access a frequently used element located at the end of the navigation list. For example, in the TripIt app, Figure 5.3(a), a user has to navigate all elements from the top to the bottom to access the *fab* icon (the icon with a plus sign, highlighted in Figure 5.3(b)). To resolve these issues, developers can define a custom navigation to reduce the interactions required to reach the important elements. For example, in the School Planner app (Figure 5.4(a)), the fab icon is located at the top of the navigation list, although its actual position on the screen is at the bottom right.

Grid Layout Grids provide an efficient layout for presenting multiple items in a small space, all of which can be accessed in no time for users without disability. However, since a grid's items are

accessed linearly by SwitchAccess, it takes a lot of interactions to reach the last element on the gird. For example, in the TodoList app, the calendar widget has 30 items in the grid that need to be visited before reaching to "CANCEL" or "OK" buttons (Figure 5.4(b)). To fix this, developers can provide different layouts for different settings, e.g., a text-based date picker when TalkBack or SwitchAccess are enabled.

Web View There is a common practice for mobile developers to reuse web content (implemented in HTML/JavaScript) for some parts of their apps using `WebView` widget [87]. However, assistive services cannot analyze web elements properly, as shown earlier in the case of *Non-Standard Implementation* category of accessibility failures. Even if improper usage of web elements does not make an app inaccessible, it can degrade the user experience. For example, in the case of Dictionary app, shown in Figure 5.4(c), the definition of a term is shown in terms of a series of web elements, and each word in the passage is a clickable Android GUI element. Consequently, TalkBack and SwitchAccess need to navigate through all of these elements to reach the end of the text.

5.4 Conclusion

This chapter described a novel, high-fidelity form of automated accessibility analysis for Android apps, called LATTE. It reuses tests written by developers to evaluate the correctness of the important use cases in their apps to also validate the accessibility of those use cases. LATTE first extracts use cases corresponding to an app's tests, and subsequently executes them with the help of assistive services. We evaluated the effectiveness of LATTE by analyzing 20 apps selected from 13 different categories from Google Play and identified 32 accessibility failures that could not be identified using the state-of-the-art technique. A qualitative analysis of the results obtained in our experiments allowed us to identify a number of interesting categories of accessibility failures and warnings, together with ways in which developers can resolve them.

Although LATTE finds accessibility failures undetected by Accessibility Scanner, it cannot detect accessibility barriers for users who do not use assistive services, e.g., low text contrast. Therefore, Accessibility Scanner cannot be replaced completely by LATTE, rather they complement each other.

The main input for LATTE, which should be designed by app developers, is GUI test cases. If these test cases are unavailable, for any reason, LATTE is not able to do any accessibility analysis on the app. The next chapter, examines the possibility of completely automated accessibility analysis of mobile apps without any manual input.

Chapter 6

Assistive-Service Crawler

The previous chapter introduces LATTE, which incorporate assistive services in evaluating the feasibility of executing GUI test cases. However, LATTE assumes the availability of GUI tests for validating the functionalities of the app under test, which are then repurposed for accessibility analysis. Unfortunately, developers do not usually write GUI tests for their apps, making their approach applicable to only situations in which GUI tests are available. Studies show that more than 92% of Android app developers do not have any GUI test for their apps [79]. Even if GUI tests are available for proprietary apps, the test cases are rarely available to the public or app store operators that may want to assess the accessibility of apps for users. Furthermore, GUI tests may fail to achieve good coverage, making their approach ineffective at finding accessibility issues in uncovered parts of the app under test.

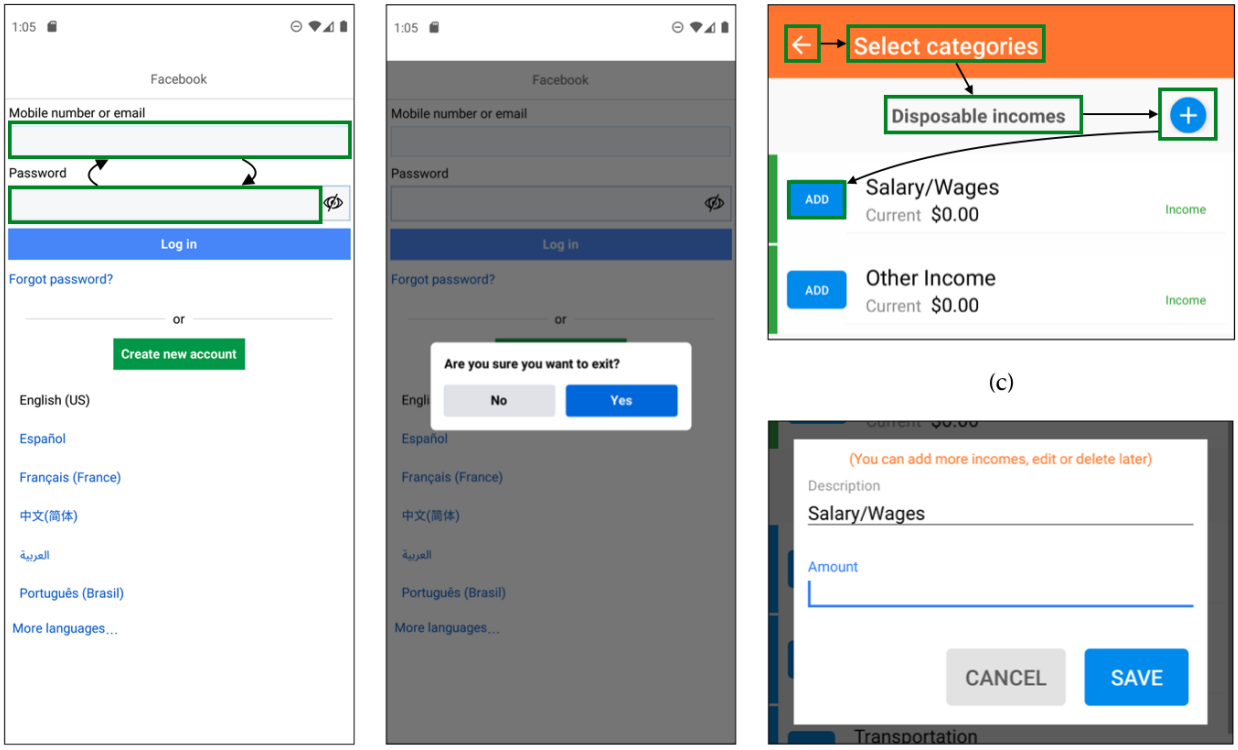
After LATTE, Alotaibi, et al. [6] have proposed a method of detecting certain accessibility failure that may occur when using TalkBack. However, their approach requires the developer to manually navigate through the app, i.e., the input to their tool is a screen of an app, rather than the app under test. Furthermore, their approach cannot detect issues related to performing actions with TalkBack.

To address the limitations of existing tools, we have developed a fully automated approach, called

GROUNDHOG, for validating the accessibility of Android apps that replicates the manner in which disabled users actually interact with apps, i.e., using assistive services. GROUNDHOG gets the app in a binary form, i.e., APK, and installs it on a Virtual Machine (VM). It utilizes an app crawler to explore a diverse set of screens to be assessed. For each screen, GROUNDHOG extracts all the possible actions and executes the same action with different interaction models, including different assistive services, to validate if the app is accessible. GROUNDHOG leverages the VM to repeatedly reevaluate the app from the same state, performing the same action using different assistive services to identify the accessibility issues that may affect users with various forms of disability.[1] In particular, GROUNDHOG checks if UI elements can be located by users, i.e., **locatability**, and all actions can be performed, i.e., **actionability**, regardless of the way users interact with the device, e.g., touch-based interaction or assistive-service interaction. Instead of just reporting violations of accessibility guidelines as in prior work, GROUNDHOG produces a summary of the accessibility issues containing a video that describes how a user with disability cannot perform an action in an app. This type of reporting can help developers to pinpoint the issue and increase their awareness of the challenges faced by users with disability.

Our empirical experiments show that GROUNDHOG can detect 293 accessibility issues that could not be detected by existing accessibility testing tools.

The rest of this chapter is organized as follows: Section 6.1 motivates this study with an example. Section 6.2 explains the details of our approach, and Section 6.3 describes the optimizations over our technique. The evaluation of GROUNDHOG on real-world apps is finally presented in Section 6.5.



(a) (b) (c) (d)

Figure 6.1: (a) The login activity of Facebook app, (b) The exit dialog appears when users press back button on Facebook app, (c) a screen in BudgetPlanner app, the highlighted boxes and arrows depicts the directional navigation to the “ADD” button by TalkBack, (d) a dialog appears after tapping “ADD” button

6.1 Motivating Example

In this section, we provide two examples to illustrate the types of accessibility issues that cannot be detected with conventional accessibility testing tools and prior studies.

Figure 6.1(a) shows the login screen of the Facebook app with more than 1 billion installs on Google Play [54]. This screen provides the ability for the user to log in, which obviously is crucial to be accessible, since it is the entry point of the app.

A user with a disability relies on an assistive service to interact with the app. As discussed in

[1]The name of our tools is inspired by the popular Hollywood movie “Groundhog Day” from 1993, where the lead character is stuck in a time loop, forcing him to relive the same day, which is akin to our repeated reevaluation of an app from the same state.

Section 4.3, a TalkBack user can navigate through the elements with four navigation modes, Linear, Touch, Jump, and Search. Using either of these navigation modes on the app screen illustrated in Figure 6.1(a), TalkBack can only detect the two text boxes, annotated in green in Figure 6.1(a), and is incapable of detecting the rest of the elements, including crucial buttons such as "Log in" or "Create new account". However, a regular user can see all the elements on the screen, provide login credentials, and tap on the buttons to log in and use the app without any problem. Interestingly, a TalkBack user cannot even exit the app using the back button as none of the elements on the exit dialog, in Figure 6.1(b), are accessible by TalkBack. This is an example of **locatability** issue, since a user with a disability cannot locate (reach) an element on the screen.

Existing accessibility testing approaches are not capable of detecting these issues. Google Accessibility Scanner [9] evaluates the top screen on a device, checks a few rules for the elements, and reports their violations as accessibility issues. In running Scanner on the screen in Figure 6.1(a) 4 issues are detected for text boxes, 2 of them are warning about their *"small touch target size"*, and 2 of them are noting the *"missing speakable text"* for them. Neither Scanner nor other rule-based accessibility testing tools [20, 17] are capable of detecting navigational issues in Android apps.

Assistive services also enable users to perform actions on elements. Unfortunately, actions performed under different interaction models may have inconsistent behaviors. Figure 6.1(c) shows a screen in a popular budget tracker app, with more than 1 million installs, where users can add income or expenses to their budgets. To add an income to the budget, a user without a disability simply taps on the "ADD" button and a form appears to input the amount, as shown in Figure 6.1(d). For the same action, a TalkBack user, first locates the "ADD" button, e.g., the Linear Navigation is shown by arrows in Figure 6.1(c)). Once the element is located, the user double taps to perform a click action through TalkBack. However, in this case, The income addition form in Figure 6.1(d) will not be shown, preventing TalkBack users from adding any income and rendering the app inaccessible for them as a result. This is an example of **actionability** issue, since the action is not supported consistently under different interaction models.

The insight underlying our work is that the two types of accessibility issues discussed above cannot be revealed accurately unless the apps are examined in the manner disabled users interact with apps, i.e., using assistive services.

6.2 Approach

Regardless of different interaction models, the ability to locate elements on the screen and perform actions consistently are fundamental needs in app accessibility. As a result, *locatability* and *actionability* form the basis of our approach. The goal of our approach is to automatically detect apps that fail to meet these accessibility requirements at runtime.

To that end, we propose GROUNDHOG, an automated assistive-service driven testing tool. Figure 6.2 shows the overview of our approach. GROUNDHOG utilizes an *App Crawler* to explore different states of the app. After each change in the app, *App Crawler* invokes the *Snapshot Manager* to capture a VM snapshot if the current state (screen) has not already been seen. *Snapshot Manager* provides the VM Snapshots to *Action Extractor*, where all the possible actions on the given state of the app are subsequently extracted. GROUNDHOG then tries to locate the elements and perform these actions on them using three different Proxy Users: Touch Proxy User, TalkBack Proxy User, and Abstract Proxy User. Finally, the new state of the app after performing the action is provided to the *Oracle* along with the initial app state. *Oracle* assesses if each Proxy User successfully performs the action and produces the final report.

In this section, we describe each component of GROUNDHOG in detail, except Proxy Users, which have been discussed in Section 4.4.

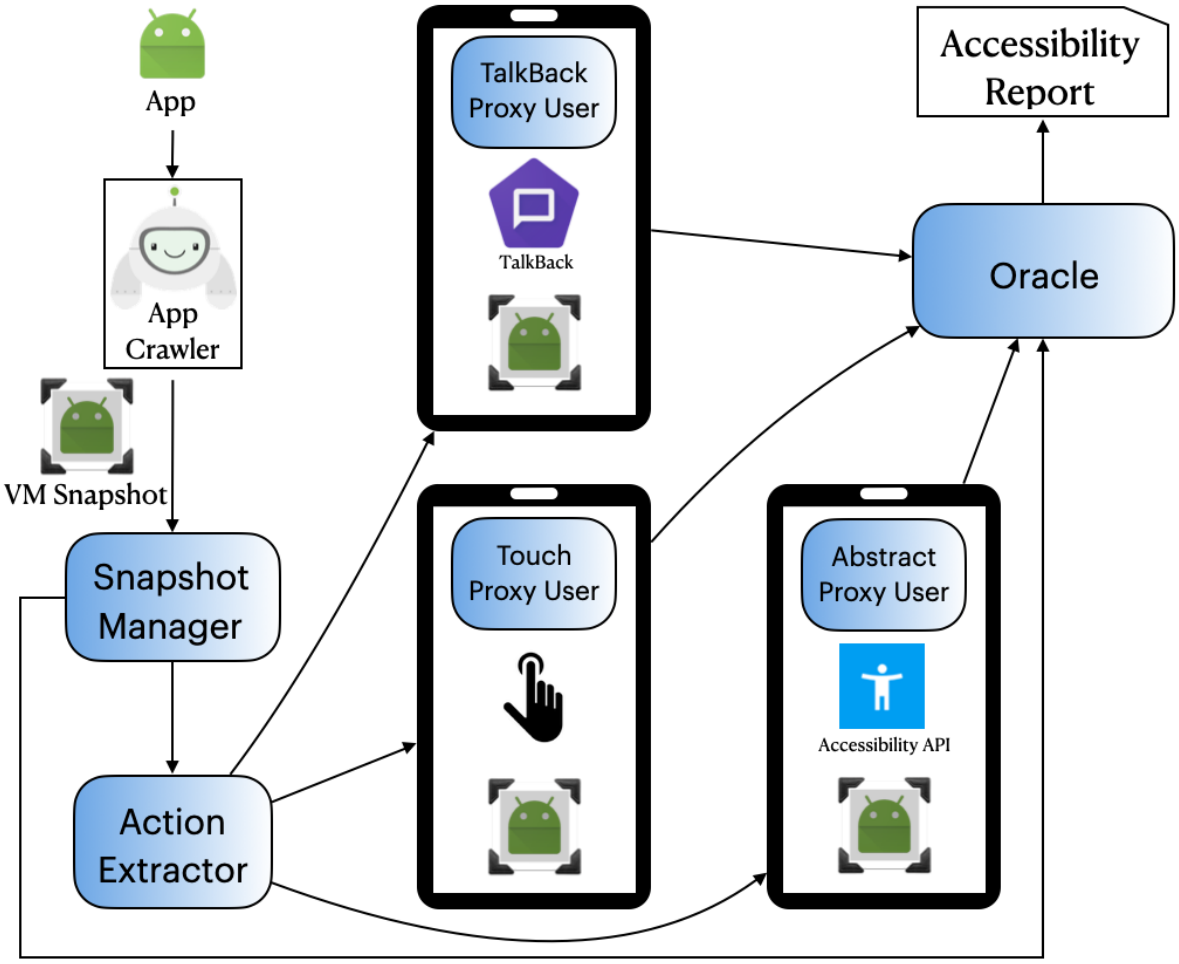


Figure 6.2: An overview of GROUNDHOG

6.2.1 Snapshot Manager

The goal of Snapshot Manager is to allow a diverse set of app states obtained through crawling to be later analyzed. Snapshot Manager is a connector between an app crawler and the rest of the system. GROUNDHOG can be integrated with any of the existing app crawling techniques like Monkey [55], Stoat [116], Ape [57], Sapienz [84], etc. These crawlers employ various techniques in modeling the app to trigger transitions between app states. For example, Stoat models app behavior as a Finite State Machine (FSM) whose nodes are UI elements and attempts to maximize node coverage as well as code coverage. In GROUNDHOG, even a human agent, e.g., developer or tester, can be involved to replace or enhance an automated app crawler to reach any desired state of the app.

For each app state, Snapshot Manager checks whether this state is a newly discovered state to take a snapshot for further analysis or not. To that end, Snapshot Manager calculates a hash value of the hierarchical representation of UI elements on the screen. Screen hash calculation in Snapshot Manager only incorporates elements and attributes that impact obtaining a diverse set of app screens. For example, elements that do not belong to the app under test, i.e., have a different package name or belong to an advertisement widget, are not included. Similarly, not all elements' attributes can distinguish different screens. For example, if the app crawler taps on an edit text box or writes a random string in it, its *focused* and *text* attributes change; however, they are not indicators of a new screen. The practice of excluding some values in defining GUI states is also widely adopted in Mobile GUI testing studies [44, 38, 39].

Snapshot Manager provides VM snapshots of diverse app screens to the rest of GROUNDHOG's components.

6.2.2 Action Extractor

The *Action Extractor* component takes a VM snapshot of an app state as input and extracts a list of available actions from it. To that end, *Action Extractor* loads the snapshot on a VM equipped with an Accessibility Service such as UIAutomator. This service runs in the background and enables capturing a hierarchical representation of UI elements. *Action Extractor* performs further analysis on the dumped hierarchy of UI elements. It explores the tree of elements and searches for those that support action, e.g., have *clickable=true* in their attributes.

An action consists of two parts: the operation, e.g., click, and an identifier of the element on which the action is performed. The target element can be identified uniquely by its *apath*, i.e., the absolute path from the root to the target node in the UI hierarchy tree. For example, assuming the target element is the first "ADD" button in Figure 6.1(c), the corresponding *apath* is */Framelayout/LinearLayout/FrameLayout[2]/Button*. Also, the operation of this action can be determined from the

"clickable" attribute. Therefore, this action can be represented as the following JSON object that is passed to proxies to be executed in different interaction modes:

```
{
    operation: 'click',
    apath: '/Framelayout/LinearLayout/FrameLayout[2]/Button'
}
```

6.2.3 Oracle

The Oracle component is responsible for analyzing each app state and corresponding execution logs to determine if an accessibility issue exists in executing an action with a proxy.

For *locatability* issue, Oracle refers to failure reports of Proxy Users to check if the Proxy User was successfully locating the element. For *actionability* issue, Oracle first analyzes event logs to check if the events which are indicating a change in the content of the UI, i.e., *TYPE_WINDOW_CONTENT_CHANGED*, and executing an action, e.g., *TYPE_VIEW_CLICKED*, occurred. It also compares the app's previous state with the new state to ensure the event occurred. In comparing app states, Oracle compares their UI hierarchy similar to *Snapshot Manager* by comparing their hash values. However, Oracle does not exclude the same attributes as *Snapshot Manager* in calculating the hash value. For example, changes in the *text* attribute are not demonstrating a new screen for *Snapshot Manager* but can indicate an action execution. In the end, if the UI hierarchy before and after the action execution is the same, and there is no corresponding *AccessibilityEvent* of the executed action, the oracle reports an *actionability* issue for a given User Proxy.

Furthermore, the Oracle compares the *actionability* of each element across different Proxy Users to check if there exists at least one Proxy User that can successfully perform the action. This way, we are assured the element is associated with behavior (it is operative) and not just a decorative element.

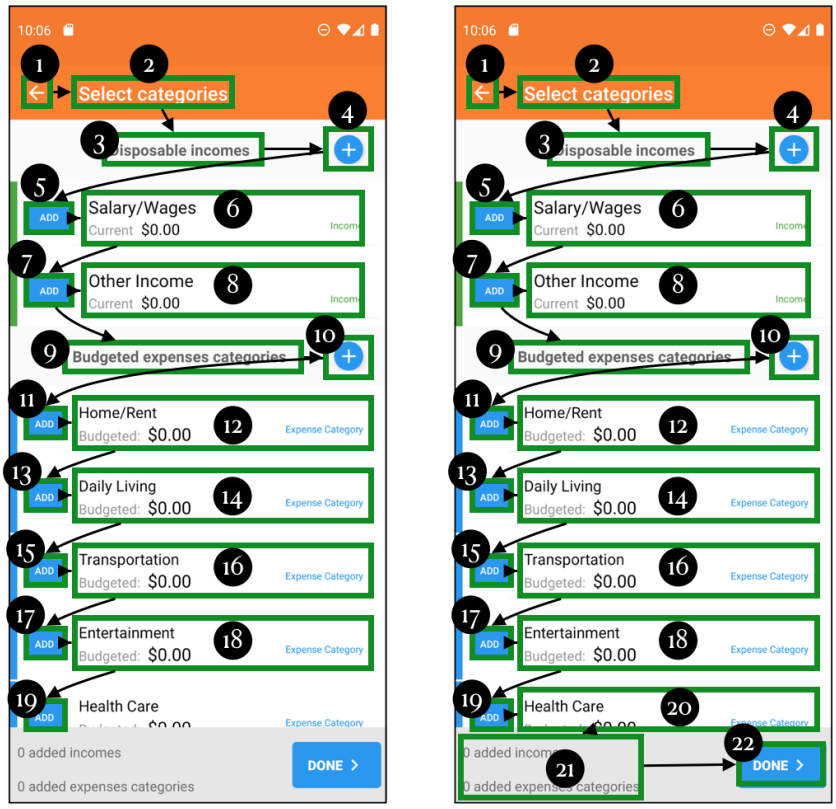


Figure 6.3: Locating (a) the last "ADD" button, and (b) the "Done" button with TalkBack Proxy in Linear Navigation. 18 Linear Navigation interactions in (b) are redundant since they have been performed in (a) already.

6.3 Optimization

In the previous section, we explained how, given a snapshot of an app, GROUNDHOG extracts all possible actions for each of them, and locates and performs the available actions using different Proxy Users. For example, Figure 6.3 depicts the process of locating two elements (a) the last "ADD" button, and (b) the "Done" button. Note that TalkBack traverses the UI hierarchy with each swipe starting from the top left element on the screen. As can be seen in Figure 6.3, the elements 1 to 19 appear both in (a) and (b). In other words, there is substantial redundancy between the steps required to locate these two elements.

We introduce an optimization technique using a memoization algorithm to minimize the number of interactions in the Linear Navigation mode without losing the accuracy of detecting locatability issues in an app. The basic idea is to memorize the elements that TalkBack has located directionally in previous action executions and start the exploration from the closest located element to the target element. To locate the target element, TalkBack Proxy first sends a direct *AccessibilityEvent*, called *ACTION_FOCUS* to element e which asks TalkBack to focus on it directly. The element e is a visited element in the past action executions of TalkBack Proxy, closest to the target element in the UI hierarchy. This way, all Linear Navigation from the start to the element e is bypassed, allowing the exploration to proceed much faster.

6.4 Implementation

GROUNDHOG is designed as a Client-Server architecture model where the server is on the host machine and the client resides on an Android device. The server side, implemented in Python, orchestrates the whole analysis from running an app crawler, taking snapshots, executing actions with Proxy Users, creating reports, and visualizing the results. The client, implemented in Java, is basically an accessibility service, i.e., Proxy Users, that controls the device to execute actions.

GROUNDHOG utilizes Android Debug Bridge (ADB) [14] to manage communications between the server and client. GROUNDHOG also modifies Stoat app crawler [116] and employs it to explore different states of the app. As discussed in Section 6.2, any app crawler can be used in GROUNDHOG. The rationale behind choosing Stoat is that it is completely open-source and conveniently works with the latest Android versions. It also has been widely used in previous studies. Lastly, Pillow [40] Python imaging library and Flask [97], python web framework, assist in visualizing the detected accessibility issues.

In our experiments, for actionability evaluation of GUI elements, we only focused on click actions

that are most commonly associated with app behaviors. However, GROUNDHOG can be similarly configured for any other type of action, e.g., long-click. Also, for the TalkBack Proxy User, we used two navigation modes (Linear and Touch) for the experiments as they are the most common ways to navigate the app.

6.5 Evaluation

We conduct several research experiments to evaluate GROUNDHOG and answer the following research questions:

RQ1. How effective is GROUNDHOG in detecting accessibility issues?

RQ2. How does GROUNDHOG compare to Google Accessibility Scanner (the official accessibility testing tool in Android)?

RQ3. What are the characteristics of the detected accessibility issues? How do they impact app usage for users with disabilities?

RQ4. What is the performance of GROUNDHOG? To what extent optimization improves its performance?

6.5.1 Experimental Setup

We evaluate GROUNDHOG on three different sets of real-world apps. First, a set of 20 random apps with more than 10 million installs in Google Play Store [19] (labeled as **P**). Second, 20 randomly selected apps from AndroZoo [3], a collection of Android apps collected from several sources including Google Play (labeled as **A**). All of these 40 apps are published in Google Play in 2021 and 2022. We also included 17 apps from the 20 apps that were evaluated by LATTE (labeled as

L).[2] Out of the 17 apps from the LATTE Latte dataset included in our study, 11 have confirmed accessibility issues.

In total, our dataset consists of 57 apps that have been published in 21 different categories in Play Store. The complete list of datasets can be found on our companion website [114]. We ran GROUNDHOG on each app until at least 10 states (screens) were captured (in total 570 different states).

To answer the research questions, we carefully examined the results to check if the reported issue is correct (true positive) or wrong (false positive). Therefore, we create a smaller set of results by selecting 5 UI states from 10 apps in each dataset (P, A, and L). In total, a set of 150 different UI states with 1,133 actions is created which can be seen in Table 6.1 (sorted based on installs).

All experiments were conducted on a typical computer setup for development (MacBook Pro, 2.8 GHz Core i7 CPU, 16 GB memory). We used the most recent distributed Android OS at the time of experiments (SDK30), and the latest versions of assistive services, i.e., TalkBack 12.1 and SwitchAccess 12.1.

6.5.2 RQ1. Effectiveness of GROUNDHOG

Table 6.1 summarizes the accessibility issues detected by GROUNDHOG. The *Actions* column represents the total number of extracted actions from all different states of the app and the number of actions that GROUNDHOG found to be operative, i.e., leading to a modification in the GUI state. As shown in the Table, on average, each snapshot has 7.5 actions to be evaluated by Proxy Users. The columns entitled *TalkBack Unlocatable*, *TalkBack Unactionable.*, and *Abstract Unactionable* represent locatability and actionability issues by TalkBack Proxy Users, and actionability issues by Abstract Proxy User, respectively. For each type of issue, we show the total number of detected

[2]We had to exclude 3 outdated apps that do not work at the time of the experiments.

Table 6.1: The evaluation subject apps with the details of detected accessibility issues by GROUNDHOG

Id	App	Category	#Installs	#Actions		TalkBack Unlocatable		Talkback Unactionable		Abstract Unactionable			#All Issues		Scanner
				Total	Operative	Total	TP	Total	TP	Total	TP	SA	Total	TP	
P1	Instagram	Social	>1B	31	17	0	0	0	0	0	0	0	0	0	9
P2	FacebookLite	Social	>1B	20	18	14	14	0	0	7	6	6	21	20	33
P4	Zoom	Business	>500M	26	25	1	0	0	0	0	0	0	1	0	13
P7	MicrosoftTeams	Business	>100M	23	19	0	0	2	0	2	0	0	2	0	6
P11	MovetoiOS	Tools	>100M	12	10	2	2	0	0	0	0	0	2	2	11
P12	Bible	Books	>50M	44	39	6	6	0	0	0	0	0	6	6	20
P13	ToonMe	Photography	>50M	48	41	18	17	1	0	0	0	0	19	17	43
P19	Venmo	Finance	>10M	24	17	0	0	0	0	0	0	0	0	0	6
P21	Lyft	Navigation	>10M	21	18	2	0	0	0	0	0	0	2	0	2
P22	Expedia	Travel	>10M	40	34	9	6	0	0	0	0	0	9	6	71
A1	YONO	Finance	>100M	92	59	54	41	9	9	1	1	1	64	51	39
A2	NortonVPN	Tools	>10M	21	16	9	8	1	0	0	0	0	10	8	8
A3	DigitalClock	Tools	>10M	57	42	7	7	0	0	1	0	0	8	7	21
A5	To-Do-List	Productivity	>5M	45	32	2	1	0	0	0	0	0	2	1	19
A6	Estapar	Vehicles	>1M	41	31	23	21	2	0	0	0	0	25	21	11
A9	MyCentsys	House	>10K	34	19	0	0	0	0	9	9	9	9	9	14
A10	HManager	Productivity	>10K	17	17	2	2	0	0	0	0	0	2	2	5
A11	Greysheet	Lifestyle	>10K	44	24	1	0	0	0	19	18	18	20	18	10
A13	MGFlasher	Vehicles	<10K	54	37	5	5	2	2	6	6	6	11	11	19
A18	AuditManager	Productivity	<10K	15	10	0	0	5	5	5	5	5	5	5	6
L3	Yelp	Food	>50M	62	56	10	9	0	0	0	0	0	10	9	9
L4	GeekShopping	Shopping	>10M	29	28	5	3	0	0	0	0	0	5	3	13
L5	Dictionary	Books	>10M	42	38	3	1	0	0	2	1	0	5	2	16
L6	FatSecret	Health	>10M	37	37	11	9	1	1	0	0	0	12	10	14
L8	SchoolPlanner	Education	>10M	52	48	8	8	0	0	1	0	0	9	8	52
L9	Checkout51	Shopping	>10M	29	22	6	6	0	0	0	0	0	6	6	4
L11	TripIt	Tavel	>5M	52	39	9	8	0	0	0	0	0	9	8	6
L12	ZipRecruiter	Business	>5M	31	27	1	0	0	0	0	0	0	1	0	5
L13	Feedly	News	>5M	63	34	34	34	14	12	24	23	23	58	57	1
L15	BudgetPlanner	Finance	>1M	27	25	2	0	6	6	6	6	6	8	6	26
Total				1133	879	244	209	43	34	83	75	74	341	293	512
Precision						0.85		0.79		0.90			0.86		

issues and the number of issues manually verified by authors or True Positives (TP).

To verify if an issue is detected correctly by GROUNDHOG, we load the corresponding snapshot on an emulator and interact with the app manually. For TalkBack locatability issues, we explored the app using TalkBack's two navigation modes, i.e., Linear and Touch, and check if the target element cannot be located in either way. Note that since Abstract Proxy directly interacts with the corresponding *AccessibilityNodeInfo* objects, it has no locatability issue.

For the actionability issues, first, we perform the action with touch (by tapping on the element) and observe the changes in the app state, e.g., by tapping on a checked box, its state changes, or by clicking a button, a new page may appear. Once we confirmed the target element is associated with an action by touch, we reload the snapshot to the same state two other times. The first time, we use TalkBack to click on the element (double tap), and the second time we send *ACTION_CLICK* to the target element using ADB and GROUNDHOG. Then we monitored all changes to see if anything happened. We follow a conservative strategy and assume that any changes after clicking (even if it is not the same as the change after tapping the element) show the element is actionable.

With the number of verified issues (TPs), we evaluated the effectiveness of GROUNDHOG in terms of Precision as the ratio of the number of TPs to the number of all detected issues. We also report Action Coverage and Recall of GROUNDHOG as follows.

Precision

The number of locatability and actionability issues that are confirmed manually are shown in Table 6.1. In total, GROUNDHOG could detect 293 true accessibility issues with a precision of 86%. Two-thirds of the apps in our test set have locatability issues. Note that, when an element is not locatable by TalkBack, it cannot be verified if it is actionable. Therefore, the number of TalkBack Proxy User actionability issues is expected to be less than Abstract Proxy actionability issues. A9 and A11 are the only two exceptions in our test set. Our further investigations of these

apps reveal that TalkBack dispatches touch events to the screen when performing *ACTION_CLICK* fails. TalkBack utilizes this workaround to overcome some accessibility issues in apps.

Our analysis of GROUNDHOG's failures showed that 39 out of 48 false positives could be fixed by rerunning GROUNDHOG on the app for the second time. The reason for these failures in the first attempt is the improper timing between performing an action and retrieving the results from the device, e.g., some of *AccessibilityEvent*s are not captured, which is a common challenge in dynamic analysis techniques due to concurrency issues.

In a few of the false positives, although the assistive services did not make any changes to the app's state, the changes by touch interaction do not contribute to any functionality of the app. For example, Figure 6.4 (a) shows the login page of MicrosoftTeams app (P7). Clicking on the email text box on the login page results in different behaviors based on the way it is performed. When a user with an assistive service clicks on the text box, nothing happens; however, if a user touches the text box, the decorative figure disappears, as shown in Figure 6.4 (b). GROUNDHOG reports this as an actionability issue. However, since this change does not impact assistive-service users, we mark it as a false positive.

Some false positives happen because of changes in the app state during exploration. For example, GROUNDHOG reports a button in a slider list of the To-Do-List app (A5) as locatabilty issue, as shown in Figure 6.4 (c). However, the reason behind this is that the element is the last item on the list and when TalkBack focuses on it, the sliding widget fetches new elements and moves the elements to the front. This changes the GUI hierarchy layout and GROUNDHOG does not realize the current first element is the same as the last element on the list seen previously. Moreover, GROUNDHOG detects a TalkBack actionability issue for the SchoolPlanner app (L8), as shown in Figure 6.4 (d), since performing a click on the focused element does not change the UI state (since the tab is already active). However, by touching on the tab, we are in fact touching on the overlay, resulting in the disappearance of the overlay element.

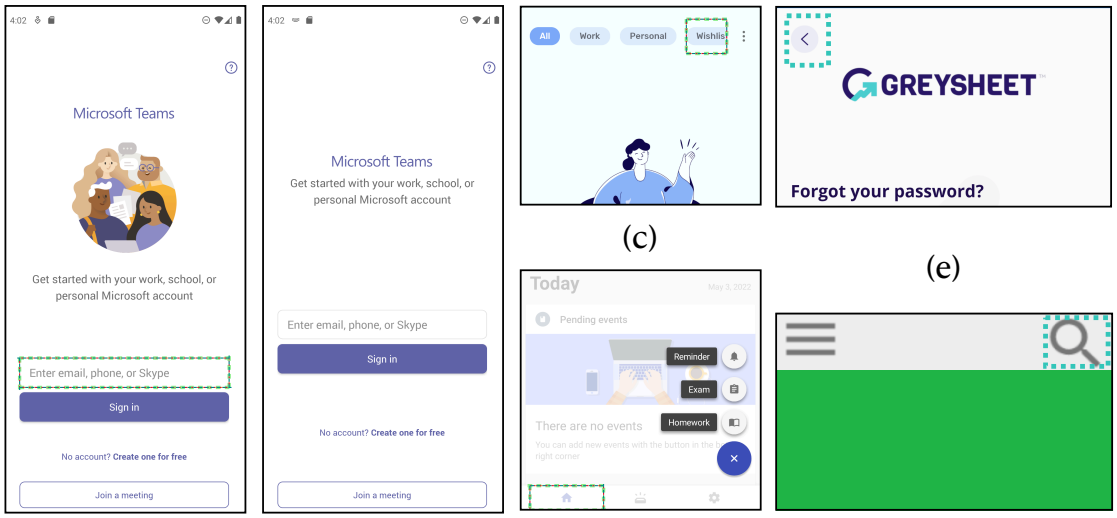


Figure 6.4: (a-d) are examples of false positives, and (e-f) are examples of missing actions in GROUNDHOG

Action Coverage

To understand the effectiveness of GROUNDHOG in extracting all possible actions from the screen, we manually examined all 150 UI states by touch interactions to extract the set of all elements that are associated with an action. In total, we found 1,149 actions, where GROUNDHOG could extract 1,133 of them (98% action coverage). In cases that GROUNDHOG missed an action, there was a custom-implemented UI widget without proper specifications for accessibility services. For example, two missing actions, back and search buttons from apps Greysheet and Feedly apps (A11 and L13), depicted in Figure 6.4(e), and (f), are layouts with attribute *clickable* set to *False*. Thus, GROUNDHOG cannot identify them as actionable elements.

Recall

To calculate the recall of GROUNDHOG in detecting real accessibility issues, we used the set of detected accessibility issues by LATTE as the ground truth. In total, LATTE found 12 accessibility

issues, where 10 of them could be detected by GROUNDHOG (83% recall in detecting existing issues). One false negative happens for the Feedly app where GROUNDHOG did not extract the search button, depicted in Figure 6.4 (e), as an action. The other false negative happens in the Dictionary app, where the accessibility issue can be revealed after performing three consecutive actions on the app. Since GROUNDHOG analyzes an app only with one action, this issue could not be detected. We also found 87 new accessibility issues in the dataset of apps from LATTE that were not detected by LATTE.

In comparison with LATTE, we can see GROUNDHOG is able to detect a much larger number of accessibility issues. This is mainly because LATTE assumes the availability of manually written GUI tests and does not achieve the same level of coverage as GROUNDHOG that uses a crawling technique. At the same time, in a few cases, GROUNDHOG is missing certain accessibility issues that are detected by LATTE because manually written tests can exercise non-native UI elements that do not have a proper specification for accessibility services (i.e., attributes of *AccessibilityNodeInfo* object are not properly set), while GROUNDHOG cannot properly analyze such elements.

6.5.3 RQ2. Comparison with Scanner

Google Accessibility Scanner [9], or Scanner for short, is the most widely used accessibility analyzer for Android. Scanner leverages Accessibility Testing Framework (ATF) [10] to evaluate screen accessibility. To compare GROUNDHOG with Scanner, we analyzed all the examined app states in Table 6.1 with Scanner and checked what it reports. The last column of Table 6.1 displays the number of issues detected by Scanner. By comparing the accessibility issues reported by GROUNDHOG against what Scanner reports, we found that there is no intersection between the type of issues each of them detects. Scanner evaluates a screen against predefined accessibility rules and reports issues such as low contrast, small touch target size, and missing speakable text for unlabeled icons. It cannot detect issues related to interactions with an app using assistive services.

However, the accessibility issues reported by Scanner are also important to be addressed to have an accessible app. We believe that GROUNDHOG complements Scanner and other ATF-based testing techniques [7, 45, 61] in evaluating app accessibility.

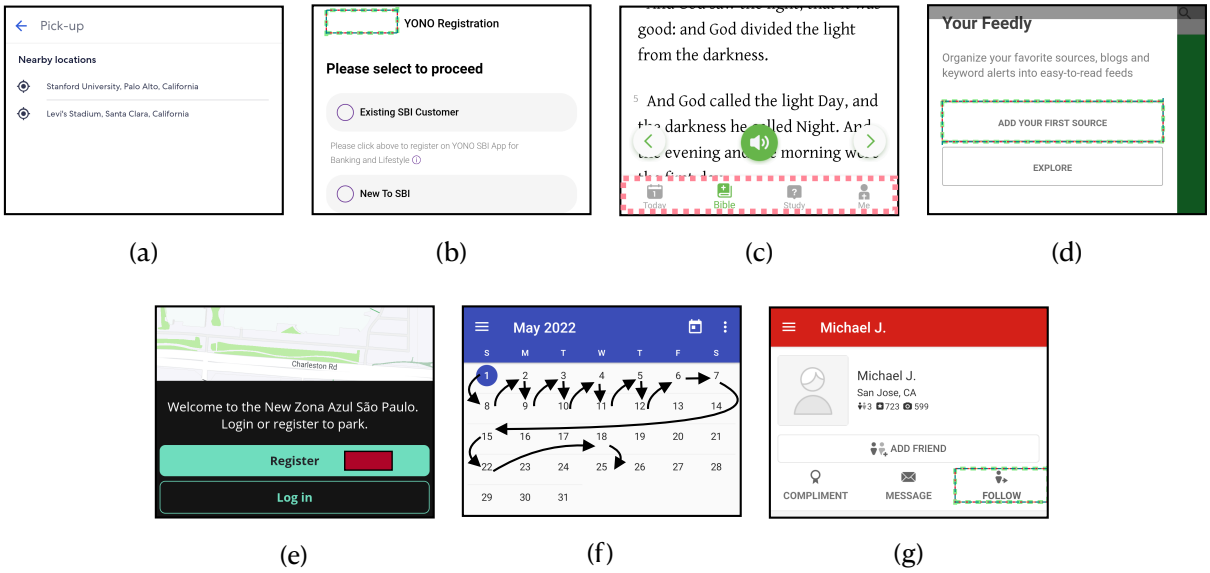
6.5.4 RQ3. Qualitative Study

We manually examined all the detected accessibility issues to understand how the issues affect users with a disability and what are their root causes. We found four different categories of issues as follows.

Unlocatable elements with TalkBack

GROUNDHOG evaluates locatability of elements by TalkBack in using both Linear and Touch Navigation modes. In severe cases, neither of these modes can locate an element. For example, Figure 6.5 (a) shows a screen in the Expedia app where none of its elements, even the back button, can be detected by TalkBack. We found that the root cause of this issue is having the *important-for-accessibility* attribute set to false, meaning that TalkBack should treat them as decorative elements and skip them in exploring the app. Developers should set this attribute properly. We found this issue in Facebook, Expedia, Checkout51, ToonMe, SchoolPlanner, and Yelp apps.

In some cases, the element can be located by Linear Navigation, but not by Touch Navigation. For example, Figure 6.5 (b) depicts the entry screen of YONO (a banking app), where the highlighted button can be located by Linear Navigation, yet, the element does not get accessibility focus when touched. This issue happens when there is an overlap among the active elements on a screen, similar to Figure 6.5 (b), where the highlighted button is placed under the top layout. Such elements confuse users about the screen's content and may also have security implications when a malicious functionality is hidden by malware authors in such elements. This finding motivates us to examine



(a) (b) (c) (d)

(e) (f) (g)

Figure 6.5: Qualitative study of GROUNDHOG's report on subject apps

the impacts and root causes of such issues, which will be explained in the following chapter. This type of issue can be found in YONO, Feedly, Dictionary, Estapar, TripIt, NortonVPN, Facebook, DigitalClock, ToonMe, AuditManager, and SchoolPlanner apps.

The remaining cases of locatability issues occur in elements that TalkBack skips in Linear Navigation but can be focused on by touch. For example, Figure 6.5 (c) shows a part of the Bible app, when the user uses TalkBack in Linear Navigation mode and reaches the end of the text, the highlighted bottom menu disappears. For a sighted user who sees all the changes on the screen, the disappearance of the menu can aid in reading the rest of the text more conveniently; however, it confuses blind users who may not even know the menu exists in the first place. This type of is issue have been seen in LATTE as well. The FatSecret, Geek, ToonMe, TripIt, Bible, MoveToiOS, and HManager apps have this type of issue.

Actionability

This issue manifests itself when an assistive service cannot be used to perform an action. GROUNDHOG could find this type of issue in Facebook, Dictionary, Feedly, BudgetPlanner, MyCentsys, Greysheet, MGFlasher, AuditManager, FatSecret apps. For example, Figure 6.5 (d) shows a button in Feedly app that can only be clicked by touch.

Generally speaking, Abstract Proxy has more capabilities than TalkBack in performing actions as it uses Accessibility API to directly click on *AccessibilityNodeInfo* object. However, this was not the case in MyCentsys and Greysheet apps. Our further investigation and study on TalkBack source code [21] revealed that TalkBack utilizes a workaround to mitigate accessibility issues in apps. Talkback first uses Accessibility API to perform and check if the action is sent successfully; otherwise, it sends a touch event to the center of the focused element. Although this workaround may address inaccessibility in some situations, it may confuse users even more in some other situations. For example, Figure 6.5 (e) highlights a button under the Register button with the text "Help". However, when a TalkBack user double taps, the Register button is clicked instead.

A common theme of apps with actionability issues is that they are developed using hybrid frameworks or utilize WebViews [23]. Hybrid frameworks enable a developer to implement mobile apps in one codebase with one language, like C# in Xamarin[91]. Similarly, WebView renders web elements that are developed in HTML, CSS, and JavaScript code in mobile apps. One of the advantages of hybrid apps and Webviews is reusing the same code on different platforms, like iOS, Android, and even the Web. We could find YONO, ToonMe, Estapar, and Greysheet apps in Apple Store with similar accessibility issues detected by GROUNDHOG, manifested by Voiceover (the iOS' official screen reader). We believe further studies are required to assess the accessibility issues resulting from hybrid frameworks.

Counterintuitive Navigation

One type of information produced by GROUNDHOG as part of its reporting is short videos in GIF format showing how Talkback navigates linearly to reach an element. Checking these videos revealed a new type of accessibility issue where developers set an unexpected traversal order for elements. For example, Figure 6.5 (f) shows the visiting order of a calendar's elements in SchoolPlanner. As seen, there is no pattern in visiting the elements. In another example, Yelp's home page has counterintuitive navigation where the search button (which is at the top of the page) will be reached when all other elements have been visited.

Inoperative Actions

We examined the inoperative actions reported by GROUNDHOG to see how they impact users with disabilities. Such clickable elements without any impact on the app content increase the number of interactions for TalkBack users to reach an element. For example, it takes 25 directional navigation to reach the farthest element in a state of DigitalClock; however, if the inoperative actions are removed by developers it can be reduced to 20 interactions, saving 20% of time spent by users with disabilities.

However, in some instances, there is a usability bug in inoperative actions which concerns users without disabilities. For example, Figure 6.5 (g) shows a profile page of a user in Yelp where GROUNDHOG detects the Follow button is not operative. Here, the other buttons in the same row (Compliment and Message) are associated with an action (the login page appears). It seems, there is a bug that makes the Follow button inoperative.

6.5.5 RQ4. Performance

We measured the time that GROUNDHOG takes to create reports to understand how GROUNDHOG can be integrated into the development lifecycle. For an app on average, GROUNDHOG takes 3,541 seconds to explore an app, execute all actions using different Proxy Users, and produce an accessibility report with visualized information. Since GROUNDHOG does not require any manual input from developers, analyzing an app in less than an hour is completely practical, and can be done on a nightly basis.

The breakdown of the execution time is as follows. The app crawler (Stoat) takes 420 seconds on average to explore different states of the app. The action extraction part virtually takes no time (less than a second). The heavy part of GROUNDHOG is executing each action via Proxy Users. GROUNDHOG executes each action in 21, 24, and 40 seconds for Abstract, Touch, and TalkBack Proxy Users, respectively. There are some common time-consuming parts for all Proxy Users: reloading snapshot takes 4.1 seconds, reconnecting ADB takes between 2 to 12 seconds, and GROUNDHOG waits for 5 seconds after each action is executed to ensure all changes in the app state are finalized. TalkBack Proxy User takes more time to execute because the communication between GROUNDHOG and TalkBack is a slow process since GROUNDHOG actually performs touch gestures and waits for TalkBack to change its internal state.

GROUNDHOG's performance can be improved significantly by parallelizing the snapshot analysis thanks to its Client-Server model. Each VM snapshot is less than 1GB of data and can be easily transferred in less than 10 seconds.

Locating an element using TalkBack Proxy takes 9.71 seconds on average per action. Without our optimization technique, it would take 26 seconds on average. In other words, the optimization improves the performance of this aspect of GROUNDHOG by more than 2.5 times per action, which reduces the app analysis time by 10 minutes on average.

6.6 Threats to Validity

External validity. A key threat to validity is preserving the state of the app under test since three different Proxy Users should perform the same action on the same element. We mitigate this threat by capturing a VM snapshot of the device used for all Proxy Users. The virtualization technique may not preserve the state of apps that update their content dynamically or retrieve information from the server. For example, in a shopping app, if one proxy adds an item to an empty shopping cart that is synchronized with an external database, the same VM snapshot may be in a different state when it is loaded for another Proxy User. We have not observed this situation occurring in our experiments; however, to prevent reporting false positives/negatives in similar cases, we check the UI hierarchy of the apps after loading the VM snapshots. If they are not exactly similar, we report a flag indicating that the VM snapshot is different and the result may not be reliable. It would be interesting for future work to examine elegant solutions for handling dynamic and online content.

Another threat resides in the variety of actions supported by GROUNDHOG. Our current implementation supports clicking action. Other touch gestures are not implemented. Although clicking is one of the most essential touch gestures for interacting with GUI elements, our claimed benefits of GROUNDHOG can be more confidently generalized by providing and evaluating support for other types of actions. However, it is worth noting that most other complex touch gestures, like pinching in/out or double-tap, are not supported by assistive services in the first place. For example, pinching can be used for zooming in on an image, but it does not have an equivalent in TalkBack since blind users may not see visual images.

Internal validity. We implemented GROUNDHOG using several libraries and tools, including *ADB*, *Android Virtual Device*, *Stoat* [57], and *AccessibilityService* in Android, which may introduce defects in the crawling and analysis steps of our implementation. Furthermore, our prototype may contain bugs in its implementation. We have tried to minimize this threat by upgrading all libraries to the latest available versions, writing automated unit tests, and conducting code reviews. In

addition, we tested the prototype extensively on numerous popular Android apps.

6.7 Conclusion

This chapter introduced GROUNDHOG, a fully automated assistive-service driven accessibility crawler to detect accessibility issues that only manifest themselves through interactions with the app. GROUNDHOG explore apps and assess the locatability and actionability of each element on the screen using different interaction modes provided by assistive services.

One result of the experiments with GROUNDHOG was detecting elements that can be accessed with TalkBack by Linear Navigation but cannot be accessed by touch interaction. This observation encouraged us to investigate the idea of over-accessibility when some information or functionalities should not be accessible to any user; however, users with assistive services can access them. The next chapter explains our studies and finding regarding this matter.

Chapter 7

Over-Accessibility Issue Detection

Principles of universal design [41] dictate that technologies and services, including mobile apps, must be accessible to everyone regardless of their abilities. These principles are often overlooked in development practices, where developers build and test their apps based on the assumption that by default, a user views the app content on the screen and interacts with it by touch. Such assumptions exclude about 15% of the world's population with some form of disability, especially users with visual and fine-motor impairments.

Prior chapters have shown that many apps are shipped with functionalities that are not accessible using assistive services. We call this the *under-access* problem. In this chapter, we look at the dual of this issue, which we call the *over-access* problem. That is, some apps are shipped with functionalities that in certain states can be accessed using assistive services but not otherwise.

An element is *Overly Accessible* (OA) when it provides more information and functionality to assistive-service users than regular users. In security-sensitive apps, OA elements can jeopardize the security of password-protected apps such as banking, investment, health, etc. Case in point, for several iOS versions, users have reported scenarios of using VoiceOver, the standard screen reader in iPhones, to bypass iOS passcode and gain access to contacts, photos, notes, etc [35, 72, 71].

Moreover, OA elements can be used to provide unauthorized access to premium functionalities in apps with in-app purchases, endangering around 60% of companies on app stores that derive revenue from such functionalities in their apps [93]. As an example, the Mediation Moments app [34] has premium articles that are available to subscribed users; however, we found that an AT user can read these articles without purchasing the subscription. Lastly, bypassing the designed workflow can result in invalid inputs to be provided to an app, breaking its logic and leading to unexpected crashes. For example, in using the Airbnb app to book a place, the "decrement" button is disabled for touch when there is only one traveler, preventing zero and negative inputs. We found that an assistive-service user can still click this button and submit a request for a room for a negative number of people.

Interestingly, over-access also degrades the accessibility of apps. Blind users utilize screen readers to navigate through the elements on a screen sequentially. Even if the OA elements are not security-sensitive, presenting information that the developer did not intend to be available on the screen can confuse the screen-reader users. OA elements also increase the number of required interactions to reach the desired element, resulting in a less optimal user experience.

Despite the severe impacts of OA elements, they have received practically no attention in prior accessibility analysis of apps or security-related studies. Neither Google Accessibility Scanner [9], nor Apple Accessibility Inspector [28] check any rules for over accessibility. They only check a set of accessibility rules (e.g., proper text size and color) on currently displayed UI elements. Most other accessibility testing studies [17, 104] extend the accessibility rules of standard scanners and cannot detect OA elements consequently. Prior security-related studies [95, 64] have investigated the feasibility of constructing malicious software (e.g., malware) to launch a security attack by exploiting accessibility APIs. No prior study has investigated the vulnerabilities caused by OA elements in benign apps that can be readily exploited by any user, and using the standard ATs.

To fill this gap, we conducted an empirical study on 100 different UIs from 20 randomly selected apps to understand OA elements and their specifications. We then developed a tool, OVERSIGHT,

to automatically detect them on a given state of the app.

OVERSIGHT first leverages the findings of our empirical study and devises a static checker to analyze currently displayed UI elements and localize *OA smells*, i.e., elements with one of the OA characteristics that may lead to revealing information or functionality that is unavailable for sighted users and available for assistive-service users. Then, OVERSIGHT validates the accessibility of these elements dynamically using TalkBack and Proxy Users. Finally, OVERSIGHT reports accessibility issues resulting from OA elements. Our empirical evaluation on 30 apps reveals that OVERSIGHT can precisely detect more than 83% of OA elements.

The remainder of this chapter is organized as follows. Section 7.1 motivates this study with an example and provides background information. Section 7.2 introduces different classes of OA elements according to our empirical study. Section 7.3 explains OVERSIGHT, an automated approach to detect OA elements. Finally, in Section 7.4, the evaluation of OVERSIGHT on real-world apps is presented.

7.1 Motivating Example

Figure 7.1 shows screenshots of AppLock [69], a popular app locker with more than 5,000,000 installations and rating of 4.2. As shown in Figure 7.1 (a), the app lists all the installed apps on a phone on its first page, enabling users to add a lock to any desired app. App lockers protect themselves and other requested apps by preventing access to their content without providing a secret pattern or passcode. When a user opens the AppLock or any locked apps, e.g., Files or Messages as shown in Figure 7.1(a), she first sees the lock screen, depicted in Figure 7.1(b), and should first unlock it with a preset pin. Many other types of apps (e.g., investment, health monitoring, diary, etc.) employ a similar protection strategy for their contents.

A user without disability can see the pin pad and the text asking to “Enter pin” on the screen.

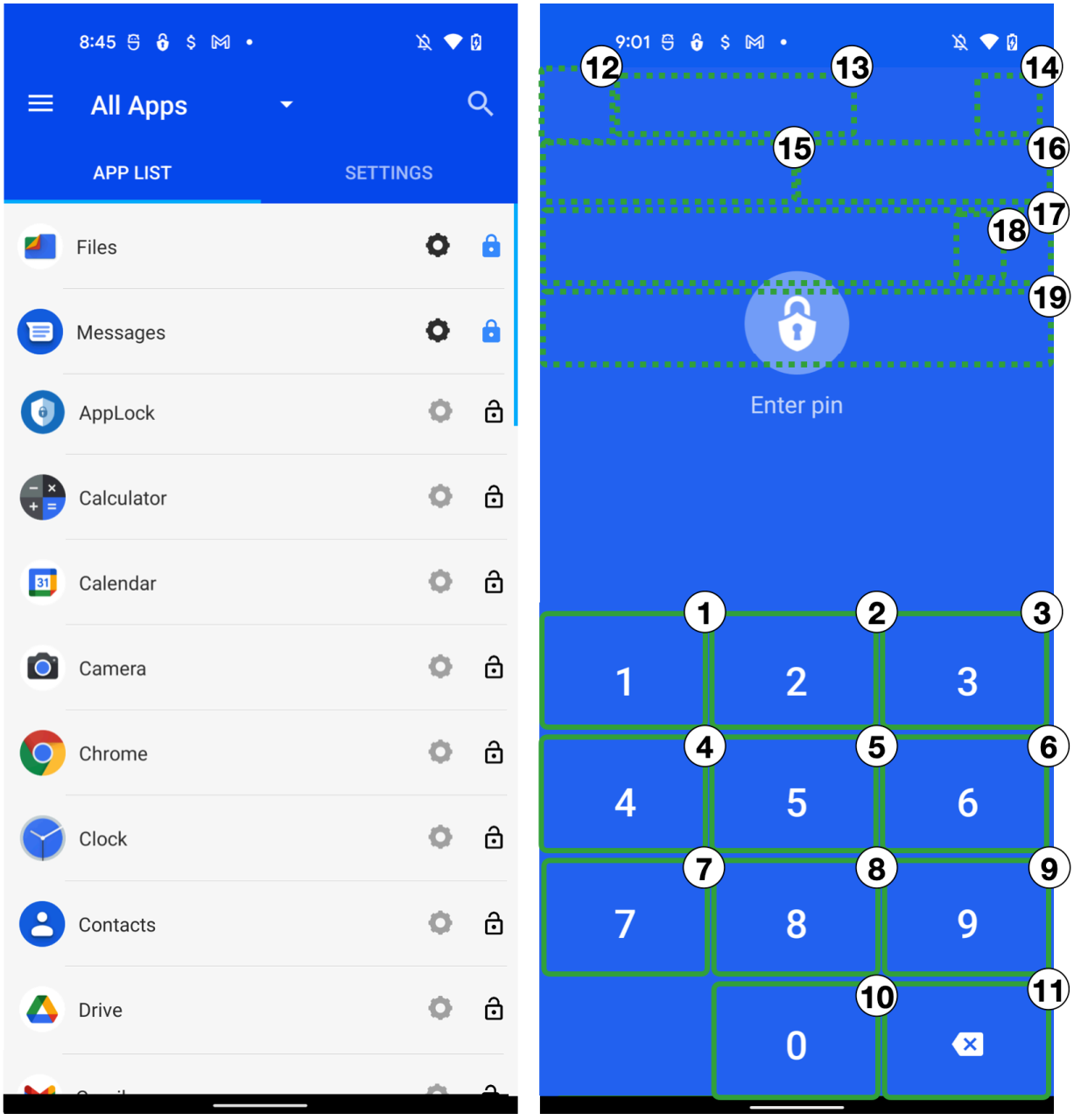


Figure 7.1: Built-in lock for a security-sensitive app.

She would try to unlock the app by entering the pin through touching the numbers on the screen. However, a user with disability has to rely on assistive services to interact with apps.

Unfortunately, developers oftentimes only test their apps' functionality under conventional ways of interaction, leading to many inaccessible functionalities in apps. A developer who is aware of the disabled users' limitations may utilize accessibility testing tools, such as Google Accessibility Scanner [9], to evaluate the accessibility of their app. For example, for the lock page of AppLock, Accessibilty Scanner reports an issue for the text contrast of "Enter pin". Accessibility Scanner may also report "missing speakable text" if there is a clickable image without a content description, or "small touch target size" if the clickable area is too small for an element. Google Accessibility Scanner, as well as all other prior accessibility testing tools (e.g., [112, 113, 5, 9, 20, 90]), are aimed at finding *under-access*, i.e., features that should be available to the user but cannot be accessed using assistive services. None of these tools report issues related to *over-access*, i.e., features that should not be available to the user but can be accessed using ATs.

In practice, a blind user may need to understand the screen content by exploring and navigating through all the elements on the screen. Figure 7.1(b) shows which elements can be focused by TalkBack. The numbers indicate the order in which elements are focused. After passing pin pad elements, TalkBack detects some elements that are not visible to sighted users. We call these elements Overly Accessible (OA) as they are not visible to sighted users or clickable by touch. Announcing these elements not only misleads the blind user about the content of the page, but in many cases also requires an exorbitant number of interactions to pass a long list of OA elements until the user reaches the visible functionality that the developer intended to be available. Such OA elements remain undetected in the prior accessibility testing tools.

These OA elements, as specified in Figure 7.1(b), can also pose security concerns. By listening to what TalkBack announces, we can understand that the OA elements correspond to the first page of AppLock as shown in Figure 7.1(a). This page contains the list of device apps and the mechanism to enable or disable their locks. For instance, element 17 in Figure 7.1(b) is the lock

toggle for the Files app. This means that, using TalkBack, a user can access the locked apps and disable their protections, without even entering the pin code. In essence, she can bypass the lock screen protection. Prior research has demonstrated how Accessibility APIs can be used by malware authors to launch a security attack [95, 64] and how to prevent such attacks [103, 99]. No prior work, however, has aimed to develop a method of assisting developers with detecting vulnerabilities caused by OA elements in benign apps that can be readily exploited by any user, and using the standard assistive services.

To fill this gap, we took a deeper look at how UI elements are represented to assistive services. In modern platforms such as Android, Accessibility Service runs in the background and provides the required information about a window's content to assistive services. From the perspective of Accessibility Service in Android, a window's content is presented as a tree of `AccessibilityNodeInfos` (nodes) [53]. Android 12 documentation lists 65 different types of information that are provided by nodes. Table 7.1 illustrates a sample set of this information. We hypothesize that nodes with peculiar specifications can lead to OA elements. For example, in Figure 7.1(b), by comparing the Bounds and DrawingOrder of elements, the second and third method in Table 7.1, we found that the layout that expands the whole window is drawn on top of some of the elements. While the elements underneath are covered for a sighted user, an assistive service can still navigate through them and announce them to the user. Our objective in this study is to study specifications of OA elements and propose an automated tool to detect such OA elements that can have severe security, privacy, and accessibility impacts on apps.

7.2 Overly Accessible Elements

An element is OA if it is exposing more information/functionality to assistive services than what is available through the conventional interaction mode. To understand to what extent node specifications can reveal OA elements, we perform an empirical study on manually detected OA elements on

Table 7.1: Sample types of information exposed from nodes to assistive services.

	Attribute	Description
1	**ActionList**	The actions that can be performed on the node.
2	**Bounds**	The coordinates of the bounding box of the node.
3	**DrawingOrder**	The drawing order of the view of this node.
4	**Text**	The text of this node.
5	**Enabled**	Whether this node is enabled.
6	**VisibleToUser**	Whether this node is visible to the user.
7	**Clickable**	Whether this node is clickable.
8	**ContentDesc**	The content description of this node.
9	**ChildCount**	The number of children.
10	**PackageName**	The package this node comes from.

some real world apps. In this section, we explain the data collection and results of this study.

7.2.1 Data Collection

Our goal is to collect all the available information from nodes to assistive services. To that end, we first developed an Accessibility service, called OVERSIGHT Service (OSS), which is capable of capturing different types of information exposed from nodes. OSS runs in the background on an Android device and receives commands from Android Debug Bridge (ADB) [14], a command line tool that ships with Android devices. Using this service, we conducted an empirical study on 100 different screens of 20 real world apps. Our app list consists of 5 apps with built-in lock from Google play and 15 randomly selected apps from 38,106 apps that were published in 2021 in AndroZoo [4] (without any intersection with apps evaluated in LATTE or GROUNDHOG). We installed each app on a Google Pixel 4 device, along with OSS. Then, we interacted with each app to find 5 different states and explored each state with TalkBack and without it. We aimed at finding elements that are not visible to sighted users but TalkBack announces them or performs an action on them. We utilized OSS to dump OA nodes screenshot and specification in the hierarchy of nodes.

We then performed open coding of these elements iteratively. We coded the elements, noting any condition that was not discovered before. To facilitate efficient coding, we developed a web application to visualize unannotated elements with search and batch tagging capabilities. In this way,

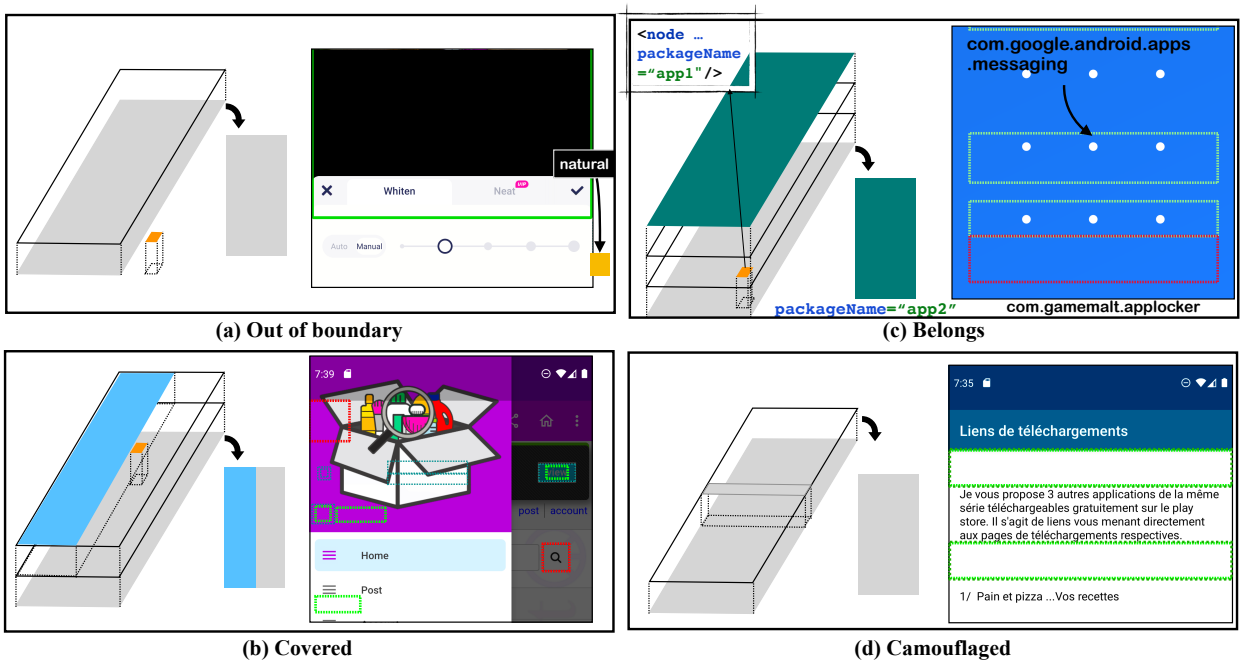


Figure 7.2: Over Accessibility Conditions.

we can search and tag elements in batches using queries specified by different types of information from nodes, for example, $\texttt{Text} \neq \emptyset \vee \texttt{ContentDesc} \neq \emptyset$ filters elements without any information.

7.2.2 Results

We categorized the conditions of OA elements that were yielded during the coding procedure into two main classes:

- **Overly Perceivable:** elements that reveal content to an assistive service that is not available through regular interaction mode.
- **Overly Actionable:** elements that provide action to an assistive service that is not available through regular interaction mode.

These classes are inline with two accessibility principles from Web Content Accessibility Guidelines (WCAG) [118]: (1) Content should be equally perceivable by different users [119], and (2) UI

elements should be equally operable by different users [120]. These principles can be violated due to bias in the level of access granted to any type of user, e.g., screen reader users vs. sighted users. While providing more access through conventional interaction modes, i.e., under-access problem, has been studied extensively and supported by a series of guidelines, not many works have investigated its counterpart, i.e., over-access problem. Our study is based on these principles and we organize detected OA elements' conditions under them. These conditions can be considered as accessibility guidelines to be later expanded or tailored to different platforms. Below, we list the conditions of OA elements we found in Android apps.

Overly Perceivable

A node with a textual data or content description is Overly Perceivable if it cannot be read or viewed by a sighted user, but can be accessed through programmatic means. We found the following conditions for such elements that are *hidden* to sighted users:

P1. Out of boundary: Nodes that are outside of the screen boundary, either with negative coordinates or with coordinates exceeding the device size. The left side of Figure 7.2(a) illustrates a schematic of the app screen with different layers corresponding to the drawing order of comprising elements. The orange element is OA as it is out of screen boundary and is not visible on the rendered screen. Figure 7.2(a) also shows a real example in our empirical study on the right.

P2. Covered: Nodes that are covered by other nodes in the rendered UI. Dashed boxes in Figure 7.1 are examples of covered nodes. Figure 7.2(b) also schematically shows how the orange OA element is covered by a blue sliding pane.

P3. Zero area: Nodes whose bounding box has zero area. These nodes will not be depicted on the screen but can be focused by an AT that will announce their content.

P4. Android invisible: Nodes that are not out of screen boundary and have positive area but they

are specified as invisible to user.

P5. Invalid bounds: Nodes whose captured bounds contradict the bounding box definition in Android documentation. The bounds attribute is supposed to be presented as the coordinates of the top-left and bottom-right points of the box. For example, if the coordinates of the ending point are smaller than the start point, the node has invalid bounds.

P6. Belongs: Nodes that belong to a package name that is different from the app under test. Left side of Figure 7.2(c) illustrates that the green screen from app2 is placed on top of the elements of app1. In the rendered screen, the elements from app1 are not visible to sighted user but may be announced by assistive services. The right side of Figure 7.2(c) shows a locker in our study, in which the elements of the Messages app are detected on the lock screen.

Overly Actionable

The `ActionList` attribute of nodes specifies the list of actions available to assistive services. When a node support click action for assistive services, the following conditions are barriers in performing that action through conventional interaction modes.

A1. Hidden: Nodes that are hidden to sighted users, i.e., with any of P1 to P6 conditions stated above.

A2. Disabled: Nodes that are disabled under certain conditions in the app and cannot be triggered by touch.

A3. Camouflaged: Empty nodes that are used as placeholders and are not detectable by sighted users, e.g., empty text boxes. Figure 7.2(d) provides the schematic placement of these nodes on the screen on the left and an example in our empirical study on the right.

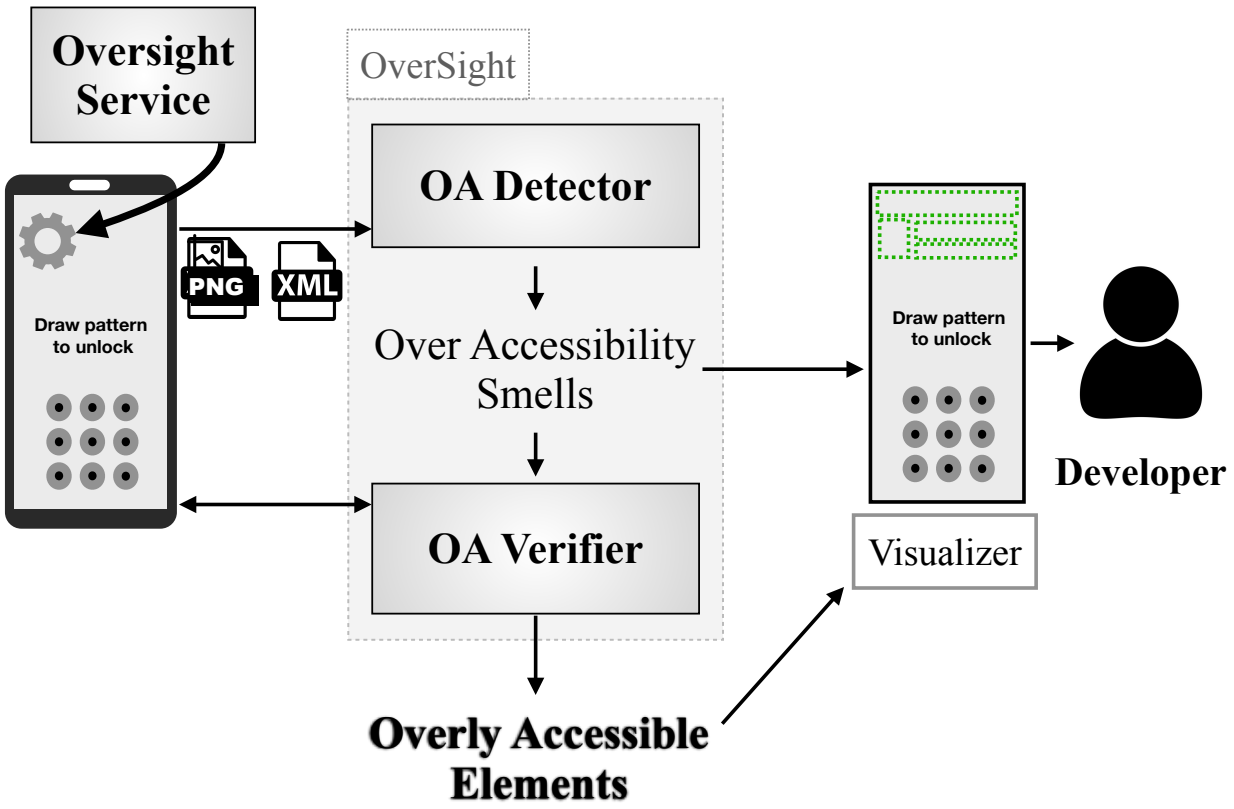


Figure 7.3: Overview of OVERSIGHT framework.

7.3 Approach

In this section, we introduce OVERSIGHT, an automated tool that gets the information from a specific state of the app and returns a list of OA elements confirmed by an assistive services. Figure 7.3 illustrates the overview of our approach. OVERSIGHT engine consists of two main components: OA Detector (Section 7.3.1) and OA Verifier (Section 7.3.2).

7.3.1 OA Detector

OA Detector gets a window's content specification in XML along with its screenshot through OVERSIGHT Service (OSS). As described in Section 7.2, OSS runs in the background, dumps hierarchical representation of nodes in an XML file, and enables communication with the device

through broadcast messages. OA Detector analyzes nodes on the window and returns *Over Accessibility Smells*, i.e., nodes that meet one of the conditions derived from our empirical study (Section 7.2). Confirming over accessibility issues in these nodes is the responsibility of OA Verifier . OA Verifier utlizes TalkBack and Abstract Proxy Users to validate the locatability and actionability of over accessibility smells. OVERSIGHT also visualizes over accessibility smells as well as OA elements on the screenshot along with their specification for developers. In the following sections, we describe the details of each component.

Our empirical study organizes a set of conditions under the basis of over-perceivability and over-actionability. OA Detector implements these conditions to automatically check the nodes against them. In this section, we explain the formula and algorithms we used to implement each condition.

For the majority of the conditions, i.e., P1, P3-P6, A1-A2, the implementation is straightforward using their definitions in Section 7.2 and the attributes of nodes mentioned in Table 7.1. Here, we elaborate on the algorithms we utilized to calculate the covered nodes (P2) and camouflaged ones (A3). Implementation details of all conditions are available with our open-source tool available at [96].

P2. Covered To find out covered elements, we investigate how Android draws elements on a window. Android draws a window starting from the root node and recursively draws the child elements according to their drawingOrder. To determine what nodes are covered, we simulate Android's drawing but in reverse order using a depth-first search algorithm. We start visiting nodes from the last drawn node to the first drawn node and keep track of covered areas. A node is "covered" if any of the covered areas obscure its bounding box.

Algorithm 1 explains our approach in details. For a given node, n, and a set of bounds that may cover it, B_C, `DetectCovered` first checks if n is covered to set all the descendants up to the leaf node as covered. (Line 2-4) If n is not covered, we will assess if its children are covered. To that end, we first sort the children in descending order based on their drawingOrder in line 5. The first

Algorithm 1: Overlap Analysis Algorithm

```
   Input: n ∈ Node(The visiting node), B_C : {b_1, ··· , b_k}(The set of covering bounds)
1  Function DetectCovered(n, B_C):
2      if n.covered then
3          ∀d ∈ n.descendants : d.covered ← True
4          return
5      ordered ← Sort n.children based on decreasing order of drawingOrder
6      foreach m ∈ ordered do
7          if m.bounds is covered by B_C then
8              m.covered ← True
9          DetectCovered(m, B_C)
10         B_C ← B_C ∪ m.bounds
```

element in the *ordered* list is the last child drawn by Android on the window among the other children. Then, in line 6, we iterated through the children and check if they are covered by any bounds in B_C. If that is the case, in the recursion call, the algorithm set all the descendants covered. Otherwise, in the recursion call, children of node m will be assessed. In line 10, we add the bounds of node m to the set of covering bounds since the other children in the for loop of line 6 may be covered by m.

A3. Camouflaged Detecting camouflaged nodes (A3) is challenging since there is no attribute in nodes indicating their color. This condition occurs when developers want to utilize some empty views as a placeholder. To detect these elements, we filter out nodes that have any child. Then, we evaluate the image associated to the remaining nodes. To get the image, we crop the screenshot based on the coordinates of the bounding box of the node. Then, we check if all the pixels of the image have the same color.

OA Detector evaluates compliance of each node with the defined conditions to find a list of nodes that has *Over Accessibility Smells*, i.e., they have symptoms that can lead to revealing information or functionality to assistive-service users that is not available to sighted users. To verify their accessibility with an assistive service, we propose OA Verifier as below.

7.3.2 OA Verifier

The behavior of different assistive services in focusing on the elements and performing an action on them cannot be predicted statically. To confirm if an assistive service can locate the detected over accessibility smells, we utilize OA Verifier . The goal of this component is to evaluate the locatability and actionability of nodes identified by OA Detector on a real device with an assistive service.

To assess the locatability of a node with TalkBack, we utilize the Linear Navigation mode in TalkBack Proxy User (defined in Section 4.4) as OA elements are not viewable on the screen to be enabled by tapping/touching. Since OA elements most likely appear after the ones that are visible to sighted users, TalkBack Proxy User first explores the screen backward by drawing "swipe left" gesture. Whenever TalkBack focuses on a node, OA Verifier calls the node TalkBack Locatable. TalkBack Proxy User continues screen exploration until either it reaches all the nodes in the given list, or sees a repetitive node.

An element is considered actionable, if 1) it is locatable and 2) a Proxy User performs the action successfully. To assess the actionability of elements, we first filter out unlocatable elements. Then use TalkBack and Abstract Proxy Users to perform click action on the remaining OA smells.

7.4 Evaluation

In this section, we evaluate OVERSIGHT on real-world apps to answer the following research questions:

RQ1. How accurate is OVERSIGHT in detecting OA elements?

RQ2. What are the potential impacts of OA elements on different apps and communities?

RQ3. What is the performance of OVERSIGHT?

7.4.1 Experimental Setup

Datasets

We evaluated our approach on 60 app screens from 30 real-world Android apps. Our test set consists of three groups of apps: (*group1*) 10 app lockers similar to the motivation example from Google Play, (*group2*) 10 apps with known accessibility issues detected in LATTE, and (*group3*) 10 randomly selected apps from different categories of Google Play. For each app, we captured two different states of the app. For apps in group1, the first state is the lock screen of the app itself, and the second state is the lock screen that protects a third-party app, e.g., Messages, when it is locked. For apps in group2, we selected two different screens of the app with the confirmed accessibility issue. Lastly, for apps in group3, we randomly explored the apps and captured two different screens. We ran our experiments on an Android emulator based on Android 11.0 and with TalkBack version 12.1 on a typical development machine, using a MacBook Pro with 2.4 GHz core i7 CPU and 16 GB memory.

7.4.2 RQ1. Accuracy of OVERSIGHT

To answer this question, we ran OVERSIGHT on each snapshot in our test set and carefully examined the reports. We separately evaluate OVERSIGHT's two main components, OA Detector and OA Verifier.

To evaluate OA Detector, we carefully checked the reported OA smells in each category and tagged them as True Positive (TP) if it was correctly detected with one of the OA conditions and False Positive (FP) otherwise. We then calculate OA Detector's precision as the ratio of the number of

nodes that were correctly detected by OA Detector to the number of all detected OA smells.

Table 7.2 summarizes the results of this experiment. Each row in this Table corresponds to one state of an app. The number of nodes in each state varies as shown in the second column (N). In our test set, it can be as few as 6 nodes and as many as 656. *Smell* column indicates the number of nodes with Overly Perceivable (P) or Overly Actionable (A) conditions on each screen. We display the precision per app state under the *DP* (Detector Precision) column, and the average precision is in the last row.

As shown in the Table 7.2, on average, OA Detector has a precision of 84.23% in detecting OA smells. For 56 different states in 28 number of apps, the precision is 100%. We analyzed the elements recognized as False Positive, i.e., with FP tag, to better understand OA Detector failures. Figure 7.4 shows some examples where OA Detector erroneously evaluates a node as OA. In Figure 7.4(a), the map and the text on it is annotated as OA. Further inspection of this layout showed us that the map is behind a transparent layout and made our algorithm classify the underlying nodes as "covered" (recall P.2 in Section 7.3.1). In Android, transparent layouts pass the touch gesture to the underlying elements so they will not be recognizable through conventional interaction modes. Since layout colors are not included in node information, OA Detector cannot distinguish transparent layouts from color-filled ones. Moreover, having a stack of transparent nodes, if not maintained properly, can cause troubles for assistive-service users. For example, if all the stacked nodes are focusable, the assistive service will focus on each of them separately, confusing assistive-service users about what is shown on the screen and resulting in a less optimal navigation experience. Partially covered nodes are another failure of OA Detector as shown in Figure 7.4(b). There is a "Sort and Filter" button covering the elements underneath. However, as the underlying texts are partially recognizable to sighted users they are tagged as FPs. OA Detector does not exclude partially covered elements in the "covered" category since a developer may have intentionally blocked access to part of a node content.

To evaluate the OA Verifier component, we investigate the nodes specified as locatable and actionable

Table 7.2: Accuracy of OVERSIGHT in running on 30 apps.

App	N	Smells		DP	TalkBack		SAT	VP	VR
		P	A		L	A	A		
...domobi...	47	0	2	0.00	0	2	2	1.00	1.00
	26	9	6	1.00	8	3	6	1.00	0.95
...alpha...	12	0	0	1.00	0	0	0	1.00	1.00
	8	0	0	1.00	0	0	0	1.00	1.00
...sp.pro...	42	0	0	1.00	0	0	0	1.00	1.00
	26	9	6	1.00	8	5	6	1.00	0.90
...thinky...	18	0	0	1.00	0	0	0	1.00	1.00
	17	1	0	1.00	0	0	0	1.00	0.50
...litetoo...	73	6	7	1.00	6	7	7	1.00	1.00
	73	0	0	1.00	0	0	0	1.00	1.00
...nevways...	55	1	1	0.00	0	1	1	1.00	1.00
	6	0	0	1.00	0	0	0	1.00	1.00
...ammy.a...	16	0	0	1.00	0	0	0	1.00	1.00
	26	9	6	1.00	8	6	6	1.00	0.95
...gsmobile...	53	0	0	1.00	0	0	0	1.00	1.00
	37	0	0	1.00	0	0	0	1.00	1.00
...cd.app...	12	0	0	1.00	0	0	0	1.00	1.00
	12	0	0	1.00	0	0	0	1.00	1.00
...saeed.ap...	13	0	0	1.00	0	0	0	1.00	1.00
	16	0	1	0.00	0	1	1	1.00	1.00
...c51	83	4	1	0.00	4	1	1	1.00	1.00
	29	0	1	0.00	0	0	1	1.00	1.00
...fatsec...	41	0	1	1.00	0	0	1	1.00	0.50
	147	14	5	1.00	2	0	5	1.00	0.70
...colpit...	18	2	0	1.00	0	0	0	1.00	0.50
	67	2	1	1.00	0	0	1	1.00	0.50
...tripit	202	55	20	0.91	6	1	20	1.00	0.54
	270	68	23	1.00	4	4	23	1.00	0.55
...contex...	52	0	26	1.00	0	0	5	1.00	0.50
	57	1	0	0.00	1	0	0	1.00	1.00
...yelp.an...	66	0	0	1.00	0	0	0	1.00	1.00
	129	15	9	0.86	0	0	6	1.00	0.50
...devhd.f...	71	19	4	1.00	4	0	4	1.00	0.60
	138	44	21	1.00	8	0	21	1.00	0.67
...ziprecr...	36	0	0	1.00	0	0	0	1.00	1.00
	63	5	5	1.00	0	0	2	1.00	0.50
...diction...	102	8	5	1.00	2	1	4	1.00	1.00
	177	98	5	0.89	98	1	1	1.00	1.00
...and...	110	16	10	0.95	0	0	8	1.00	0.50
	69	16	10	0.53	16	9	9	1.00	1.00
...airbnb...	42	0	1	1.00	0	0	0	1.00	0.50
	56	0	3	1.00	0	0	1	1.00	0.50
...carfax...	30	0	0	1.00	0	0	0	1.00	1.00
	20	0	0	1.00	0	0	0	1.00	1.00
...expedi...	53	0	2	1.00	0	0	1	1.00	0.50
	98	6	0	0.33	4	0	0	1.00	0.83
...houzz	22	0	0	1.00	0	0	0	1.00	1.00
	169	37	27	1.00	9	3	5	1.00	0.85
...mcdona...	42	1	0	1.00	0	0	0	1.00	0.50
	126	35	10	0.32	35	5	5	1.00	0.94
...meditat...	22	3	2	1.00	0	0	2	1.00	0.50
	46	15	1	0.75	14	0	1	1.00	0.94
...pinterest	32	1	1	1.00	1	1	1	1.00	1.00
	24	0	0	1.00	0	0	0	1.00	1.00
...popular...	36	5	3	1.00	0	0	3	1.00	0.50
	158	40	19	1.00	21	0	2	1.00	0.68
...theathl	20	0	1	1.00	0	1	1	1.00	1.00
	77	35	5	1.00	34	5	5	1.00	0.99
...weawow	32	2	2	0.00	0	0	2	1.00	1.00
	656	280	52	1.00	194	3	5	1.00	0.87
Average:				84.23%				100%	83.27%

with TalkBack and Abstract Proxy Uses. To check the reported nodes by OA Verifier, we load the corresponding snapshots of the app states on the emulator and utilize an assistive service, e.g., TalkBack, to explore the app and assess Locatability (L) and Actionability (A) of OA smells. In terms of locatability, if TalkBack can focuses on a node, we consider it locatable. For actionability, the node is actionable if it is locatable and is clickable, i.e., the click gesture, such as double tap in TalkBack, broadcasts a click event. When an element is clicked successfully in Android, an `AccessibilityEvent`, called `VIEW_CLICKED`, is created and sent to `AccessibilityServices`. To determine if the action was performed, OVERSIGHT service captures the events and shows if an event of type `VIEW_CLICKED` or `WINDOW_CONTENT_CHANGED` is logged. Since OA Verifier follows the same strategy in detecting clicked nodes, the accuracy of OA Verifier equals to its accuracy in detecting locatable nodes. Thereby, we label the output of OA Verifier as true if it matches with our manual investigation and false otherwise. Using these tags, we calculate precision and recall of OA Verifier as follows: Precision is the ratio of number of nodes that correctly verified to be locatable to the number of locatable nodes detected by OA Verifier, while recall is the ratio of number of nodes that correctly verified to be locatable to the number of OA smells that are manually verified to be locatable.

Table 7.2 shows the average precision and recall of OA Verifier using TalkBack and Abstract Proxy Users in the last two columns, VP (Verifier Precision) and VR (Verifier Recall). As shown in the last row, the average precision and recall on all apps is 100% and 83.27% respectively. While OA Verifier is 100% precise in its reports, the recall shows that it has missed some issues. Figure 7.4(c) shows an example of a set of nodes that were erroneously detected to be unlocatable by OA Verifier. On this state of the "Weawow" app, there is a map of all the cities that a user can get the weather information for. When TalkBack reaches to this widget, it navigates through all the nodes on the map, as depicted by number annotations on the map, and gets stuck there in an infinite loop. Thus, all the nodes on the second half of the screen were reported as unlocatable. OVERSIGHT attempted to address such issues for scrollable widgets by navigating both forward and backward on the screen. However, backward navigation on this app does not help since the app content loads dynamically

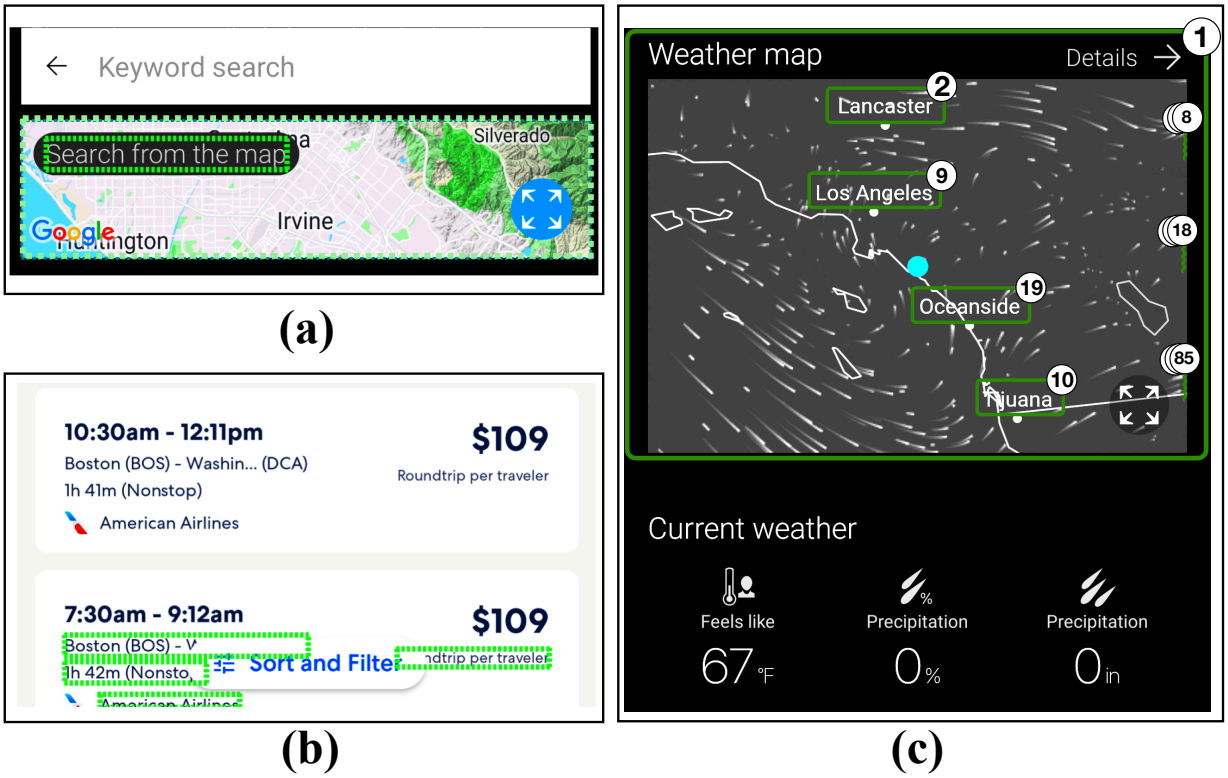


Figure 7.4: OVERSIGHT Failures. (a) and (b) are false positives of OA Detector, where dashed green boxes are erroneously detected as covered. (c) is a false negative of OA Verifier. TalkBack stuck in the world map.

in forward navigation, while scrolling to the bottom. OA Verifier also has a similar issue in web apps such as Dictionary. In this app, every time the app is scrolled forward, it fetches a totally new UI specification which although looks visually similar, uses different *apath*s for nodes, making the logged information inaccurate.

Further investigation of conditions of detected OA elements revealed that the "covered" condition (recall P.2 in Section 7.3.1) is the most frequent symptom of OA elements. 18 apps out of 30 had at least one "covered" OA element. According to Android documentation, Android attempts to evaluate whether a node is visible to user [51] (recall row 6 of Table 7.1) to be announced by TalkBack. However, our review of Android's source code [52] indicates the platform only compares the bounds of a child node with its parents to evaluate if they are visible to user (i.e., the corresponding `VisibleToUser` flag is set to true). However, such a comparison does not exist for nodes that are siblings or children of siblings. We believe Android platform should reassess its

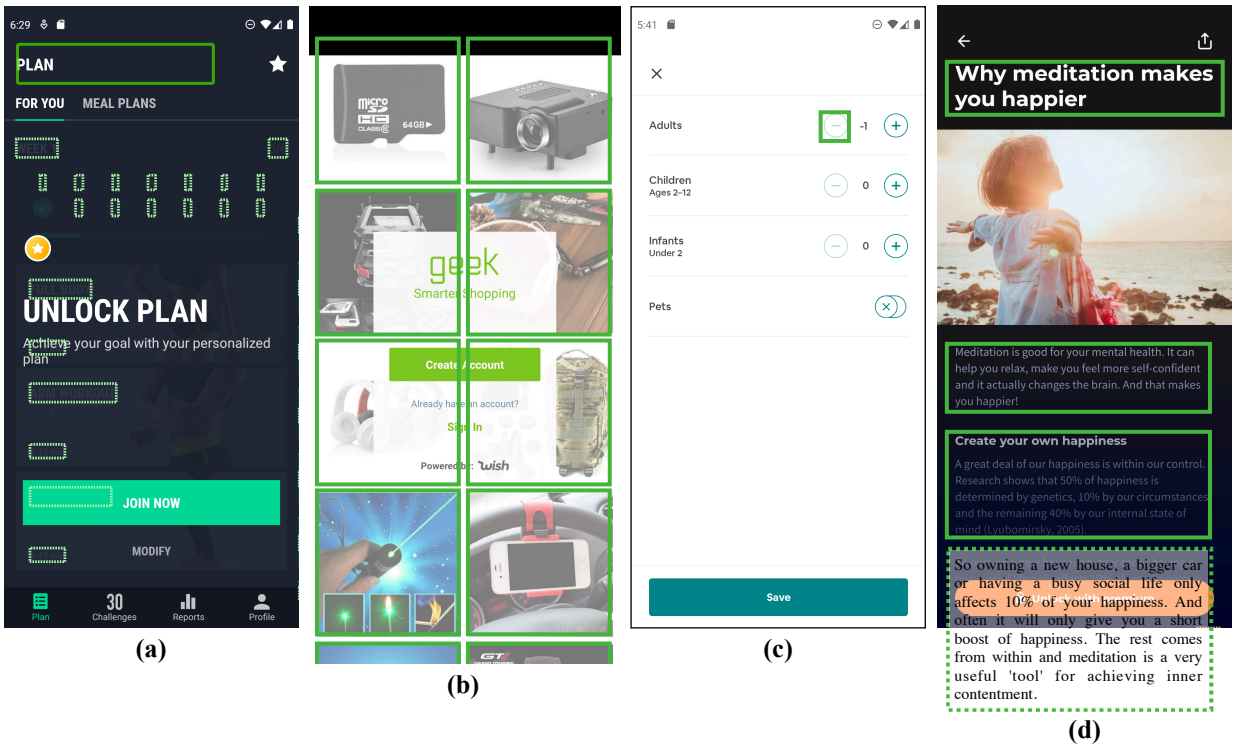


Figure 7.5: Impacts of OA elements in different apps. (a) Accessibility issue of overly perceivable elements. (b) Accessibility issue of overly actionable elements. (c) Workflow violation, giving access to premium content (d) Workflow Violation, breaking app logic by submitting a hotel request for negative number of travelers.

strategy of detecting visible nodes to minimize such issues.

7.4.3 RQ2. Qualitative Analysis of Detected OA Elements

We manually examined all reported OA elements by OVERSIGHT in Table 7.2 and study how they impact users with disabilities and app developers.

Users with disabilities

Both over perceivable and over actionable elements degrade app accessibility, hindering disabled users' ability to explore the app conveniently. For example, in "30 days workout" app, Figure 7.5(a), a blind user has to navigate through the covered elements, highlighted in green. Although these OA

elements, requiring paid subscription to access, are not actionable, a user who wants to understand the app content would be confused of what is shown on the screen. Moreover, if she wants to reach a specific button, e.g., Profile, she has to pass through all OA elements, resulting in a less optimal user interaction. A similar scenario happens in the welcome page of "iSaveMoney" app. The intended use-case is for the user to follow the introductory steps and get familiar with different parts of the app; however, the information from next steps are available to assistive-service user in the first step, making the introduction complicated. School Planner, ZipRecruiter, and McDonlads have a similar issue. It is worth mentioning that OA elements in app lockers discussed in RQ2 not only undermine app functionality for assistive-service users but also complicate their interaction with apps. When they utilize app exploration by swipe, there is no lock preventing them to access apps. However, app exploration by touch will not activate the OA elements that supposedly exist on the screen.

In some cases, OA elements provide actions to assistive-service users. Case in point, background images in Geek, shown in Figure 7.5(b), are not accompanied with any textual data but are actionable. Although none of them are associated with any functionalities, i.e., they do not change the screen content when triggered, they complicate app exploration for assistive-service users who believe there are real buttons on the screen. Interestingly, this app was also diagnosed with under-access problem in a prior work [112] because of a rolling dynamic widget that all these background images belong to. The assistive-service user gets stuck in an infinite loop and cannot login if she wants to explore the screen by swiping.

Developers

Besides Developers design a workflow by which users interact with the apps. Violating such workflows can (1) break app logic, (2) provide unauthorized access to premium content.

Developers restrict access to some functionalities to avoid false inputs and gather required information from users. For example, in the Airbnb app depicted in Figure 7.5(c), when the number of

passengers is zero, the decrease button for the number of travelers is disabled. However, using an assistive service, one can decrease the number of passengers to less than zero. Similarly, in Expedia and FatSecret apps, the continue button is disabled until the user enters the required information at each step. Using assistive services, users can pass invalid inputs, which can result in the app malfunctioning or crashing.

In some cases, the workflow violation targets developer's revenue model. Figure 7.5(d) illustrates an article in a meditation app which is only available fully for the subscribed users. However, TalkBack announces the whole content of this article and scrolls through it without asking for a subscription. The same issue exists for the premium articles in the "The Athletics" app. While these examples are related to the restricted scroll functionality, the same issue threatens any other blocked functionalities that are intended to be available to subscribed users.

7.4.4 RQ3. Performance

The time-consuming component of OVERSIGHT is OA Verifier which needs to interact with the device and perform actions on elements. On average, it takes 54 seconds for OA Verifier to perform an action. The execution time varies in different apps as their number of nodes and OA smells are different. For the apps in our test set, the average execution time of OVERSIGHT is 571 seconds, which can be effectively used in practice. Any dynamic analysis tool, including OVERSIGHT, is costly in time compared to simple static checkers. The OA Detector runs very fast, under one second. By identifying the OA smells, OA Detector reduces the number of nodes that need to be verified by 84% on average. Without OA Detector, an expensive verifier would need to assess every single node on the screen.

7.5 Conclusion

Assistive services help disabled users have equal access to mobile apps by providing alternative modes of interaction. An inconsistency between different interaction modes may result in both under-access as well as over-access problems. The former has been extensively studied in prior works (including LATTE and GROUNDHOG), concerning inaccessible data and functionality. However, this chapter presented the latter and discussed the threats of overly accessible elements, enabling an assistive-technology user to get access to app content or functionality that is not available otherwise. We also studied the characteristics of overly accessible elements and proposed OVERSIGHT to automatically detect them in mobile apps with high accuracy. Our evaluation reveals overly accessible elements have severe impacts on both disabled users and developers.

In the last three studies that led to building the automated tools LATTE, GROUNDHOG, and OVERSIGHT, the main focus was to evaluate the accessibility of mobile apps with minimal manual effort using assistive services. However, removing human knowledge from these tools limits their ability to detect accessibility issues that require an intelligent oracle. The next chapter tries to tackle this limitation by empowering manual testers with automated tools to assess the accessibility of mobile apps with ease.

Chapter 8

Assistive-Service Aided Manual Testing

Modern mobile devices are equipped with touchscreens, providing rich experiences for users; however, they also force developers to test and validate the functionality of their apps either manually or using automated tools. In the testing process, developers may neglect to evaluate their software for approximately 15% of the world's population with disabilities [124], many of whom cannot use conventional interaction methods, such as touch gestures.

In order to understand how people with disabilities use mobile apps, developers are encouraged to conduct user studies with users (preferably with disabilities) using assistive services, such as screen readers. Despite the fact that software practitioners acknowledge the importance of human evaluation in accessibility testing, they admit that end-user feedback is difficult to obtain [31]. Furthermore, for small development teams with limited resources, finding users with various types of disabilities and conducting such evaluations can be prohibitively challenging and expensive.

The key insights that guide our research are (1) mobile developers and testers still prefer manual testing in-app development [67, 81, 70], (2) assistive services need to be incorporated for evaluating apps' accessibility, and (3) there is a lack of expertise and knowledge among many mobile developers and testers on how to properly evaluate the accessibility of their apps with guidelines, automated

tools, and assistive services. A survey found that 48% of Android developers cite lack of awareness as the main reason for accessibility issues in apps [7]. Another survey found that 45% of accessibility practitioners are experiencing problems related to accessibility development and design, such as inadequate resources and experts [31].

Informed by the above-mentioned insights, this chapter introduced a new form of automated accessibility analysis, called A11YPUPPETRY, that aids developers in gaining insights on accessibility issues of their apps. Developers and testers can evaluate their apps manually by using touch gestures, while A11YPUPPETRY records these interactions. After that, A11YPUPPETRY interacts with the app on another device using an assistive service to perform the equivalent actions on behalf of the testers, regardless of their knowledge and expertise in accessibility and assistive services. A11YPUPPETRY is inspired by Record-and-Replay (RaR) techniques, such as [50, 59, 107], where a program records the user actions on an app and replays the same actions on the same app in another device. However, to the best of our knowledge, all existing RaR techniques replay the recorded actions exactly as they are performed in the primary device. For example, if the user touches specific coordinates of the screen, the replayer program also sends a touch event for the same coordinates. A11YPUPPETRY is different from these techniques since the replaying part is completely done by an alternative way of interaction, e.g., a screen reader. More importantly, A11YPUPPETRY generates a fully visualized report for developers after replaying the recorded use case with assistive services, which are augmented by accessibility issues.

The rest of this chapter is organized as follows: Section 8.1 motivates this study with an example and explains the challenges that we are facing. The Section 8.2 provides an overview of our approach and the following sections explain the details of our approach. The evaluation of A11YPUPPETRY on real-world apps is presented in Section 8.3. Finally, Section 8.4 provides a discussion on our findings in our user studies.

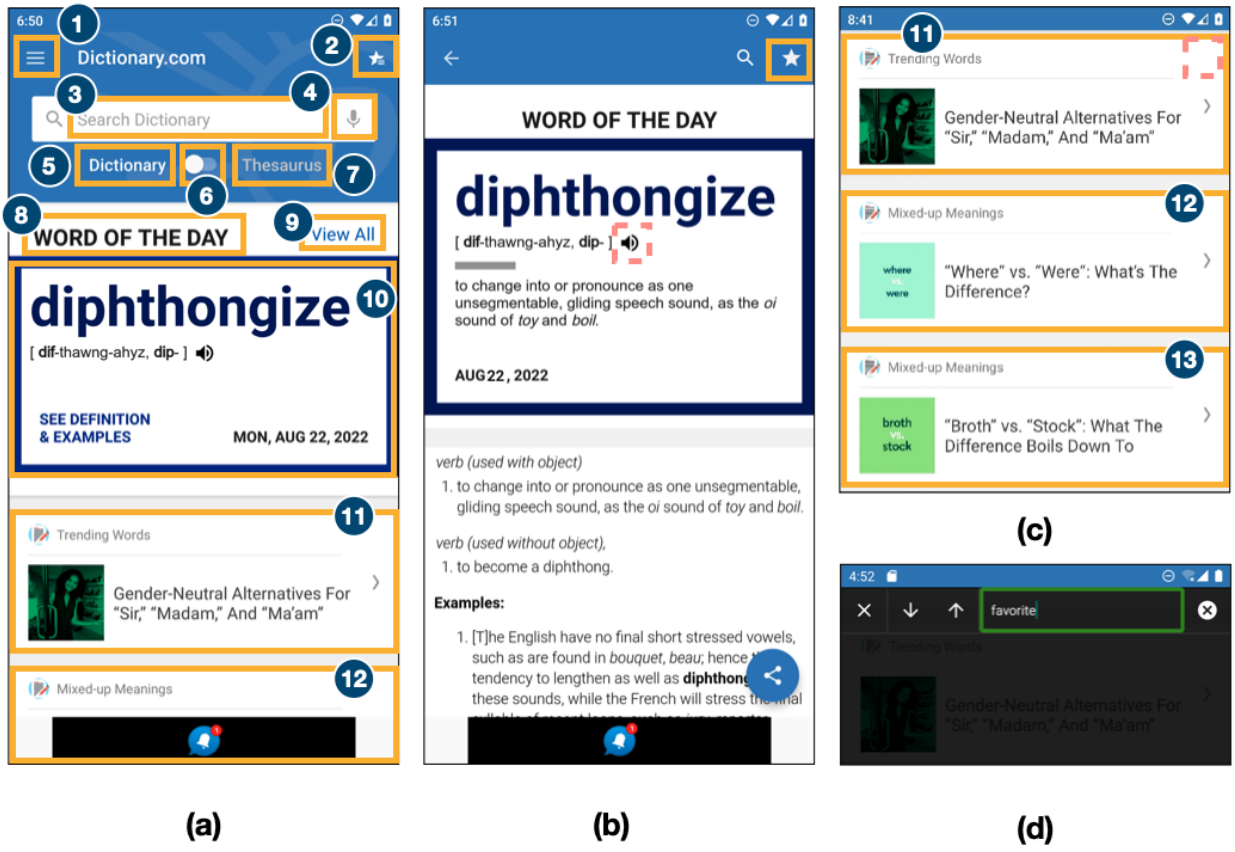


Figure 8.1: (a) The main page of Dictionary app, (b) The page after tapping on the word of the day, (c) Upper menu disappears when users scroll down the page, (d) The Search Navigation provided by TalkBack

8.1 Motivating Example

This section demonstrates a couple of accessibility issues that manifested by using assistive-services which cannot be detected by conventional accessibility testing tools. Then, we elaborate on the challenges of automatically recording touch gestures and replaying them with a screen reader.

Figure 8.1(a) shows the home page of the Dictionary.com app with more than ten million users in the Android Play store [43]. Assume a tester wants to validate the correctness of a use case which consists of 3 parts: Selecting the "word of the day" and listening to its pronunciation, and marking the word as a favorite, and reviewing or removing favorite words.

A user without a disability who can see all elements on the screen and perform any touch gestures can perform this use case fairly easily. First, she taps on the word of the day, box 10 in Figure 8.1(a), then the app goes to Figure 8.1(b). Next, she taps on the speaker button to listen to the pronunciation, pink-dashed box in Figure 8.1(b). Then to mark the word as a favorite, she taps the star button, yellow-solid box in Figure 8.1(b), and she can get back to the home page, Figure 8.1(a), by pressing the back button. Next, to see the list of favorite words, she taps on box 2. The rest of this use-case (removing the favorite word) has been discussed in Section 4.1.

To perform the same use case, users with visual impairments, particularly blind users, have a completely different experience. As mentioned in Section 4.3, using *Linear Navigation* in TalkBack, the user can navigate to the next and previous element of the currently focused element by swiping right and left on the screen. For example, to reach the "word of the day" in Figure 8.1(a), which is *diphthongize*, the user can start from box 1 (top left icon) and navigate to the next elements until it reaches box 10. Note that TalkBack may group elements for a more fluent announcement, like here, where a couple of textual elements are grouped into box 10. After getting to the word of the day page, to listen to the pronunciation, the user needs to locate the speaker button, pink-dashed box in Figure 8.1(b). However, the element cannot be focused on by TalkBack since it is a WebView element customized by developers; therefore, this functionality is inaccessible for TalkBack users. While the unlocatablity of this element by TalkBack is a critical accessibility issue, Google's Accessibility Scanner, the most widely used accessibility analyzer for Android, cannot detect it since the scanner does not consider assistive services like TalkBack into account.

Assuming the mentioned accessibility issue does not exist, the blind user continues the rest of the use case by selecting the favorite button, the yellow-solid box with the star icon in Figure 8.1(b), and then returns to the home page. After returning to the home page, the user needs to find Favorites List or box 2 in Figure 8.1(a). However, since the user was previously on this page, box 10 is focused. By navigating to the next elements, boxes 11 and 12, TalkBack automatically scrolls forward to fetch the items below; however, the app makes the upper menu disappear as shown in Figure 8.1(c).

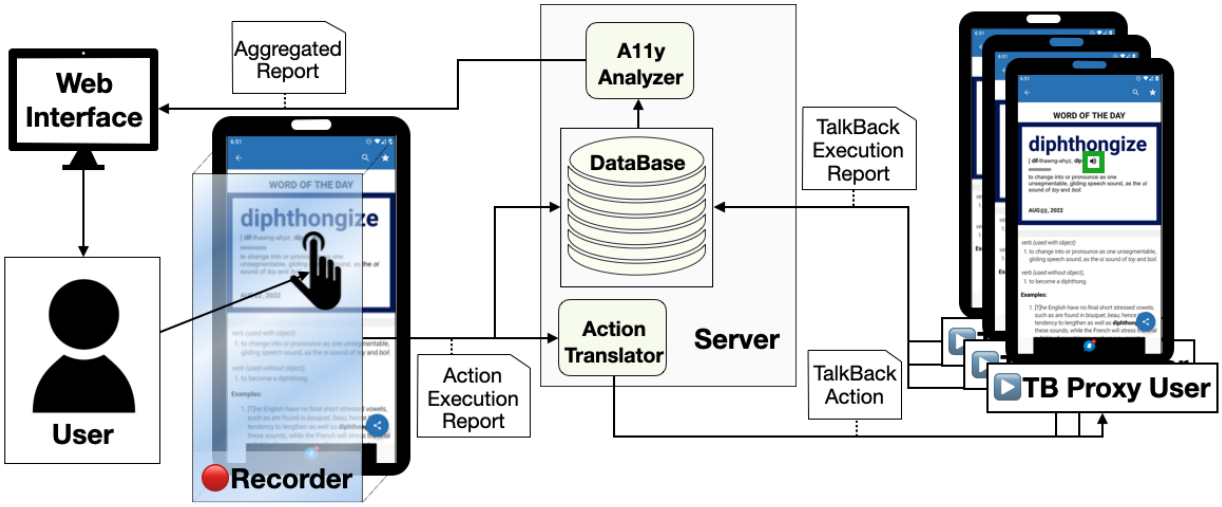


Figure 8.2: An overview of A11YPUPPETRY

A sighted user can notice this major change in the screen since he can observe all parts of the screen; however, a blind user may not notice it. Consequently, the blind user cannot locate the favorites list button, initially located at the top right of the display. Even if the user searches for the word "Favorite", Figure 8.1(d), there is no result since the favorites list button does not exist on the screen anymore. This is another example of accessibility issues that could not be detected without considering TalkBack in runtime.

While it is straightforward for most app developers and testers without disabilities to perform the aforementioned use case with touch gestures, none of the accessibility issues above could be detected unless the same use case is performed using a screen reader. Recall that our idea is to record touch gestures from an arbitrary app tester, execute them using a screen reader automatically, and generate a report with detected accessibility issues. Now, we explain the possible challenges to realizing this idea. The following section provides an overview of our approach, A11YPUPPETRY.

8.2 Approach Overview

A11YPUPPETRY consists of four main phases, (1) Record, (2) Action Translate, (3) Replay, and (4) Report. In this section, we provide an overview of the approach and in the next four sections, we explain the details of each phase.

Figure 8.2 depicts an overview of A11YPUPPETRY. The process starts with the Record phase when the user interacts with a device enabled with the Recorder service. The Recorder service listens to UI changes events and adds a transparent GUI widget overlay on top of the screen to record the user's touch gestures. After receiving a touch gesture on the overlay, the Recorder replicates the gesture on the underlying app, and sends the recorded information to the server as an *Action Execution Report*. The server will store the recorded information in the database.

In the second phase, Action Translation, the *Action Translator* component receives the *Action Execution Report* from the Recorder (containing UI hierarchy, screenshot, and the performed gesture) and translates it to its equivalent *TalkBack Action*. For example, touching on the coordinates of the favorite button in Figure 8.1(b) will be translated to focusing on the favorite button and performing a double-tap gesture.

In the Replay phase, the TalkBack Action is sent to several replayer devices that perform the action. Each replayer device has a running TalkBack Proxy User that receives TalkBack Action from the server, creates and maintains a TENG of the app (defined in Section 4.4), and performs the received actions with a navigation mode, i.e. Linear, Touch, Jump, and Search. Once an action is performed, a TalkBack Execution Report will be stored in the database. The TalkBack Execution Report consists of actions that are executed with TalkBack, screenshots, and UI hierarchy files of the different states of the app before, during, and after execution.

In the final phase (Report), the *A11y Analyzer* component reads the stored information in the database, i.e., Action and TalkBack Execution Reports, and produces an *Aggregated Report* of the

recording, replaying, and the detected accessibility issues. The user can access this report using a web application.

8.2.1 Recorder

In a nutshell, the recorder component uses two different ways to record the user's actions, (1) through a transparent overlay placed on top of the app and (2) by listening to system events related to the changes on the screen. We implement the recorder on top of Sugilite [75], a programming by demonstration tool for Android apps.

We implemented the recorder component as an *AccessibilityService* to understand the user's action. When the recorder is enabled, it creates an overlay of the screen's size, which is an *android.view.object* and attaches it to the foreground window. This overlay acts as an echo component; it performs any received touch gestures on the app with Accessibility API. A touch gesture event, e.g., PointGesture $\mathbb{PG}$, is captured by *onTouchListener* which is enabled for the overlay. Once the touch gesture is received, a copy of the touch gesture, the UI hierarchy of the current screen, and a screenshot image are combined and packed as *Action Execution Report*.

Although the overlay object can record touch gestures, a few other actions such as adjusting volume with physical buttons or typing with a keyboard cannot be captured. To that end, the recorder listens to all *AccessibilityEvent*s and records the events that represent actions performed by the user. For example, when the user types on an EditText with a keyboard, the recorder will receive *AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED* containing the typed text. Similar to touch gestures, these events, along with the UI hierarchy and screenshot of the app, are packed and sent as *Action Execution Report*s.

Once the *Action Execution Report* is created, either by the overlay screen or *AccessibilityEvent*, the recorder sends it with WebSocket to the Server. Note that the recorder is an app inside an Android

device or emulator, and all the storing, analysis and broadcasting is done on the external remote server.

8.2.2 Action Translation

In the second phase of A11YPUPPETRY, the *Action Translator* component (Figure 8.2) translates actions recorded from the user using touch gestures to their counterparts in TalkBack. We propose a mapping from touch gestures, i.e., Point and Line Gestures, to TalkBack actions (defined in Section 4.3). The other types of gestures, like TwoFingerLine or Circular gestures do not have any equivalent in TalkBack.

PointGesture

PointGestures, like single-tap or long-press, can be mapped to ElementBased actions in Talkback since a PointGesture is usually associated with a GUI element. In some cases, the PointGesture is not associated with a single element and the exact coordinate of the touched surface is important. For example, a painting app may have a large canvas where the user can paint and draw shapes by touch gestures. Although the underlying element of all these gestures is the canvas, the exact coordinate of the gesture is important to draw the lines precisely. We exclude these cases in this work since they require a fine visual perception of the screen to pinpoint the desired coordinates.

However, to precisely find the equivalent of a PointGesture, we also need to find the element associated with the touch gesture. To find the associated element, we list all elements in the UI hierarchy (recall that the Action Execution Report has the UI hierarchy of the app before the execution). Then filter the elements that enclose the touched point and sort them based on their z-index. An element with a greater z-index is always in front of an element with a lower z-index [47]. Then we iterate the list to find an element that has a matching attribute to the action that is

performing. For example, if the PointGesture is single-tap or long-press, then the element should have a *clickable* or *long-clickable* attributes respectively. If no such element can be found, we choose the first element in the list.

LineGesture

LineGestures can be mapped to either TouchGestureReplication or PredefinedActions. For example, a swipe-up touch gesture can be performed in TalkBack either by swiping up with two fingers or performing the predefined action, swipe right then left.

Once the input action is translated into a TalkBack action, it will be sent to replayer devices, in particular, to their TalkBack Proxy Users.

8.2.3 Replayer

The third phase of A11YPUPPETRY replays the received *TalkBack Action* with TalkBack. Before the user starts interacting with the app, the recorder and replayer devices are in the same state, i.e., the app under test is installed and opened. In the replayer device, TalkBack and TalkBack Proxy User services are enabled. For each navigation mode, i.e., Linear, Jump, Search, and Touch, there is one replayer device receiving the inputs from the server. Section 4.4 explained how TalkBack Proxy User can perform an action in details.

Once a replaying use case is finished, TalkBack Proxy Users compile a set of information and send it to the server, including the UI hierarchy, screenshot, TENG, and performed actions in all stages.

8.2.4 Report

In the final phase of A11YPUPPETRY (Report), the A11y Analyzer component in Figure 8.2 analyzes all information stored in the database and generated from the Recorder and TalkBack Proxy Users, compiles and aggregates them, and shows the final report to the user via a web interface. Since the target users of A11YPUPPETRY are developers and testers with limited knowledge on accessibility, we implemented the following features to illustrate the accessibility barriers in their apps.

- **Annotated Video.** Once the record and replay for an app is completed, A11y Analyzer create videos using the captured screenshots, then animates the touch gesture on the image. Moreover, it adds a short description of the gesture whenever it is performed. For the replayer vidoes, it also annotate the focused elements by TalkBack during the navigation.
- **Blindfold Mode.** The replayer video cannot represent the issues that blind users may face, especially the ones related to the semantics of the app. For example, when visual icons have content descriptions that are irrelevant to their corresponding buttons' functionality, blind users may become confused and not understand the app. We provided a blindfold mode in our report which lists the textual description of the items that have been navigated with TalkBack. For example, the Blindfold Mode report of Linear Navigation for Figure 4.3(a) would be "1. Back button, double tap to activate 2. 1 Selected, 3. Delete button, , double tap to activate 4. Favorite ..., 5. Select, 6. diphtongize, not checked, checkbox, double tap to toggle"
- **State Comparison.** A11y Analyzer also compares the state of the apps in the recorder and replayer devices to see if there is any difference between them. Ideally, if all actions are performed correctly in all replayers, there should be no difference between the states. The comparison is done by checking the UI hierarchy of the apps before performing any action. In case of different between states, the web interface shows a warning sign near the state to show the issue.

Table 8.1: The evaluation subject apps with the detected accessibility issues

App	Category	#Installs	#Actions	#User Issues	#Scanner Issues	#LATTE Issues	#A11YPUPPETRY Issues				
							Linear	Touch	Jump	Search	Total
ESPN	Sports	>50M	24	11	18	6	6	2	13	6	17
DoorDash	Food	>10M	23	8	22	10	9	1	13	9	15
Expedia	Travel	>10M	33	8	89	4	2	3	19	7	22
Dictionary	Books	>10M	21	8	113	6	4	2	13	5	15
iSaveMoney	Finance	>1M	21	5	35	2	10	9	10	2	11

8.3 Evaluation

This section explains our experiments and user studies to evaluate the effectiveness and limitations of A11YPUPPETRY.

We selected five Android apps with possible accessibility issues detected in GROUNDHOG or online social media [68]. For each app, we designed a task (consisting of 20 to 40 actions) according to the functionalities of the app. Also, we included the parts of the app that were reported inaccessible in the task. The first four columns of Table 8.1 show some information about the subject apps and the number of actions involved in the designed tasks.

We use A11YPUPPETRY on each task of these five apps. We used an Android emulator with Android 11 and TalkBack (version 12.1) for both recording and replaying devices. Our prototype of A11YPUPPETRY enables us to perform the experiments synchronously (recorder and replayers are running simultaneously) or asynchronously (the recording can be done before the replaying). For the experiments, we use the asynchronous mode to not introduce any problem caused by network or other concurrency issues; however, in practice, the synchronous mode is more promising since the results can become ready much faster. Furthermore, we scan each stage of the app during the execution with Accessibility Scanner (version 2.3).

To compare A11YPUPPETRY with existing work, we used LATTE (our use-case driven testing tool) and Accessibility Scanner. Since LATTE requires GUI test cases for the analysis, we transformed recorded use cases to GUI test cases. Scanner is not a use-case driven tool and scans the whole screen; therefore, we ran Scanner on the screens of the app after each interaction. Moreover, since

in this experiments we are focused on blind users who uses TalkBack, we filter out issues that are not related to blind users, like small touch target size or low text contrast.

Besides experiments with these tools, we conducted two user studies with users with visual impairment who have experience working with TalkBack in Android. To connect to such users, we used the third-party service Fable [1]. Fable is a company that connects tech companies to users with disabilities for user research and accessibility testing. Fable compensates all user testers and is committed to fair pay for the testers [2].

We used two services of Fable: Compatibility Test and User Interview. In the compatibility test, we provided the designed tasks and apps to Fable, then Fable distributed each task to three TalkBack users with visual impairments. The users performed the tasks, and for each step of the task, they reported any issues they faced. Once we gathered all detected issues from A11YPUPPETRY and compatibility tests in Fable, we did a preliminary analysis and produced a comprehensive list of accessibility issues for each step. Then for each app, we sent requests for user interviews with Fable, where Fable scheduled a one-hour online interview with a blind user who uses TalkBack. During the interview, the user shared his/her Android phone screen. We asked the users to perform the designed tasks and explain their thoughts and understanding of the app's pages. When they faced an accessibility issue that prevented them from continuing the task, we intervened and guided them to skip to the next step. Once the users finished the tasks, we started a conversation and asked them some specific questions about the tasks or general questions about their experience in working with screen readers and apps. In summary, each app is assessed four times: three users in compatibility tests and one user in an online interview.

The source code of A11YPUPPETRY, a demo of the web interface, designed tasks, apps, and user responses can be found in our companion website [1].

We would like to understand how A11YPUPPETRY can help detect accessibility issues confirmed

[1] https://www.makeitfable.com

[2] https://makeitfable.com/article/why-fair-pay-for-testers-matters/

Table 8.2: The percentage of the intersection of user-confirmed issues detected by Scanner, LATTE, and A11YPUPPETRY to the total number of user-confirmed issues.

App	% Intersection with User-Confirmed Issues				
	Scanner	LATTE	A11YPUPPETRY		
			Detected	Evidence	Total
ESPN	10%	18%	18%	45%	**63%**
DoorDash	25%	25%	50%	37%	**87%**
Expedia	12%	25%	50%	12%	**62%**
Dictionary	25%	50%	50%	37%	**87%**
iSaveMoney	40%	40%	40%	20%	**60%**

by users with visual impairment. As discussed before, all five tasks from five subject apps are assessed by users with disabilities, Accessibility Scanner, LATTE, and A11YPUPPETRY. For A11YPUPPETRY, we used four navigation modes (Linear, Touch, Jump, and Search). For user feedback, if at least one user expresses an issue with a certain action, we assume the action has an accessibility issue. The number of reported issues for each app can be found in Table 8.1. The last column represents the number of actions that at least one of the navigation modes in A11YPUPPETRY reported an issue. As can be seen, the issues detected by LATTE and A11YPUPPETRY are proportional to the number of actions; however, Scanner reported many issues that can be difficult for testers to examine and verify.

Table 8.2 summarizes the effectiveness of Scanner, LATTE, and A11YPUPPETRY in detecting issues confirmed by actual users. For each tool, we calculate the number of user-confirmed problems that the tool could automatically detect. The key insight for designing A11YPUPPETRY was that a human tester interacts with it and interprets the results to locate accessibility issues that could require human knowledge to detect. Therefore, for A11YPUPPETRY, we also calculate the number of user-confirmed issues that evidence of the such problems exists in the report of A11YPUPPETRY. Table 8.2 shows the ratio of these calculated numbers to the total number of user-confirmed issues. As can be seen, A11YPUPPETRY, even its automated detected results, outperforms the existing tools. On average, A11YPUPPETRY could detect more than 70% of issues confirmed by users.

To have a better understanding of the detected issues, we manually analyzed all reported issues

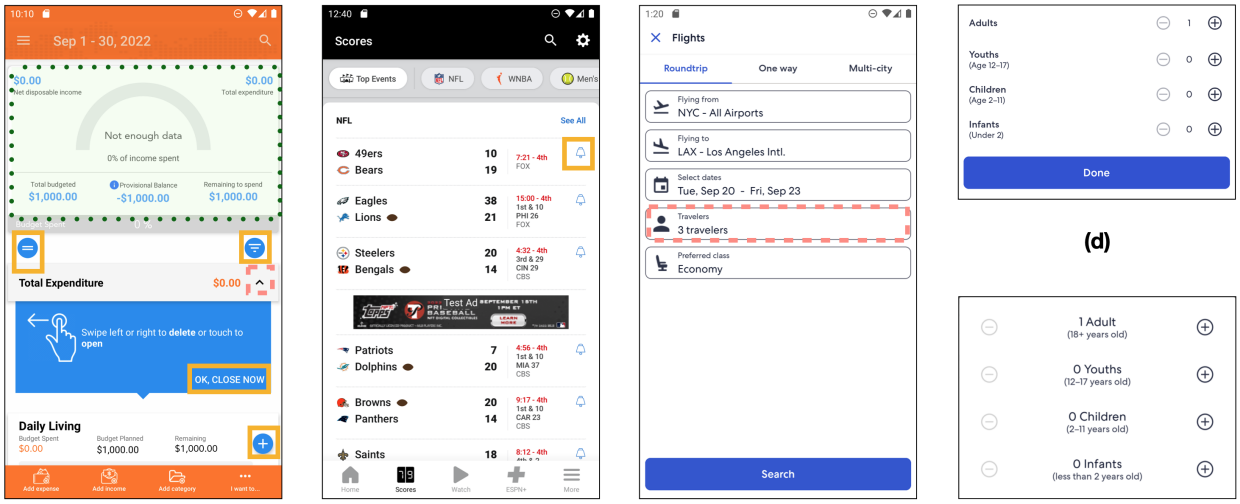


Figure 8.3: (a) the toggle button in iSaveMoney is not focusable and buttons indicated by yellow-solid boxes have ineffective action, (b) The content description of the notification icon in ESPN has unsupported characters, (c) The textual description of travelers numbers are different in Expedia, (d) (e) different fragments showing to different users

and categorized them into five categories: (1) **Automated Detection** the ones that both users and A11YPUPPETRY reported, (2) **Evidence Provided** the ones that users reported and A11YPUPPETRY provide some evidence of the existence of such issue in its report which can guide the tester to detect the issue, (3) **Unsettled Issues** that A11YPUPPETRY reported, but users did not find significant, (4) **Flaky Issues** that A11YPUPPETRY mistakenly reported as issues, and (5) **Undetected Issues** are the one that users reported but A11YPUPPETRY did not provide any evidence of such issue. In the following, we explain the subcategories of each of these categories and provide illustrative examples.

Automated Detection

Missing Speakable Text This issue (a visual element without the content description) is among the most common types of accessibility issues in mobile apps [36]. Due to the nature of this issue, existing accessibility testing techniques, like Accessibility Scanner, can detect this issue by only analyzing the layout of the app without considering assistive services. A11YPUPPETRY detects

such issues using the Search navigation, i.e. if an element is not associated with a textual description, it cannot be searched with TalkBack.

Unfocusable Element In this issue, an element associated with functionality or data cannot be focused by TalkBack; as a result, TalkBack users cannot access them or even realize such an element exists. In Section 8.1, we gave an example of such an issue (the speaker button in Figure 8.1(b)). Note that this issue cannot be detected by Accessibility Scanner since it requires assessing whether the element is focusable by TalkBack in runtime.

Sometimes, the unfocusable element belongs to a minor feature that the user may not need, for example, the collapse button in the iSaveMoney app that hides the details of expenses (red-dashed box in Figure 8.3(a)). However, sometimes this issue becomes critical, for example, on one of the search pages of Expedia, none of the elements on the screen, including Navigate Up Button, are focusable, making the user confused. A user mentioned: "After typing New York and pressing the search button, I am unable to move around the screen at all. None of the gestures that I use to navigate or read the screen work."

Ineffective Action Sometimes elements are focused on by TalkBack; however, the intended action cannot be performed. For example, in the iSaveMoney app, many buttons, including all yellow-solid boxes in Figure 8.3(a), can be focused by TalkBack; however, after clicking them by double tap, nothing happens. It seems the underlying reason behind this issue is the customized implementation of the buttons, which are sensitive to touch gestures and not click actions. The issue is also found in Doordash when the user wants to change the delivery option to pick-up.

Evidence Provided

The following issues are reported by users and not by A11YPUPPETRY; however, the aggregated report of A11YPUPPETRY, including the annotated video and blindfold mode, provides evidence of these issues. The report can help accessibility testers to reveal the issue faster without the need to interact with apps multiple times.

Uninformative Textual Description The main purpose of content description of elements is to help users with visual impairment understand the app better; as a result, merely having a content description does not improve accessibility. A11YPUPPETRY is not capable of analyzing the semantics of content descriptions; however, its blindfold mode lists the texts that are announced while exploring the app. A developer/tester can determine whether the textual descriptions are informative or not by reading the blindfold mode report. Here are some examples of this type of issue confirmed by users.

- The textual element has some random or irrelevant data. For example, the notification icon in ESPN, highlighted button in Figure 8.3(b), has a content description "Í", which is not informative
- The elements associated with a functionality, e.g., button, checklist, or tab, should express their functionality. While TalkBack takes care of standard elements like *android.widget.button*, it does not announce the functionality of non-standard elements, e.g., a button which is a *android.widget.TextView*. Doordash app has many of these issues, e.g., "Save" without button or "Pickup/Delivery" without announcing toggle.
- The textual description should describe the purpose of the element completely. For example, on the renting page of Expedia, there is a compound element described as "Pick-Up"; however, it is unclear if it is related to location or date. A sighted user can easily recognize it by looking at the pinpoint icon inside this element which hints this element is related to the location of picking up.

- Sometimes, the textual descriptions provide complete information; however, they can be incorrect. For example, the traveler's element, highlighted in Figure 8.3(c), clearly shows there are 3 travelers selected; however, its textual description is "Number of travelers. Button. Opens dialog. 1 traveler" which is not correct.

Difficulties in Reading Besides the textual description of elements, the way the texts are announced by TalkBack is important for understanding an app. We found a few accessibility issues reported by the users that make it difficult for them to perceive the text. This kind of issue can be detected by testers by manually analyzing the blindfold mode and annotated videos. For example, in Dictionary, paragraphs of texts cannot be read as a whole; the user has to read a long text word by word. Or in the Doordash app, yellow-solid boxes in Figure 8.4(a), each category on the main page is announced two times, one time the visible text, e.g., Grocery or Chicken, another time the image which didn't have textual description, announced as "unlabeled". In another example, all content of the summary block in the iSaveMoney app, green-dotted highlighted box in Figure 8.3(a), are announced altogether in an unintuitive order, and the user had to change the reading mode to understand each word. Although these issues do not make the app incomprehensible, they are barriers to blind users from using these apps conveniently. We asked one of the interviewees how they felt about this kind of inaccessibility, and he said he could deal with them "but we, blind people or deaf people, deserved the same amount of dignity as others."

Unsettled Issues

A11YPUPPETRY detected some issues that the users in our user studies did not find them significant issues. Mainly these issues belong to Jump and Search navigation modes. In the Jump navigation mode, TalkBack Proxy User tries to locate the element using jump navigation (going to the next control or heading element); however, sometimes, it is not possible to reach to element since it does not have proper attributes, e.g., it is not a button. TalkBack Proxy User with Search navigation mode

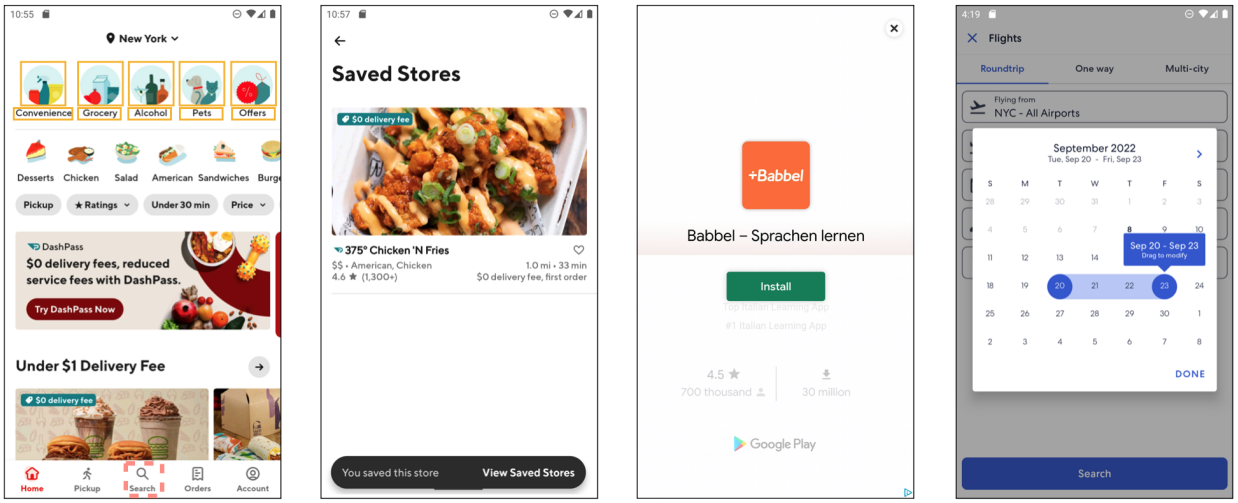
tries to locate the elements by searching their textual description; however, when there are multiple elements with the same description, this mode cannot locate the element correctly. Although users mentioned it would be nice if the attributes were set properly so they could use different navigation modes; they did not find these issues important since they usually do not use Jump and Search navigation modes. We further examined why users do not use these modes often in Section 8.4.

Flaky Issues

Sometimes A11YPUPPETRY reports issues that are not correct, which is caused by technical problems with the experiments. The main characteristic of this category is that by rerunning A11YPUPPETRY, the issue may not be reported again. There are three main technical problems. First, TalkBack sometimes freezes and does not respond properly and on time, making A11YPUPPETRY think the app has accessibility issues that do not let TalkBack continue the exploration. Secondly, the recorder may record incorrect elements to perform; for example, on the signup page of the ESPN app, instead of recording a button, it records a transparent view covering the button, which does not interfere with the touch interaction. Lastly, the apps can be changed and be in different states on TalkBack Proxy User devices. Mainly this issue is caused by A/B testing, where developers dynamically show different pages to different users to measure some metrics about their product. For example, Figure 8.3(d) and (e) are two different fragments of changing the number of travelers in the Expedia app. If the recorder records the action in Figure 8.3(d), the same element cannot be found in Figure 8.3(e) since the structure is totally different.

Undetected Issues

As expected, A11YPUPPETRY cannot provide all accessibility-related feedback, and the best way to evaluate the accessibility of apps is by conducting user studies with disabled users. We categorized the limitation of A11YPUPPETRY in the following categories.



(a) (b) (c) (d)

Figure 8.4: (a) After pressing the search tab in DoorDash, a new search page appears without any announcement, (b) List of saved stores in DoorDash, (c) The interstitial ad in Dictionary app and the close tab is not focusable by TalkBack, (d)The accessible calendar in Expedia

Improper Change Announcement As users interact with mobile apps, the layout constantly changes. A sighted user can monitor all these changes to understand the latest state of the app; however, it is much more difficult for users with visual impairment to realize something is changed in the app. During our interview, users reported a couple of these kinds of issues. For example, when the user presses the search tab in the Doordash app, the red-dashed box in Figure 8.4(a), a completely new search page appears without any announcement for TalkBack users. One participant mentioned "My preference is that whenever something like that happens, [TalkBack] moves the focus up to where the new content begins because someone as a screen reader won't necessarily [realize the app is changed]."

Excessive Announcement On the other hand, it can be problematic and annoying when TalkBack announces content more than users need. For example, in the Expedia app, when a user types a name in the search edit box, TalkBack interrupts the user by announcing "Suggestions are being loaded below". Although it is informative for users to know the search results are loaded on the fly, it is annoying to interrupt constantly.

Temporary Visible Elements Sometimes, apps introduce new elements for a short period to notify the user something has changed and let the user undo or do something relevant to this change. For example. in the Doordash app, when the user saves a restaurant as her favorite, a pop-up box appears, Figure 8.4(b), notifying the user the store is saved and disappears momentarily. A blind user is informed of this change; however, she does not have enough time to focus on the appearing dialogue.

8.4 Discussion

The previous section demonstrates the effectiveness of A11YPUPPETRY in providing insights and detecting accessibility issues. This section discusses other findings from the user studies that might be insightful for future research work.

TalkBack Interaction Preferences. We further examine how users with visual impairments interact with apps using TalkBack. We asked the interviewees of the user interview to explain the different ways they use TalkBack. If they did not mention any of the navigation ways that we found in TalkBack documentation, we ask them if they are aware of them.

Generally, the primary way of navigation mode for all participants is Linear navigation. A user mentioned "I'm more into the flick, element to element, to explore an app and understand its layout." This mode is especially used when the user interacts with an app or page that is unfamiliar.

The next favorite way of navigating is through Touch mode; however, it is usually used in certain scenarios. For example, when the user knows about the possible location of elements, use Touch navigation mode. One participant mentioned "The back buttons are always at the top left, usually so... I'm going to put my finger at the top left to find that back button.". Also, when the user cannot find the element or is stuck in a loop, use touch to find the target element.

Some interviewees said they might use Jump navigation for headings in the apps that they are familiar with. One participant said "If I don't know [the app] well enough ... I'm going to flick through the whole thing to figure out the layout. If I know it well enough, then I probably would switch to the heading option and then search by heading". However, almost none of the participants are willing to use the Search navigation mode. One user mentioned "I know [search] is there. But I prefer to just hunt for [the elements]. It gives me a more experience with the app."

Besides the element locating, we realized the users do not want to use other actions like scrolling since scrolling confuses them in understanding the new state of the app. A user said "[I use scrolling] if I know an app really well. But sometimes I find that when I do the scrolling thing, it'll get me into something else... sometimes it'll get me where I really don't want to be. So I have a tendency not to want to do it."

Context. A common accessibility issues in mobile apps are missing speakable text [36, 7]. Although missing speakable text degrades the user experience and ability to locate elements, sometimes users can figure out the functionality of the unlabeled button given their context. For example, the user can view the list of saved stores in Doordash and remove any of them, as depicted in Figure 8.4(b). The element for removing a store is an icon with the shape of a heart without a content description. However, our interviewee did not have a problem with locating this button. He mentioned "That is a good layout, an accessible checkbox next to [the restaurant], which is checked unchecked. I have seen these checkboxes on the home screen. I don't like them on the home screen because the user doesn't know what that checkbox actually does. The common sense here would tell you I'm in the saved stores' section. So if I uncheck a box, it's going to remove that." Anyway, this observation should not encourage developers not to care about missing content descriptions; on the contrary, it emphasizes the importance of context for users with visual impairment to understand the app better.

Advertisement. In the experiments of recording and replaying with A11YPUPPETRY, we did not observe any ads. However, if an interstitial ad appears during the replay process, A11YPUPPETRY may fail to continue as the appearance of ads is random and irregular. For example, for the Dictionary app, the interstitial ad, such as Figure 8.4(c), might appear when the user searches for a word. Disabled users have difficulty noticing the occurrence of the ads until they get stuck in the ads window for a few minutes. Even if they are aware of the ads, closing them and returning to the previously interrupted use case is challenging. One of the interviewees tried to locate the app with Linear and Touch navigation modes; however, the ad's close button seems not focusable by TalkBack. As a result, the user had to restart the app (close and open again) to continue the task.

All the interviewees are cautious about the in-app advertisements. As one stated, "I tend not to open [the in-app advertisements] because half of the time, these advertisements cause problems." In addition, most interviewees admitted that they are willing to pay for the ad-free version if the price is not high, so they do not need to deal with ads while navigating apps. A user mentioned: "If the app gives me the option to do without ads with a small price, I pay the small price just so I don't have to deal with the ads. Most of the time [the ads] don't work with the screen readers." Nevertheless, previous research indicates that some apps still contain ads even if users pay ad-free fees [58].

To the best of our knowledge, only one previous research investigated the impact of ads on disabled users. The research found that most ads are represented in GIFs, and more than half of the sampled ads have no ALT tag [117]. Therefore, screen readers can not read the contents of the ads to blind users. Other researchers investigated the impact of ads on the whole user group, not just disabled users. The negative influences of ads include privacy threats, significant battery consumption, slowing down the app, and disabling an app's normal function [58, 48]. We argue that the negative influences mentioned above could worsen for disabled users as they rely on assistive technologies to accomplish the task. Remember, the previous section illustrates disabled users usually need more steps and time to achieve an equivalent action that users without disability perform.

There are some design implications for in-app advertisements. Generally, ads that take the entire screen are named interstitial ads, while ads that are represented as horizontal strips are named banner ads. The ads should be announced correctly via Assistive Services so disabled users can know the occurrence of the ads. In addition, developers are encouraged to design banner ads since the banner ads usually will not disable an app's functionality. By contrast, interstitial ads significantly attract users' attention and even require users to close the ad manually [58]. Developers should also ensure the ads disappear after users pay ad-free fees.

Guided Navigation. The interviewees enjoyed interacting with an app when the app guided them through the process. In particular, Expedia did a great job in reserving flights: it consists of several steps like asking about the origin and destination airports, and dates. Once each step is done, the focus is changed to the next question and also announces the changes. Users are also able to get out of this selection and get back to the search page to change or view other information. One of the interviewees was especially happy about the calendar, Figure 8.4(d), and mentioned "That was one of the coolest mobile calendars I've ever used because it walked me through where I was. I selected the start date, and it told me that, and then it said, pick your end date, and then it summarized with states, like September 19th Start date or September 20th in the trip."

Alternative Suggestion. As we discussed before, there are many complex touch gestures that do not have equivalent in TalkBack, e.g., dragging or pinching in. Developers are recommended to provide alternative interactions for complex gestures. For example, the calendar widget in Expedia, Figure 8.4(d), suggests that sighted users modify the selected dates by dragging the start to end dates. Moreover, for TalkBack users, it announces "Select dates again to modify" which is an alternative way for modifying the dates.

Common Sense. During the interviews, we noticed participants sometimes locate elements much faster than the other elements. In particular, for elements like "Search" or "Back", instead of using

Linear navigation, they explored certain parts of the app by Touch navigation and locating the element. We asked how they locate these elements and they generally respond they predict the location of these elements with the help of common sense. For example, the back button or open navigation drawer is usually located on the top left of the element, or menus are located in the footer. Common sense is not limited to similar elements on the screen. In the interview for the Doordash app, the interviewee found the button that shows the address of a restaurant pretty fast, even though the button was unlabeled. When we asked how he found such an element, he responded "A normal company would put the address on top, you know. So I'm using it. That's common sense." Therefore, it is important for developers to not change the spatial aspects of UI elements without considering users' habits.

8.5 Conclusion

This chapter introduced A11YPUPPETRY, a semi-automated record-and-replay technique for detecting accessibility issues in mobile app use cases using TalkBack, the official screen reader in Android. A11YPUPPETRY records the user touch gestures in a device, translates the gestures into their equivalent action in TalkBack, and performs them on four different devices with four navigation modes in TalkBack. Finally, A11YPUPPETRY analyzes reports of the recorder and replayers and generates aggregated and visualized reports for developers, even without knowledge of accessibility. We evaluated A11YPUPPETRY by conducting user studies with users with visual impairments. We realized A11YPUPPETRY is capable of detecting various types of accessibility issues that cannot be detected by existing tools, and also can provide evidence for some accessibility barriers that can be provided to accessibility experts, without taking their time to spend time on interacting with TalkBack.

Chapter 9

Conclusion

In this dissertation, I introduced the idea of using assistive services for the automated evaluation of accessibility in mobile apps and presented a set of automated testing tools built on this idea. I showed that although existing guideline-based automated testing is helpful in detecting various types of accessibility issues, they are limited to detecting issues that only are manifested in runtime with an assistive service– the way users with disabilities interact with an app.

In order to utilize assistive services automatically, I introduced Proxy User. A Proxy User is a program that interacts with a mobile device using an assistive service. Powered by Proxy Users, I applied the idea of assistive-service-driven testing on three UI testing areas. First, I introduced LATTE, which reuses existing UI test cases by translating them to human-readable use case specifications, then executing them with two official assistive services in Android: TalkBack and SwitchAccess. Second, I designed fully automated test input generation focused on accessibility (GROUNDHOG and OVERSIGHT) to evaluate mobile apps' under- and over-accessibility issues by crawling the apps and perform actions with and without assistive services. Finally, I introduced A11YPUPPETRY, a record-and-replay technique to aid manual testers with limited knowledge of accessibility and assistive services assess the accessibility of mobile apps.

In the remainder of this chapter, I conclude my dissertation by enumerating the contributions of my work and avenues for future work.

9.1 Research Contribution

- **Fundamental contribution to accessibility testing** I proposed a novel approach to assess the accessibility of mobile apps that focus on the way users with disabilities interact with devices. The concept of Proxy User abstracts away the engineering details of working with assistive services programmatically. I built a prototype of Proxy Users for the Android platform; however, this concept can be easily applied to other platforms like iOS or the Web. Moreover, the applicability of Proxy User is realized in the proposed four different techniques (LATTE, GROUNDHOG, OVERSIGHT, and A11YPUPPETRY).

- **Tools and experiments.** I designed and built the proposed testing techniques for the Android platform and evaluated them on real-world and popular apps. These tools are implemented in Client-Server architecture and are capable of parallelization, especially on the cloud. The source code of all tools and supplementary materials (like subject apps in the experiments) are publicly available [110, 114, 96, 1]; moreover, all experiments are written as Python scripts to help other researchers reuse, replicate, and extend the proposed approaches.

- **User study.** I conducted user studies to create a more realistic understanding of what users with disabilities need and want. The insights from these studies may help other researchers to focus on undiscovered and significant accessibility problems. **Dataset.**

  - A dataset of 50 UI test cases on 20 apps associated with detected accessibility issues (already reused in other researchers' work [5])

  - A dataset of 30 apps with various accessibility issues, including super popular apps like Facebook

- A dataset of 40 accessibility issues confirmed by blind users

9.2 Future Work

Fixing Accessibility Issues A straightforward research direction on top of accessibility issue detection mechanisms is fixing techniques. Some issues are trivial to find, like missing speakable text (the most common accessibility issue in mobile apps). My colleague and I already designed a machine learning tool to fix this issue in CoALa [89]. However, the proposed techniques in this dissertation are capable of detecting more complex issues automatically. As a future research idea, a search-based algorithm can be designed to search for a source code that does not have navigational issues detected by one of the proposed tools. Since GROUNDHOG and OVERSIGHT are entirely automated and do not require manual input, they can be great choices as oracles for automated accessibility fixing techniques.

Regression Accessibility Testing Regression testing is a practical approach for validating software quality as it evolves, and accessibility is one of the essential software qualities. Any change in a mobile app may change the UI, which blind users are familiar with, or even introduce accessibility issues. Future research work can optimize and extend the proposed assessment technique to add a wholly automated accessibility testing component to regression testing. In particular, LATTE can be used as the base of such a technique since it is designed to work with UI test cases.

Education An effective way to encourage developers to build more accessible apps is to make them aware and knowledgeable regarding accessibility and what users with disability need and want. Mobile developers without disabilities are likely to design, develop, and test their products for users without disabilities since they are not aware of how users with disability interact with an app. The proposed techniques in this dissertation, in particular A11YPUPPETRY, can be used and

extended to help developers and even students know their design's impact on other users. A future research direction may study the effects of visualization tools on developers' empathy toward users with disability.

Bibliography

[1] A11yPuppetry. A11ypuppetry companion website. https://a11ypuppetry.github.io/, 2022. Last Accessed: September 15, 2022.

[2] P. Ackland, S. Resnikoff, and R. Bourne. World blindness and visual impairment: despite many successes, the problem is growing. *Community eye health*, 30(100):71, 2017.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, Austin, TX, 2016. IEEE, IEEE.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[5] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond. Automated detection of talkback interactive accessibility failures in android applications. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 232–243, Virtual, 2022. IEEE, IEEE.

[6] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond. Automated detection of talkback interactive accessibility failures in android applications. In *15th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2022.

[7] A. Alshayban, I. Ahmed, and S. Malek. Accessibility issues in android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*, pages 1323–1334, Virtual, 2020. ICSE.

[8] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[9] Android. Accessibility scanner - apps on google play. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US, 2022. Last Accessed: May 6, 2022.

[10] Android. Accessibility testing framework. https://github.com/google/Accessibility-Test-Framework-for-Android, 2022. Last Accessed: May 6, 2022.

[11] Android. Accessibilityservice in android. https://developer.android.com/guide/topics/ui/accessibility/service, 2022. Last Accessed: May 6, 2022.

[12] Android. Andoird accessibility api, accessibilityaction. https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo.AccessibilityAction, 2022. Last Accessed: October 12, 2022.

[13] Android. Android accessibility overview. https://support.google.com/accessibility/android/answer/6006564, 2022. Last Accessed: May 6, 2022.

[14] Android. Android debug bridge. https://developer.android.com/studio/command-line/adb, 2022. Last Accessed: May 6, 2020.

[15] Android. Build more accessible apps. https://developer.android.com/guide/topics/ui/accessibility, 2022. Last Accessed: May 6, 2022.

[16] Android. Control your android device with switch access. https://support.google.com/accessibility/android/answer/6122836?hl=en, 2022. Last Accessed: May 6, 2022.

[17] Android. Espresso : Android developers. https://developer.android.com/training/testing/espresso, 2022. Last Accessed: May 6, 2022.

[18] Android. Get started on android with talkback - android accessibility help. https://support.google.com/accessibility/android/answer/6283677?hl=en, 2022. Last Accessed: May 6, 2022.

[19] Android. Google play. https://play.google.com/store/apps, 2022. Last Accessed: May 6, 2022.

[20] Android. Improve your code with lint checks. https://developer.android.com/studio/write/lint?hl=en, 2022. Last Accessed: May 6, 2020.

[21] Android. Talkback source code by google. https://github.com/google/talkback, 2022. Last Accessed: May 6, 2022.

[22] Android. Use touch gestures. https://developer.android.com/develop/ui/views/touch-and-input/gestures, 2022. Last Accessed: August 29, 2022.

[23] Android. Webview - android documentation. https://developer.android.com/reference/android/webkit/WebView, 2022. Last Accessed: May 6, 2022.

[24] appetizerio. Replaykit. https://github.com/appetizerio/replaykit, 2022. Last Accessed: September 2, 2022.

[25] Appium. Mobile app automation made awesome. http://appium.io/, 2020.

[26] Apple. Accessibility on ios. https://developer.apple.com/accessibility/ios/, 2022. Last Accessed: May 6, 2021.

[27] Apple. Apple accessibility. https://www.apple.com/accessibility/iphone/, 2022. Last Accessed: May 6, 2020.

[28] Apple. Debug accessibility in ios simulator with the accessibility inspector. https://developer.apple.com/library/archive/technotes/TestingAccessibilityOfiOSApps/TestAccessibilityiniOSSimulatorwithAccessibilityInspector/TestAccessibilityiniOSSimulatorwithAccessibilityInspector.html#//apple_ref/doc/uid/TP40012619-CH4-SW1, 2022. Last Accessed: May 6, 2022.

[29] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660, 2013.

[30] F. Behrang and A. Orso. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 54–65. IEEE, 2019.

[31] T. Bi, X. Xia, D. Lo, J. Grundy, T. Zimmermann, and D. Ford. Accessibility in software practice: A practitioner's perspective. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–26, 2022.

[32] N. P. Borges, M. Gómez, and A. Zeller. Guiding app testing with mined interaction models. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 133–143. IEEE, 2018.

[33] G. Brajnik. Comparing accessibility evaluation tools: a method for tool effectiveness. *Universal access in the information society*, 3(3-4):252–263, 2004.

[34] M. M. B.V. Meditation moments. https://play.google.com/store/apps/details?id=com.meditationmoments.meditationmoments&hl=en_US&gl=US, 2022. Last Accessed: March 10, 2022.

[35] M. Campbell. Lock screen bypass enables access to Notes in iOS 15, 2021.

[36] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, and G. Li. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*, page 322–334, Virtual, 2020. ICSE.

[37] P. T. Chiou, A. S. Alotaibi, and W. G. Halfond. Detecting and localizing keyboard accessibility failures in web applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 855–867, Virtual, Athens, Greece, 2021. ACM New York, NY, USA.

[38] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices*, 48(10):623–640, 2013.

[39] W. Choi, K. Sen, G. Necul, and W. Wang. Detreduce: minimizing android gui test suites for regression testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 445–455, Gothenburg, Sweden, 2018. IEEE, IEEE.

[40] A. Clark and Contributors. Pillow, python imaging library. `https://pillow.readthedocs.io/en/stable/`, 2022. Last Accessed: May 6, 2022.

[41] B. R. Connell. The principles of universal design, version 2.0. *http://www. design. ncsu. edu/cud/univ_design/princ_overview. htm*, 1997.

[42] F. Y. B. Daragh and S. Malek. Deep gui: Black-box gui input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 905–916. IEEE, 2021.

[43] Dictionary.Com. Dictionary.com english word meanings & definitions. `https://play.google.com/store/apps/details?id=com.dictionary`, 2022. Last Accessed: August 29, 2022.

[44] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, pages 1–12, Seoul, South Korea, 2020. IEEE.

[45] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, pages 116–126, Västerås, Sweden, 2018. ICST.

[46] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. Barista: A technique for recording, encoding, and running platform independent android tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 149–160, Tokyo, Japan, 2017. IEEE, IEEE.

[47] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Tivos: Trusted visual i/o paths for android. *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[48] C. Gao, J. Zeng, F. Sarro, D. Lo, I. King, and M. R. Lyu. Do users care about ad's performance costs? exploring the effects of the performance costs of in-app ads on user experience. *Information and Software Technology*, 132:106471, 2021.

[49] N. Ghorbani, R. Jabbarvand, N. Salehnamadi, J. Garcia, and S. Malek. Deltadroid: Dynamic delivery testing in android. *ACM Trans. Softw. Eng. Methodol.*, sep 2022. Just Accepted.

[50] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 72–81, San Francisco, CA, USA, 2013. IEEE, IEEE.

[51] Google. Accessibilitynodeinfo. `https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo#isVisibleToUser()`, 2020. Last Accessed: March 6, 2022.

[52] Google. Accessibilityinteractioncontroller.java. https://android.googlesource.com/platform/frameworks/base/+/80943d8/core/java/android/view/AccessibilityInteractionController.java#680, 2022. Last Accessed: May 3, 2022.

[53] Google. Accessibilitynodeinfo. https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo, 2022. Last Accessed: March 12, 2022.

[54] Google. Facebook lite - apps on google play. https://play.google.com/store/apps/details?id=com.facebook.lite&hl=en_US&gl=US, 2022. Last Accessed: May 6, 2022.

[55] Google. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey, 2022. Last Accessed: May 6, 2022.

[56] U. Government. U.s. revised section 508 standards. https://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-ict-refresh/final-rule/text-of-the-standards-and-guidelines, August 20, 2020.

[57] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280, Montreal, Canada, 2019. IEEE, IEEE.

[58] J. Gui, M. Nagappan, and W. G. Halfond. What aspects of mobile ads do users care about? an empirical study of mobile in-app ad reviews. *arXiv preprint arXiv:1702.07681*, 2017.

[59] J. Guo, S. Li, J.-G. Lou, Z. Yang, and T. Liu. Sara: self-replay augmented record and replay for android in industrial cases. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*, pages 90–100, Beijing, China, 2019. Association for Computing Machinery.

[60] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224, Philadelphia, PA, USA, 2015. IEEE, IEEE.

[61] 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 *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217, Bretton Woods, New Hampshire, USA, 2014. ACM New York, NY, USA.

[62] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 349–366, Auckland , New Zealand, 2015. Association for Computing Machinery.

[63] Y. Hu and I. Neamtiu. Valera: an effective and efficient record-and-replay tool for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 285–286, 2016.

[64] J. Huang, M. Backes, and S. Bugiel. A11y and privacy don't have to be mutually exclusive: Constraining accessibility service misuse on android. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3631–3648, 2021.

[65] IBM. Ibm accessibility requirements. `https://www.ibm.com/able/guidelines/ci162/accessibility_checklist.html`, 2022. Last Accessed: May 6, 2020.

[66] W. Inc. Geek - smarter shopping. `https://play.google.com/store/apps/details?id=com.contextlogic.geek&hl=en_US`, 2020.

[67] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Baltimore, MD, USA, 2013. IEEE, IEEE.

[68] K. Kaja. Doordash issue tweet. `https://twitter.com/kirankaja12/status/1551710324016836608`, 2022. Last Accessed: September 15, 2022.

[69] KewlApps. Applock. `https://play.google.com/store/apps/details?id=com.gamemalt.applocker`, 2022. Last Accessed: March 10, 2022.

[70] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, Graz, Austria, 2015. IEEE, IEEE.

[71] F. Koroy. Another BAD iOS 12 Passcode Bypass! 12.1/12.0.1 (Works on XS), 2018.

[72] F. Koroy. iOS 12 Passcode Bypass! Photos & Contacts (Works on XS), 2018.

[73] W. Lachance. Orangutan. `https://github.com/wlach/orangutan`, 2022. Last Accessed: September 2, 2022.

[74] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie. Record and replay for android: Are we there yet in industrial cases? In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 854–859, 2017.

[75] T. J.-J. Li, A. Azaria, and B. A. Myers. Sugilite: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 6038–6049, Bremen , Germany, 2017. Association for Computing Machinery.

[76] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[77] Y. Li, Z. Yang, Y. Guo, and X. Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019.

[78] J.-W. Lin, R. Jabbarvand, and S. Malek. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53. IEEE, 2019.

[79] J.-W. Lin, N. Salehnamadi, and S. Malek. Test automation in open-source android apps: A large-scale empirical study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1078–1089, Virtual, Australia, 2020. ACM New York, NY, USA.

[80] J.-W. Lin, N. Salehnamadi, and S. Malek. Route: Roads not taken in ui testing. *ACM Trans. Softw. Eng. Methodol.*, sep 2022. Just Accepted.

[81] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, Shanghai, China, 2017. IEEE, IEEE.

[82] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[83] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[84] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, Saarbrücken, Germany, 2016. ACM New York, NY, USA.

[85] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas. An evolutionary approach to adapt tests across mobile apps. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 70–79. IEEE, 2021.

[86] G. Material Design. Gestures. `https://material.io/design/interaction/gestures.html#principles`, 2022. Last Accessed: August 29, 2022.

[87] O. Matters. Mobile app backlog is directly damaging revenue in the enterprise. `http://www.bizreport.com/whitepapers/mobile_app_backlog_is_directly.html`, 2020. Last Accessed: September 15, 2020.

[88] F. Mehralian, N. Salehnamadi, S. F. Huq, and S. Malek. Too much accessibility is harmful! automated detection and analysis of overly accessible elements in mobile apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*, Michigan, USA, 2022. IEEE, ACM New York, NY, USA.

[89] F. Mehralian, N. Salehnamadi, and S. Malek. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 107–118, Virtual, Athens, Greece, 2021. ACM New York, NY, USA.

[90] Microsoft. Accessibility insights for android. `https://accessibilityinsights.io/docs/en/android/overview/`, 2022. Last Accessed: March 13, 2022.

[91] Microsoft. An app platform for building android and ios apps with .net and c#. `https://dotnet.microsoft.com/en-us/apps/xamarin`, 2022. Last Accessed: May 6, 2022.

[92] D. T. Milano. Culebra. `https://github.com/dtmilano/AndroidViewClient/wiki/culebra`, 2022. Last Accessed: September 2, 2022.

[93] M. Miller. Monetization insights from app professionals. `https://www.data.ai/en/insights/app-monetization/app-marketers-developers-survey-2/`, 2017.

[94] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[95] M. Naseri, N. P. Borges Jr, A. Zeller, and R. Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. In *The 19th Privacy Enhancing Technologies Symposium, Jul 2019, Stockholm, Sweden*. Springer, 2019.

[96] OverSight. Oversight. `https://sites.google.com/view/oversight2`, 2022.

[97] Pallets. Flask, the python micro framework for building web applications. `https://github.com/pallets/flask`, 2022. Last Accessed: May 6, 2022.

[98] H. Petrie and O. Kheir. The relationship between accessibility and usability of websites. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 397–406, San Jose, California, USA, 2007. CHI.

[99] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1120–1136, 2018.

[100] C. Power, A. Freire, H. Petrie, and D. Swallow. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 433–442, Texas, USA, 2012. CHI.

[101] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 571–582, Texas, Austin, 2016. Association for Computing Machinery.

[102] Ranorex. ranorex. `https://www.ranorex.com/mobile-automation-testing/android-test-automation/`, 2022. Last Accessed: September 2, 2022.

[103] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. In *NDSS*, 2017.

[104] Robolectric. robolectric/robolectric, Jul 2019.

[105] RobotiumTech. robotiumrecorder. `https://github.com/RobotiumTech/robotium`, 2022. Last Accessed: September 2, 2022.

[106] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*, pages 2–11, Baltimore, MD, USA, 2017. ASSETS.

[107] O. Sahin, A. Aliyeva, H. Mathavan, A. Coskun, and M. Egele. Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 128–138, San Diego, CA, USA, 2019. IEEE, IEEE.

[108] N. Salehnamadi, A. Alshayban, I. Ahmed, and S. Malek. A benchmark for event-race analysis in android apps. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 466–467, 2020.

[109] N. Salehnamadi, A. Alshayban, I. Ahmed, and S. Malek. Er catcher: a static analysis framework for accurate and scalable event-race detection in android. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 324–335. IEEE, 2020.

[110] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte companion website. `https://github.com/seal-hub/Latte`, 2020.

[111] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–11, Virtual, Okohama, Japan, 2021. ACM New York, NY, USA.

[112] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2021. Association for Computing Machinery.

[113] N. Salehnamadi, F. Mehralian, and S. Malek. Groundhog: An automated accessibility crawler for mobile apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*, Michigan, USA, 2022. IEEE, ACM New York, NY, USA.

[114] N. Salehnamadi, F. Mehralian, and S. Malek. Groundhog companion website. `https://github.com/seal-hub/Groundhog`, 2022. Last Accessed: September 1, 2022.

[115] C. Silva, M. M. Eler, and G. Fraser. A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, pages 286–293, Thessaloniki, Greece, 2018. DSAI.

[116] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, Paderborn, Germany, 2017. ACM New York, NY, USA.

[117] D. Thompson and B. Wassmuth. Accessibility of online advertising: a content analysis of alternative text for banner ad images in online newspapers. *Disability Studies Quarterly*, 21(2), 2001.

[118] W3. Web content accessibility guidelines (wcag) overview. `https://www.w3.org/WAI/standards-guidelines/wcag/`, 2020. Last Accessed: August 20, 2020.

[119] W3. Principle 1: Perceivable. `https://www.w3.org/TR/WCAG20/#perceivable`, 2022. Last Accessed: March 15, 2022.

[120] W3. Principle 2: Operable. `https://www.w3.org/TR/WCAG20/#operable`, 2022. Last Accessed: March 15, 2022.

[121] W3. Web content accessibility guidelines (wcag) overview. `https://www.w3.org/WAI/standards-guidelines/wcag/`, 2022. Last Accessed: May 6, 2022.

[122] W3. Xml path language. `https://www.w3.org/TR/2017/REC-xpath-31-20170321/`, 2022. Last Accessed: May 6, 2022.

[123] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 738–748. IEEE, 2018.

[124] WHO. World report on disability. `https://www.who.int/disabilities/world_report/2011/report/en/`, 2011. Last Accessed: May 6, 2022.

[125] X. Zhang, A. S. Ross, and J. Fogarty. Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 609–621, Berlin, Germany, 2018. UIST.