UNIVERSITY OF CALIFORNIA,
IRVINE

Automated Techniques for Improving the Accessibility of Android Applications for Screen Reader Users

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Forough Mehralian

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I extend my gratitude to my advisor, Professor Sam Malek. He was patient with my mistakes, generous in giving chances, and believed in me more than I believed in myself. His wisdom shines through in his ability to discern when to offer praise to bolster your confidence and when to push you to achieve even greater heights.

I want to thank my committee members, Professors Iftekhar Ahmed and Anne Marie Piper, for their invaluable feedback on my dissertation. I am also deeply grateful for the support and guidance of Professors Cristina Lopes and Sameer Singh throughout this journey. Being a member of the SEAL lab at UCI has been a privilege, and I appreciate the opportunity to learn from and collaborate with my colleagues, especially Dr. Navid Salehnamadi, my collaborator, mentor, and friend, whose unwavering support has been invaluable.

This dissertation was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation. Also, I would like to thank Richard N. Taylor, Informatics Department, and UCI Grad Division for partly supporting my research financially.

This dissertation would not have been possible without the support of my friends over the past six years. I am especially grateful to Dr. Zahra Montazeri, who has been like family to me and a tremendous source of inspiration throughout this journey.

Finally, I am deeply thankful to my mom and dad, who, despite being far away, supported me with their love and sacrifices. I am also grateful to my wonderful brother, Ehsan, and my inspiring sister, Fahimeh.

VITA

Forough Mehralian

EDUCATION

Doctor of Philosophy in Software Engineering	**2024**
University of California, Irvine	*Irvine, CA*
Master of Science in Software Engineering	**2018**
Sharif University of Engineering	*Tehran, Iran*
Bachelor of Science in Computational Sciences	**2015**
Sharif University of Engineering	*Tehran, Iran*

RESEARCH EXPERIENCE

AI/ML Research Intern	**March 2024–August 2024**
Apple Inc.	*Seattle, Washington*
Graduate Research Assistant	**2019–2024**
University of California, Irvine	*Irvine, California*
Graduate Research Assistant	**2015–2018**
Sharif University of Technology	*Tehran, Iran*

TEACHING EXPERIENCE

Teaching Assistant	**2018–2019, 2023**
University of California, Irvine	*Irvine, California*
Teaching Assistant	**2017-2018**
Sharif University of Technology	*Tehran, Iran*

REFEREED CONFERENCE PUBLICATIONS

Automated Accessibility Analysis of Dynamic Content Changes on Mobile Apps **Apr 2025**
International Conference on Software Engineering (ICSE)

MA11y: A Mutation Framework for Web Accessibility Testing **Sep 2024**
International Symposium on Software Testing and Analysis (ISSTA)

Groundhog: An Automated Accessibility Crawler for Mobile Apps **Oct 2022**
Automated Software Engineering (ASE)

Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps **Oct 2022**
Automated Software Engineering (ASE)

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

Automated Construction of Energy Test Oracle for Android **Nov 2020**
The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

ABSTRACT OF THE DISSERTATION

Automated Techniques for Improving the Accessibility of Android Applications for Screen Reader Users

By

Forough Mehralian

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2024

Professor Sam Malek, Chair

Universal design principles mandate that technologies and services, including mobile apps, should be accessible to all users, regardless of their abilities. However, these principles are often overlooked in development practices. This dissertation addresses the significant gap in mobile app accessibility for users with visual impairments, particularly those relying on Assistive Technologies (ATs) like screen readers. While existing guidelines and tools aim to improve accessibility, they often fall short in real-world scenarios, especially for dynamic content and interactive elements that require more than static rule-based analysis. This work advances the field of accessibility testing and repair by introducing three key contributions: (1) COALA, a deep learning approach for generating informative labels for unlabeled icons, overcoming biases in previous methods; (2) Groundhog and OverSight, automated tools that detect inconsistencies in app accessibility when using ATs, identifying issues related to both under-access and over-access problems; and (3) TimeStump, a framework for detecting and addressing accessibility challenges caused by dynamic content changes in apps. Through these innovations, this research enhances the accessibility of mobile apps for screen reader users, ensuring a more inclusive experience.

Chapter 1

Introduction

Principles of universal design [54] 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 impairments.

People with visual impairments rely on assistive technologies (ATs) such as screen readers to understand app content and interact with it. Mobile operating systems such as Android and iOS have also integrated screen readers to assist universal access to the app services. ATs provide alternative interaction modes for the users to explore the app. For example, TalkBack, the official screen reader in Android, navigate through UI elements as the user swipes and describe each element to the screen reader user. For image button, TalkBack announces textual labels that are provided by developers. Without a proper description of the functionality initiated by these buttons, screen reader users are unable to interact with an app, compromising the app accessibility.

To improve app accessibility, various technology institutes and companies such as World Wide Web Consortium [159], Apple [33], and Google [27] have released accessibility guidelines and best practices. These guidelines are accompanied by accessibility analysis tools that automatically analyze an app's compliance with the guidelines and identify accessibility issues [25, 17, 31, 34]. These tools scan the graphical user interface (GUI) of an app and produce reports of accessibility guideline violations, such as the lack of textual labels for image buttons.

However, research has shown that developers often fail to adopt these accessibility analysis tools or disregard the reported results due to the sheer number of relevant and irrelevant warnings [12]. This lead to many apps suffering from issues such as missing informative labels [136, 45]. With an automated label generation technique, we can tackle this problem by reducing the burden on developers and enhancing app accessibility.

Moreover, previous studies have shown that guidelines only cover half of the accessibility issues [129], and worse yet, less than half of these guidelines are checked by the existing accessibility analysis tools [157]. For instance, certain accessibility issues that arise when interacting with the app using ATs cannot be captured by existing tools. Case in point, a button that is easily visible to sighted users and clickable by touch, may not be localized or selected by screen readers. Additionally, these tools do not account for dynamic app content changes that may affect accessibility. Despite of sighted users that can see the entire screen at a glance, blind users, with the help of screen readers, can only perceive on one element at a time. Therefore, they may not be aware of a newly appearing button on top of the screen. Further automated detection approaches are necessary to localize such issues that hinder screen reader users' ability to interact with apps.

This research proposes to advance app accessibility testing and repair by: (1) introducing an automated accessibility repair technique to generate informative labels for unlabeled icons, (2) enhancing automated accessibility testing by considering ATs and how users interact with them, and (3) analyzing dynamic content changes and their impact on screen reader users.

1.1 Dissertation Overview

This dissertation proposes a three-pronged approach to advance testing and repair of accessibility issues in android apps related to screen reader users.

First, it addresses the challenge of labeling icons in mobile apps, a crucial aspect for users relying on screen readers. Current research reveals a significant issue with missing or non-informative icon labels, which impedes app usability for blind users. Building upon existing techniques, this dissertation introduces COALA, a deep learning approach designed to improve automatic label generation for icons. Through an extensive empirical study of 9,658 Android apps, it was found that previous methods, such as LabelDroid, suffered from a data-driven bias, often predicting predefined labels that are not useful for unlabeled icons. COALA overcomes these limitations by incorporating various data sources and context-aware learning, achieving a 24% improvement in label accuracy over its predecessor.

Second, this work introduces automated techniques to detect accessibility issues by analyzing inconsistencies between using the app with and without assistive technologies. This approach overcomes the limitations of rule-based accessibility analysis tools, which often miss accessibility problems that occur when interacting with the app via screen readers. Specifically, this method led to the development of Groundhog, which automatically identifies issues related to locatability and actionability during app exploration with assistive technologies. Additionally, OverSight addresses the inverse problem—identifying elements that are accessible with assistive technologies but not otherwise. User studies confirmed that the issues detected by Groundhog and OverSight were indeed problematic for screen reader users.

Third, it investigates the impact of dynamic visual content on screen reader users, particularly focusing on how dynamic changes can cause accessibility challenges for screen reader users. Traditional accessibility guidelines and tools have largely overlooked this issue. To address this gap, the dissertation introduces TimeStump, an automated framework that detects issues related to dynamic content changes in Android apps. TimeStump uses a crawler to monitor app states before, during, and after actions, identifying problematic dynamic changes that affect screen reader users. By analyzing these changes, TimeStump helps developers recognize and rectify accessibility issues that arise from dynamic content, ensuring a more inclusive user experience.

1.2 Dissertation Structure

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

- 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 (ESEC/FSE), Athens, Greece, August 2021 [112].

- 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), Michigan, USA, October 2022 [143].

- 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), Michigan, USA, October 2022 [111].

- Forough Mehralian, Ziyao He, and Sam Malek, Automated Accessibility Analysis of Dynamic Content Changes on Mobile Apps, To appear at 2025 International Conference on Software Engineering (ICSE), Ottawa, Canada, April 2025.

The following publications are not included in the dissertation but are related:

- Mahan Tafreshipour, Anmol Deshpande, Forough Mehralian, Iftekhar Ahmed, and Sam Malek, MA11y: A Mutation Framework for Web Accessibility Testing, ISSTA 2024, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, September 2024 [151].
- Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek, Automated Construction of Energy Test Oracle for Android, ESEC/FSE 2021, the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Sacramento, CA, November 2020 [89].

Chapter 2

Research Problem

Mobile apps are a vital component of our technology-driven world, and it is essential that they provide equal access to all individuals, including people with visual impairments who utilize screen readers to interact with the app. Unfortunately, numerous studies have shown that accessibility issues persist in mobile apps, hindering their use by screen reader users.

To address this problem, various automated accessibility analysis techniques have been proposed to detect accessibility issues in mobile apps. These tools typically use predefined rules derived from accessibility guidelines to identify potential problems. However, while it is important for developers to follow these guidelines, the rules alone may not fully capture all the accessibility challenges that blind users face.

Therefore, it is necessary to examine accessibility barriers resulting from the inconsistencies between how blind users and sighted users interact with the app. Automated accessibility assessment techniques that consider the GUI context, interaction modes, and dynamic changes can help detect and address these issues effectively, resulting in more accessible apps.

2.1 Problem Statement

Mobile apps have become an integral part of our daily lives, and it is essential to ensure that they are accessible to all, including individuals with visual impairments. However, integrating accessibility into software development is a challenging task, especially in agile app development processes where examining app accessibility with real users is a time-consuming and resource-intensive process.

To assist developers in designing and developing accessible software, various accessibility guidelines have been published [159, 33, 27]. Additionally, automated accessibility analysis tools have been designed to check for violations of these guidelines in apps [25, 17, 31, 34]. However, despite such tools, several apps are still riddled with accessibility issues [12, 136, 45]. These studies also demonstrate wide-spread violation of label-based accessibility concerns such as missing labels, making it difficult for screen reader users to understand app functionalities and interact with them. This necessitates the need for automated label generation techniques to predict labels for image-based buttons.

Although accessibility guidelines and automated tools are helpful in developing and testing accessible apps, they do not cover the various accessibility challenges faced by screen reader users while interacting with the app. Studies have shown that accessibility guidelines only consider about 50.4% of the problems encountered by users [129], as few as 40% of which were covered by automated evaluation tools [157]. These tools cannot identify accessibility issues that manifest themselves when using ATs. Furthermore, app content that changes over time poses accessibility challenges for disabled users, which are not addressed in traditional testing techniques.

Therefore, the challenge is to develop automated accessibility testing techniques that can detect accessibility issues that real users face while navigating through the app and provide effective repair strategies. This problem is summarized as follows:

"Mobile app accessibility is essential to prevent the exclusion of individuals with disabilities from the digital world. Although examining app accessibility with real users is the most reliable evaluation technique, it is often challenging to incorporate into agile app development processes. This has resulted in an increased demand for automated tools that can assist with accessibility testing. While accessibility guidelines and existing accessibility analysis tools have been suggested to aid in designing and testing accessible apps, many apps still have accessibility issues, such as missing textual labels for image buttons. An automated label generation technique is required to alleviate this issue by providing hints to developers during app development or prior to publishing the app. Furthermore, accessibility guidelines do not encompass the various accessibility challenges that actual users face. Automated tools that are based on these guidelines do not take into account how users navigate the app with ATs. Additionally, these tools are unable to evaluate accessibility challenges of app content that changes dynamically over time for screen reader users. Thereby, there is a need for automated accessibility testing techniques that can identify accessibility issues that only arise at real-time, when interacting with the app using ATs."

2.2 Thesis Statement

This dissertation is motivated by two key observations. First, despite the existence of accessibility guidelines and automated accessibility testing tools, there remain many accessibility issues that these tools cannot detect, particularly those that arise during interactions with screen readers at specific points in time. Second, existing accessibility analysis tools can detect issues such as missing labels, but they do not offer automated repair solutions, which limits their adoption or leads to their reported issues being ignored. This research aims to address these limitations as summarized below.

"The goal of this research is to improve the state-of-the-art in software accessibility testing

and repair by addressing the limitations of existing automated tools. Specifically, this research aims to incorporate GUI context in generating high-quality labels for unlabeled icons, and develop an automated tool to address this issue. Additionally, this work seeks to improve accessibility testing by incorporating ATs and their navigation modes, and to develop techniques for detecting accessibility issues that arise due to dynamic content changes for screen reader users. By addressing these gaps in the current literature, this dissertation aims to promote the development of accessible apps for screen reader users."

2.3 Research Hypothesis

This research investigates the following hypotheses:

- Built-in screen readers in mobile devices facilitate interactions with mobile apps by reading the screens out loud for the users. Similar to Alt-Text for web images [172], the embedded textual labels for GUI images are essential for enabling usage of these apps by the blind. These labels are even more critical for *functional* icons [3], which are images that developers utilize to convey the availability of an action, not to convey information. Without a proper description of the functionality initiated by icons, screen-reader users are unable to interact with an app. App accessibility is thus directly affected by the lack of informative icon labels.

  Prior label generation technique [45] is context-agnostic, i.e., it predicts labels merely based on the images. This model is not able to generate informative labels for similar icons in different apps. My study aims to incorporate usage context of icons in predicting an informative label for them.

  Hypothesis 1: Incorporating multiple sources of information from icon usage context can improve the accuracy of generating informative labels for unlabeled icons using deep learning models.

To demonstrate the feasibility of this hypothesis, the first step will be to automatically create a dataset of labeled icons in Android and incorporate information from various sources such as icon location, neighboring element, and screen title. Through data exploration, I will study the characteristics of icon labels and the relationship between words of icon labels and other contextual information from icons. I will then design a deep learning model called COALA to leverage additional sources of information from icons in addition to their graphical representation to determine the probable words in the label.

- ATs are crucial for testing app accessibility, as they provide alternative interaction modes for users. To ensure equal access for all users, developers need to verify that their app behaves similarly with and without ATs. When certain functionalities are inaccessible to AT users or only provided to ATs, this violates the equal access principle, resulting in accessibility issues. The former is referred to as the "under-access" problem, while the latter is the "over-access" problem. Over-access not only degrades app accessibility by confusing AT users but also poses severe security and privacy implications, allowing one to bypass protected functionalities using ATs, such as using VoiceOver to read notes on a locked phone.

  Hypothesis 2: The development of an automated tool that compares the behavior of Android apps with and without ATs can identify discrepancies that lead to under-access and over-access problems.

  To test this hypothesis, two automated tools, GROUNDHOG and OVERSIGHT, will be designed to interact with the app both with and without ATs. While GROUNDHOG examines the accessibility of all functionalities available without ATs, focusing on the under-access problem, OVERSIGHT is designed to detect the dual issue, namely the over-access problem.

- Dynamic content refers to UI elements that are updated as a result of external events and may not always be triggered by user interaction. These changes can pose significant accessibility challenges for visually impaired users who rely on screen readers to access app content. Screen reader users may not be aware of changes occurring in other parts of the app, which can lead to confusion or missed information. For example, time-delayed close buttons for ads may appear a few seconds after the page is loaded, remaining unknown to screen reader users. In this study, I will investigate accessibility issues related to dynamic content changes and develop an automated tool to detect these issues in mobile apps.

  Hypothesis 3: It is possible to devise an automated tool to track dynamic content changes on the GUI and detect accessibility issues for screen reader users.

  To validate this hypothesis, I propose the design of an automated tool, called TimeStump, which will track real-time content changes on the GUI and verify their accessibility to screen readers. The tool detects dynamically changed content that is not announced to screen reader users.

Chapter 3

Context-Aware Label Generation

Mobile apps are playing an increasingly important role in our daily lives, including the lives of approximately 304 million users worldwide that are either completely blind or suffer from some form of visual impairment. These users rely on screen readers to interact with apps. Screen readers, however, cannot describe the image icons that appear on the screen, unless those icons are accompanied with developer-provided textual labels. A prior study of over 5,000 Android apps found that in around 50% of the apps, less than 10% of the icons are labeled. To address this problem, a recent award-winning approach, called *LabelDroid*, employed deep-learning techniques to train a model on a dataset of existing icons with labels to automatically generate labels for visually similar, unlabeled icons. In this work, we empirically study the nature of icon labels in terms of distribution and their dependency on different sources of information. We then assess the effectiveness of LabelDroid in predicting labels for unlabeled icons. We find that icon images are insufficient in representing icon labels, while other sources of information from the icon usage context can enrich images in determining proper tokens for labels. We propose the first context-aware label generation approach, called COALA, that incorporates several sources of information from the icon in generating accurate labels. Our experiments show that although COALA significantly outperforms LabelDroid in both user study and automatic evaluation, further research is needed. We suggest that future studies should be more cautious when basing their approach on automatically extracted labeled data.

3.1 Introduction

There is an increased onus on app developers to make their products accessible for users with a wide range of disabilities, including the approximately 304 million users worldwide that are blind or visually impaired [40]. Blind users rely on screen readers to interact with apps. Built-in screen readers in mobile devices facilitate interactions with mobile apps by reading the screens out loud for the users. Similar to Alt-Text for web images [172], the embedded textual labels for GUI images are essential for enabling usage of these apps by the blind. These labels are even more critical for *functional* icons [3], images that developers utilize to convey the availability of an action, not to convey information. Without a proper description of the functionality initiated by icons, screen-reader users are unable to interact with an app. App accessibility is thus directly affected by the lack of informative icon labels.

Recently, several projects have studied the extent of accessibility issues in mobile apps [136, 13, 45], among others demonstrating widespread violation of label-based accessibility guidelines in Android apps. Missing labels, duplicate labels, and non-informative labels are different types of label-based accessibility concerns, among which missing labels is the most prominent one. Missing label occurs when an icon is not accompanied with a textual label (a.k.a., *content description* in Android) describing its functionality. Ross et al. [136] report that in 50% of their assessed apps, less than 10% of icons were labeled.

The increasing awareness of this accessibility issue has instigated some recent efforts towards alleviating it [63, 185, 45]. Notably, Chen et al. [45] developed a promising method, called

LabelDroid, which employs deep-learning techniques to train a model on a dataset of existing icons with labels to automatically generate labels for visually similar, unlabeled icons.[1]

For data-driven approaches, such as LabelDroid, data exploration is a prerequisite. Perfect model architectures may deliver misleading or unexpected results if the dataset is not carefully examined. Although prior works have empirically studied the severity of the missing labels [136, 13], none have studied (1) the characteristics of natural language labels for the labeled icons extracted from thousands of automatically explored apps in terms of their categories, uniqueness, distribution, and dependency to other icon properties, which can be relevant to the problem of automatic label generation, (2) effectiveness of existing repair approaches in predicting different categories of labels, and (3) impact of incorporating different sources of information in generating icon labels.

To fill this gap, we conducted a large empirical study on icons extracted from 9,658 android apps to understand the characteristics of icons and labels in Android apps. We then assessed the effectiveness of learning and generating natural language labels using this dataset.

Our empirical study reveals that the dataset of automatically extracted icon labels is highly imbalanced, resulting in a severe data-driven bias in the LabelDroid model. It is striking that the introduced bias in LabelDroid is toward predicting *predefined labels* that are shipped with icons in the widgets and templates of Android's Standard Development Kit. Since in practice it is extremely unlikely for these icons to be unlabeled, generating labels for them is pointless. We found that excluding these predefined labels drops the LabelDroid's accuracy by 34%.

Besides, our empirical study shows the necessity but insufficiency of images in representing icon labels. We found that incorporating different information sources for icons can enrich their representation by providing their usage context, substantially improving the identifica-

[1]LabelDroid received the ACM SIGSOFT Distinguished Paper Award at ICSE 2020 [4].

tion of correct tokens in labels.

These findings subsequently informed the development of COALA—a *deep learning (DL)* approach to generate **co**ntext-**a**ware **la**bels for icons in Android apps. COALA automatically extracts high-quality labels for icons from the raw dataset of app screens and layouts, from which it learns how to incorporate different sources of information and translate the image to a textual label. It then utilizes the learned model to generate informative labels for unlabeled icons. Our experiments show that COALA outperforms LabelDroid by 24% in generating labels for unlabeled icons that exactly match the ground truth.

This chapter makes the following contributions:

- An empirical study of the nature of labels and how different sources of information contribute to predicting a correct label;
- An analysis of data imbalanceness, and how it invalidates the results reported in the evaluation of LabelDroid [45];
- COALA, the first context-aware label generation approach to generate textual labels for Android icons and its implementation which is publicly available [142];
- Experimental results corroborating the superiority of COALA in comparison to LabelDroid in generating high-quality labels for icons.

The remainder of this chapter is organized as follows. Section 3.2 provides the background of this study using an illustrative example. Section 3.3 explains the empirical study of natural language labels and other sources of information of icons. Then, Section 3.4 describes COALA in detail, which will be evaluated in Section 3.5 along with the existing deep learning model, LabelDroid. Section 3.6 explains the threats to the validity of the research. The chapter concludes with a discussion of the related research and avenues of future work.

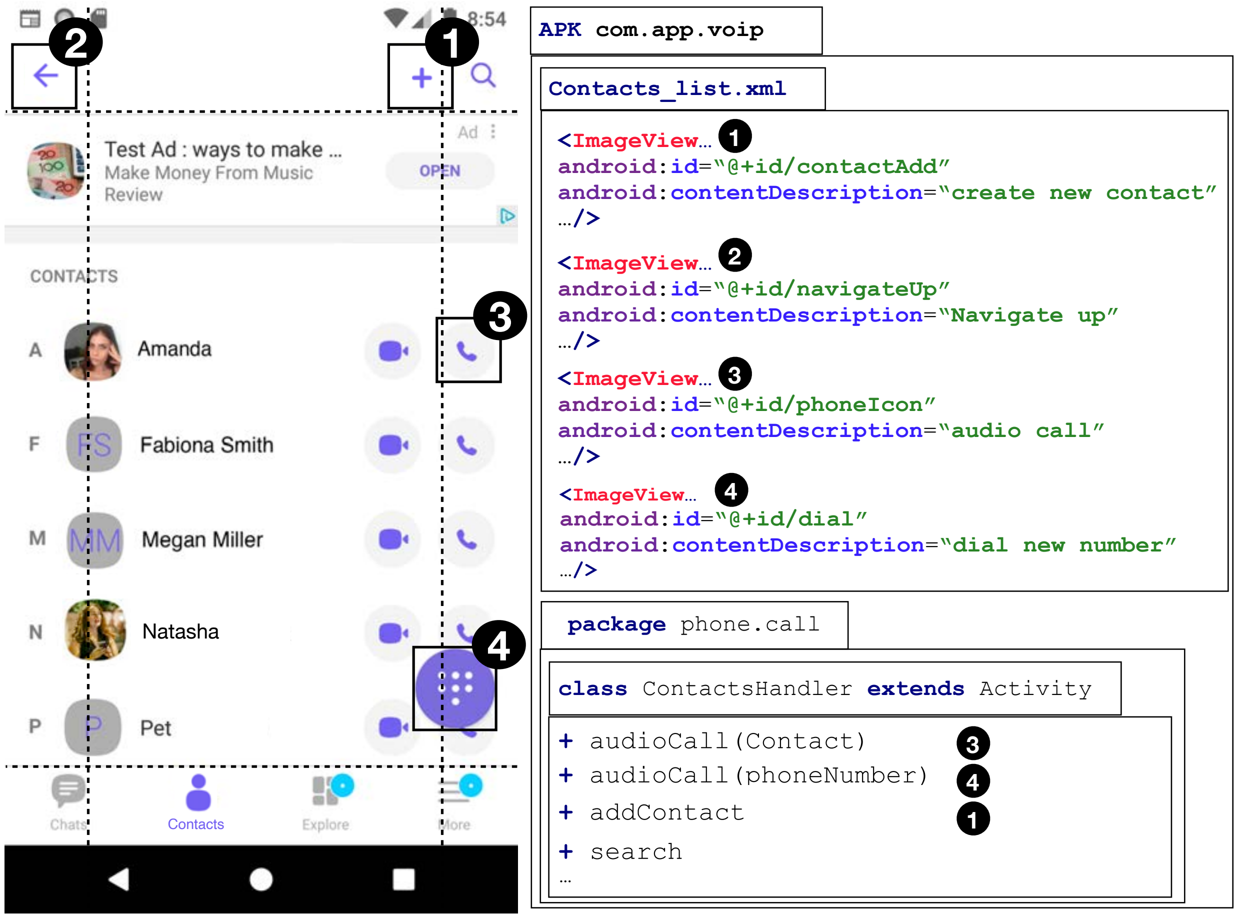
3.2 Background

An Android app's user interface (UI) is implemented in terms of one or more *activity* components, where each activity represents a screen. Figure 3.1 shows an activity for a messenger app along with snippets of its XML *layout* and source code. A *layout* file specifies the placement and design of UI elements in an activity. UI elements such as `ImageView` are objects in the XML tree structure of layout.

Visually impaired users rely on screen readers, like Google's *TalkBack* service, to interact with apps. Screen readers describe images for blind users by announcing the developer-provided textual label in `android:contentDescription` field of UI elements such as `ImageView` and `ImageButton`. Figure 3.1 illustrates four icons in a messenger app, along with their content descriptions in the layout.

Note that the icons comprising Android Standard Development Kit's UI widgets and templates, such as *action bars*, come with *predefined* labels. The textual description of icon number 2 in Figure 3.1, "Navigate up", is an example of such predefined labels.

Prior studies [136] show that missing label (i.e., content description) is a prevalent accessibility issue in Android apps, rendering screen readers inoperable. The plus icon in Figure 3.2 suffers from this issue. TalkBack screen reader describes this button as "unlabeled button", seriously hindering a blind user's ability to use the app. To tackle this issue, Chen et al. proposed LabelDroid [45], which predicts natural language labels for icons given their images using deep-learning techniques. In their work, the authors extracted thousands of labeled icons from 7,594 Android apps. They then trained an image captioning deep-learning model to transform icon images to natural language labels. However, images cannot fully represent icon labels. For example, while the plus icon in Figure 3.1 and Figure 3.2 are visually similar, a proper label for the former, e.g., "create new contact", is different from the label for the latter, i.e., "add a playlist". The distinction between two labels comes from the usage



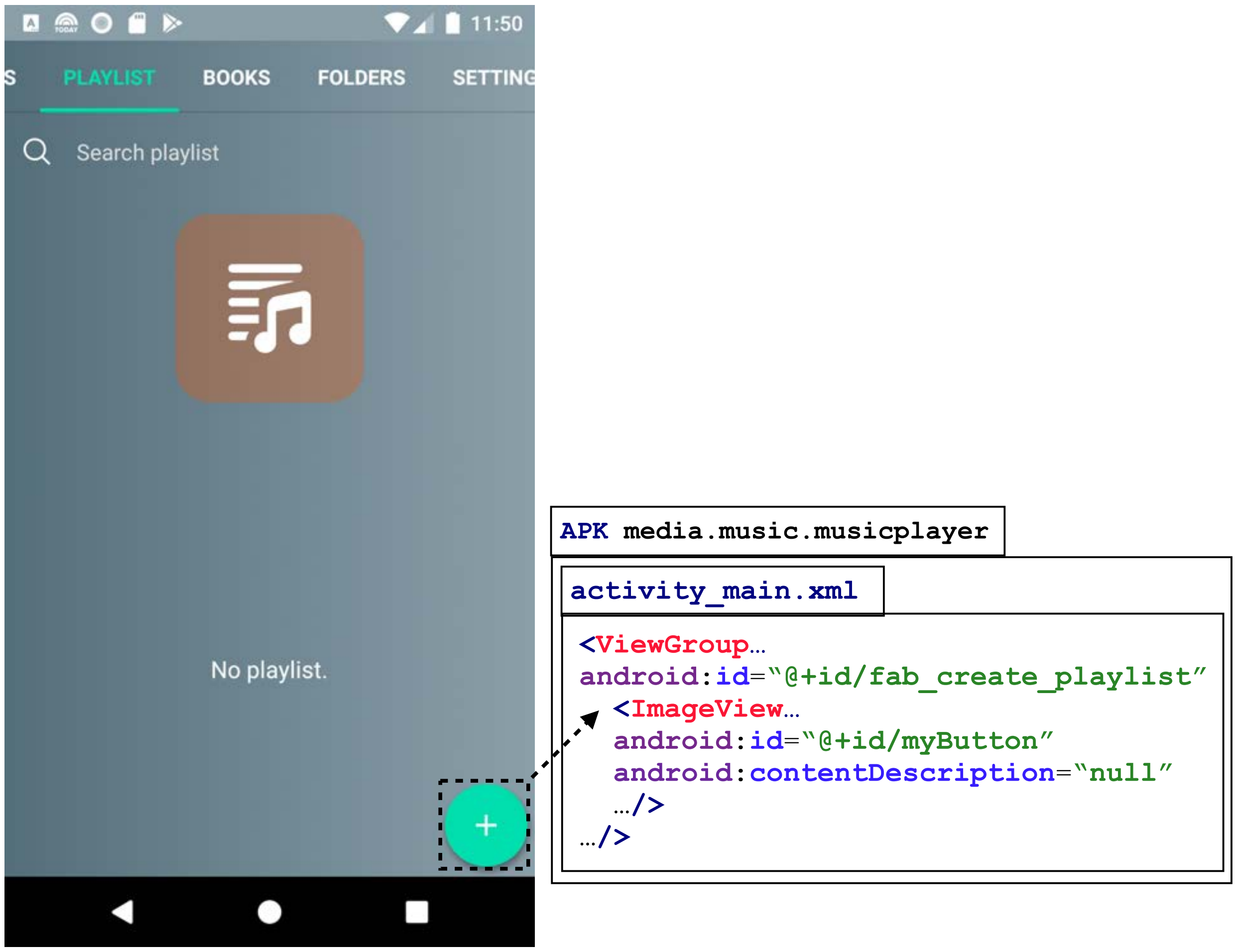
```
APK com.app.voip
Contacts_list.xml
<ImageView… 1
android:id="@+id/contactAdd"
android:contentDescription="create new contact"
…/>
<ImageView… 2
android:id="@+id/navigateUp"
android:contentDescription="Navigate up"
…/>
<ImageView… 3
android:id="@+id/phoneIcon"
android:contentDescription="audio call"
…/>
<ImageView… 4
android:id="@+id/dial"
android:contentDescription="dial new number"
…/>

package phone.call
class ContactsHandler extends Activity
+ audioCall(Contact)          3
+ audioCall(phoneNumber)      4
+ addContact                  1
+ search
…
```

Figure 3.1: Icons and their content descriptions in a messenger app

context of icons. The former is used in the contacts page of a messaging app, while the latter is used in a music player app. Our objective is to understand the nature of natural language labels for icons, study the effectiveness of the prior work, and propose an automated label generation model to alleviate the shortcomings of the prior work and motivate the need for further studies.

3.3 Data Exploration

To develop a better understanding of icon labels and whether different sources of information have predictive impact on them, we conducted an empirical study to answer the following



```
APK media.music.musicplayer
activity_main.xml
<ViewGroup…
android:id="@+id/fab_create_playlist"
  <ImageView…
  android:id="@+id/myButton"
  android:contentDescription="null"
  …/>
…/>
```

Figure 3.2: Plus icon in a music player app

research questions:

RQ1. What are the characteristics of labels regarding their uniqueness and distribution?

RQ2. How similar are the labels of icons with similar images?

RQ3. To what extent different sources of information from icon context can reveal the label?

3.3.1 Experimental Setup

We conduct our study on a set of icons extracted from 15,087 Android apps included in the LabelDroid dataset [45]. This dataset was collected through dynamic GUI exploration of apps. We extracted visible, clickable Android `ImageView` and `ImageButton` icons from XML layouts and screenshots to form our primary dataset for this study. Each icon in our dataset corresponds to a triple of $< image, label, usage\_context >$. For image, we crop the screenshot based on the coordinates of an icon's boundary box as specified in the `bounds` property of the icon considering the orientation of the device. For label, we use the value of `contentDescription` property associated with the icon. For contextual information, we extract several parameters in three levels, i.e., app-level (app category), activity-level (activity name, screen title), icon-level (Android id, screen region, ancestor id, siblings id/text).

The majority of contextual parameters directly map to a property in the XML layout, e.g., `id` and `text`. To find the parent and siblings of an icon, we refer to the hierarchical structure of XML layouts. Icons that share a parent node in the XML tree are considered to be siblings. For the screen title, we refer to the `text` property of the top, leftmost `TextView` element in the layout. For the screen region, we refer to the `bounds` property of an icon to determine in which of the 9 screen regions, as specified with dashed lines in Figure 3.1, it belongs. Activity names and package names are available in GUI exploration artifacts. We then use package names to extract app categories from Google Play using BeautifulSoup [133] crawler.

To improve data integrity, we performed text normalization steps on textual information of icons. Labels and textual parameters extracted using the above-mentioned techniques are not restricted to follow a standard or commonly accepted structure. Developers may use *camelCase* [175] or *snake_case* [176] conventions, or even other types of characters as delimiters. We transformed the textual information to lower case, replaced all the special characters with a space and applied spell correction and lemmatization to their tokens. We also filtered *meaningless labels* as introduced in the prior work [45].

In our dataset, icons with the same label in the same app would be counted once. Thus, our dataset consists of 21,864 icons extracted from 17,839 different screens of 9,658 apps.

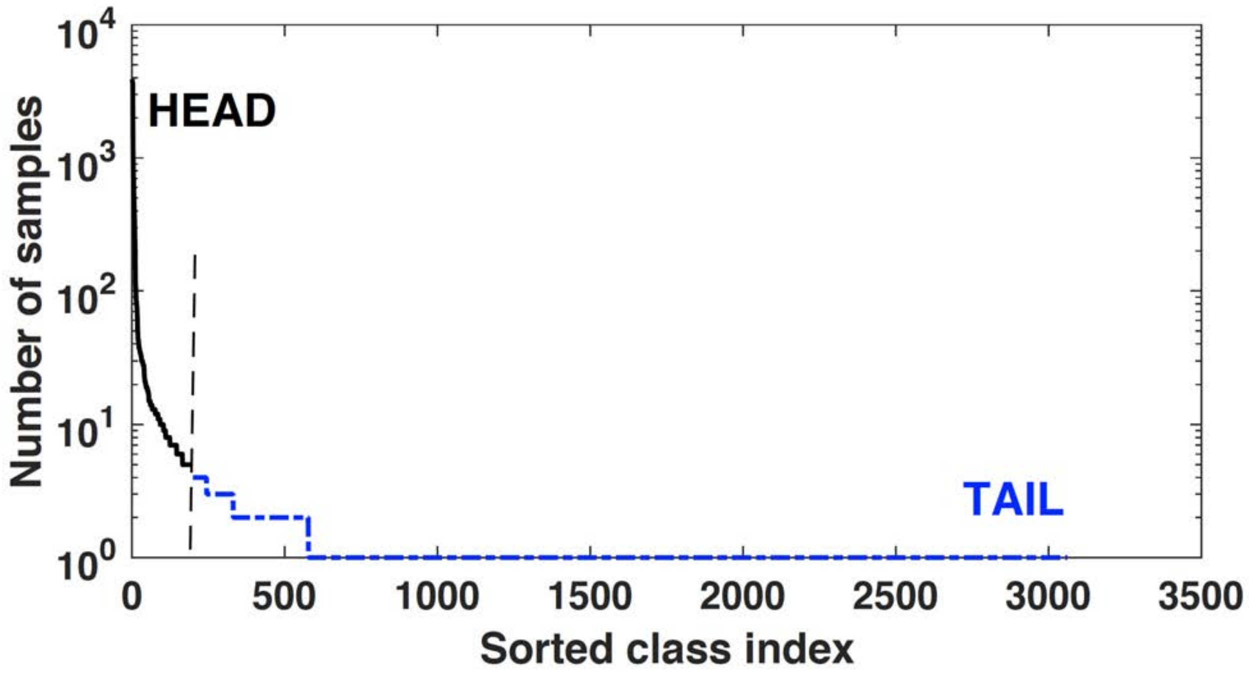


Figure 3.3: Imbalanced distribution of labels for icons. To the left are the few dominant classes, and to the right is the long tail. The cutoff separates the labels with more than 5 samples.

3.3.2 RQ1. Characteristics of Labels

For this research question, we first study the distribution of icon labels. In our dataset, we found 3,061 different labels with high-class imbalance. By considering each of these 3,061 labels as a class for icons, we observed a *long-tailed* dataset as shown in Figure 3.3, in which 51.57% of the data comes only from 3 most frequent classes, while 93.5% of the classes have less than 5 samples in the whole dataset and 2,484 out of 3,061 occurred only once. The average number of samples per class is 7.14 and the median is 1. The gap between mean and median also indicates a left-skewed data distribution.

By further exploration of dominant classes of labels, we found that some Android icons come with a predefined label. For example, if a developer uses an up button in the action bar, similar to icon 2 in Figure 3.1, it comes with “Navigate up” label. We manually extracted predefined labels for all icons in Android Studio, the most widely used IDE for Android development. This analysis produced the following labels: {*"navigate up"*, *"more options"*, *"next month"*, *"previous month"*, *"open navigation drawer"*, *"close navigation drawer"*, *"search"*, *"clear query"*, *"interstitial close button"*}. If we exclude these labels, the average number of samples per class drops by 63.79% and changes to 2.58.

This observation alerts us to the erroneous conclusions we may draw from the evaluation of a label prediction approach. To facilitate the explanation of the issue, imagine 90% of the data has label X. In that case, a model that just predicts X is already 90% accurate and its effectiveness may not be interpreted realistically. This issue would be exacerbated if our goal was to label unlabeled icons, but the model is only good at predicting dominant labels that happen to be the aforementioned predefined labels.

In addition to dominant classes, it is also important to pay attention to low-frequency labels. The long tail of label distribution demonstrates the labels for which the dataset may not be representative enough. When we exclude predefined labels, 31.47% of the remaining labels have only one sample in the dataset. That means, a proper label for them cannot be simply retrieved from the previously seen data, challenging a deep learning model for prediction of labels.

Observation 1: The data is highly skewed towards a limited set of labels, threatening the validity of the evaluation of a label prediction model. Furthermore, low-frequency and unique labels in the long tail challenge the construction of an effective learning-based prediction model.

When we studied the distribution of tokens in the whole dataset, we observed the same long-tailed distribution. That is, the tokens of predefined labels are the dominant classes. We further studied the tokens for unique labels to determine to what extent the tokens of unique labels can be derived from previously seen tokens. We found that for 53.99% of unique labels, *all* of the tokens were observed in the previously seen tokens, and for 86.5% of the unique labels, at least one non-trivial token was previously seen. By trivial tokens, we mean stop words such as “for”, “the”, “to”, etc. Thus, a token-based label prediction model may hold promise in correctly generating low-frequency and unique labels.

Observation 2: Substantial portion of tokens comprising the unique labels can be found in the existing vocabulary of tokens, suggesting a token-based prediction model may be effective in correctly generating low-frequency and unique labels.

3.3.3 RQ2. Labels of Visually Similar Icons

In order to study labels of icons with similar images, we trained an image classifier and annotated the icons with their image class. Liu et al.[106] identified 99 common image classes shared across apps through manual open coding of Android icons. They then trained a Convolutional Neural Network (CNN) to classify icon images. Their model is 94% accurate on average in predicting class of icon images. Rico dataset [58] provides the output of this image classifier for icons existing in their dataset of Android screens. We augmented our dataset with the class of icons to study the diversity of labels of visually similar icons.

On average, each class contains 225.4 icons with 20.03 different labels. For example, among 178 icons in class “add” (the plus icon), 61 different labels exist. In terms of tokens, the average number of unique tokens for labels in each class of icons is 28.6 with the median of 16. For instance, there are 61 unique tokens comprising the labels of icons in class “add”.

Observation 3: While image similarity restricts the set of probable labels and tokens for icons, there is still substantial level of diversity among labels and tokens for visually similar icons.

3.3.4 RQ3. Labels and Icon Information

We study the relationship between tokens of icon labels and other contextual information from the icon. We extract the contextual information in three levels: (1) *App level* that contains the categories of apps, (2) *Activity level* including screen title and activity name, and (3) *Icon level* that contains the identifier name of the icon itself, its parents, and its siblings along with the region that the icon is located and the texts of its siblings.

To measure the dependency between two random variables, we calculate their *Mutual Information (MI)* [123]. It quantifies the amount of information obtained about one random variable through observing another random variable. In the context of our problem, MI determines which parameter, C, has the highest likelihood of predicting correct tokens in the label, T. $MI(C,T)$ is defined as $H(T) - H(T|C)$, where H is the *entropy*, representing the uncertainty in a random variable. Although MI tells us how important the parameters are in predicting tokens, it does not tell us whether the contextual information is a predictor of presence or absence of tokens in icon labels. For that, we calculate the Pearson correlation coefficient, $\rho(T,C)$, to see how changes in the icon information result in *predictable* changes in the tokens.

Table 3.1 summarizes the result of this experiment. As shown in Table 3.1, we observe different degrees of correlation between the various sources of information and the tokens. The identifier name of the icon and its parent have the highest correlation with tokens in the labels. While app category does not appear to associate with tokens. Apart from app category, activity-level parameters have least correlations.

Table 3.1: Correlation, ρ, and Mutual Information, MI, between icon information, C, and tokens in labels, T.

Information Sources		$MI(T, C)$	$\rho(T, C)$
App level	Category	0.0908	0.1469
Activity level	Screen title	0.3474	0.0953
	Activity name	0.3924	0.1808
Icon level	Android id	0.7337	0.4221
	Screen region	0.2840	0.3187
	Siblings id	0.5673	0.3266
	Siblings text	0.4519	0.2814
	Parent id	0.5699	0.426

Observation 4: Different information sources exhibit different degrees of effectiveness in empowering a probabilistic model in predicting tokens for icon labels.

3.3.5 Summary

Our findings in data exploration motivated us to conduct further studies on the models and develop COALA, the first context-aware label generation approach.

First, the empirical study showed that the dataset of icons and labels in Android is highly imbalanced towards predefined labels. Thus, we will study the impact of learning on such data on the model fairness.

Second, the empirical study showed that while a significant subset of labels in the dataset are unique, substantial portion of tokens comprising these unique labels can be found in the existing vocabulary of tokens. To overcome the challenge posed by unique and low-frequency labels for a DL approach, we devise a learning model to generates labels in terms of their constituent tokens.

Third, the empirical study showed that while there exists a wide variety of labels for visually similar icons, there are other sources of information available for improving the representation power of a probabilistic model. Our approach, in turn, leverages additional sources of information of an icon in addition to its graphical representation to determine the probable tokens in its label. This is akin to the intuition that sighted users can easily distinguish the functionality of similar icons by virtue of their knowledge of each icon's usage context.

In the following section, we describe the architecture of COALA. We then study further research questions related to the fairness and effectiveness of DL models in generating textual labels for icons in Section 3.5.

3.4 COALA

In the following section, we introduce `COALA`—a *deep learning (DL)* approach to generate **co**ntext-**a**ware **la**bels for icons in Android apps. Figure 3.4 provides an overview of `COALA`, which consists of two main modules: *Data Pre-Processing* and *DL Architecture* .

Data Pre-Processing module in `COALA` is responsible for extracting a dataset of labeled icons along with their information in their usage context. After finding icon specifications from thousands of XML layouts, it filters improper and duplicate icons similar to the prior work [45] and creates a dataset of icons for the DL module.

The DL Architecture is responsible for encoding the icons to be later decoded to textual labels. The encoding step has two phases: *Image Encoder* and *Context Encoder*, each of which is tailored to compute the embedding of a specific type of data. These representations are then fused in a *Fully Connected Layer* to prepare a vector, from which *Label Decoder* generates the corresponding textual label.

In the remainder of this section, we describe the details of each module in our DL Architecture as illustrated in Figure 3.4.

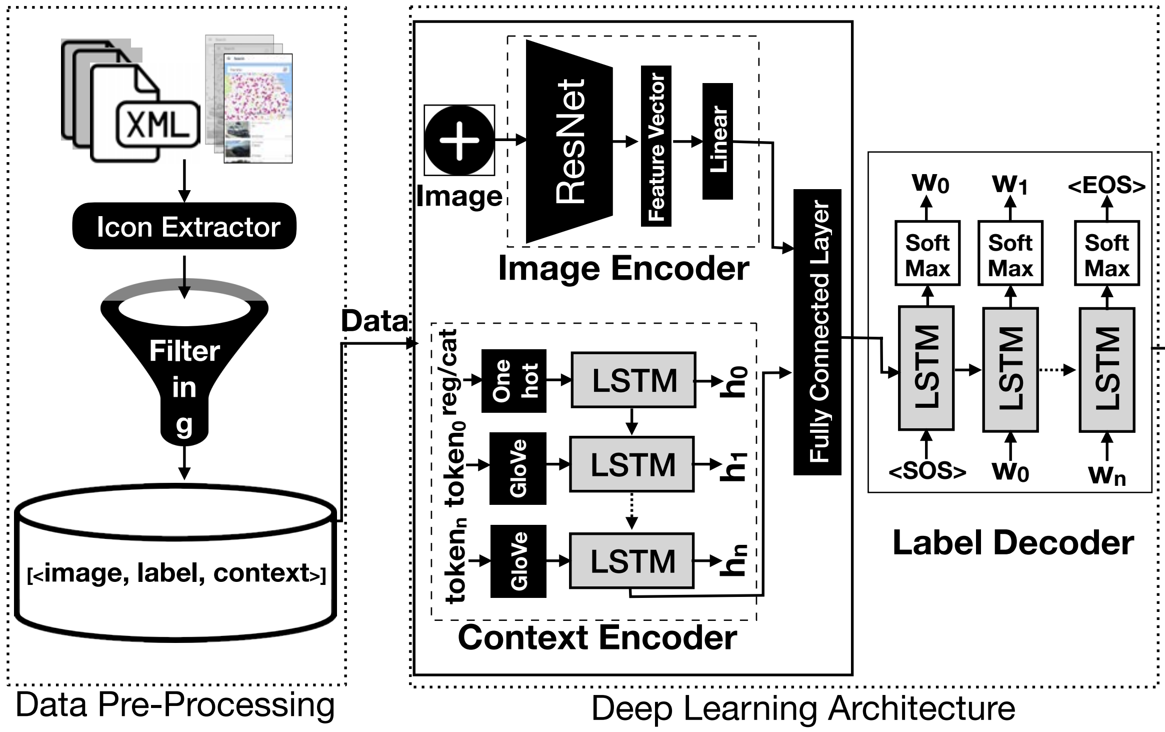


Figure 3.4: Overview of COALA framework

3.4.1 Image Encoder Module

Embedding visual data into low-dimensional space has been extensively studied using the *Convolutional Neural Networks (CNN)* [100]. This type of network receives input images as a matrix of their pixel values and extracts a feature vector of the input image through various convolutional and pooling layers.

COALA encodes images in the same fashion. However, instead of training a CNN model from scratch, it utilizes the *transfer learning* technique. Transfer learning allows us to leverage pre-trained CNN models such as ResNet [85], without the need for dealing with technical challenges of training a model from scratch on a big enough set of training data for sufficient amount of time. That means, we leverage the knowledge of a pre-trained ResNet model and re-purpose it for icon image classification, a.k.a., fine-tuning.

To that end, we prepared a dataset of icon images with their annotated classes from Rico dataset [58]. Using a tool in prior work [106], Rico augmented more than 66,000 screenshots with semantic annotations which consist of icon classes. We extracted annotated icons in Rico dataset and manually checked the consistency of images with their annotated class. We also downsampled dominant classes of icon images to have a balanced dataset. We trained and fine-tuned a pre-trained ResNet18 classifier [85] using this data and used it in COALA for encoding icon images.

3.4.2 Context Encoder Module

The input of Context Encoder is the sources of information from usage context of the icon as shown in Table 3.1. The purpose of this module is to embed these parameters into a feature vector. In choosing a proper model for Context Encoder, we should consider three main characteristics of parameters. First, these parameters have two different types: categorical and textual. Second, textual parameters have a variable length. Third, often only a subset of parameters are available for a given icon.

To support both categorical and textual parameters, Context Encoder utilizes two different input embedding components, namely: one hot encoder [81] and a word embedding model, specifically *GloVe* [127].

One hot encoder maps the categorical parameters, i.e., *category* and *screen region*, to a binary vector. *Category* can take either one of the 53 categories that exist in Google Play or none if it was removed from Google Play by the time we checked. *Screen region* is one of the 9 zones, as shown in Figure 3.1, that an icon can belong to. Thus, the encoded vector should be at least 63 bits in length. However, it is zero-padded to have the same length as the vector representation of other parameters.

For textual parameters, similar to [119], Context Encoder first summarizes all the parameters into one sentence by joining the textual phrases with a dot, ".", as a delimiter. For example, for the plus icon in Figure 3.2, joining the page title, i.e., "playlist", and its cleaned android id, i.e., "create playlist", results in the summary of "playlist.create playlist." for textual parameters of the icon (we filtered "fab" token since it specifies the icon type, i.e., floating action button, and is not informative). Then, a pre-trained GloVe model is responsible for mapping each token in the summarized sentence to its vector representation. GloVe is an unsupervised pre-trained model that has proven its ability in capturing syntax and semantic regularities using vector arithmetic [127]. This is mainly because (1) it is not needed for the model to learn the exact vocabulary of these tokens, and (2) using a semantic preserving word embedding enables the model to better generalize to unseen tokens whose synonyms exist in the dataset.

Given the vector representation of icon information, Context Encoder utilizes a *Recurrent Neural Network (RNN)* [86]. RNNs are known for their chain-like structure, which makes them capable of learning from variable-length sequences of data. Specifically, we use a type of RNNs, *Long Short-Term Memory (LSTM)* networks, shown to be superior to the standard RNNs by avoiding the long-term dependency problem caused by Vanishing Gradients [38].

LSTMs consist of a chain of repeating modules, a.k.a., *LSTM cells*. The vector representation of icon information passes through the LSTM cells in which four neural network layers interact in a special way. Several adjustable weights control the information each cell remembers, forgets, or passes to the next cell (a.k.a., hidden state). During training, the model tunes these internal weights towards decreasing the overall training loss.

The output of the last LSTM cell is fused in a fully connected layer with the image embedding to provide the input for the Label Decoder module.

3.4.3 Label Decoder Module

Given the image and other information of an icon, the Label Decoder is responsible for generating natural language labels. Icon labels are variable-length sequences of tokens. Thus, a proper model should be able to generate accurate tokens of labels sequentially. To that end, `COALA` employs an LSTM network with an internal loop that lets it iteratively generate icon labels token by token.

At each time step, the Label Decoder chooses the most probable token from a vocabulary of tokens. `COALA` builds this vocabulary based on frequent tokens in the labels of the training set. This vocabulary also includes SOS (Start of Sequence), EOS (End of Sequence), and UNK (Unknown). Then, each label will be represented by a sequence of ids of its comprising tokens surrounded by SOS and EOS tokens. UNK stands for tokens of labels that are not in the vocabulary.

Label Decoder initializes the hidden state of the first LSTM cell with the encoder output. Similar to the LSTM network of Context Encoder module, four internal neural network layers with adjustable weights regulate the information flow through the network. Label Decoder also sends an SOS token to the first LSTM cell to signal the start of the label generation process. Next the LSTM cells get the previously generated token as input and calculate the score of choosing each token in the vocabulary. A Softmax [131] layer gets the LSTM cell output to transform the scores to a probability function. The most probable token is the final output at each step. Label Decoder terminates this procedure when EOS token is the output of current time step.

To train DL model, Label Decoder calculates cross-entropy loss function to measure the extent to which the predicted probability diverges from the actual token. `COALA` trains the whole DL architecture end-to-end. Thereby, this loss function back propagates through the whole network, i.e., encoder and decoder, to adjust the internal weights.

During training, the aforementioned process of passing the last generated token to the next LSTM cell can be followed. However, this process results in model instability and slow convergence [70]. To alleviate these issues, `COALA` uses *Teacher Forcing* strategy. Teacher forcing is a training-time procedure in which the model receives the ground truth output y_t as input at time step $t+1$ [178]. This means Label Decoder passes the t_{th} token of the target label as the input to the $t+1_{\text{th}}$ LSTM cell during training. However, in the testing phase, we pass the previously generated tokens to the next LSTM cell and we only use the ground truth labels to calculate the performance of the model.

3.5 DL Model Assessment

To study the impact of imbalanced training data on the state-of-the-art model, LabelDroid, and also the effectiveness of our context-aware approach, we study the following research questions:

RQ4. How effective is LabelDroid in practice?

RQ5. To what extent is the DL architecture of LabelDroid capable of coping with imbalanced data?

RQ6. How effective is the context-aware model of COALA in label prediction? To what extent COALA outperforms the context-agnostic model of LabelDroid?

RQ7. To what extent the labels provide an informative explanation of the icon functionality?

RQ8. How long does it take for COALA to train and predict labels?

Table 3.2: Details of COALA dataset

	App	Activity	Icon
Train	7,728	14,230	17,462
Test	965	1,821	2,274
Validation	965	1,788	2,128
Total	9,658	17,839	21,864

3.5.1 Experimental Setup

Datasets

We split the dataset of icons introduced in Section 3.3.1 into three separate sets with respect to apps. In this way, train, validation, and test set are respectively 80%, 10%, and 10% of apps selected randomly. Table 3.2 shows the details of our dataset. Note that we run our experiments 5 times using different random partitioning of the data to minimize evaluation bias. This means in a different random partitioning, the number of icons and screens may be slightly different in each partition since we split based on apps.

Moreover, as we aim to study the effects of imbalanced data on LabelDroid, inspired by prior work [55], we created balanced datasets by downsampling dominant classes of data. A parametric Sigmoid function on the inverse label frequency manages the balanceness of the dataset. Sigmoid parameter adjusts the number of frequent labels included in the sampled data by controlling the steepness of its curve. We experimented with 5 different parameters of Sigmoid, resulting in balanced datasets of size {2,557, 3,931, 7,599, 11,601, 15,495}. The smallest dataset is completely balanced with one instance for each label.

DL Implementation and Configurations

Our DL model is implemented in PyTorch [126], a popular open-source Machine Learning library for Python. We utilized Adam optimizer [95] to update the internal weights iteratively based on the cross-entropy loss function. To prevent the predefined labels from overwhelming the network during training and producing a biased model, we adapted weighted cross-entropy [105] to enforce the model learn from the labels in minority. Each DL model has several configurable parameters, a.k.a., hyperparameters, that can impact the performance of the model. To tune these hyperparameters of the model, we also performed a guided grid search strategy to choose the values that had the best performance on the validation data. The details of our configurations are available on COALA's website [142].

Evaluation Metrics

We evaluate the effectiveness of DL models using 4 evaluation metrics that are commonly used for image captioning problems, namely: BLEU [124], METEOR [36], ROUGH [103], CIDEr [154]. We also report *exact match*, i.e., the percentage of data for which the generated label is an exact match of the ground truth. That means, it only awards the model if *all* the tokens of the ground truth appear in the generated label with the same order. BLEU score, however, focuses on n-gram overlaps to measure the quality of the generated label. It calculates precision for n-grams, denoted by BLEU-1, BLEU-2, and BLEU-3, for n in {1, 2, 3}.

BLEU score has some drawbacks, for example in not considering sentence structure or word meanings, which has led to the advent of other evaluation metrics. METEOR (Metric for Evaluation of Translation with Explicit ORdering) is based on the harmonic mean of unigram precision and recall [36]. ROUGH is a set of recall-oriented measures, from which we use ROUGH_L that is based on the Longest Common Subsequence in the ground truth and the generated label [103]. CIDEr (Consensus-based Image Description Evaluation) leverages term frequency-inverse document frequency (tf-idf) to measure the similarity of the ground truth and the generated label [154].

The implementation of these metrics is available in a Python library, called NLGEval library [147], which we have used to evaluate COALA.

3.5.2 RQ4. LabelDroid's Effectiveness

Our empirical study (Section 3.3) revealed a severely imbalanced label distribution for icons. We conducted an experiment to study whether overlooking this data-driven bias leads to misinterpretation of prior work's effectiveness. For that purpose, we trained LabelDroid [45] on their dataset using their default configurations and evaluated the model on generating (1) *predefined* labels, which as introduced in Section 3.3.2 correspond to the default catalog of icons in widgets that come with Android Studio, and (2) *non-predefined* labels, i.e., all the icons in the test set except the ones with predefined labels. Among 1,876 icons in their test set, 866 of them have predefined labels and the remaining 1,010 icons have non-predefined labels. Note that for this research question, we use the original dataset of LabelDroid to only study the impact of data balanceness and keep their approach as close as possible to their original version. For next questions, we use the dataset introduced in Section 3.5.1, which is the extended version of LabelDroid's dataset since the dataset they used in their work lacks additional sources of information from icons.

Table 3.3 summarizes the results of this experiment. As shown in Table 3.3, the effectiveness of LabelDroid, in all metrics, is significantly higher in generating predefined labels than the non-predefined labels. The unfortunate outcome is that this variation impacts the overall result: resulting in an incorrect interpretation of the model's effectiveness in predicting proper labels for unlabeled icons.

For a better illustration of the impact of imbalanced data on the overall effectiveness of LabelDroid, consider the stacked bar chart of Figure 3.5. Here, each bar indicates the ratio of correctly predicted labels according to a different metric. Within each bar, the solid

Table 3.3: LabelDroid's effectiveness in generating predefined/non-predefined labels in their test set.

	Exact_match	BLEU-1	BLEU-2	BLEU-3	METEOR	ROUGE_L	CIDEr
Predefined	0.90	0.90	0.90	0.82	0.69	0.91	4.54
Non-predefined	0.17	0.24	0.18	0.18	0.12	0.27	0.89
All	0.51	0.55	0.53	0.41	0.32	0.55	2.58

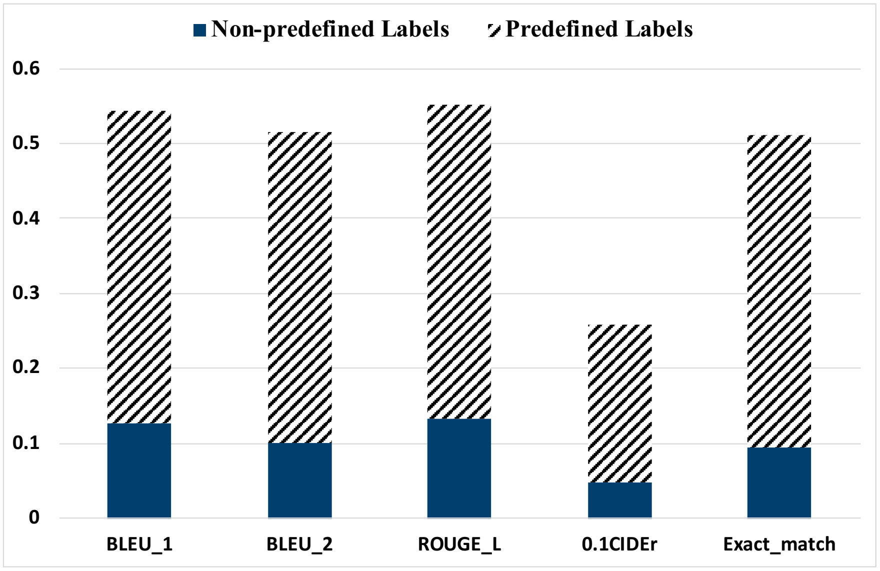


Figure 3.5: Biased effectiveness of LabelDroid towards predefined labels

blue fill indicates the ratio of correctly predicted non-predefined labels, while the dashed fill indicates the ratio of correctly predicted predefined labels. Figure 3.5 clearly shows the overall evaluation is highly impacted by the model's effectiveness on predefined labels.

This observation indicates that the imbalanced data produced a biased model for predicting predefined labels. But the issue is substantially more severe than it may appear at first blush, since there is no point in predicting icons with predefined labels. After all, these are the labels of icons that are shipped with the popular Android Studio. In practice, it is a rare occurrence for Android Studio icons to appear in apps with no labels. This would only occur if the developer intentionally removes the label automatically associated with the icon by the IDE. The ultimate goal of label prediction is to generate labels for unlabeled icons, which often have non-predefined labels.

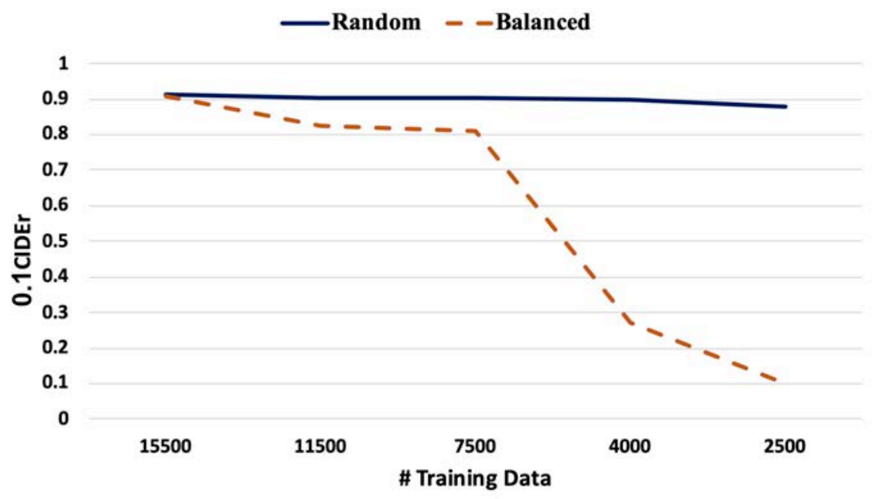


Figure 3.6: Overall evaluation of LabelDroid model trained on re-sampled data

3.5.3 RQ5. Impact of Balanced Data on LabelDroid's Effectiveness

To further evaluate LabelDroid, we also studied its performance if it were to be trained on balanced data in predicting non-predefined labels. To that end, we performed balance sampling with regard to distinct labels of the training data to downsample dominant labels. Then, we trained a new model on the newly sampled training set and evaluated the model on the test set.

Figure 3.6 depicts the results of training LabelDroid on balanced data using CIDEr metric. The orange dashed line corresponds to the effectiveness of the model trained on balanced data. We should note that in balanced sampling, as well as changing the data distribution, we are reducing the amount of training data, which affects the model's ability to learn. To monitor this variable, we also performed random downsampling on the training data to make the dataset have the same number of training data as in our balanced datasets. In figure 3.6, the blue solid line depicts the performance of the model trained on randomly sampled data. Figure 3.6 shows that having balanced training data does not improve LabelDroid's effectiveness. The general trend in the blue solid line shows that reducing the

Table 3.4: Comparison of COALA and LabelDroid effectiveness in generating non-predefined labels

model	Exact_match	BLEU-1	BLEU-2	BLEU-3	METEOR	ROUGH_L	CIDEr
COALA	0.38	0.4	0.2	0.15	0.21	0.489	1.34
LabelDroid	0.14	0.21	0.125	0.09	0.12	0.24	0.7

training data slightly degrades the model's effectiveness. However, increasing the data balanceness drastically drops the model's effectiveness. The same behavior was observed in the model's effectiveness in generating non-predefined labels and when we used other evaluation metrics. However, to comply with the page limits, those results are available on COALA's website [142].

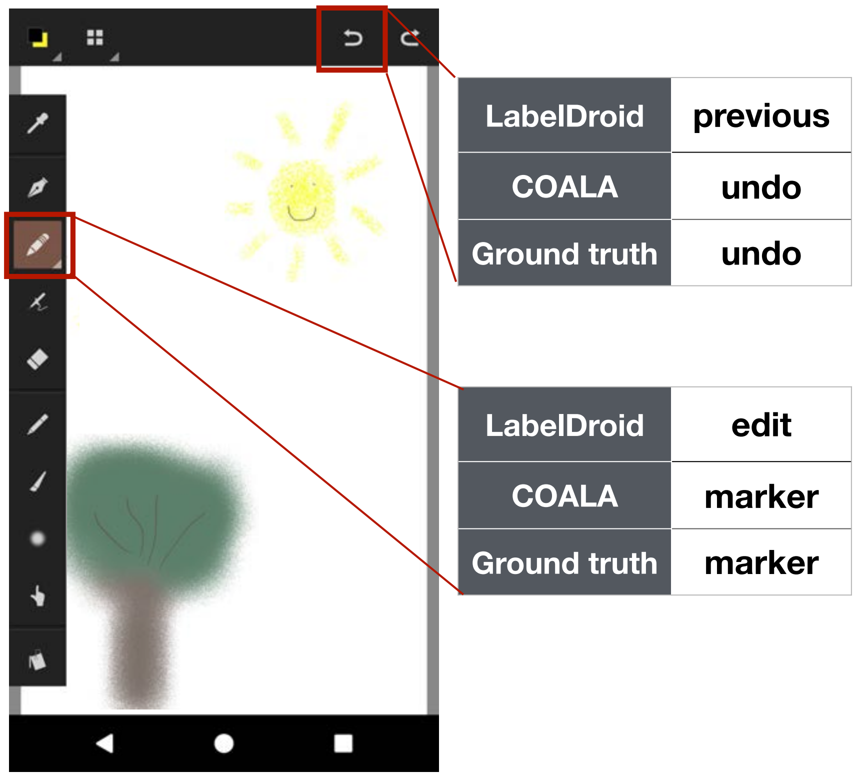
This concludes that learning from balanced data did not provide any remarkable improvement in the effectiveness of LabelDroid in generating non-predefined labels.

3.5.4 RQ6. Effectiveness of coala

The ultimate goal of COALA is to generate labels for unlabeled icons. Thus, our main objective is to evaluate COALA's effectiveness in generating non-predefined labels and compare it with the prior work [45].

Table 3.4 summarizes the effectiveness of COALA and LabelDroid in generating non-predefined labels and shows the superiority of COALA over LabelDroid in all metrics. Figure 3.7 illustrates two icons in a painting app in our test set for which COALA was able to generate correct labels, but LabelDroid failed.

In this app, there are different icons for different painting tools such as a marker in a vertical, left toolbar, as well as other essential icons in the horizontal, top toolbar. It is clear that without considering the context of these icons, generating the correct label may be impossible. For example, the Marker icon in the vertical toolbar has been widely used



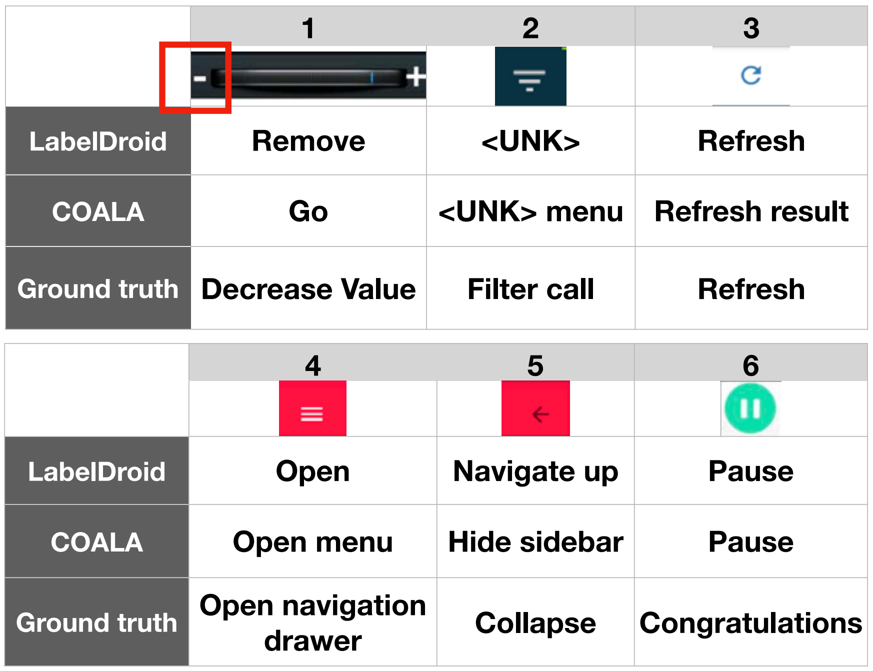
LabelDroid	previous
COALA	undo
Ground truth	undo

LabelDroid	edit
COALA	marker
Ground truth	marker

Figure 3.7: Examples of inability of the context-agnostic model, LabelDroid, in generating correct labels.

to denote the "Edit" icon. However, by considering the adjacent icons, COALA was able to detect the icon as a painting tool. For the Undo icon, in addition to commonly co-occurred icons, existing hints in their textual information, such as Android *id*, enabled the model to generate accurate labels. More examples of LabelDroid's failures, for which COALA was able to generate correct labels are available on our website [142].

We also examined common failure scenarios for COALA. As shown in Figure 3.8, COALA has generated incorrect labels, particularly for icons 1 and 2. Due to the black-box nature of DL models, pinpointing the exact reason for these failures is not possible. However, examining the icons gives us an overview of probable culprits. In the first snapshot, the contextual information for minus icon is not informative. Thereby, COALA was unable to recognize its functionality. Furthermore, the existence of the token "wheel" in the identifier name of its sibling may have confused the model to generate a wrong label. There are also some failures



	1	2	3
LabelDroid	Remove	<UNK>	Refresh
COALA	Go	<UNK> menu	Refresh result
Ground truth	Decrease Value	Filter call	Refresh

	4	5	6
LabelDroid	Open	Navigate up	Pause
COALA	Open menu	Hide sidebar	Pause
Ground truth	Open navigation drawer	Collapse	Congratulations

Figure 3.8: Examples of failures in label generation using COALA and LabelDroid

in generating labels for infrequent icons, e.g., icon 2. Not having sufficient training data to help the model learn the icon is a probable cause of this failure.

Besides these model failures, there are certain cases that are not mistakes, yet penalize the effectiveness of our model in the manner evaluated here. Case in point, consider icon 3 in Figure 3.8, where COALA has generated more valid tokens, possibly conveying more useful information to a blind user than the ground truth. Additionally, the generated labels may be semantically valid alternatives, as in icons 4 and 5 in Figure 3.8. There are also some cases, such as icon 6, in which the ground truth is invalid, not the generated label. These examples indicate that in practice COALA is more effective than the NLP metrics suggest, since they are simply comparing the generated labels against the ground truth. These metrics do not account for situations in which COALA either generates a semantically equivalent label as the ground truth label, or the ground truth itself is wrong. This observation motivated us to conduct a user study to better evaluate COALA.

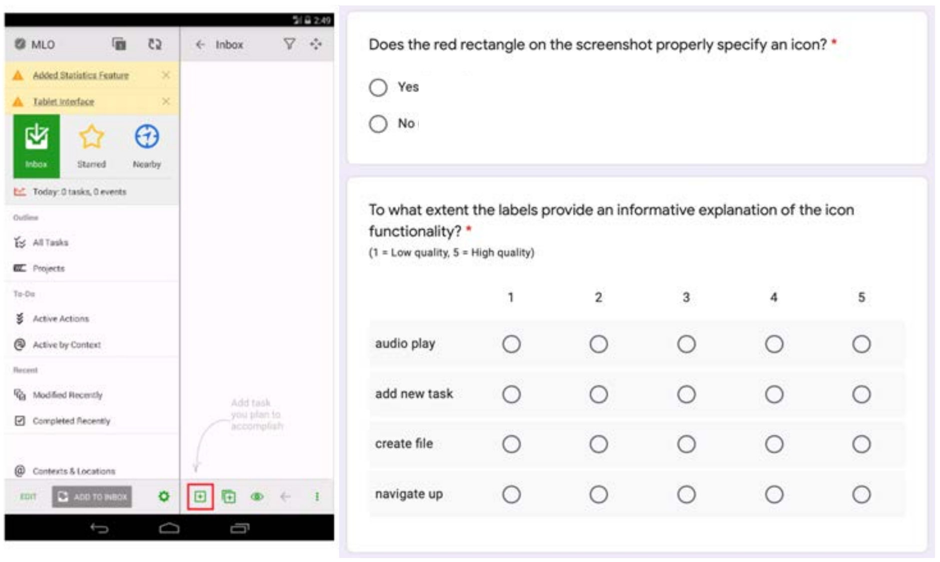
3.5.5 RQ7. Informative Explanation for Users

In addition to the automated evaluation of models using metrics described in Section 3.5.1, we conducted a user study to understand the quality of automatically generated labels by LabelDroid and COALA. Since manual investigation of all icons in our test set was not practical, we randomly selected a sample from the test data based on the image classes of icons. This ensures having a representative and balanced dataset of icons with different images (e.g., plus, backward-arrow, etc.). We used our icon image classifier introduced in Section 3.4.1 to get image classes and randomly selected up to 10 icons from each class, resulting in 198 icons from 61 distinct image classes. Next, we highlighted the icon under investigation on the screenshot of the app as shown in Figure 3.9. For each icon, we show four types of labels (1) *ground truth*, which is the content description extracted from the XML layout, (2) the generated label by LabelDroid, (3) the generated label by COALA, and (4) a *random* label, different from the ground truth, selected from all labels in the dataset. We then used Google form to display the icons on the screenshots, as well as four different labels to the users and get their responses. We shared the survey on social media and asked volunteers to rate the quality of the labels from 1 to 5 after reading an instruction (the instruction is available at [142]). To avoid bias, we collected and aggregated responses of at least three different users for each icon, resulting in 730 answers in total.

Prior to analysis, we filtered out the unreliable data as follows:

- **Incomprehensible icons.** The highlighted area on a screenshot may not specify a proper icon due to capturing the screenshot of an app at a transition point during app exploration. Alternatively, the user may not be able to determine the functionality of a designated icon from the screenshot. We remove such images from our analysis by asking users if the icon on the screenshot is valid and understandable.

- **Inconsistent scores.** We expect users to assign the same score to the same labels of



Does the red rectangle on the screenshot properly specify an icon? *

○ Yes

○ No

To what extent the labels provide an informative explanation of the icon functionality? *
(1 = Low quality, 5 = High quality)

	1	2	3	4	5
audio play	○	○	○	○	○
add new task	○	○	○	○	○
create file	○	○	○	○	○
navigate up	○	○	○	○	○

Figure 3.9: A sample question from the user study.

an icon; otherwise, there is inconsistency in scores. For example, the ground truth and COALA's candidates for an icon can both be "collapse", and users should assign the same score to both of them. We remove all scores of users who have inconsistent scores since such users are not reliable.

- **Insufficient rating.** We have a threshold of three responses for each icon. Thus, if applying the prior filtering steps drops the number of responses for an icon below three, we remove the icon entirely.

Our filtering criteria removed 216 responses, resulting in 514 responses for 156 icons. We then aggregated the users' responses by calculating the average of all the scores for each icon. Therefore, for each type of labels in {ground truth, COALA, LabelDroid, and Random} there is a list of 156 scores corresponding to each icon, which we call *score list*. The average of all score lists of ground truth, COALA, LabelDroid, and Random are 3.91, 3.15, 2.83, and 1.13, respectively.

To determine if the differences observed between the means of score lists are statistically significant, we performed hypothesis testing. Since the scores failed the Shapiro-Wilk nor-

Table 3.5: Statistical analysis of scores. Given the significance level of 0.05, the scores of COALA's labels are significantly better than the scores of LabelDroid's labels. $\bar{\mu}_{Diff}$ is the average of difference between score lists.

H0 (Null Hypothesis)	$\bar{\mu}_{Diff}$	p-value
Ground truth - COALA = 0	0.76	5.14e-6
Ground truth - LabelDroid = 0	1.08	8.68e-10
COALA- LabelDroid = 0	0.33	1.7e-2
Ground truth - Random = 0	2.78	2.94e-25
COALA- Random = 0	2.02	7.49e-20
LabelDroid - Random = 0	1.67	6.43e-16

mality test [138], we performed non-parametric testing using Wilcoxon signed-rank test [177] with significance level of 0.05. Table 3.5 shows the result of this analysis. As seen on the upper half of the table, the quality of ground truth labels is better than the quality of labels generated by COALA and LabelDroid, since the mean of ground truth's score list is significantly better than the mean of COALA's and LabelDroid's score list (p-value equal to 5.14e-6 and 8.68e-10 respectively). Similarly, it shows that the quality of labels generated by COALA is significantly higher than the quality of labels generated by LabelDroid (p-value=1.7e-2). Moreover, the lower half of the table shows that labels of ground truth, COALA, and LabelDroid have higher quality than random labels.

An outcome of this analysis is that although COALA significantly improves the state-of-the-art technique in generating natural language labels for icons, it is still not as good as the labels provided by actual developers. This observation suggests that there is still room for improvement and further research in this area.

3.5.6 RQ8. Performance

To answer this research question, we evaluated the time required to train a new model and used the resulting model to generate a label for an icon. We ran the experiments on a Ubuntu computing cluster using an NVIDIA GP102 GPU and 128G memory. It took 241 minutes on average for COALA to train a new model on our dataset. This time includes evaluating the model at each time step on the validation set for model selection purposes. However, it took only 17 milliseconds on average for the trained model to generate the label of an icon given its specification. This indicates that COALA is efficient for use in a variety of settings, including automated repair of inaccessible apps, and inclusion in screen readers to dynamically resolve unlabeled icons.

3.6 Threats to Validity

Sampling bias: The selection of Android apps in this study may introduce bias. We mitigated this threat by exploring another dataset, RICO [58]. We obtained similar results as that reported here. The results of our study on RICO dataset are available online [142]. Moreover, both LabelDroid and RICO datasets consist of more than 20 thousands apps selected from various categories of Google Play store.

Learner bias: For the empirical study on DL models, we use two architectures, COALA and LabelDroid. One possible threat to the validity of our results is the choice of the neural-network modules and hyper parameters of our models. For COALA, our focus was studying the impact of incorporating different sources of information using a well-known architecture. Also, for training LabelDroid, we used their original implementation and hyper-parameters.

Evaluation bias: We evaluated LabelDroid and COALA under the same evaluation metrics used in LabelDroid's publication and consistent with natural language processing literature. We report the effectiveness of models on a test set, left out from the whole dataset of labeled icons. However, the generalizability of models on unlabeled icons needs to be studied further. This signifies the need for creating high quality benchmarks of icons in Android apps in future. We further reduce the evaluation bias by running our experiments 5 times and averaging the results.

3.7 Related Work

Accessibility issues have been extensively studied for websites [91, 82] and more recently for mobile apps either in specific categories such as e-government [146], smart cities [117], and health [170] or in general [125, 136, 57, 181, 155, 13, 139]. The increased awareness of the prominence of accessibility issues has motivated the development of several accessibility guidelines, and accessibility assessment and repair tools.

Accessibility guidelines: The World Wide Web Consortium (W3C) [159] is the main organization in determining protocols and standards for websites whose primary initiative is to develop accessibility standards. They have provided detailed tutorials for constructing inclusive web pages [158]. For mobile apps, Google and Apple, the primary organizations facilitating the app marketplace, have published accessibility guidelines for Android and iOS developers [21, 32]. Despite the existence of these guidelines, according to [13], developers are still not aware of the issues or find it costly to address them.

Android accessibility evaluation and repair tools: Accessibility evaluation tools leverage static analysis and/or dynamic analysis techniques to report various accessibility issues. Lint [23] is a static tool that checks project files and warns the developers about missing labels. Espresso [22], Robolectric[134] and Accessibility Scanner [17] are based on Accessibility Testing Framework of Android [2], with the capability to dynamically scan the app for accessibility issues [20]. PUMA [83], MATE [63], and IBM Mobile Accessibility Checker (MAC) [181] are other dynamic testing frameworks that check accessibility issues at runtime.

Despite several tools for accessibility assessment, only a few repair tools are available to fix accessibility issues for blind users. To enable runtime accessibility repair and enhancement for Android, Zhang et al. [184] proposed interaction proxies to layer on top of the original implementation of app. In their subsequent work [185], they utilize this platform for social annotation of GUI elements for missing labels. Different from their work, COALA is capable of automatically generating labels for icons by learning from the previously labeled ones.

Furthermore, Liu et al. [106] utilize a deep learning classifier to semantically annotate icon images based on around 100 categories they defined for the icons. Although their initiative was not fixing accessibility issues, screen readers can take advantage of their proposed textual annotations for unlabeled icons. Unlike their work, COALA generates textual labels not only from icon images but also from their usage context. In this work, we leverage from their annotated icons to fine-tune an image classifier which is capable of embedding icon images.

The most relevant work to our study is LabelDroid [45], which is a context-agnostic model for generating labels for icons. As discussed heavily throughout the paper, LabelDroid's insufficient representation of Android icons only by their images, as well as its biased model negatively affect its effectiveness.

3.8 Conclusion and Future Work

Missing labels seriously hinder blind users' ability to interact with mobile apps. In this work, we studied the characteristics of icon labels and demonstrated how overlooking the imbalanced nature of labels result in a biased deep-learning model. We also presented COALA, a context-aware label generation approach for icons in Android. Our experimental results show that by incorporating additional sources of information, COALA could outperform the prior work [45] in automatically generating labels for unlabeled icons.

In future, we will explore incorporating additional sources of information from source code to enrich icon representation and improve the accuracy of our model by studying different DL models. We also aim to integrate our model with (1) IDE analysis tools, such as Lint, to not only detect missing labels, but to also recommend fixes, and (2) screen readers to facilitate blind users' interactions with apps.

Our research artifacts are available to the public [142].

Chapter 4

AT-Aware Accessibility Testing

Accessibility is a critical software quality affecting more than 15% of the world's population with some form of disabilities. Modern mobile platforms, i.e., iOS and Android, provide guidelines and testing tools for developers to assess the accessibility of their apps. The main focus of the testing tools is on examining a particular screen's compliance with some predefined rules derived from accessibility guidelines. Unfortunately, these tools cannot detect accessibility issues that manifest themselves in interactions with apps using assistive services, e.g., screen readers. A few recent studies have proposed assistive-service driven testing; however, they require manually constructed inputs from developers to evaluate a specific screen or presume availability of UI test cases. In this work, we propose an automated accessibility crawler for mobile apps, Groundhog, that explores an app with the purpose of finding accessibility issues without any manual effort from developers. Groundhog assesses the functionality of UI elements in an app with and without assistive services and pinpoints accessibility issues with an intuitive video of how to replicate them. Our experiments show Groundhog is highly effective in detecting accessibility barriers that existing techniques cannot discover. Powered by Groundhog, we conducted an empirical study on a large set of real-world apps and found new classes of critical accessibility issues that should be the focus of future work in this area.

4.1 Introduction

The ever-growing reliance of people on mobile apps to perform daily tasks necessitates app accessibility for all, notably for more than 15% of the population who have disabilities [174]. 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 [45, 135, 12].

To assist developers in enhancing app accessibility, technology institutes such as World Wide Web Consortium [163] and companies such as Apple [33] and Google [27] have published accessibility guidelines and best practices. These guidelines are backed by accessibility analysis tools to automatically analyze app compliance with guidelines and detect accessibility issues [25, 17, 31, 34]. For instance, by analyzing User Interface (UI) elements, they can report whether the contrast between elements and their backgrounds are above a certain threshold or the area of a button is smaller than a specific area defined in the guidelines.

Unfortunately, guidelines cannot detect about %50 of the accessibility issues that users with disabilities may encounter while interacting with apps [130]. 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. For example, users with visual impairment rely on screen readers, i.e., TalkBack in Android, to navigate through UI elements and perform actions on them. TalkBack users click on a desired element by double tap gesture. When this gesture entails no change in the app state, the element is not actionable by TalkBack and can render the app inaccessible.

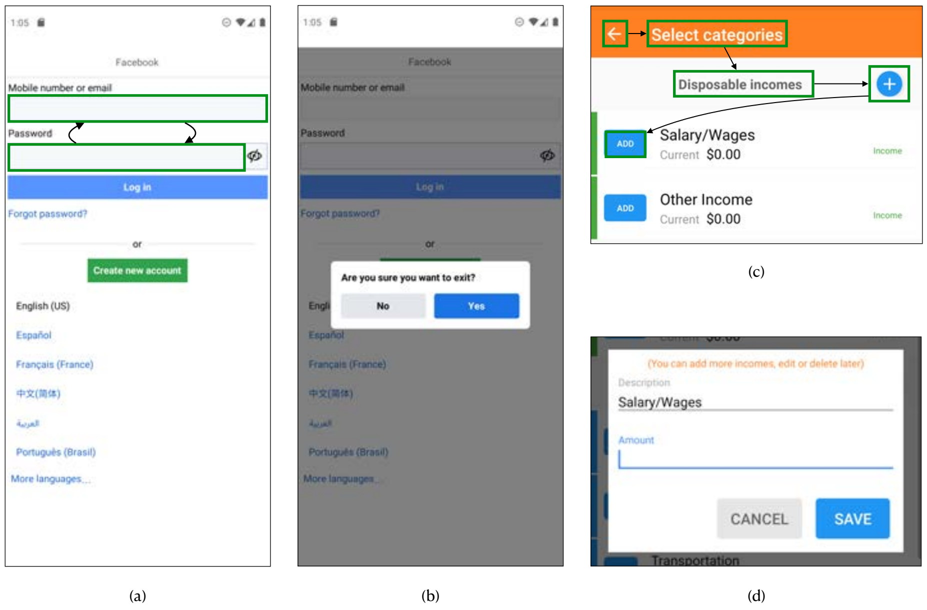


Figure 4.1: (a) The login activity of Facebook app, (b) The exit dialog appears when users press back button on Facebook app, (c) a screen in BudgetPlanner app, the highlighted boxes and arrows depicts the directional navigation to the "ADD" button by TalkBack, (d) a dialog appears after tapping "ADD" button

The great majority of prior automated accessibility testing techniques do not take assistive services into account in their analysis. Salehnamadi, et. al [139] incorporate assistive services in evaluating the feasibility of executing GUI test cases. Their work, however, 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 [104]. 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. The work by Alotaibi, et. al [11] also utilizes TalkBack to find inaccessible elements in navigating sequentially through all the elements on the screen without GUI tests. This work is limited to analyzing a single screen that developers should provide manually.

Moreover, it cannot detect other types of reachability issues that may occur while exploring the app with TalkBack by touch or with other assistive services. Their work also does not consider the different ways of performing actions with and without assistive services, potentially resulting in unactionable elements for assistive services

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.

This chapter makes the following contributions:

[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.

- A novel, high-fidelity, and fully automated form of automated accessibility analysis that evaluates the accessibility of mobile apps from the perspective of users with various forms of disability.
- A publicly available implementation of the above-mentioned approach for Android, called Groundhog [144];
- An empirical evaluation on a large set of real-world Android apps, showing the effectiveness of Groundhog in detecting new accessibility issues in popular apps (even with more than 1 billions installs on Google Play), and
- A qualitative study of the different types of accessibility issues that can be detected by Groundhog, which can aid future researchers with developing automated means of fixing these specific kinds of issues.

The rest of this chapter is organized as follows: Section 4.2 motivates this study with an example. Section 4.3 provides a background on accessibility testing and Android fundamentals. Section 4.4 explains the details of our approach, and Section 4.5 describes the optimizations over our technique. The evaluation of Groundhog on real-world apps is finally presented in Section 4.7. The chapter concludes with a discussion of the related research and avenues of future work.

4.2 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 4.1(a) shows the login screen of the Facebook app with more than 1 billion installs on Google Play [76]. 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. For example, a blind user utilizes TalkBack [72], the standard screen reader in Android, to perceive the screen content by listening to what TalkBack announces for each element on the screen. A TalkBack user can navigate through the elements sequentially by swipe (*Directional Exploration*), or focus on a specific element by touch (*Touch Exploration*). Using either of these exploration strategies on the app screen illustrated in Figure 4.1(a), TalkBack can only detect the two text boxes, annotated in green in Figure 4.1(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 4.1(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 [17] 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 4.1(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 [23, 22] are capable of detecting navigational issues in Android apps.

Assistive services also enable users to perform actions on elements. When there are no locatability issues, Assistive services such as TalkBack can focus on the desired element. In the case of TalkBack, double-tap gesture triggers the "Click" action on the focused element. Unfortunately, actions performed under different interaction models may have inconsistent behaviors. Figure 4.1(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 4.1(d). For the same action, a TalkBack user, first locates the “ADD” button, either by touch exploration (tapping on the location of the button) or directional exploration (swiping right until the target element is focused, as shown by arrows in Figure 4.1(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 4.1(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.

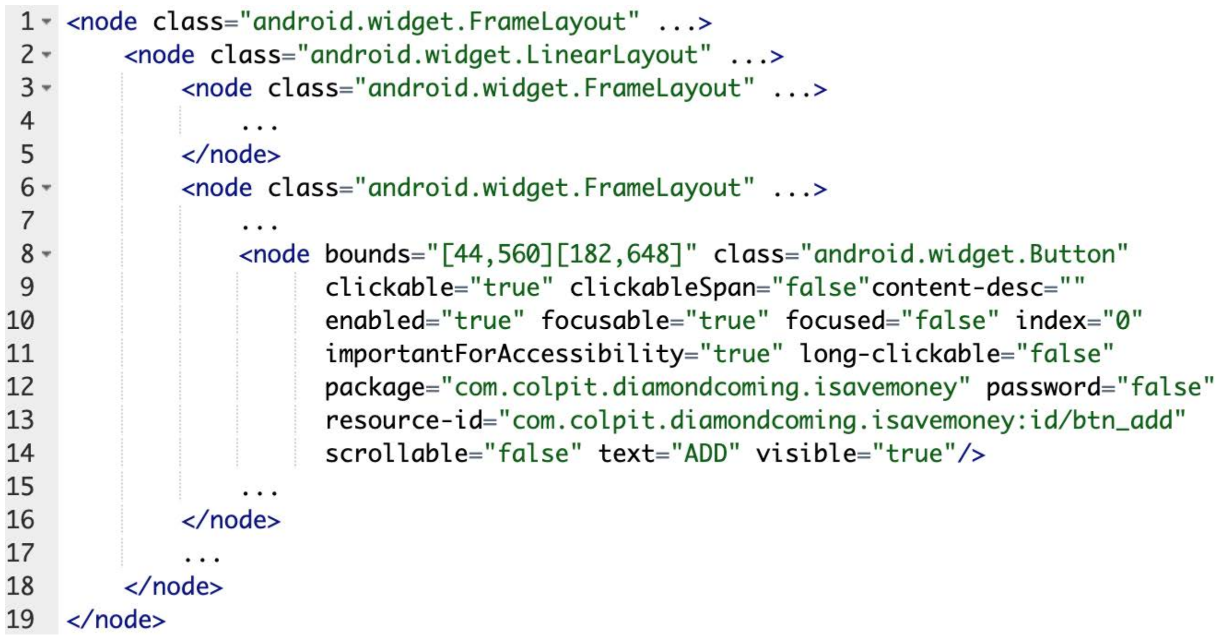
4.3 Background

We provide a brief overview of User Interface (UI) components and accessibility support in Android to help the reader understand the material that is presented later.

4.3.1 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. This section provides background on UI components and GUI testing in Android.

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 check



```
1  <node class="android.widget.FrameLayout" ...>
2      <node class="android.widget.LinearLayout" ...>
3          <node class="android.widget.FrameLayout" ...>
4              ...
5          </node>
6          <node class="android.widget.FrameLayout" ...>
7              ...
8              <node bounds="[44,560][182,648]" class="android.widget.Button"
9                    clickable="true" clickableSpan="false"content-desc=""
10                   enabled="true" focusable="true" focused="false" index="0"
11                   importantForAccessibility="true" long-clickable="false"
12                   package="com.colpit.diamondcoming.isavemoney" password="false"
13                   resource-id="com.colpit.diamondcoming.isavemoney:id/btn_add"
14                   scrollable="false" text="ADD" visible="true"/>
15             ...
16         </node>
17         ...
18     </node>
19 </node>
```

Figure 4.2: A part of the excerpted XML representation of UI structure in the Budget Tracker app shown in Figure 4.1(c).

boxes. The non-leaf nodes, on the other hand, are invisible to user. These non-leaf nodes are called *ViewGroups* or *Layouts* and 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 provide description for widgets without textual representation or *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 4.2 shows part of the UI structure in the Budget Tracker app. Lines 8-14 represents the first "ADD" button in Figure 4.1(c) where its attributes such as clickable or text are represented.

XPath [1] (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 the structural information. For example, the first "ADD" button in Figure 4.1(a) 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 ommitted from classes). Since the class of an android widgets cannot be changed at runtime, the absolute path in

XPath, or in short *apath*, is a reliable identifier of widgets in Android.

4.3.2 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 built on top of the accessibility API. We briefly describe accessibility API in Android and how an assistive service like TalkBack 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, are also implementations of this abstract service [18]. *AccessibilityService* works as a wrapper around an Android device interacting with it (performing actions on and receiving feedback from it).

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 device and manage 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 "ADD" button (Figure 4.1(c)), 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 4.2.

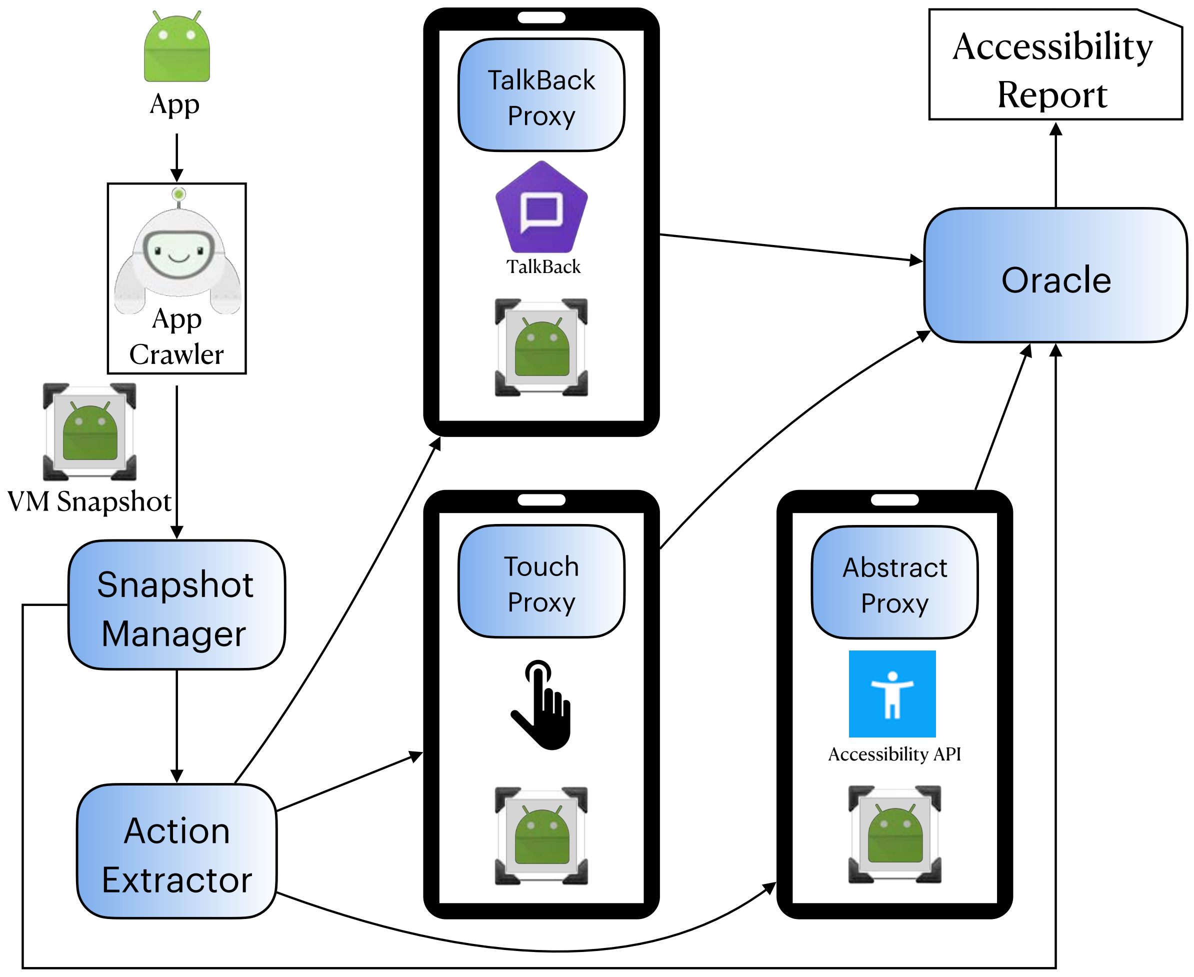


Figure 4.3: An overview of GROUNDHOG

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.

4.4 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 4.3 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 proxies: Touch Proxy, TalkBack Proxy, and Abstract Proxy. 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 successfully performs the action and produces the final report.

In this section, we describe each component of GROUNDHOG in detail.

4.4.1 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 [77], Stoat [149], Ape [80], Sapienz [108], 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 [60, 51, 52].

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

4.4.2 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, similar to Figure 4.2. *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 4.1(c), the corresponding *apath* is */Framelayout/LinearLayout/FrameLayout[2]/Button*, as shown in lines 8-14 in Figure 4.2. Also, the operation of this action can be determined from the "clickable" attribute (line 9 in Figure 4.2). 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'
}
```

4.4.3 Proxies

Proxies represent various interaction models with a device. The goal of each Proxy is to execute a given action on a given app state and return the app state after performing the action along with the execution logs. To that end, each Proxy utilizes Android's *AccessibilityService* to support two main functionalities: (1) locating the element specified in action and (2) performing the intended action on the element. In this study, we consider three different models: Touch, TalkBack, and an Abstract assistive service with all the capabilities of accessibility API. The details of each Proxy are as follows.

Touch Proxy

This Proxy interacts with the system from the standpoint of users without disabilities. These users do not use any assistive service and see the elements that are depicted on the screen to locate them. Touch Proxy first determines the coordinates of the bounding box of the element on the screen to locate the element. It then sends a tap gesture event to the element to simulate the touch-based interaction model.

To locate an element, Touch Proxy searches for the corresponding node of the target element identified by its *apath* in the UI hierarchy. It starts from the root node of the screen and follows the address specified in the apath in a depth-first traversal order of the UI tree. If at a level, no branch matches the apath, it means the node is not *locatable*.

To perform an action on the located element, Proxy calculates a tap point considering the bounding box of the element. To that end, it calculates the coordinate of the center of the element using its bounding box coordinates. For example, the coordinate for the tap action on the "ADD" button (113, 604) can be calculated from line 8 in Figure 4.2. Once the coordinate of the target element is determined, Touch Proxy performs the click operation by sending a tap gesture event for that coordinate.

TalkBack Proxy

This Proxy utilizes TalkBack to interact with the device. TalkBack supports two UI exploration modes: Directional Exploration (by swiping) and Touch Exploration (touching different screen parts). Similarly, TalkBack Proxy leverage both exploration modes. When we enable TalkBack, it focuses on the first node on the screen. Swipe right (left) changes the focus on the next (previous) element on the screen. The Proxy first employs directional exploration to locate an element, i.e., iteratively draws swipe right gestures using the Accessibility API to navigate to the desired element. The Proxy terminates navigation if it focuses on the desired node or visits an element twice. The latter case indicates either there is a navigation loop or all existing elements have been visited once. When this process fails in locating the element, there is a *locatability* issue in using directional exploration. For example, a revolving list of elements can cause a navigation loop for a TalkBack user, preventing the user from reaching the elements residing afterward. To alleviate this problem, in practice, disabled users transition to explore by touch mode to focus on a random element outside of the loop and resume directional exploration forward or backward from there. This Proxy, similarly, tries to use touch exploration.

If the element is not found in Directional Exploration, TalkBack Proxy tries Touch Exploration mode by touching on the coordinates of the target element. If the element cannot be focused, TalkBack Proxy reports a violation of *Locatable* rule for the element. Once the element is located (focused), TalkBack Proxy uses Accessibility API to perform the intended operation., e.g., perform a double tap on the screen to click on the focused element. If the target element cannot be focused by Directional Exploration or Touch Exploration modes, a *locatability* failure is reported using TalkBack.

Abstract Proxy

As mentioned in Section 4.3, all assistive services in Android are built on top of the Accessibility API. To evaluate the app accessibility, given all the capabilities of Accessibility API, we introduce Abstract Proxy. Accessibility issues revealed for Absract Proxy exist for all other assistive services (e.g., SwitchAccess [16] for users with motor impairment) since they use Accessibility API to locate elements and perform actions on them.

For locating an element, Abstract Proxy locates the elements by their *apath* similar to what was explained for Touch Proxy. Then, it sends the event corresponding to the action, e.g., *ACTION_CLICK* to the located node using Accessibility API.

All the proxies return the next state of the app along with the execution logs to the Oracle component to be further analyzed. The execution logs contain all the triggered events, the action specification, node information, and failure reports.

4.4.4 Oracle

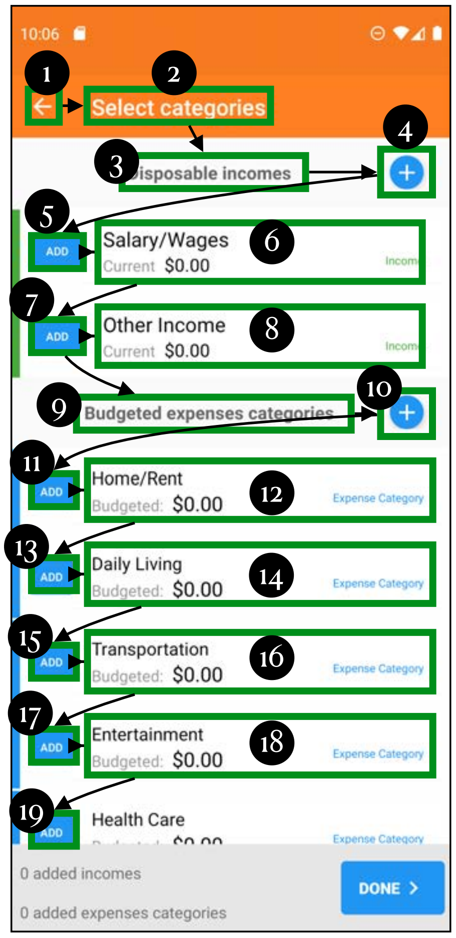
The Oracle component is responsible for analyzing each app state and corresponding execution logs to determine if an accessibility issue exists in executing an action with a proxy.

For *locatability* issue, Oracle refers to failure reports of proxies to check if the Proxy 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 proxies to check if there exists at least one Proxy 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.

4.5 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 proxies. For example, Figure 4.4 depicts the process of locating two elements (a)



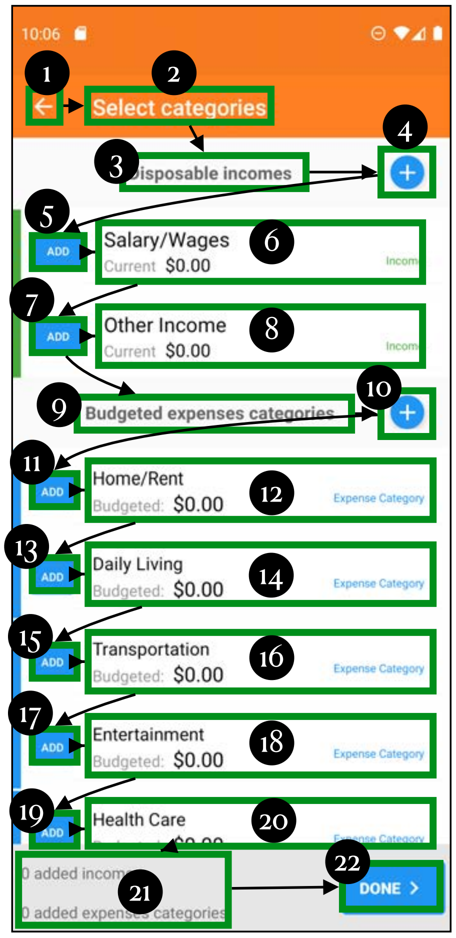


Figure 4.4: Locating (a) the last "ADD" button, and (b) the "Done" button with TalkBack Proxy in directional navigation. 18 directional navigation interactions in (b) are redundant since they have been performed in (a) already.

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 4.4, 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 Directional Exploration strategy 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 directional navigation from the start to the element e is bypassed, allowing the exploration to proceed much faster.

4.6 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 proxies, creating reports, and visualizing the results. The client, implemented in Java, is basically an accessibility service, i.e., proxies, that controls the device to execute actions.

Groundhog utilizes Android Debug Bridge (ADB) [26] to manage communications between the server and client. Groundhog also modifies Stoat app crawler [149] and employs it to explore different states of the app. As discussed in Section 4.4, 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 [53] Python imaging library and Flask [122], 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.

4.7 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?

4.7.1 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 [28] (labeled as **P**). Second, 20 randomly selected apps from AndroZoo [8], 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 [139] (labeled as **L**).[2] Latte is a related prior tool, discussed in Section 5.1, to which we compare against. Out of the 17 apps from the Latte dataset included in our study, 11 have confirmed accessibility issues.

[2]We had to exclude 3 outdated apps that do not work anymore.

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 [144]. 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 4.1 (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 (SDK30), and the latest versions of assistive services, i.e., TalkBack 12.1 and SwitchAccess 12.1.

4.7.2 RQ1. Effectiveness of Groundhog

Table 4.1 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 proxies. The columns entitled *TalkBack Unlocatable*, *TalkBack Unactionable.*, and *Abstract Unactionable* represent locatability and actionability issues by TalkBack Proxy, and actionability issues by Abstract Proxy, respectively. For each type of issue, we show the total number of detected issues and the number of issues manually verified by authors or True Positives (TP).

Table 4.1: 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		

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 exploration modes, i.e., Directional and Touch Exploration strategy 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 4.1. 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 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 4.5 (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 4.5 (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 4.5 (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 4.5 (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.

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 4.5(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 confirmed accessibility issues by Latte [139] 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 4.5 (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.

4.7.3 RQ2. Comparison with Scanner

Google Accessibility Scanner [17], or Scanner for short, is the most widely used accessibility analyzer for Android. Scanner leverages Accessibility Testing Framework (ATF) [24] to evaluate screen accessibility. To compare Groundhog with Scanner, we analyzed all the examined app states in Table 4.1 with Scanner and checked what it reports. The last column of Table 4.1 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 [12, 64, 84] in evaluating app accessibility.

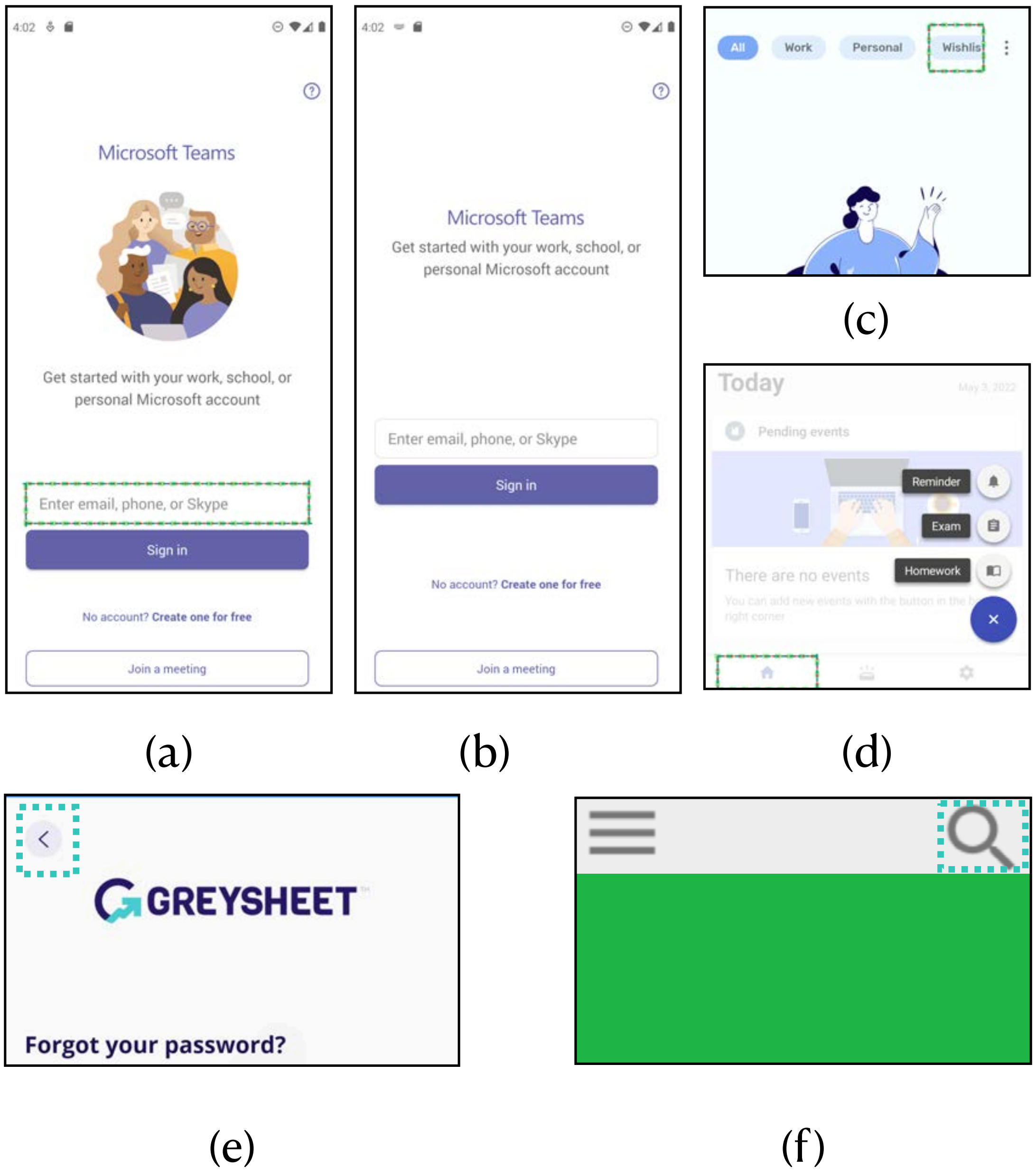

4:02
Microsoft Teams
Get started with your work, school, or personal Microsoft account
Enter email, phone, or Skype
Sign in
No account? Create one for free
Join a meeting
(a)
4:02
Microsoft Teams
Get started with your work, school, or personal Microsoft account
Enter email, phone, or Skype
Sign in
No account? Create one for free
Join a meeting
(b)
All
Work
Personal
Wishlis
(c)
Today
Pending events
Reminder
Exam
Homework
There are no events
(d)
GREYSHEET
Forgot your password?
(e)
(f)

Figure 4.5: (a-d) are examples of false positives, and (e-f) are examples of missing actions in Groundhog

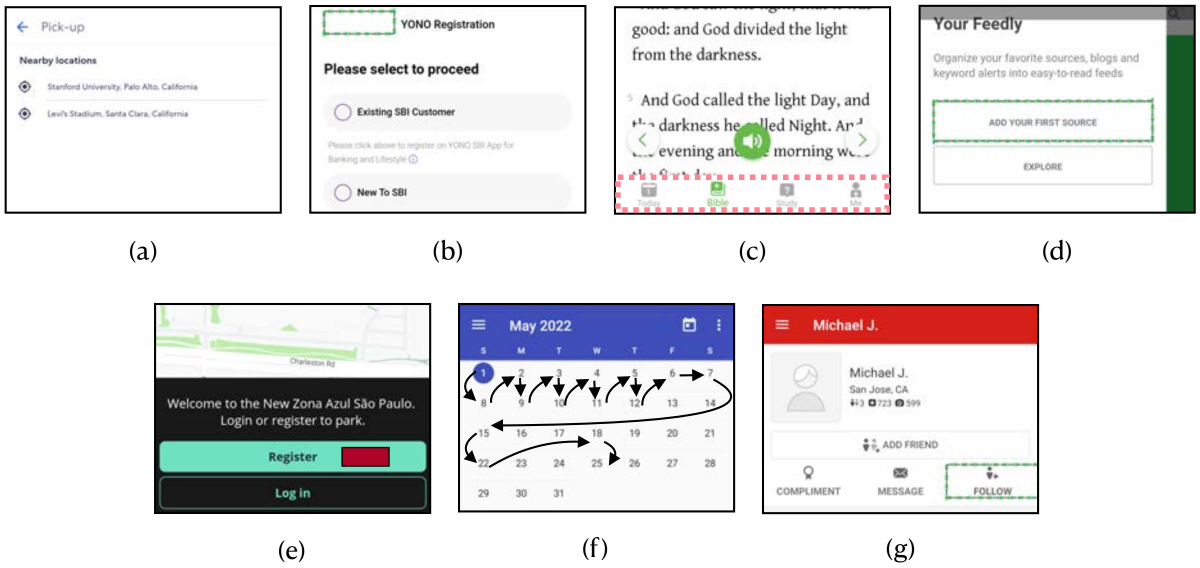

Pick-up
Nearby locations
Stanford University, Palo Alto, California
Levi's Stadium, Santa Clara, California
(a)
YONO Registration
Please select to proceed
Existing SBI Customer
New To SBI
(b)
good: and God divided the light from the darkness.
And God called the light Day, and the darkness he called Night.
Today
Bible
Study
Me
(c)
Your Feedly
Organize your favorite sources, blogs and keyword alerts into easy-to-read feeds
ADD YOUR FIRST SOURCE
EXPLORE
(d)
Welcome to the New Zona Azul São Paulo. Login or register to park.
Register
Log in
(e)
May 2022
(f)
Michael J.
San Jose, CA
ADD FRIEND
COMPLIMENT
MESSAGE
FOLLOW
(g)

Figure 4.6: Qualitative study of GROUNDHOG's report on subject apps

4.7.4 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 directional and touch exploration strategies. In severe cases, neither of these strategies can locate an element. For example, Figure 4.6 (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 directional exploration, but not by touch exploration. For example, Figure 4.6 (b) depicts the entry screen of YONO (a banking app), where the highlighted button can be located by directional exploration, 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 4.6 (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. The security implications of this accessibility issue are further studied in [111]. 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 directional exploration but can be focused on by touch. For example, Figure 4.6 (c) shows a part of the Bible app, when the user uses TalkBack in Directional exploration 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. 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 4.6 (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 [19] 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 4.6 (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 [29]. Hybrid frameworks enable a developer to implement mobile apps in one codebase with one language, like C# in Xamarin[114]. 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 directionally 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 4.6 (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 regular users. For example, Figure 4.6 (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.

4.7.5 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 proxies, 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 proxies. GROUNDHOG executes each action in 21, 24, and 40 seconds for Abstract, Touch, and TalkBack Proxy, respectively. There are some common time-consuming parts for all proxies: 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 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.

4.8 Threats to Validity

External validity. A key threat to validity is preserving the state of the app under test since three different proxies should perform the same action on the same element. We mitigate this threat by capturing a VM snapshot of the device used for all proxies. 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. 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* [80], 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.

4.9 Related Work

Empirical studies on mobile accessibility [12, 155, 137, 45] have revealed the prevalence of various accessibility issues in mobile apps, preventing disabled users from utilizing their services. These findings have motivated the research community to develop techniques to automatically detect accessibility issues [64, 12, 23, 48], and to repair the detected issues [45, 112, 10, 183].

In general, automated accessibility testing techniques evaluate app compliance with accessibility guidelines [163] using static or dynamic analysis approaches [148]. Static analysis approaches such as Lint [23] 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 [17] and other tools that are built on top of Accessibility Testing Framework [84, 88, 64] 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 [130]. 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 [12, 64] or provide the ability to write app exploration scenarios in form of GUI tests [22, 134].

A related prior work is Latte [139], which was already discussed in Section 4.1 and empirically compared against in Section 4.7. Alotaibi, et al. [11] have proposed a method of detecting certain accessibility failures that may occur when using TalkBack. However, in contrast to Groundhog, 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 unactionable elements. Such manual exploration is expensive, time-consuming, and may not result in good coverage.

Unlike prior testing techniques, Groundhog is a fully automated accessibility testing technique that only requires app in binary form and detects accessibility issues in interactions with the app using several interaction models. Groundhog can be generalized to any assistive service in the context of Android and with different exploration modes to evaluate all GUI elements at each state.

4.10 Conclusion

Prior accessibility testing tools can only point out a small portion of the problems that people with disabilities encounter while interacting with an app [109]. In this work, we proposed Groundhog, a fully automated assistive-service driven accessibility crawler to detect accessibility issues that only manifest themselves through interactions with the app. Groundhog explores apps and assesses the locatability and actionability of each element on the screen using different interaction modes provided by assistive services. Our future work involves evaluating the extent to which the ideas presented here can be applied to other computing domains (e.g., iOS, Web), and expanding Groundhog's support to additional assistive services and more complex gestures.

Chapter 5

AT-Aware Accessibility Testing: Over-Accessibility Issues

Mobile apps, an essential technology in today's world, should provide equal access to all, including 15% of the world population with disabilities. Assistive Technologies (AT), with the help of Accessibility APIs, provide alternative ways of interaction with apps for disabled users who cannot see or touch the screen. Prior studies have shown that mobile apps are prone to the *under-access* problem, i.e., a condition in which functionalities in an app are not accessible to disabled users, even with the use of ATs. We study the dual of this problem, called the *over-access* problem, and defined as a condition in which an AT can be used to gain access to functionalities in an app that are inaccessible otherwise. Over-access has severe security and privacy implications, allowing one to bypass protected functionalities using ATs, e.g., using VoiceOver to read notes on a locked phone. Over-access also degrades the accessibility of apps by presenting to disabled users information that is actually not intended to be available on a screen, thereby confusing and hindering their ability to effectively navigate. In this work, we first empirically study overly accessible elements in Android apps and define a set of conditions that can result in over-access problem. We then present OVER-

Sight, an automated framework that leverages these conditions to detect overly accessible elements and verifies their accessibility dynamically using an AT. Our empirical evaluation of OverSight on real-world apps demonstrates OverSight's effectiveness in detecting previously unknown security threats, workflow violations, and accessibility issues.

5.1 Introduction

Principles of universal design [54] 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. To facilitate disabled users' interaction with apps, mobile platforms support Assistive Technologies (AT) such as screen readers or special physical keyboards, which utilize the information exposed by Accessibility APIs to provide an alternative interaction model.

Prior studies have shown that many apps are shipped with functionalities that are not accessible using ATs [140, 12]. We call this the *under-access* problem. In this paper, 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 ATs but not otherwise.

An element is *Overly Accessible* (OA) when it provides more information and functionality to AT 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 [43, 97, 96]. 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 [115]. As an example, the Mediation Moments app [42] 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 AT 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 [17], nor Apple Accessibility Inspector [34] check any rules for over accessibility. They only check a set of accessibility rules (e.g., proper text size and color) on displayed UI elements. Most other accessibility testing studies [22, 134] extend the accessibility rules of standard scanners and cannot detect OA elements consequently. A recent accessibility testing study proposed Latte [140], an accessibility testing framework to examine the accessibility of UI elements by executing a specific use case using AT. Nevertheless, OA elements are not a concern of Latte as it focuses on finding inaccessible elements.

Prior security-related studies [118, 87] 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 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 AT users. Then, OverSight validates the accessibility of these elements dynamically using a custom AT with all the capabilities of Accessibility API and Talkback, which is the standard screen reader on Android devices. 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.

This chapter makes the following contributions:

- First study that introduces the problems caused by apps that are overly accessible.
- An empirical study of OA elements and their characteristics.
- The first automated tool, called OverSight, for localizing and detecting OA elements in Android apps, which has been made publicly available [121].
- An empirical evaluation on real-world apps, corroborating the effectiveness of OverSight

in detecting OA elements.

The remainder of this chapter is organized as follows. Section 5.2 motivates this study with an example and provides background information. Section 5.3 introduces OA elements according to our empirical study. Section 5.4 explains OverSight, an automated approach to detect OA elements. In Section 5.5, the evaluation of OverSight on real-world apps is presented. The chapter concludes with a discussion of the related research and avenues of future work.

5.2 Motivating Example & Background

Figure 5.1 shows screenshots of AppLock [92], a popular app locker with more than 5,000,000 installations and rating of 4.2. As shown in Figure 5.1 (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 5.1(a), she first sees the lock screen, depicted in Figure 5.1(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. 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 ATs to interact with apps. Mobile platforms such as Android have integrated ATs such as TalkBack [72]—the standard Android screen reader—and SwitchAccess [16]—a special keyboard with two keys, *Next* and *Select*—to enable app exploration for disabled users. Both of these ATs focus on each element on

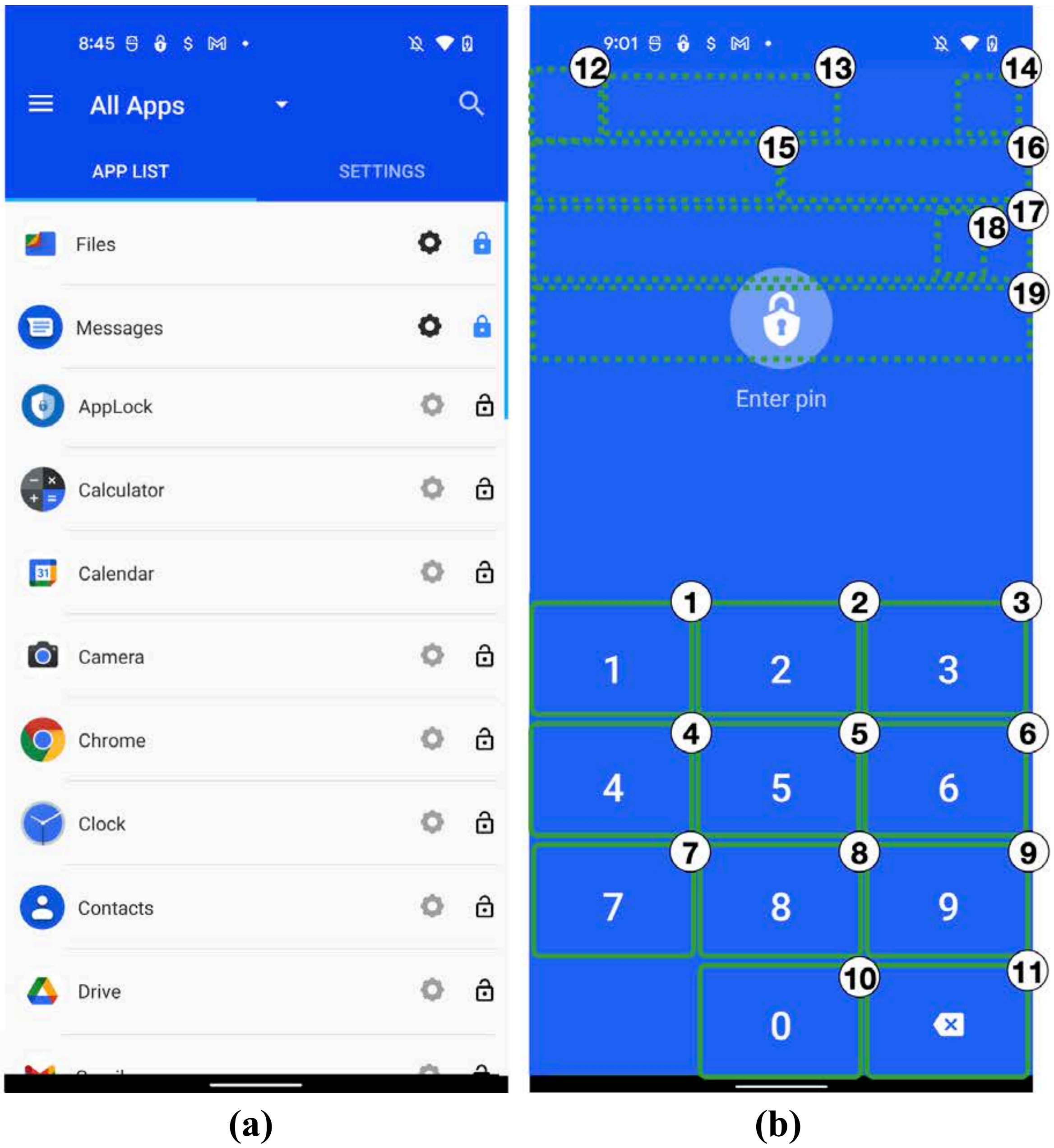


Figure 5.1: Built-in lock for a security-sensitive app.

the screen and navigate through them sequentially, from top left to bottom right. The *Select* switch in SwitchAccess or the *double tap* gesture in TalkBack perform the *Click* action that is similar to touching the element without ATs. To represent each element to blind users, TalkBack also announces a textual description of the focused element on the screen. For visual elements like icons, these textual descriptions, which are called *Content Description* in Android, should be provided by developers in the UI specification, a hierarchical structure of elements represented in an XML file.

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 [17], 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., [140, 17, 23, 113]), are aimed at finding *under-access*, i.e., features that should be available to the user but cannot be accessed using ATs. 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.

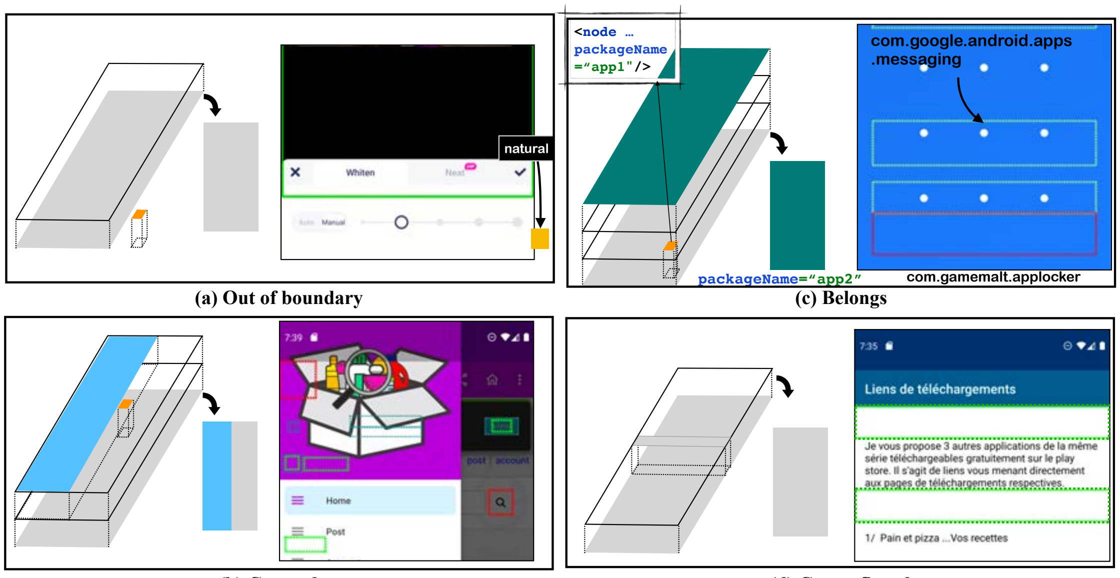
In practice, a blind user may need to understand the screen content by exploring and navigating through all the elements on the screen. Figure 5.1(b) shows which elements can be focused by TalkBack. The numbers indicate the order in which elements are focused. After passing pin pad elements, TalkBack 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 5.1(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 5.1(a). This page contains the list of device apps and the mechanism to enable or disable their locks. For instance, element 17 in Figure 5.1(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 [118, 87] and how to prevent such attacks [132, 128]. 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 ATs.

To fill this gap, we took a deeper look at how UI elements are represented to ATs. In modern platforms such as Android, Accessibility Service runs in the background and provides the required information about a window's content to ATs. From the perspective of Accessibility Service in Android, a window's content is presented as a tree of `AccessibilityNodeInfo`s (nodes) [75]. Android 12 documentation lists 65 different types of information that are provided by nodes. Table 5.1 illustrates a sample set of this information. We hypothesize that nodes with peculiar specifications can lead to OA elements. For example, in Figure 5.1(b), by comparing the Bounds and DrawingOrder of elements, the second and third method in Table 5.1, 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 AT can still navigate through them and announce them to an AT user. Our objective in this study is to study specifications of OA elements and propose an automated tool to detect such OA

Table 5.1: Sample types of information exposed from nodes to ATs.

	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.



Je vous propose 3 autres applications de la même série téléchargeables gratuitement sur le play store. Il s'agit de liens vous menant directement aux pages de téléchargements respectives.

(a) Out of boundary **(c) Belongs**

(b) Covered **(d) Camouflaged**

Figure 5.2: Over Accessibility Conditions.

elements that can have severe security, privacy, and accessibility impacts on apps.

5.3 Overly Accessible Elements

An element is OA if it is exposing more information/functionality to ATs 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 some real world apps. In this section, we explain the data collection and results of this study.

5.3.1 Data Collection

Our goal is to collect all the available information from nodes to ATs. 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) [26], 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 [9]. We installed each app on a Google Pixel 4 device, along with OSS. Then, one author 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. Two authors of the paper 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, authors 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. After the initial coding, the authors discussed disagreements to reach a consensus.

5.3.2 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 AT that is not available through regular interaction mode.
- **Overly Actionable:** elements that provide action to an AT that is not available through regular interaction mode.

These classes are inline with two accessibility principles from Web Content Accessibility Guidelines (WCAG) [159]: (1) Content should be equally perceivable by different users [160], and (2) UI elements should be equally operable by different users [161]. 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. On the left, Figure 5.2(a) illustrates a schematic of the screen in 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 5.2(a) also shows an 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 5.1 are examples of covered nodes. Figure 5.2(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. 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.

P5. Android invisible: Nodes that are not out of screen boundary and have positive area but they are specified as invisible to user.

P6. Belongs: Nodes that belong to a package name that is different from the app under test. Left side of Figure 5.2(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 ATs. The right side of Figure 5.2(c) shows a locker in our study, in which the elements of the Messages app are detected on the lock screen.

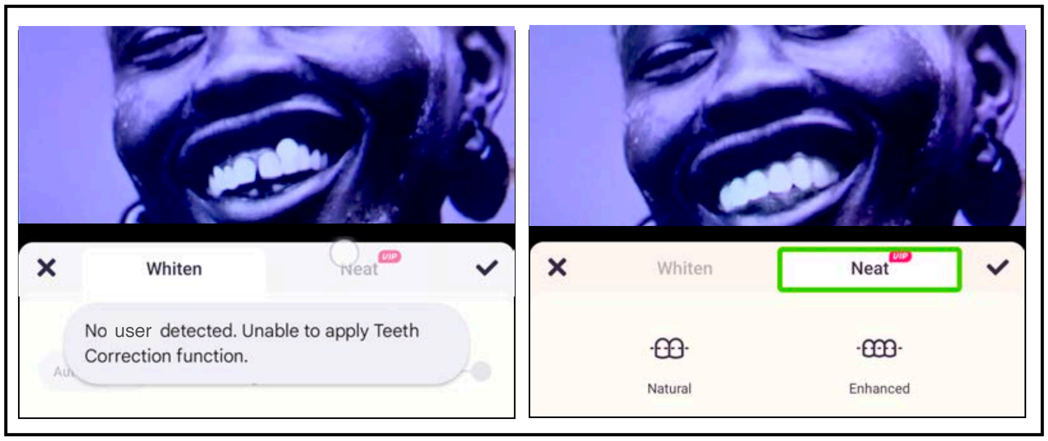

Whiten
Neat
VIP
No user detected. Unable to apply Teeth Correction function.
Whiten
Neat
VIP
Natural
Enhanced

Figure 5.3: Neat button is not working when touched by enabled users but is available to TalkBack users.

Overly Actionable

The `ActionList` attribute of nodes specifies the list of actions available to ATs. When a node support click action for ATs, 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. Figure 5.3 provides an example for this condition, where the teeth correction function is disabled for unsubscribed user but using TalkBack, the user can activate it.

A3. Camouflaged: Empty nodes that are used as placeholders and are not detectable by sighted users, e.g., empty text boxes. Figure 5.2(d) provides the schematic placement of these nodes on the screen on the left and a real example on the right.

5.4 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 AT. Figure 5.4 illustrates the overview of our approach. OverSight engine consists of two main components: OA Detector (Section 5.4.1) and OA Verifier (Section 5.4.2).

OA Detector gets a window's content specification in XML along with its screenshot through OverSight Service (OSS). As described in Section 5.3, 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 5.3). Confirming over accessibility issues in these nodes is the responsibility of OA Verifier . Our approach only relies on available information to AccessibilityServices; therefore, it is applicable to any app regardless of its technology or even if it is obfuscated. OA Verifier communicates with the device and explores the window with an AT to validate the reachability 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.

5.4.1 OA Detector

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.

Here, we describe the details of P1, P3-P6, and A1-A2 as implemented, and also elaborate

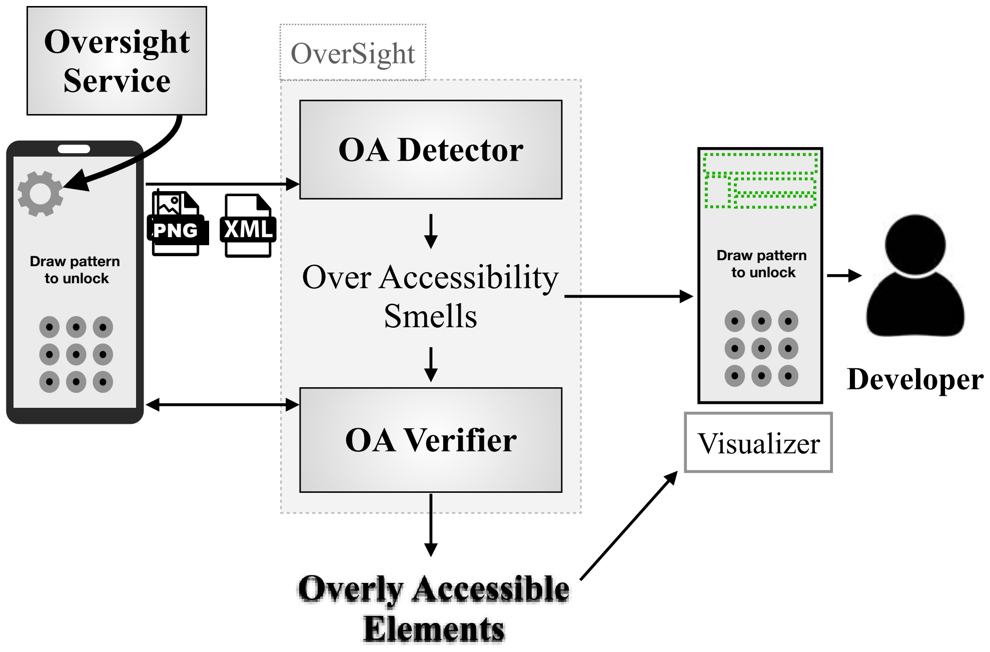


Figure 5.4: Overview of OVERSIGHT framework.

on the algorithms used to calculate covered nodes (P2) and camouflaged nodes (A3). Implementation details of all conditions are available with our open-source tool available at [121]. First, we define the conditions for perceivable and actionable nodes, and then we formally define all over-perceivable and over-actionable nodes. Recall from the description of these conditions in the previous section that all P1 to P6 nodes must be perceivable, and all A1 to A3 nodes must be also actionable.

Perceivable: A node is perceivable if it has a textual information. Based on Table 5.1, the attributes *text* and *ContentDesc* may contain such information.

$$\begin{aligned} &\forall n \in Node; n.Text \neq \emptyset \vee n.ContentDesc \neq \emptyset \\ &\Rightarrow perceivable(n) = True \end{aligned} \tag{5.1}$$

Actionable: A node is actionable if it has an attribute that is associated with an action, e.g., *Clickable*, *LongClickable*, or the existence of these actions in *ActionList*.

$$\begin{aligned}&\forall n \in Node; n.Clickable \vee n.LongClickable \vee \\ &\{CLICK, LONGCLICK\} \cap n.ActionList \neq \emptyset \\ &\Rightarrow actionable(n) = True\end{aligned} \tag{5.2}$$

P1. Out of boundary: To detect these nodes, we compare the boundary of the node with the size of the window, i.e. $Window.width$, $Window.height$. The bounds of an element is shown as the coordinates of the top left and bottom right of its bounding box.

$$\begin{aligned}&\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1]; \\ &x_0 < 0 \vee x_1 > Window.width \vee y_0 < 0 \vee y_1 > Window.height \\ &\Rightarrow out\_of\_bound(n) = True\end{aligned} \tag{5.3}$$

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 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 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

Algorithm 1: Overlap Analysis Algorithm

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

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.

P3. Zero area: The bounding box of any node forms a rectangle which can have a zero area.

$$\begin{aligned} &\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1]; \\ &x_0 = x_1 \vee y_0 = y_1 \Rightarrow zero\_area(n) = True \end{aligned} \tag{5.4}$$

P4. Invalid bounds: We use Equation 5.5 to find the nodes whose bounding box – bottom-left and top-right coordinates – is not a rectangle.

$$\begin{aligned} &\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1]; x_0 > x_1 \vee y_0 > y_1 \\ &\Rightarrow invalid\_bounds(n) = True \end{aligned} \tag{5.5}$$

P5. Android invisible: To detect nodes with this condition, we look for nodes without any of the above-mentioned conditions that are marked as invisible to user in node attributes.

(Recall row 6 in Table 5.1)

$$\begin{aligned}&\forall n \in Node; \neg n.Visible \wedge \neg out\_of\_bound(n) \wedge \\ &\neg n.covered \wedge \neg zero\_area(n) \wedge \neg invalid\_bounds(n) \\ &\Rightarrow android\_invisible(n) = True\end{aligned} \tag{5.6}$$

P6. Belongs: We compare the package name of nodes with the package name of the UI under test (UIUT) to find if nodes belong to its corresponding app.

$$\begin{aligned}&\forall n \in Node; n.pkgName \neq UIUT.pkgName \\ &\Rightarrow belongs(n) = True\end{aligned} \tag{5.7}$$

A1. Hidden: An actionable node that has any conditions in Equations 5.3 to 5.7 is considered hidden, since a sighted user cannot perform any touch gesture on it.

$$\begin{aligned}&\forall n \in Node; out\_of\_bounds(n) \vee n.covered \vee zero\_area(n) \vee \\ &android\_invisible(n) \vee invalid\_bounds(n) \vee belongs(n) \\ &\Rightarrow hidden(n) = True\end{aligned} \tag{5.8}$$

A2. Disabled: The *enabled* attribute of a disabled actionable node should be *False* to be considered as over-actionable (Recall row 5 in Table 5.1).

$$\forall n \in Node; \neg n.enabled \Rightarrow disabled(n) = True \tag{5.9}$$

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. With the advent of advanced computer vision and machine learning algorithms, analyzing app screenshots has been recently studied in prior works [46, 179, 44]. Such techniques can infer not only UI nodes, but also their structure from screenshots. While these advanced UI analysis techniques can be adopted here, we opt for the simple aforementioned technique that can effectively detect empty boxes without the need for complex models.

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

5.4.2 OA Verifier

The behavior of different ATs in focusing on the elements and performing an action on them cannot be predicted statically. To confirm if an AT can reach the detected over accessibility smells, we utilize OA Verifier . The goal of this component is to evaluate the reachability and actionability of nodes identified by OA Detector on a real device with an AT. To interact with the device, we expand the capabilities of OverSight Service (OSS) that was previously only responsible for capturing information from nodes. OSS receives commands from OA Verifier, perform the required gestures on the device, broadcast commands and return the results.

To achieve its objective, OA Verifier uses two subcomponents: 1) Reachability Analyzer, and 2) Actionability Analyzer. The first component verifies if an AT can focus on the node, while the second one checks if the AT can perform the action on it. In this work, we describe our approach for TalkBack as the standard screen reader in Android, and a custom

AT, called Super AT (SAT), as it has all the information and functionalities provided by Accessibility Service. As briefly mentioned in Section 5.2, Accessibility Service in Android runs in the background and provides the required information to ATs. Each AT specifies a list of *flags* [73] to request for the corresponding information and capabilities from the Accessibility Service. For example, `flagRetrieveWindowContent` is required to be able to get the events indicating that something on the window has changed. In this work, we give all the capabilities to SAT, making it a representative of all ATs that are using a subset of its capabilities. In other words, SAT-verified nodes show what can *potentially* be accessible to different ATs, while OA nodes verified by TalkBack show that any user, who utilizes the standard platform screen reader, can get access to their content. The input for both of these subcomponents is an emulator snapshot, captured from a specific state of the UI under test, and a list of nodes to be verified, i.e., over accessibility smells.

Reachability Analyzer

If an AT can focus on a node, we call the node reachable by that AT. In Android, Accessibility API can perform actions on given nodes by calling the `performAction` method. OA Verifier identifies nodes by their `XPath`, i.e., their absolute path from the root node, and performs the focus action, `ACTION_ACCESSIBILITY_FOCUS`, on them. If this focused node, returned by `AccessibilityService`, is the desired node, OA Verifier determines the node reachable by SAT.

To assess the reachability of a node with TalkBack, we utilize the "Explore by swiping" strategy instead of the touch exploration 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, OA Verifier first explores the screen backward by drawing "swipe left" gesture. Whenever TalkBack focuses on a node, OA Verifier calls the node TalkBack Reachable. TalkBack continues screen exploration until either it reaches all the nodes in the given list, or sees a repetitive node.

Some UI components such as scrollable widgets may render some elements on the app unreachable. In practice, to break such infinite loops, a screen-reader user can touch on an element outside of the loop and resume exploring the app. To work around these loops, OA Verifier performs both forward and backward navigation from the top of screen when it does not meet its stopping criteria. Eventually, nodes that TalkBack cannot focus on by either backward or forward app exploration are determined to be unreachable by TalkBack.

Actionability Analyzer

An element is considered actionable, if it 1) is reachable and 2) performs the action successfully. Thus, Actionability Analyzer first evaluates reachability of over actionability smells using the same strategy as Reachability Analyzer.

Once Actionability Analyzer determines reachable nodes, it attempts to performing the action on them. This means it requires to first focus on the element and trigger the action using TalkBack or SAT. Since reachability of these nodes have already confirmed, we directly put the accessibility focus on the node under test using Accessibility API. Then, we utilize the specific AT to perform action. For TalkBack, OA Verifier performs a double-tap gesture to click the focused node. For SAT, OA Verifier calls `performAction`(ACTION_CLICK) on any given node. To verify if the action was performed successfully, OA Verifier listens to the AccessibilityEvents and denotes the node clickable by either TalkBack or SAT if VIEW_CLICK event or WINDOW_CONTENT_CHANGED was logged.

5.5 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. How prevalent are over-access problems in security-concerned apps?

RQ3. What are the potential impacts of OA elements on different apps and communities?

RQ4. What is the performance of OverSight?

5.5.1 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 in a prior study [140], 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. For the second question, we mainly focus on app lockers, security-critical apps that are responsible for protecting user apps. We picked 5 highest-rated, 5 lowest-rated, and 5 randomly selected app lockers from Google Play and followed the same strategy as group1 to capture two different states from each app. We did not incorporate the low-rated app lockers in RQ1 to keep the quality of apps in that study consistent.

Implementation details

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. OverSight Service is implemented in Kotlin and communicates with OA Detector and OA Verifier components, implemented in Python, using ADB [26].

5.5.2 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.

OA Detector: 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 5.2 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.

Table 5.2: Accuracy of OverSight in running on 30 apps.

App	N	Smells		DP	TalkBack		SAT	VP	VR
		P	A		R	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%

As shown in the Table 5.2, 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 5.5 shows some examples where OA Detector erroneously evaluates a node as OA. In Figure 5.5(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 5.4.1). In Android, transparent layouts pass the touch gesture to the underlying elements so that they are not 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 AT users. For example, if all the stacked nodes are focusable, AT will focus on each of them separately, confusing AT 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 5.5(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 if OA Detector fails to detect any issues, i.e., False Negatives, we ran OA Detector on a set of apps with known issues. OA Detector's False Negatives are the OA elements that OA Detector fails to report. Since no prior dataset exists, we take our apps from the empirical study that were manually investigated for OA elements (recall Section 5.3). We investigated the manually confirmed OA issues that do not appear in the list of OA smells. Figure 5.3 shows the only case that the OA Detector failed to detect. The reason for this failure is that instead of disabling the button, the app intercepts the click event at runtime when it is touched to show an error message. This means the button performs the click action successfully with and without AT. However, its inconsistent behavior cannot be detected by OA Detector. We also noticed in some cases the issue was captured not in the first attempt but after the second attempt. This issue is due to the challenges of interacting with the device using OSS and analyzing the results at a proper time. To mitigate such validity threats, we ran our experiments 3 times on each app.

Further investigation of conditions of detected OA elements revealed that the "covered" condition (recall P.2 in Section 5.4.1) 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 [71] (recall row 6 of Table 5.1) to be announced by TalkBack. However, our review of Android's source code [74] 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 strategy of detecting visible nodes to minimize such issues.

OA Verifier: To evaluate the OA Verifier component, we investigate the nodes specified as reachable and actionable with TalkBack and SAT. To check the reported nodes by OA Verifier, we load the corresponding snapshots of the app states on the emulator and utilize an AT, e.g., TalkBack, to explore the app and assess Reachability (R) and Actionability (A) of OA smells. In terms of reachability, if the AT can focuses on a node, we consider it reachable. For actionability, the node is actionable if it is reachable 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 reachable 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 reachable to the number of reachable nodes detected by OA Verifier, while recall is the ratio of number of nodes that correctly verified to be reachable to the number of OA smells that are manually verified to be reachable.

Table 5.2 shows the average precision and recall of OA Verifier using TalkBack and SAT 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 5.5(c) shows an example of a set of nodes that were erroneously detected to be unreachable 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 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 mistakenly reported by OA Verifier as unreachable or not over accessible (False Negative). 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 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 XPaths for nodes, making the logged information inaccurate.

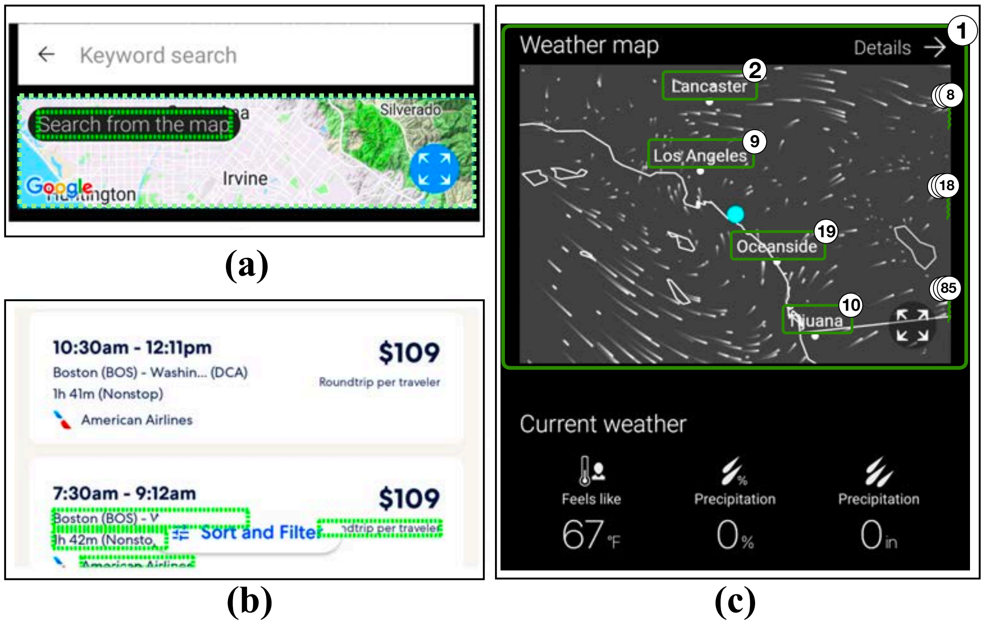


Figure 5.5: OVERSIGHT Failures: (a) and (b) are false positives of OA Detector, where dashed green boxes are erroneously detected as covered; (c) is a false negative of OA Verifier, where TalkBack is stuck in the world map.

5.5.3 RQ2. OA Elements in Security-Sensitive Apps

OA elements in security-sensitive apps, such as app lockers, put the privacy and security of both users and apps at stake by divulging private content or granting access to functionality that they are supposed to protect. To understand the prevalence of such critical issue in these apps, we utilize OVERSIGHT to test 15 real-world app lockers. All app lockers require two permissions from the users to work, 1) "Usage Access", to track what other apps the users are using, 2) "Display Over other Apps", to place their lock screen on top of the other apps. We grant the required permissions to the apps and evaluate two main functionalities of lockers: 1) locking the locker app itself, and 2) locking another third-party app that they are to protect. Table 5.3 summarizes the test set and results. The table contains three groups of 5 app lockers — most popular, randomly selected and least popular — from a list of 125 lockers we got from the Google Play store. The list includes the app locker from our empirical study, marked by '*' in the table. 'x' indicates that the protection could be bypassed by an AT, while '✓' indicates no OA element was detected by OVERSIGHT. We also manually confirmed the automatically diagnosed over-access problem by OverSight and reported all the issues to the developers.

As Table 5.3 shows, 13 cases out of 30 different states have over-access problems not only by SAT, but also by TalkBack, the standard screen reader. 5 of these issues belong to the most popular apps, endangering the security of hundreds of millions of users.

We also observed that apps with lower rating and installation number are not as robust as popular ones. For example, we had to reopen the locked app using "com.saeed..." multiple times to finally see the lock screen. On the other hand, interestingly, the over-access problem is not as common in the last 5 apps. We realized that app lockers utilize different strategies in providing a lock screen. For example, in the last app in Table 5.3, we found that the app locker first puts the app in the background and then displays the lock screen. In this way, OA elements still exist, yet they will not endanger the target app as they are the nodes on the home screen. Such strategy is time and energy consuming and would be less appealing to users. Among other apps, some inflate a full-screen overlay on the locker without creating a new Activity such as "com.gamemalt.ap..." or "com.litetools...", while the other ones such as "com.sp.protec...", "com.domobile...." and "com.ammy..." create a new Activity for the lock screen. Android provides mechanisms for both approaches to manage the hierarchy of nodes for the UI elements. The default behavior in inflating an overlay on the same Activity results in appearance of all the elements of the Activity, including those that should not be accessible, in the UI hierarchy. Thus, developers need to take proper actions to avoid that by setting their nodes not important for accessibility for example. However, by default, the hierarchy of nodes for a new activity only incorporates nodes that are specified in this activity and will not leak the elements from prior activities.

Developer's decision in utilizing these strategies can impact app stability, robustness and usability. We strongly encourage them to consider security threats of OA elements, resulting from their design decisions as well as the other app qualities.

Table 5.3: Over accessibility issues in app lockers.

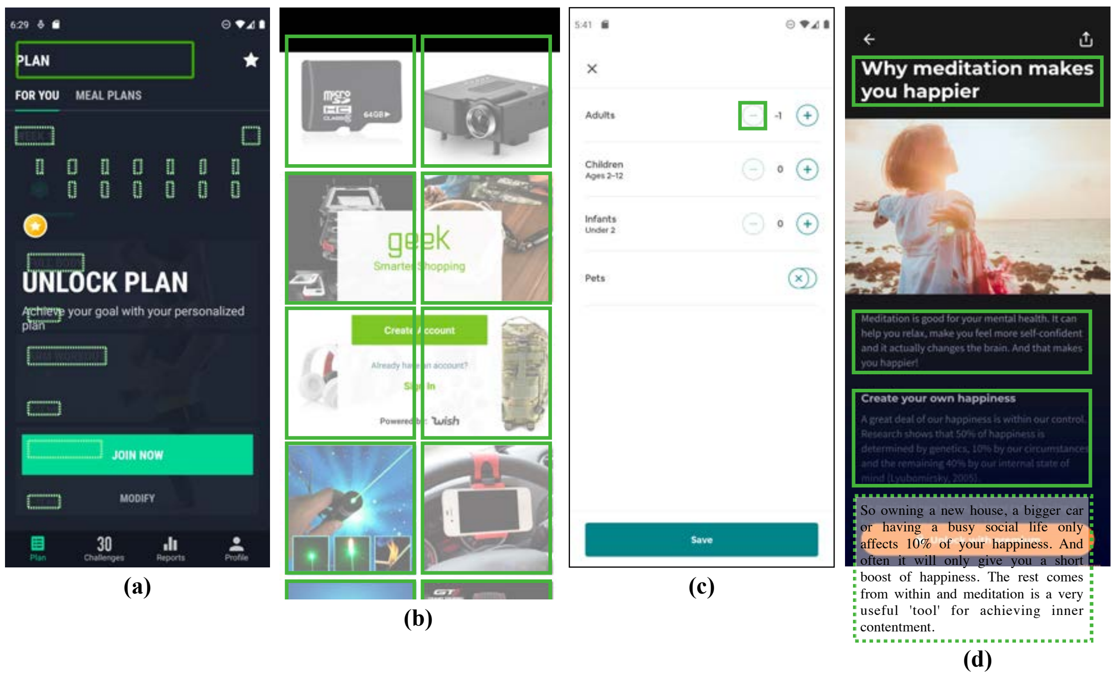
App	Version	#Installed	Rate	State 1		State 2	
				TB	SAT	TB	SAT
`com.netqin.ps`	293	+100M	4.3	✓	✓	✓	x
`com.domobile....`	2021052001	+100M	4.2	✓	✓	x	x
`com.alpha.app...`	412	+50M	4.7	✓	✓	✓	✓
`com.sp.protec...`	231	+50M	4.4	✓	✓	x	x
`com.thinkyeah...`	166	+10M	4.6	✓	✓	✓	✓
`com.litetools...`	91	+10M	4.3	x	x	✓	✓
*`com.gamemalt...`	108	+5M	4.3	x	x	x	x
`com.nevways.a...`	92	+5M	4.3	✓	✓	✓	✓
`com.ammy.app....`	151908296	+1M	4.6	✓	✓	x	x
`com.gsmobile....`	34	+500K	4.5	✓	✓	✓	✓
`me.ibrahimsn....`	134	+50K	4.0	✓	✓	✓	x
`com.cd.applo....`	2	+10K	4.5	✓	✓	✓	✓
`com.saeed.app...`	4	+10K	4.4	✓	✓*	✓	✓*
`com.applockli...`	8	+10K	4.0	✓	✓	✓	✓
`com.mms.applo...`	1	+5K	4.0	✓	✓	✓	✓
`app.lock.hide...`	6	+5K	3.5	✓*	✓*	✓*	✓*

5.5.4 RQ3. Qualitative Analysis of OA Elements

We manually examined all reported OA elements by OverSight in Table 5.2 and categorized them based on their impact on disabled users and app developers in terms of app accessibility, app security, and work flow violations.

App Accessibility

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 5.6(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



Meditation is good for your mental health. It can help you relax, make you feel more self-confident and it actually changes the brain. And that makes you happier!

A great deal of our happiness is within our control. Research shows that 50% of happiness is determined by genetics, 10% by our circumstances and the remaining 40% by our internal state of mind (Lyubomirsky, 2005).

So owning a new house, a bigger car or having a busy social life only affects 10% of your happiness. And often it will only give you a short boost of happiness. The rest comes from within and meditation is a very useful 'tool' for achieving inner contentment.

Figure 5.6: Impacts of OA elements. (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.

is for the user to follow the introductory steps; however, the information from next steps are available to AT user from the beginning, 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 AT users but also complicate their interaction with apps. When they explore app by swipe, there is no lock preventing their access. 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 AT users. Case in point, background images in Geek, shown in Figure 5.6(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 AT users who believe there are real buttons on the screen. Interestingly, this app was also diagnosed with under-access problem in a prior work [140] because of a rolling dynamic widget in the background. The AT user gets stuck in an infinite loop and cannot login if she wants to explore the screen by swiping.

App Security

In RQ2, we extensively explained the critical impact of OA elements on the security of app lockers. OA elements in such apps can reveal the screen content of other apps that they are designed to protect. They can also provide access to the settings page, where the AT user can disable the lock totally. As the app lockers are mainly responsible for protecting app content, OA elements put a vulnerable app's reputation at stake. The issue is, however, not limited to app lockers. For example, parental control apps, which provide a mechanism to lock specific apps on the child's device, or variety of built-in locks in apps such as banking are also vulnerable to OA elements.

Workflow Violations

Developers design a workflow by which users interact with 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 5.6(c), when the number of passengers is zero, the decrease button for the number of travelers is disabled. However, using AT 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 ATs, 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 5.6(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.

5.5.5 RQ4. 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.

5.6 Threats to Validity

External validity. An important threat is the completeness of OA elements' conditions, extracted from examining 100 different states of 20 randomly selected apps. To mitigate this issue, we carefully selected a diverse set of app states considering the limitations of manual exploration. The extracted conditions were organized under two main classes, Over Perceivability and Over Actionability, inspired by accessibility guidelines. Although this process gives us confidence that the conditions provide a good coverage for different variations of app states, having a larger set of app states would increase the validity of generalization of our findings.

Another threat is the generalizability of the reported results of OverSight on real world apps. Our evaluation dataset for RQ1 consists of 60 screens from 30 apps. While including more apps and screens would increase the validity of this experiment, we have attempted to mitigate this threat by selecting the apps from three different sources: (1) apps with confirmed under-accessibility issues, (2) apps with an intention to make users' information secure, and (3) a diverse selection of popular apps – 30 apps in 16 different categories in total. The first two groups are intentionally selected, since they are related to under- and over-accessibility, respectively.

While Oversight mainly relies on XML layout of a screen to detect OA conditions, for detecting camouflaged elements it requires a screenshot of the screen, which is not possible for apps that restrict the ability to capture screenshot. This tends to be the case for apps displaying copyrighted content. In such situations, Oversight may miss OA elements with the camouflaged condition.

Internal validity. Our implementation of OverSight is built on top of several tools, like *ADB* and *AccessibiltiyService*, which can introduce bugs in the process of OA element detection. Moreover, it is possible there are defects in our implementation of the prototype. To address these threats, we used the latest versions of third-party tools, conducted code review on our implemented program via Github, and extensively tested the prototype in a set of apps (with no intersection with the empirical study or the evaluation data sets).

5.7 Related Work

Accessibility Analysis of Mobile Apps. Analyzing mobile app accessibility has been an active research area with the focus on proposing accessibility guidelines [159], empirical study [12, 155, 135], automated testing [23, 17, 84, 22, 134, 64], and repair techniques [45, 112, 10, 183]. Although accessibility principles [162] have implications for both under-access and over-access problems, there is no guideline regarding over accessibility and prior studies and tools are merely aimed at analyzing inaccessible functionalities in apps. OverSight is the first work in introducing the over-access problem and the first attempt to detect them.

The biggest challenge in detecting these issues is that OA elements manifest themselves in interactions involving ATs. However, the majority of accessibility testing tools are AT-agnostic. Static analysis tools like Lint [23] parse screen content and configuration files upon compilation to identify accessibility violations in code. To find issues that are undetectable in code, dynamic approaches [17, 22, 134, 64, 88] have been developed that analyze the rendered UI components on the screen, either after manual navigation to the target state of the app [88, 17, 22] or with an automated crawler [64]. These techniques, however, do not consider the use of ATs like screen readers and external keyboards in app exploration.

A prior technique, called Latte [140], utilizes ATs to evaluate if an app's functionalities, generated from its UI test cases, can be performed by disabled users. However, test cases are not always available, without which assessing UI elements with ATs is a time and memory intensive process. To mitigate this issue, Groundhog [143] proposes an optimized app exploration approach for accessibility testing. However, both Latte and Groundhog only focus on inaccessible elements. OverSight's contribution is in taking advantage of characteristics of OA elements to detect and verify over accessibility issue and its impacts on mobile apps.

Security Studies on Accessibility. Accessibility has also been studied in a security context, considering how accessibility APIs on mobile platforms can be exploited by attackers [90]. Kraunelis et al. [99] showed how malicious apps can abuse accessibility service to detect app launches and bypass security measures [156]. Researchers have also investigated the potentials of using accessibility APIs in designing attacks such as ClickJacking attacks [15, 182, 93]. These targeted the `BIND_ ACCESSIBILITY_SERVICE` permission to take full control of the UI, as demonstrated by Cloak and Dagger technique [68]. Studies have devised defense schemes [180] and solutions to this attack [132, 128, 87].

These security studies approached accessibility from a malware's perspective, designing potential attacks, analyzing the framework and proposing solutions based on the assumption that the assistive app is malicious. Conversely, we introduced OA elements as a new vulnerability that can be exploited using a benign app or standard ATs such as TalkBack. The privacy implications of such elements for users as well as their negative impact on developers' revenue and reputation has remained unnoticed in security studies. While prior security studies have shed light on what attackers can do through exploitation of accessibility APIs, our paper aims to provide software engineers with a tool to detect and eliminate vulnerabilities due to over-access in their apps.

5.8 Conclusion

Assistive Technologies 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, concerning inaccessible data and functionality. However, in this study, we 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. They can degrade app accessibility, endanger app security, and put developers' reputation and revenue at stake. To avoid such issues, in the future, we investigate the application of automatic program repair techniques in resolving the over-access problem in mobile apps.

Chapter 6

Time-Aware Assessment of App Accessibility

With mobile apps playing an increasingly vital role in our daily lives, the importance of ensuring their accessibility for users with disabilities is also growing. Despite this, app developers often overlook the accessibility challenges encountered by users of assistive technologies, such as screen readers. Screen reader users typically navigate content sequentially, focusing on one element at a time, unaware of changes occurring elsewhere in the app. While dynamic changes to content displayed on an app's user interface may be apparent to sighted users, they pose significant accessibility obstacles for screen reader users. Existing accessibility testing tools are unable to identify challenges faced by blind users resulting from dynamic content changes. In this work, we first conduct a formative user study on dynamic changes in Android apps and their accessibility barriers for screen reader users. We then present TimeStump, an automated framework that leverages our findings in the formative study to detect accessibility issues regarding dynamic changes. Finally, we empirically evaluate TimeStump on real-world apps to assess its effectiveness and efficiency in detecting such accessibility issues.

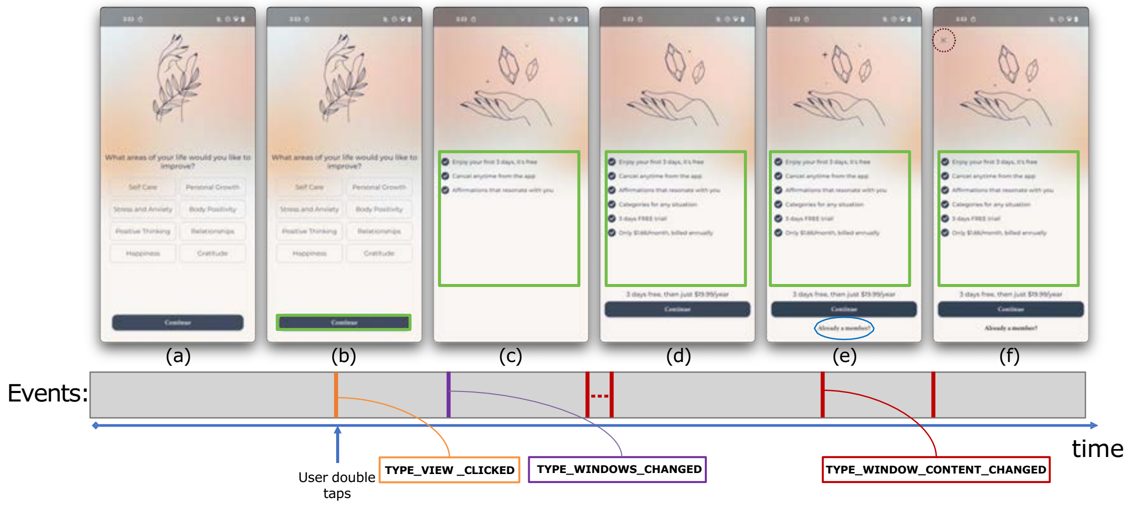

What areas of your life would you like to improve?
Self Care
Personal Growth
Stress and Anxiety
Body Positivity
Positive Thinking
Relationships
Happiness
Gratitude
Continue
Enjoy your first 3 days, it's free
Cancel anytime from the app
Affirmations that resonate with you
Categories for any situation
3 days FREE trial
Only $1.66/month, billed annually
3 days free, then just $19.99/year
Already a member?
(a)
(b)
(c)
(d)
(e)
(f)
Events:
User double taps
TYPE_VIEW _CLICKED
TYPE_WINDOWS_CHANGED
TYPE_WINDOW_CONTENT_CHANGED
time

Figure 6.1: Evolution of content loading on the screen across various states over time: (a) represents the initial screen state before the user initiates an action, (b) captures the moment when the user interacts with the app by clicking on a button, and (c) to (f) illustrate the gradual appearance of new screen content over time. Notably, in (f), the close button, indicated by a dashed red circle, appears above the accessibility focus. Since it is not tagged with `liveRegion` attribute, it is also not announced, and a screen reader user does not notice it.

6.1 Introduction

Dynamically changing visual content of screen (e.g., through animation) is a commonly used technique for enhancing the visual aesthetics of an app and to guide users' attention to specific parts of the app. However, these visually appealing techniques should not come at the cost of making apps inaccessible. In adherence to legal frameworks [120, 5], established guidelines [159, 33, 27], and ethical principles, the digital realm should be inclusive and accessible to all. This is especially crucial for the approximately 15% of the global population with some form of disability, including more than 300 million users that are blind or visually impaired [174].

Visually impaired users rely on assistive technologies like screen readers to interact with mobile apps. These tools enable users to navigate to a specific element on the screen and listen to the content in focus. However, the tunnel-like focus provided by screen readers may lead to unawareness of dynamic changes occurring elsewhere on the screen. A known example of such dynamic content is error notifications. When an app assesses user inputs and provides feedback, such as an error message through a notification, these changes may go unnoticed by screen reader users. Mobile platforms let developers designate these dynamically changing parts of a screen as “live regions”, assisting screen readers to detect and announce such changes to users. Unfortunately, developers often neglect using proper accessibility attributes, posing significant accessibility challenges for the blind.

Earlier studies and guidelines addressing software accessibility have only scratched the surface of this critical issue. The related accessibility guidelines on this matter primarily center on designating live regions for screen reader announcements. Specifically, in scenarios involving error messages, Web Content Accessibility Guidelines (WCAG) success criterion 3.3.1 emphasizes the crucial need for users to be informed about errors and comprehend what went wrong and recommends techniques such as annotating error notifications as live regions [168]. However, the challenge extends beyond these scenarios. Dynamic changes have been neglected from prior studies and tools that rely on screen captures from an app to detect accessibility issues [106, 110, 59]. GUI crawlers and app explorers typically capture screenshots of an app under test after it is in stable conditions by waiting for certain amount of time [143, 65]. Unfortunately, these approaches fail to capture app states during the entire rendering process, overlooking changes that occur over time on the screen. Consequently, they are not capable of detecting accessibility issues caused by dynamic contents.

To bridge this gap, we initiated a formative study aimed at identifying various types of dynamic changes and assessing their impact on screen reader users. This study revealed characteristics of accessibility issues related to dynamic content changes that negatively impact blind users. Building on these insights, we developed TimeStump, an automated framework designed to detect such issues in Android apps. TimeStump comprises an automated crawler, randomly exploring diverse app states and capturing data before, during, and after each action. Subsequently, this data undergoes processing using the identified patterns from our initial study to pinpoint problematic dynamic changes for screen reader users. The identified issues are then reported and visualized for developers.

This chapter makes the following contributions:

- The first study on accessibility issues arising from dynamic content changes in Android apps.
- The introduction of the first automated crawler capable of capturing dynamic content changes, complemented by the creation of the initial dataset cataloging such behaviors.
- The development and public release of the first automated tool, named TimeStump, designed for localizing and detecting accessibility issues related to dynamic content changes in Android apps [30].
- An empirical evaluation on real-world apps, corroborating the effectiveness of TimeStump in detecting accessibility issues induced by dynamic screen changes.
- A user study involving blind participants to assess the impacts of dynamic screen change on app accessibility.

The remainder of this chapter is organized as follows. Section 6.2 provides the background information. Section 6.3 describes our formative user study that motivated this work. Section 6.4 presents TimeStump, an automated approach for detection of problematic dynamic content changes. Section 6.5 details the evaluation of TimeStump on real-world apps and in collaboration with blind participants. This chapter concludes with a discussion of threats to validity, related research, and future work.

6.2 Background

Mobile platforms offer the possibility of dynamic content changes, allowing developers to alter the screen content in real-time. Figure 6.1 displays an Android app called “I Am” [116] that provides daily affirmations for users and has more than 5 million downloads. When the screen reader focuses on the continue button as shown in Figure 6.1(b), the user can double-tap to perform the click gesture. Soon after clicking the button, the window changes, and some promotional content appears gradually, such as text, buttons, and other elements. For example, the “already a member” button, dotted in blue in Figure 6.1(e), and the close button, dashed red circle in Figure 6.1(f), appear after the bullet points are displayed.

This kind of screen rendering can pose severe challenges to screen reader users. Blind users utilize screen readers to interact with apps, and when encountering an unfamiliar app, they navigate through the on-screen elements sequentially to understand the app’s layout. The swipe right and left gestures allow the screen reader to move to the next or previous element, respectively, highlighting it with a green box as shown in Figure 6.1(b). When an element is focused, the screen reader vocalizes its textual description, enabling blind users to gauge its functionality in a manner analogous to how sighted users depend on the visual cues of an element. Should the textual description align with their expectations, blind users execute a double-tap, mirroring the single-tap action typical of sighted users. The following example illustrates the challenges blind users can face when dealing with dynamic changes. In Figure 6.1, as the user navigates to screen (c), the top element, which is the text view component, receives the accessibility focus. Screen reader users explore the screen by moving through the elements sequentially from top to bottom using a swipe-right gesture. However, the close button annotated in dashed red is not recognizable as it appears on top of the screen and users are less likely to traverse backward, to the area they already visited. Such barriers can lead to unintentional interactions with ads or difficulties navigating away from them.

In Android, the guidelines suggest using an attribute called `liveRegion` to help screen readers recognize the appeared content. When an element is annotated as `liveRegion`, it is announced by the screen reader. Android system utilizes an event-based model to inform screen readers of changes in live regions. A GUI element emits an `AccessibilityEvent` when there are changes to its state, which is received by assistive technologies such as a screen reader. Figure 6.1 illustrates several different types of events that can be triggered during the loading of app content, with each color representing a distinct event. For example, `TYPE_VIEW_CLICKED` events occur after a view is clicked, `TYPE_WINDOWS_CHANGED` events happen when the app transitions to a different window, and `TYPE_WINDOW_CONTENT_CHANGED` events take place after the content inside a window changes.

Assistive technologies can also identify the element that is the source of events. In Android, GUI elements are represented by a tree of `AccessibilityNodeInfo` objects that mirror the XML hierarchy of elements and their attributes.

`AccessibilityNodeInfo` tree can be likened to the Document Object Model (DOM) tree in the case of web pages, offering a hierarchical representation of rendered elements on a web page. Prior studies on exploring various states of web apps for testing purposes have characterized dynamic content changes as modifications to the DOM that occur without reloading the page [107, 98]. These changes include updating or disappearing content [167], reordering elements [166], and inserting specific elements [165], as outlined in the WCAG guidelines. Similar to WCAG guidelines, Android suggests using accessibility attributes to notify screen reader users of such changes [169].

However, these guidelines only scratch the surface of the issues that may arise as a result of dynamic contents. For instance, when a temporary button, like an undo button, pops up on the bottom of the screen, users may struggle to locate it within the brief time frame of its visibility. This challenge intensifies when content disappears before users become aware of its existence. Merely relying on accessibility attributes does not fully resolve this issue.

6.3 Formative Study

We conducted a formative study to investigate the impact of various dynamic content changes on screen reader users.

6.3.1 Study Design

Prior studies and guidelines on Web defined dynamic content changes as modifications to the DOM that occur without reloading the page [107, 98]. Consequently, in Android apps, dynamic content changes encompasses any modifications to the hierarchical representation of elements, i.e., a tree of `AccessibilityNodeInfos`, in a rendered window. These modifications include adding, removing, or changing attributes of elements. To have a better understanding of different types of dynamic content changes in Android, two authors conducted an empirical analysis of 50 Android apps. These apps were randomly chosen from the Google Play Store, representing various app categories. Additionally, to ensure the significance and popularity of the apps studied, each app selected had a minimum of 1 million downloads. For each app, two authors manually explored the app screen using a combination of actions, such as clicking, scrolling, and typing, to observe if that would trigger dynamic content changes. If so, a recording from the screen is taken with a brief description of the dynamic content change. Following this, two authors engaged in an iterative open-coding process to categorize the types of dynamic content changes identified.

Through our empirical analysis, we identified 5 types of dynamic content changes.

Appearing Content. This content change type is characterized by an element that initially is not present on a loaded window, but appears a few moments later and remains on the screen. For example, in Figure 6.1(f), the close button, marked by a red circle, appears after a few seconds.

Disappearing Content. This content change type describes an element that disappears either after a set period or as a result of user interaction. For example, in Figure 6.2(i), the navigation bar at the bottom including element **a** as well as the more button on top, element **b**, disappear when users navigate through the list of items.

Short-Lived Content. This content change type relates to an element that initially is not present on the screen but appears and remains on the screen only for a brief duration. Due to its transient nature, we refer to this as short-lived content. For example, as illustrated in Figure 6.2(iii), when users save a restaurant, a notification message annotated as **e** appears to notify them that they have successfully saved the store and to offer an option to view all saved stores. This message disappears after a few seconds.

Moving Content. This content change type refers to an element that is initially visible on the screen but subsequently gets relocated to a different part of the screen. For instance, in Figure 6.2(ii), the app-related information, marked as **c**, is shifted to the bottom of the screen and goes out of screen bounds after user presses the install button. Users must locate that information at a different position within the sequence of elements on the screen, as perceived by screen readers.

Content Modification. This content change type pertains to an element that remains on the screen but its attributes change. For example, in Figure 6.2(ii), the `TextView` annotated as **d**, continuously refreshes its text to display the progress of the app installation process.

Having identified different types of dynamic content changes in Android, our objective was to understand their impacts on screen reader users. To this end, we selected 4 apps that collectively represented all identified types of dynamic content changes. We then designed 5 specific tasks that would involve interactions with these dynamic changes. Our objective was to understand whether the users can perform the tasks, whether they can perceive the dynamic changes in the apps, and to generally develop a better understanding of how the dynamic changes impact app accessibility. To recruit participants, we leveraged the

Fable platform [101], which connects tech companies with disabled users for accessibility testing. Each user interview session was conducted over a one-hour period. During these sessions, we requested participants to share their phone screens and perform our designed tasks while vocalizing their thoughts and actions, aka think aloud [153]. This think-aloud method, combined with in situ questioning, enabled us to observe their understanding of the dynamic content changes and assess their ability to complete the tasks successfully. Our 3 blind participants included one female and two males, all of whom demonstrated a high proficiency in using the TalkBack.

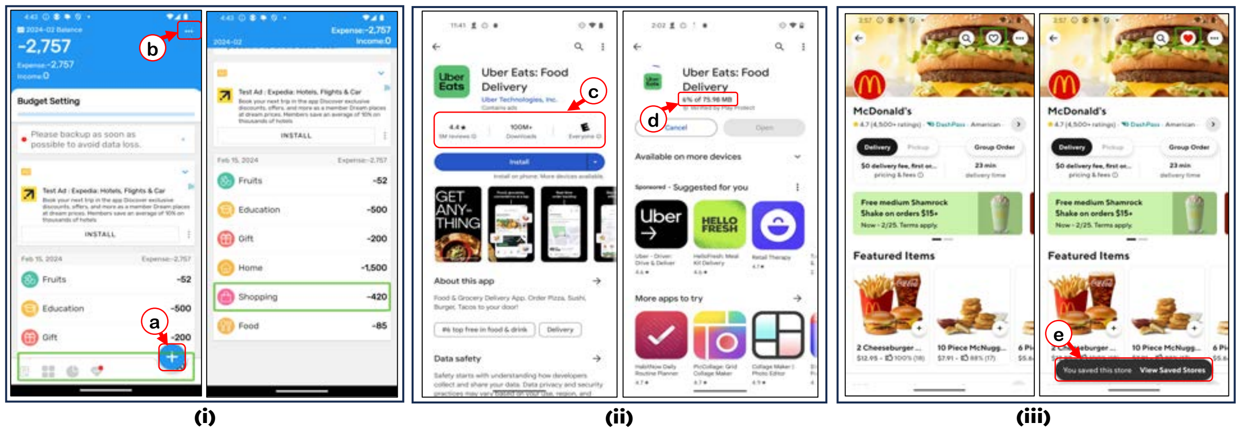


Figure 6.2: Examples of dynamic content changes: (i) the add button (annotated as a) and the more button (annotated as b) disappear when users continue exploring the screen, (ii) the app information (marked as c) moves to the bottom of the screen after hitting the Install button; the text (annotated as d) constantly changes to indicate installation progress, (iii) the short-lived notification (annotated as e at the bottom) after saving a restaurant.

6.3.2 Results

The user interviews focused on all the apps depicted in Figures 6.1 and 6.2. To generate a comprehensive list of accessibility issues related to dynamic content changes, two authors thoroughly examined each interview session. Initially, they independently identified areas where screen reader users faced confusion and attempted to ascertain the underlying reasons. Then, they engaged in discussions to reach a consensus. The following list outlines the dynamic content changes that proved challenging for screen reader users in our study.

Latent Appearing Content. When content appears without being annotated as a live region and is situated in an area previously explored by the user, it remains latent or unknown. For instance, the close button on an app's promotional page, as shown in Figure 6.1(f), emerges after a few seconds without alerting blind users. During the interview, blind users had already navigated past it, possibly interacting with elements located lower on the screen. This led to confusion when attempting to exit the promotional page, requiring users to employ various strategies such as using the TalkBack back gesture or re-navigating the screen.

Latent Disappearing Content. Content that vanishes before the user explores that region of the app stays undiscovered. In Figure 6.2(i), annotated buttons **b** and **a** disappear as users swipe through the list of items. The disappearance of the *add* button (element a) presented specific challenges for blind participants, as the button disappears in an unexplored area. As a result, they were unable to locate the element to add a new cost entry to the list. Conversely, the *more* button (element b) remains accessible, as users had already visited that element before its disappearance and when navigating backward, the more button becomes visible again.

Latent Short-Lived Content. When an element appears temporarily, it may be inaccessible to the user, especially if the element is actionable. The time it takes for the user to navigate to that element and perform the action might exceed the visibility period of short-lived content, leading to accessibility issues. In Figure 6.2(iii), we observe a brief notification annotated as **e** that emerges after users save a restaurant. Our user interviews revealed that participants were aware of this notification, understanding that they had successfully saved the restaurant and that the app offered an option to view all saved stores. However, this notification disappeared within a few seconds. For this type of dynamic content, participants only partially grasped the situation. While they recognized the appearance of the notification, thanks to the live region annotation, they did not fully understood its transient nature and were unable to click on the *View Saved Stores* button. One interviewee expressed, “It was a flash or pop up, and it went away. I would expect to be able to navigate to that button, but when I move back and forth, it is gone.”

Latent Moving Content. When the location of a previously visited element changes, it can cause confusion for blind users. As illustrated in Figure 6.2(ii), the app-related information, including app rating and download number, relocates to the bottom of the screen after pressing the install button. In our study, blind users were assigned the task of installing an app and finding its download number. After installation, they navigated backward, relying on the previous announcement by TalkBack about the download number during their journey to the install button. However, to their surprise, upon navigating back, the information they sought was no longer present, resulting in confusion and a period of being stuck on the page. None of the participants completed the task, with one participant believed that he could eventually locate the download number by navigating further down, acknowledging it would take more time.

Latent Content Modification. Changes in attributes of elements that are noticeable by sighted users may go unnoticed by users relying on screen readers. During the formative study, the `TextView` (element **d**) in Figure 6.2(ii) continuously updated its text to reflect the progress of the app installation. The proper implementation of the `liveRegion` feature ensured that changes in the `TextView` content were effectively announced to blind participants, enabling them to accurately comprehend the status of the installation process. Conversely, this suggests that if the `liveRegion` feature is not correctly implemented, it becomes challenging for screen reader users to understand content modification, such as the installation status. In addition. changes in attributes such as size and color, which are not perceptible by screen readers, do not impact their perception of the app.

6.4 Approach

Relying on the insights gained from formative interviews with screen reader users, we developed an automated framework called TimeStump, designed to identify accessibility issues associated with dynamic content changes. Figure 6.3 provides an overview of TimeStump, highlighting the three distinct phases of its operation. In the initial phase, we install an Android app on a Virtual Machine (VM) and utilize a GUI crawler to automatically explore the app, generating a diverse set of states in an app. The Snapshot Recorder tracks app states and records snapshots of distinct screens. In the second phase, we extract the list of actionable elements in each recorded snapshot. Then, the Interaction Automator systematically executes each action, capturing information before, during, and after the action. This rich dataset is passed to the third phase, where the Localizer component assesses the gathered information, precisely flagging accessibility issues stemming from dynamic content changes.

We now describe the details of each phase.

6.4.1 Phase 1: Capturing Unique App Screens

The main goal of this phase is to navigate through an app and explore its different states for subsequent examination. Interacting with the app leads to various state changes, ranging from subtle modifications in attributes, such as selecting a checkbox on the screen, to more significant changes like transitioning to an entirely new screen, resulting in a completely different hierarchical structure of elements. In this phase, our focus is on identifying a diverse set of screens from each app that have undergone significant changes. Detailed assessment of minor changes is reserved for subsequent phases.

To facilitate the testing of GUI apps, a variety of tools, such as STOAT [149], Monkey [77],

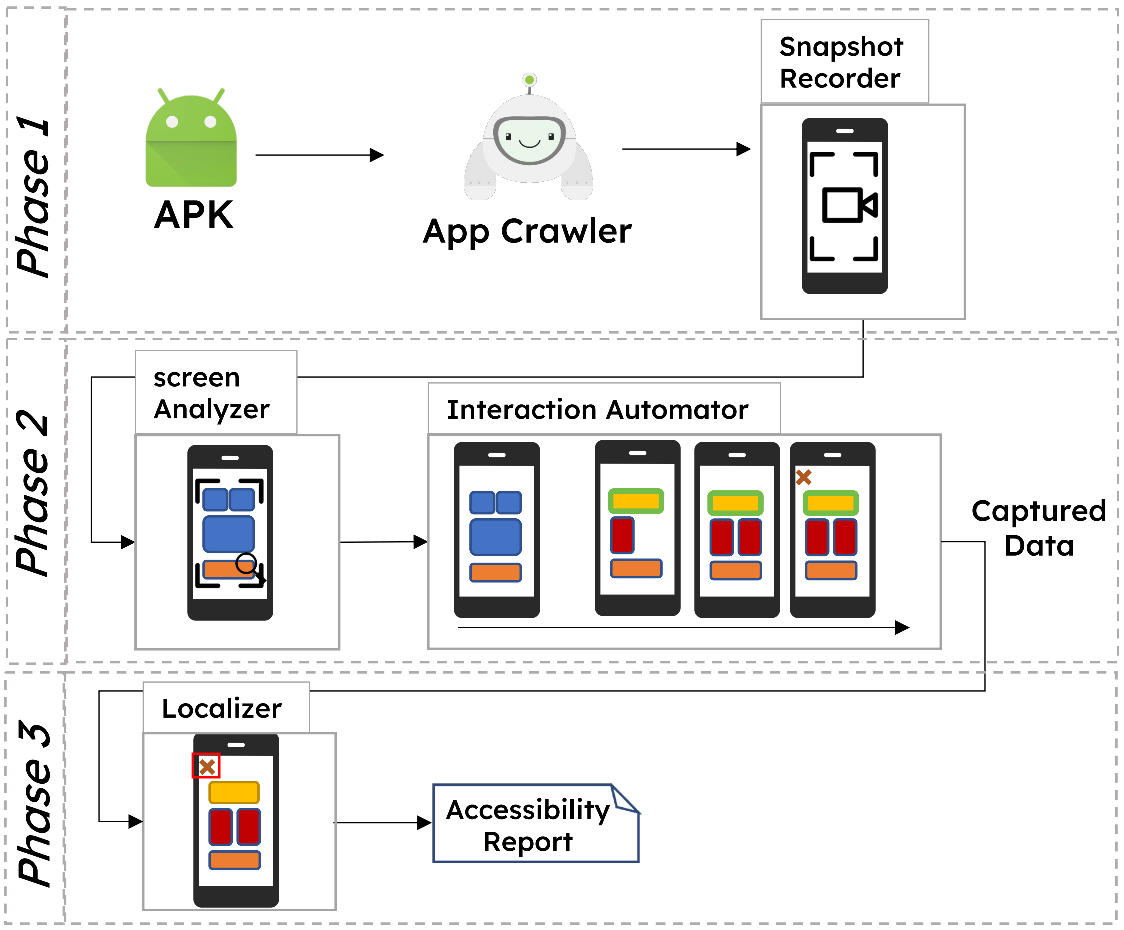


Figure 6.3: TIMESTUMP's approach overview.

Spaienze [108], and APE [80], have been specifically designed to traverse the expansive domain of an app's different states. TimeStump seamlessly integrates with any existing app exploration tool, providing the flexibility to traverse various app states. Additionally, TimeStump allows manual testers to explore the app with diverse scenarios in mind or leverage existing GUI test cases.

The Snapshot Recorder plays a pivotal role in tracking alterations. As the crawler interacts with the app, Snapshot Recorder keeps track of the screen structure and *Activities*, i.e., an Android component representing a single screen. It then captures VM snapshots from app states involving changes to activity names or the hash value of hierarchical structure of elements. Similar to previous studies [143], the hash function excludes nodes that are not important for accessibility, i.e., those not notified by screen readers, as well as attributes such as `checked` or `enabled` that do not contribute to recognizing a different screen in an app. VM Snapshots enable us to load the app from the exact state and perform further analysis.

6.4.2 Phase 2: Monitoring Apps in Action

During this phase, each potential interaction within a given app snapshot is automatically executed, all while monitoring the app for dynamic content changes.

To achieve this, we first load a VM snapshot. The Screen Analyzer employs an accessibility service to extract and dump the hierarchical structure of elements on the screen. The outcome is a tree structure wherein each node represents an element, accompanied by various attributes such as clickability. Parsing this node tree, the Screen Analyzer identifies all interactable elements and enumerates the types of actions they support, such as click, type, or swipe.

Every element is uniquely identified by its resource-id and a set of other attributes such as text, content description, and class name. The resource-id serves as a distinctive marker for locating each element. In instances where developers have not assigned a resource-id, the combination of other attributes can help in locating the element.

The Interaction Automator receives the comprehensive list of actions identified by the Screen Analyzer and executes them on the app while collecting certain data before, during, and after each action. The Interaction Automator consists of two main components: Controller and Accessibility Service.

The Controller functions as a server, sending commands to the client—the Accessibility Service, which operates in the background. The Accessibility Service is responsible for interacting with elements on the device. The client captures two frames of the app: *first frame* and *last frame*. The first frame corresponds to the initial state of the window, while the last frame corresponds to the app's state once all the content has finished rendering. The first frame primarily reflects the state of the app before any action takes place. However, if an action triggers a window change, this first frame then denotes the app's status immediately following the action. To accomplish this, TimeStump tracks accessibility events that signal either the loading of a new window or shifts in accessibility focus. When a window change occurs, the first frame is recorded immediately after detecting the event that indicates a change in the window. For example, in Figure 6.1, the TYPE_WINDOWS_CHANGED event, highlighted in purple, signifies a window transition. Consequently, Figure 6.1(c) is identified as the first frame. In the absence of such events, the first frame defaults to the state of the app before executing the action.

For the last frame, TimeStump listens for accessibility events indicating window content changes and, if none occur for more than 5 seconds, it captures that final state. For instance, Figure 6.1(f) is designated as the last frame, indicating that all changes on the app screen have been finalized. This practice is common in prior studies [143] and automation tools [79], ensuring app stability before data capture. The waiting time can be configured to be as long as necessary, or even adaptive to the specific app to optimize efficiency [65]. In instances of continual changes, such as animations or ads, a timeout period is implemented to bypass waiting.

The Controller stores these frames, as well as real-time logs of accessibility events generated by the Accessibility Service throughout the entire action execution period. Leveraging this extensive dataset empowers the Localizer to pinpoint accessibility issues arising from dynamic content changes.

6.4.3 Phase 3: Localizing Problematic Dynamic Changes

TimeStump analyzes the collected data to identify various categories of latent content changes for screen reader users. This analysis is conducted across captured frames of the app, as well as the accessibility events captured during the execution of each action on the screen.

When a screen element undergoes a change, it triggers an event of type `Window_Content_Changed`. TimeStump identifies sources of such events within the captured frames of the app, forming the initial set of candidate elements within a window that undergo a problematic change. The Localizer then compares elements in the final frame against those in the first frame to pinpoint the problematic changes.

As explained in Section 6.4.2, if the execution of an action results in a window transition, the first frame is the newly loaded window, i.e., the frame captured immediately after performing the action, similar to Figure 6.1(c). To detect window transitions, Localizer examines accessibility events of type `Windows_Changed` and `Window_State_Changed`, which are also used in the Android source code to detect the appearance of a new window [78]. Subsequently, we elaborate on the logic employed for detecting various types of problematic changes.

Due to space limits, we provide an intuitive explanation of how TimeStump localizes each issue here, and provide the detailed algorithms on the companion website [30].

Latent Appearing Content: As explained in Section 6.2 and Section 6.3, if certain content appears in previously explored areas (i.e., above the accessibility focus in default navigation order) and is not designated as a live region, it remains unknown to the screen reader user. The Localizer classifies elements in the final frame that trigger a content change event as latent appearing content if (1) they do not appear in the first frame, (2) are not designated as live regions, and (3) are positioned before the current accessibility focus.

Latent Disappearing Content: When an element disappears from the screen, a change event is triggered, similar to the case of appearing content. However, in this scenario, the event's source is the container of the vanishing element. For example, if a button within a linear layout disappears, the event source will be the linear layout, potentially covering the entire screen. Consequently, the Localizer evaluates all the children of an event publisher node in the first frame to verify their presence in the final frame. A child node is categorized as latent disappearing content if (1) it is not observed in the last frame, (2) it is not designated as a live region, and (3) it is positioned after the current accessibility focus.

Latent Short-lived Content: Elements that appear and disappear have a brief visibility period. Even if this content is announced, navigating to them and interacting with them using screen readers is challenging. The Localizer identifies these elements by searching for pairs of change events and localizing their sources denoted as <S1, S2> in two consecutive frames, checking whether S2 is the container of S1. For any such found pair, an element S1 is categorized as latent short-lived content if (1) S1 is not present in the first frame, (2) its container S2 is observed in the second frame, and (3) S1 is not designated as a live region or is actionable.

Latent Moving Content: The Localizer examines elements displaying a shift in their position on the screen across different frames while maintaining consistent identifiers such as resource-id, and content description. Elements are flagged as problematic if their changed position is above the accessibility focus or if they move beyond the screen boundaries.

Latent Content Modification: The change of attributes in an element refers to any modification in the properties that define an element's behavior, or metadata within a UI. Localizer uses a hash function to detect such modifications across different frames. The hash function encodes the element attributes that are important in exploring the app with screen readers. A discrepancy in the hash values of an element between any two frames signifies a modification in the element's content. When the `liveRegion` attribute is absent, the change remains unknown to screen reader users.

6.5 Evaluation

We evaluated TimeStump on real-world apps and with the help of several blind users to answer the following questions:

RQ1. How accurate is TimeStump in detecting dynamic content changes and different categories of accessibility issues?

RQ2. How do the issues reported by TimeStump impact the screen reader users?

RQ3. What is the performance of TimeStump? Can TimeStump efficiently be used in practice?

6.5.1 RQ1. Accuracy of TimeStump

Experimental Setup

For this experiment, we utilized STOAT [149] as the app exploration tool. We evaluated our approach on 30 real-world Android apps. Our test set consists of two groups of apps: (group1) 10 apps with manually verified dynamic content changes from different categories of Google play store, (group2) 20 randomly selected apps with known accessibility issues from a prior study [143]. For apps in group1, the authors installed top rated apps in different categories of Google play store and manually explored each app, looking for dynamic content. They captured a VM snapshot of each app at that state, with the accessibility focus set on the target element such that performing the action on the target element results in the dynamic change. For apps in group2, we set the crawler to automatically get VM snapshots from two random unique states of the app. The tool then explores the actionable elements in each state to get the real-time data.

The precision of the captured data directly impacts the ability of TimeStump to detect changes in dynamic content. Before evaluating TimeStump's effectiveness, we manually reviewed the captured data from a test set to confirm alignment with our definitions of the first and last frames, and encountered no issues. For this experiment, we chose 15 unique app states from 5 random apps from a prior study [65], encompassing a range of app transitions such as implicit loading, explicit loading, and transitions. Additional information about this study can be found on the companion website [30].

All experiments were conducted on a typical computer setup for development (MacBook Pro, Apple M1 Max, 32 GB memory). We used the most recent distributed Android OS (SDK34), and the latest versions of Android screen reader.

Results

To answer this question, the authors manually examined the issues reported by TimeStump and tagged them as False Positive (FP) if the reported issue is not correct, and True Positive (TP) if the reported issue correctly detects and categorizes problematic dynamic content changes. The authors used an emulator to load the captured snapshot and manually interacted with the app using TalkBack. This process allowed for the identification of legitimate dynamic elements in exploring the app with screen readers. We then report precision as the ratio of the number of TPs to the number of all detected issues. As shown in Table 6.1, the overall precision over all the elements in 130 actions of apps is 0.94. To compute recall, we manually reviewed apps in both group 1 and group 2 to identify dynamic changes and establish the ground truth. In group 1, snapshots were captured during manual exploration of the app, revealing states with dynamic content changes. In contrast, for group 2 apps, we manually inspected automatically captured snapshots of random app states for dynamic content changes. Dynamic elements missed by TimeStump were identified and manually labeled as False Negatives (FN). The overall recall across 130 actions is 0.92, as depicted in Table 6.1.

Figure 6.4 presents examples of problematic dynamic elements detected by TimeStump. In Figure 6.4(a), the four elements highlighted by orange boxes emerge above the accessibility focus after tapping on the *plus* button. Figure 6.4(b) illustrates short-lived elements, including a button, indicated by blue boxes, appearing after adding a song to favorites. As all of those elements are actionable, TimeStump reports them as problematic. In Figure 6.4(c), a `TextView`, annotated by the black box, is updated following the tap on the *CALCULATE* button. Since this element is not tagged as live region, it remained unannounced while exploring the app with TalkBack. Thus, TimeStump reports it as problematic.

We analyzed the failures of TimeStump and identified issues falling into two main themes, resulting in both false positives and false negatives.

The first pattern relates to inadequate identifiers assigned by developers to elements. For instance, in the Fuelio app, a `FrameLayout` serves as a container for its child elements, lacking essential identifiers such as `resource_id`, `text`, and `content_description`. As a result, its unique identification relies solely on its screen bounds. However, when an action triggers a layout modification, the element's screen bounds also change, leading TimeStump to mistakenly interpret this as an appearing element, thus generating a false positive. Moreover, the absence of sufficient identifiers can result in false negatives. In the ESPN app, certain elements possess identifiers that are neither empty nor unique. TimeStump primarily depends on these identifiers to match an element, but since they are not distinctive, TimeStump fails to differentiate between elements on the screen, consequently failing to report associated issues.

Another category of failures occurs when the screen displays multiple windows, such as step-by-step guidelines overlaid on the main app. ADB allows us to capture the `AccessibilityNodeTree` of the foremost window only, thereby missing content in other windows. This limitation contributes to both false positives and false negatives.

6.5.2 RQ2. Qualitative Study

To assess the impact of the issues detected by TimeStump on screen reader users, we conducted 15 user studies. We randomly selected three apps from RQ1, representing various types of dynamic content changes, corresponding to IDs *P8*, *G1*, and *G7*, as listed in Table 6.1. Our qualitative study consisted of 10 self-guided tasks and 5 user interviews with blind testers, recruited through the Fable platform [101]. In the self-guided tasks, testers were given concise task descriptions to execute offline on apps while recording their screens and articulating their thoughts aloud. This approach, without a moderator present during

Table 6.1: The accuracy of TimeStump on subject apps

ID	App	Category	#Installs	# Issues	TP	FP	FN	Precision	Recall
P1	Autozone	Auto & Vehicles	>5M	5	5	1	0	0.83	1
P2	Duolingo	Education	>500M	19	18	1	1	0.94	0.94
P3	Forest	Productivity	>10M	7	5	0	2	1	0.71
P4	Gratitude	LifeStyle	>1M	8	8	0	0	1	1
P5	Motivation	Health & Fitness	>5M	3	3	0	0	1	1
P6	Starbucks	Food & Drink	>10M	4	4	1	0	0.8	1
P7	TicketMaster	Events	>10M	1	1	0	0	1	1
P8	Spotify	Music & Audio	>1B	5	4	0	1	1	0.8
P9	H&M	Lifestyle	>50M	1	1	0	0	1	1
P10	File Manager	Tools	>1B	20	11	0	9	1	0.55
G1	Booking.com	Travel & Local	>500M	39	36	0	3	1	0.92
G2	Easy Bills Reminder	Finance	>100K	2	2	0	0	1	1
G3	Burn	Education	NA	6	2	0	4	1	0.66
G4	Dictionary.com	Books & Reference	>10M	4	4	0	0	1	1
G5	ESPN	Sports	>50M	3	2	0	1	1	0.66
G6	Calorie Counter by FatSecret	Health & Fitness	>50M	58	58	5	0	0.92	1
G7	Fuelio	Auto & Vehicle	>1M	101	92	5	9	0.94	0.91
G8	Life360	Lifestyle	>100M	4	4	0	0	1	1
G9	Master Lock Vault Enterprise	Lifestyle	>100K	4	1	0	3	1	0.25
G10	Nike	Shopping	>50M	25	24	0	1	1	0.96
G11	Weee! Asian Grocery Delivery	Food & Drink	>1M	8	8	0	0	1	1
G12	Norton Secure VPN	Tools	>10M	2	2	0	0	1	1
G13	TripIt	Travel & Local	>5M	7	7	1	0	0.87	1
G14	ToonMe photo cartoon maker	Photography	>50M	2	2	0	0	1	1
G15	Vimeo	Entertainment	>10M	6	6	0	0	1	1
G16	Yelp	Food & Drink	>50M	1	1	0	0	1	1
G17	The Clock	Productivity	>10M	56	55	11	1	0.83	0.98
G18	King James Bible	Books& Reference	>50M	23	21	2	2	0.91	0.91
G19	Lyft	Maps & Navigation	>50M	11	11	0	0	1	1
G20	To-Do List - Schedule Planner	Productivity	>10M	50	47	2	3	0.95	0.94
Overall				485	445	29	40	0.94	0.92

sessions, helped mitigate interviewer bias. Additionally, user interviews were conducted to explore incidents in different states of one app, *G7*, and ask follow-up questions. Due to the limited size of the tester pool on the platform, some tasks involving different apps were assigned to the same tester. However, no tester evaluated the same app multiple times. In total, 8 distinct testers participated in this study: 6 males, 2 females, with 7 identifying as White and 1 as Asian. Participant ages ranged from 20 to 45. Below, we first outline the issues that users confirmed. We then discuss the observed shortcomings and insights gained.

User Confirmed Issues

Of the 30 issues identified, users directly confirmed 25, yielding a confirmation rate of 0.83.

Appearing elements in explored areas. Figure 6.4(a) depicts an instance from this category. Blind testers were tasked with locating the `Gas` entry button, which appears after tapping the `Plus` button. The target button, along with three other elements highlighted with orange boxes on Figure 6.4(a), emerged in areas previously explored by users, without any notification. Consequently, users felt as if nothing had changed after tapping the `Plus` button. One participant expressed, *"It's very confusing and disorienting when the screen changes without any audio feedback from the screen readers"* Another noted, *"Usually, new elements appear below [the current TalkBack focus], but in this case, they appeared above."* The appearance of elements in previously explored areas caused confusion for screen reader users, resulting in longer times to locate the desired element. Two out of five interviewees were unable to find the targeted button and complete the task, while others had to explore the screen multiple times to do so. A similar issue in self-guided tasks resulted in confusion for all the participants. For elements that appear dynamically, blind testers recommended setting the `liveRegion` attribute appropriately. They suggested moving the TalkBack focus to the first new element on the screen in cases of significant window changes. For minor window changes, introduce dynamic elements in unexplored screen areas.

Disappearing elements in unexplored areas. In one of the test apps, activating a switch at the top caused some form entries to disappear. Testers interacting with the switch were not informed of the changes and were confused as to why they could not find certain elements. Four out of five interviewees were unable to complete the task, concluding that the required element was not present on the screen. Conversely, one interviewee managed to find the desired element by turning off a switch, leveraging his prior experience with such controls. Among the interviewees who failed to perform the task, one person remarked, *"[I] Thought the Recurrence section was either not on the screen or not visible to TalkBack. I just couldn't find it."* The tester expressed a preference for receiving a notification indicating that *"new controls are available or shown"* once the checkbox is ticked. This would enable them to recognize that the layout of the app has changed on the same screen and understand how to revert the layout to its original configuration.

Short-lived buttons. As depicted in Figure 6.4(b), when users add a song to their favorites, a notification pops up and let them revert the action by tapping on the `Change` button. Three of the users became aware that they could potentially use this element. Two participants missed the button because TalkBack simultaneously announced three short-lived elements, which overwhelmed them. Moreover, during self-guided tasks, none of the participants could interact with the `Change` button as it disappeared quickly. As a result, three participants could not accomplish the task for removing a song from the favorites. Two participants were able to remove the song through an alternative method, using the ticked button, annotated by a green box in Figure 6.4(b). Therefore, short-lived elements should not overwhelm blind users with excessive information. Additionally, it is recommended to avoid including clickable elements in a short-lived manner, as blind users navigating sequentially with a screen reader are likely to miss them before they vanish.

Unannounced Short-lived Elements. In the Spotify app, when users removed a song from their favorites, a short-lived notification, with the text *Removed from Liked Songs*, appeared signaling this change, providing immediate feedback to sighted users. However, TalkBack did not announce the change to the screen reader user, leading to confusion. Only two participants advanced to the step of removing a song from the favorites in our self-guided tasks. For those screen reader users, they were uncertain if the song had been successfully removed and felt compelled to navigate through the entire screen to verify that. When a song has not been added to favorites, the `Plus` button is labeled with the content description *Add Item.* Conversely, when the song is in the favorites, its description changes to *Item Added.* As a result, the blind users need to navigate the screen to check if the content description has reverted to *Add Item.* in order to confirm that their action was successful. Their experience suggested that the `liveRegion` attribute should be appropriately configured for short-lived notifications.

Unannounced Content Modification In Figure 6.4(c), tapping the `CALCULATE` button triggers an update of the result, indicated by the black box, appearing above the button.

However, this change is not communicated to screen reader users, forcing them to navigate back to check the calculation result. Although all participants in our self-guided tasks managed to find the calculation result by navigating back and forth, they reported it as confusing and inconvenient. Additionally, if users press the `CALCULATE` button without providing any input for prior entries, they receive an error message advising them to input values before proceeding. One participant noted, *"For the sake of consistency, having the calculation result announced just like the error message would not disappoint me."* Proper utilization of the `liveRegion` attribute would alleviate such issues.

Observed Shortcomings

The user study also shed light on TimeStump's shortcomings and enhancement opportunities.

Reverting Changes. TimeStump evaluates the changes resulting from each action. However, a series of actions may have counteractive impacts. For instance, For example, in the Spotify app, top views move up as users navigate toward the bottom. As soon as they attempt to navigate back to those top elements, the views are restored to their original position, and users do not perceive any problem. Our manual exploration of our test set reveals 8 elements with similar issues that may not be problematic for users. However, for users who rely on alternative interaction modes, such as *explore by touch* where TalkBack shifts its focus to the coordinates of the touch gesture, these cases can still be confusing.

Severity of Issues. The dynamic elements identified as problematic exhibit varying degrees of severity and impact on blind users, with TimeStump unable to prioritize them by severity. Factors such as the frequency of the issue among different apps and users' familiarity with it contribute to its severity. For instance, during interactions with the Booking.com app, switching tabs changes the content of the window without altering the screen reader focus or providing any notification. While this issue caused confusion for participants, they relied on their intuition and manually adjusted the TalkBack focus to the top of the screen to access the new content. However, all participants expressed that it would be helpful if the focus were automatically moved to the newly appeared content. Another factor influencing the severity of the issues is the distance of the changed element from the accessibility focus. In Figure 6.4 (c), navigating one element back and forth could help users find the results, while if the appearance of the result is far from the current focus, it may become impossible for users to locate it.

Navigation Order. TimeStump relies on the default navigation order of elements for TalkBack to determine if a dynamic change is problematic. However, users may have their own interaction preferences when using screen readers. In our study, some users rely on their prior knowledge and tap on specific parts of the app to find the requested element. One interviewee mentioned that it would be helpful if the change was announced, but he could still locate the dynamically updated `TextView`. Additionally, although customization of the navigation order of elements was not observed in the apps used in our experiments, it is important to note that developers can override the default TalkBack navigation order, e.g., allowing the topmost element to be designated as the last element focused by TalkBack on a screen. If so, the button in Figure 6.1(e), circled in red, would not trouble blind users as it would be in an unexplored area.

6.5.3 RQ3. Performance

The performance evaluation of our tool, TimeStump, is structured into three phases: app screen capturing (Phase 1), monitoring apps in action (Phase 2), and localizing problematic dynamic changes (Phase 3). In RQ1, Phase 1 involves automated capturing of two distinct app states from group2 apps, requiring an average of 167 seconds using STOAT [149]. Phase 2's analysis of each state for the number of actions is rapid; however, executing each action,

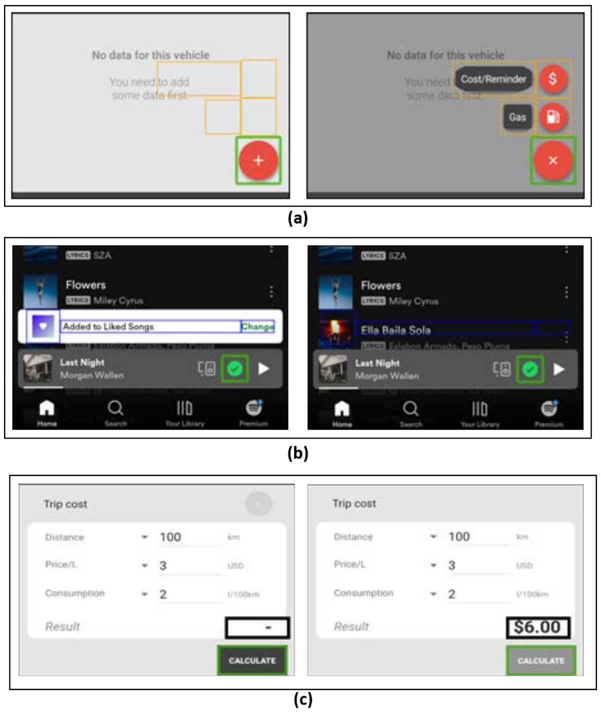


Figure 6.4: Examples of detected issues by TimeStump: (a) Appearing Content, (b) Short-Lived Content, and (c) Content Modification

capturing data, and transferring it to the server consumes about 66 seconds per action on average. The bulk of this time, approximately 20 seconds, is dedicated to dumping and transferring data, especially the captured video for each action. Developers can opt to disable video capturing in the tool, relying instead on screenshots, to significantly improve the tool's performance. In Phase 3, the post-analysis of the technique includes analysis of collected frames as well as the accessibility events, completing in approximately 7.5 seconds for each action.

6.6 Threats to Validity

External Validity. In this study, we examined dynamic content changes following action execution or screen transition. However, ad-related pop-ups or random rating requests may appear without user actions. Investigating these requires analyzing the source code and library calls to find them. Future research could focus on identifying and understanding these instances.

Another concern is the completeness of our work, both in terms of the types of dynamic content changes and the challenges they pose. We carefully selected and manually explored a diverse range of apps to identify the types of dynamic change, ensuring these aligned with web testing definitions related to element structure and attribute modifications.

Moreover, we utilized interviews to pinpoint scenarios where different types of dynamic content change might pose issues for screen reader users. Although our initial findings were extracted from three interviews, the subsequent user study in RQ2 reaffirms the validity of our conclusions.

Similarly, a concern related to RQ1 is the completeness of the identified accessibility issues. The ground truth was manually created due to the lack of preexisting datasets. To validate the manual construction of the ground truth, two authors independently reviewed the snapshots, including the `AccessibilityNodeInfo` tree and `AccessibilityEvents`, to identify problematic dynamic changes. Subsequently, they engaged in discussions to ensure agreement in their evaluations.

Internal Validity. TimeStump integrates various libraries and tools, including Stoat, ADB, AVD, and AccessibilityService, raising potential risks of defects. Additionally, there is a possibility of defects in our prototype's implementation. To counteract, we utilized the latest version of third-party tools, conducted Github code reviews, and tested on varied apps. We also assessed data capture accuracy on apps with different transitions, detailed on our website [30]. For rigorous testing, we used different sets of apps for our formative studies, accuracy evaluations, and tool assessments.

6.7 Related Works

Web Pages: Web accessibility testing primarily relies on the WCAG guidelines [159]. These guidelines have led to the development of tools assessing web page accessibility compliance [41, 69, 7, 37, 173, 6]. However, the guidelines overlook various accessibility challenges encountered by assistive technology users, especially in the context of dynamic changes. While a few criteria mandate developers to ensure dynamically displayed error/success messages are accessible to all, they fail to address other issues arising from dynamic content changes. Existing tools [41, 69, 7, 37, 173, 6] cover only a fraction of the standards, thus inadequately detecting these issues on web pages.

To address limitations of guidelines, dynamic techniques have been proposed to assess apps while interacting with them. They resulted in studies that detect accessibility issues during interactions with web pages [152, 39, 62] or evaluate and infer correct accessibility attributes, like ARIA labels [164], for web content [61, 35]. In the evaluation of interaction issues, recent studies have attempted to utilize assistive technologies, similar to how an end user explores the app. They also account for changes introduced by JavaScript by evaluating multiple states that a single web page can take [171, 66, 67, 145, 48, 49, 50]. These studies focus on interaction failures which are only a subset of the challenges posed by dynamic changes. While exploring the app dynamically, they overlook real-time changes, like unnannounced buttons appearing in previously explored areas. Furthermore, these studies miss changes like content modification that does not involve altering the DOM structure.

Mobile Apps: Similar to web accessibility testing, various automated tools are designed for mobile apps to assess specific app states and report their adherence to accessibility guidelines [23, 17, 84, 22, 134, 64, 47, 94, 56, 102]. Recognizing the limitations of accessibility guidelines and the unique interaction modes of assistive technologies, recent studies have focused on identifying inaccessible content by utilizing assistive technologies to navigate various app states and comparing it with exploring the app without assistive technologies [150, 143, 111, 139, 11, 14, 141]. However, no technique tackles the challenges of dynamic content changes.

6.8 Conclusion

The broad impacts of dynamic content changes on accessibility issues have not been thoroughly examined in prior research. We presented TimeStump, an automated framework identifying accessibility issues due to dynamic screen changes. TimeStump navigates through app states, collects data before, during and after each action, and applies a set of rules to detect dynamic screen changes that may lead to accessibility problems for the blind. An empirical study on real-world apps and a user study with blind participants prove its efficacy.

Future directions involve extending our work to ad-related pop-ups and unexpected rating requests that may appear without user actions, and expanding our implementation to other platforms, such as Web and iOS.

Our research artifacts are available publicly [30].

Chapter 7

Conclusion

In this dissertation, I proposed advancements in automated techniques to repair and test accessibility issues for screen reader users. To address the common problem of missing labels in Android apps, I developed COALA, a context-aware label generation technique for unlabeled icons. I also demonstrated that while prior automated tools can easily detect issues like missing labels, they fail to identify various other accessibility issues that screen reader users encounter. To bridge this gap, I introduced AT-aware and time-aware accessibility testing, which automatically detects issues that only arise during runtime when interacting with the app using assistive technologies.

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

7.1 Research Contribution

- **Introducing notable factors in assessing and repairing accessibility issues** In advancing automated accessibility testing and repair, I introduced three critical parameters essential for comprehensive accessibility assessment. I demonstrated how considering app context can enhance automated icon labeling. Additionally, I highlighted Assistive Technologies and Time as crucial factors for detecting accessibility issues that only manifest at runtime. I designed and built the proposed tools for the Android platform and evaluated them on real-world, popular apps.

- **Automated tools.** I designed automated tools to repair and test accessibility issues that prior techniques were unable to detect. The source code for these tools is publicly available, facilitating adoption by different companies and across various platforms.

- **User study.** I conducted user studies to assess the significance of the issues and the effectiveness of the proposed tools in a realistic context. Insights from these studies can guide other researchers to focus on previously unexplored and significant accessibility problems.

- **Dataset.** Each of the accessibility challenges highlighted in my work is accompanied by a dataset to evaluate the effectiveness of my automated tools. These datasets include 60 apps with under-access problems, 30 apps with over-access problems, and 30 apps with problematic dynamic content changes. These datasets pave the way for future researchers to build upon and enhance my work in this domain.

7.2 Future Work

Fixing Accessibility Issues While COALA addresses the most common accessibility issue for screen reader users, many other issues still need to be fixed. Single-purpose solutions are not capable of addressing the wide range of issues present in mobile apps. In this dissertation, I highlighted critical issues such as over-accessibility, under-accessibility, and latent dynamic changes. We can develop solutions for these problems and use my accessibility testing tools to verify the automated fixes. Additionally, to ensure that fixing one issue doesn't introduce new ones, we can explore fix architectures that tackle multiple types of issues simultaneously.

Automated testing of usability and accessibility Accessibility issues are inherently challenging to repair as their fixes can impact the design, ease of use, and aesthetics of the app. For example, WCAG guideline 1.4.4 requires apps to support text resizing up to 200%. Reordering elements to provide enough space for each one on the screen might break the app's integrity. We need automated testing techniques that can not only fix the accessibility issues but also ensure the preservation of app functionality and usability. Multi-objective testing techniques can be a promising direction to extend my single-purpose accessibility testing tools. These tools can also serve as an automated oracle to evaluate the effectiveness of automatically generated fixes.

Early integration of accessibility in software development An effective way to encourage developers to build more accessible apps is to assist them in incorporating accessibility from the early stages of software development and testing. This approach reduces the cost of building, testing, and repairing accessible apps. Currently, design tools allow specifying accessibility annotations such as focus order. Assessing and recommending such annotations can ensure that accessibility is a priority rather than an afterthought. Additionally, integrating testing tools into app-building tools can be beneficial. At present, developers can simulate the design of each screen. Providing a simulation of interacting with these designs using ATs can help developers understand and address accessibility problems more quickly, as they will know where to modify the code.

Optimize Accessibility Crawler through Hybrid approaches Evaluating accessibility issues for each interaction in an app dynamically is very expensive. In OverSight, I demonstrated a hybrid approach by locating over-accessibility smells statically and then assessing them using ATs at runtime. This approach can be extended by identifying patterns in UI specifications that potentially lead to accessibility issues. Advancements in AI and existing datasets of various accessibility issues can assist in learning these patterns and locating them in apps.

Bibliography

[1] Xml path language. `https://www.w3.org/TR/2017/REC-xpath-31-20170321/`, 2020. Last Accessed: May 6, 2022.

[2] Accessibility testing framework for android. `https://github.com/google/Accessibility-Test-Framework-for-Android`, 2021.

[3] Functional images. `https://www.w3.org/WAI/tutorials/images/functional/`, 2021.

[4] Labeldroid - icse best paper award. `https://tinyurl.com/yxndyovk`, 2021.

[5] A guide to disability rights laws. `https://www.ada.gov/cguide.htm`, August 20, 2020.

[6] accessiBe. accessscan - website accessibility checker - free & instant - accessibe. https://accessibe.com/accessscan, 2023. Last Accessed: December 5, 2023.

[7] A. M. Agency. Access monitor plus. https://accessmonitor.acessibilidade.gov.pt/, 2021. Last Accessed: December 5, 2023.

[8] 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.

[9] 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.

[10] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond. Automated repair of size-based inaccessibility issues in mobile applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 730–742, Virtual, Australia, 2021. IEEE, IEEE.

[11] 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.

[12] 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.

[13] 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*. IEEE, 2020.

[14] A. Alshayban and S. Malek. Accessitext: automated detection of text accessibility issues in android apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 984–995, 2022.

[15] Y. Amit. Accessibility clickjacking–android malware evolution.(2016), 2018.

[16] Android. About switch access for android. `https://support.google.com/accessibility/android/answer/6122836?hl=en`, 2020.

[17] Android. Accessibility scanner - apps on google play. `https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US`, 2020. Last Accessed: May 6, 2022.

[18] Android. Accessibilityservice in android. `https://developer.android.com/guide/topics/ui/accessibility/service`, 2020. Last Accessed: May 6, 2022.

[19] Android. Talkback and switchaccess source code by google. `https://github.com/google/talkback`, 2020. Last Accessed: May 6, 2022.

[20] Android. Test your app's accessibility. `https://developer.android.com/guide/topics/ui/accessibility/testing`, 2020. Last Accessed: August 20, 2020.

[21] Android. Build more accessible apps. `https://developer.android.com/guide/topics/ui/accessibility`, 2021.

[22] Android. Espresso : Android developers. `https://developer.android.com/training/testing/espresso`, 2021. Last Accessed: May 6, 2022.

[23] Android. Improve your code with lint checks. `https://developer.android.com/studio/write/lint`, 2021. Last Accessed: May 6, 2020.

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

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

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

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

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

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

[30] Anonymous. Timestump companion website. https://github.com/timestump/timestump, 2024. Last Accessed: Mar 7, 2024.

[31] Apple. Apple accessibility - iphone. `https://www.apple.com/accessibility/iphone/`, 2020.

[32] Apple. Content. `https://developer.apple.com/design/human-interface-guidelines/accessibility/overview/content/`, 2021.

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

[34] Apple. Apple accessibility scanner. `https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html`, 2022.

[35] M. Bajammal and A. Mesbah. Semantic web accessibility testing via hierarchical visual analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1610–1621. IEEE, 2021.

[36] S. Banerjee and A. Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005.

[37] C. Benavidez. Examinator, 2015.

[38] Y. Bengio, P. Simard, P. Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 1994.

[39] J. P. Bigham, J. T. Brudvik, and B. Zhang. Accessibility by demonstration: enabling end users to guide developers to web accessibility solutions. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, pages 35–42, Orlando, USA, 2010. Association for Computing Machinery.

[40] R. Bourne, J. Adelson, S. Flaxman, P. Briant, M. Bottone, T. Vos, K. Naidoo, T. Braithwaite, M. Cicinelli, J. Jonas, H. Limburg, S. Resnikoff, A. Silvester, V. Nangia, and H. Taylor. Global prevalence of blindness and distance and near visual impairment in 2020: progress towards the vision 2020 targets and what the future holds. *Investigative Ophthalmology and Visual Science*, 2020.

[41] G. Broccia, M. Manca, F. Paternò, and F. Pulina. Flexible automatic support for web accessibility validation. *Proceedings of the ACM on Human-Computer Interaction*, 4(EICS):1–24, 2020.

[42] 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.

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

[44] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676, 2018.

[45] 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. IEEE, ICSE.

[46] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[47] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu. Accessible or not an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 48:3954–3968, 2021.

[48] 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*, ESEC/FSE 2021, pages 855–867, Virtual, Athens, Greece, 2021. ACM New York, NY, USA.

[49] P. T. Chiou, A. S. Alotaibi, and W. G. Halfond. Bagel: An approach to automatically detect navigation-based web accessibility barriers for keyboard users. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2023.

[50] P. T. Chiou, A. S. Alotaibi, and W. G. Halfond. Detecting dialog-related keyboard navigation failures in web applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1368–1380. IEEE, 2023.

[51] 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.

[52] 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.

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

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

[55] Y. Cui, M. Jia, T.-Y. Lin, Y. Song, and S. Belongie. Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[56] H. N. da Silva, S. R. Vergilio, and A. T. Endo. Accessibility mutation testing of android applications. *Journal of Software Engineering Research and Development*, 10:8–1, 2022.

[57] R. J. P. Damaceno, J. C. Braga, and J. P. Mena-Chalco. Mobile device accessibility for the visually impaired: problems mapping and recommendations. *Universal Access in the Information Society*, 2018.

[58] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology*, 2017.

[59] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[60] 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.

[61] C. Duarte, A. Salvado, M. E. Akpinar, Y. Yeşilada, and L. Carriço. Automatic role detection of visual elements of web pages for automatic accessibility evaluation. W4A '18, New York, NY, USA, 2018. Association for Computing Machinery.

[62] F. Durgam, J. Grigera, and A. Garrido. Dynamic detection of accessibility smells. *Universal Access in the Information Society*, pages 1–12, 2023.

[63] 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*. IEEE, 2018.

[64] 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.

[65] S. Feng, M. Xie, and C. Chen. Efficiency matters: Speeding up automated testing with gui rendering inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 906–918. IEEE, 2023.

[66] N. Fernandes, D. Costa, C. Duarte, and L. Carriço. Evaluating the accessibility of web applications. *Procedia Computer Science*, 14:28–35, 2012.

[67] N. Fernandes, D. Costa, C. Duarte, and L. Carriço. Evaluating the accessibility of web applications. *Procedia Computer Science*, 14:28–35, 2012.

[68] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057, 2017.

[69] G. Gay and C. Q. Li. Achecker: open, interactive, customizable, web accessibility checking. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, pages 1–2, Raleigh, USA, 2010. Association for Computing Machinery.

[70] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

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

[72] Google. Get started on android with talkback - android accessibility help. `https://support.google.com/accessibility/android/answer/6283677?hl=en`, 2020. Last Accessed: May 6, 2022.

[73] Google. Accessibilityflags. `https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo#attr_android:accessibilityFlags`, 2022. Last Accessed: March 16, 2022.

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

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

[76] 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.

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

[78] Google. clickandwaitfornewwindow. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/cmds/uiautomator/library/core-src/com/android/uiautomator/core/InteractionController.java#252, 2024. Last ACCESSED: Feb 18, 2024.

[79] Google. Dumpcommand. https://android.googlesource.com/platform/frameworks/testing/+/jb-mr2-release/uiautomator/cmds/uiautomator/src/com/android/commands/uiautomator/DumpCommand.java#87, 2024. Last Accessed: Feb 18, 2024.

[80] 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.

[81] J. T. Hancock and T. M. Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 2020.

[82] V. L. Hanson and J. T. Richards. Progress on website accessibility? *ACM Transactions on the Web*, 2013.

[83] 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*, 2014.

[84] 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.

[85] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[86] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[87] 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.

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

[89] R. Jabbarvand, F. Mehralian, and S. Malek. Automated construction of energy test oracles for android. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 927–938, 2020.

[90] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–115, 2014.

[91] S. K. Kane, J. A. Shulman, T. J. Shockley, and R. E. Ladner. A web accessibility report card for top international university web sites. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility*, 2007.

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

[93] S. Khandelwal. New ransomware not just encrypts your android but also changes pin lock, Oct 2017.

[94] KIF. Keep it functional - an ios functional testing framework. https://github.com/kif-framework/KIF, 2023. Last Accessed: December 7, 2023.

[95] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

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

[98] K. J. Koswara and Y. D. W. Asnar. Improving vulnerability scanner performance in detecting ajax application vulnerabilities. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–5. IEEE, 2019.

[99] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao. On malware leveraging the android accessibility framework. In I. Stojmenovic, Z. Cheng, and S. Guo, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 512–523, Cham, 2014. Springer International Publishing.

[100] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[101] F. T. Labs. Fable — digital accessibility, powered by people with disabilities. https://makeitfable.com/, 2023. Last Accessed: December 7, 2023.

[102] L. Li, R. Wang, X. Zhan, Y. Wang, C. Gao, S. Wang, and Y. Liu. What you see is what you get? it is not the case! detecting misleading icons for mobile applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 538–550, 2023.

[103] C.-Y. Lin and E. Hovy. Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003.

[104] 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.

[105] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, 2017.

[106] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar. Learning design semantics for mobile apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018.

[107] N. F. Malik, A. Nadeem, and M. A. Sindhu. Achieving state space reduction in generated ajax web application state machine. *Intelligent Automation & Soft Computing*, 33(1), 2022.

[108] 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.

[109] D. A. Mateus, C. A. Silva, M. M. Eler, and A. P. Freire. Accessibility of mobile applications: evaluation by users with visual impairment and by automated tools. In *Proceedings of the 19th Brazilian Symposium on Human Factors in Computing Systems*, pages 1–10, Diamantina, Brazil, 2020. ACM New York, NY, USA.

[110] A. Mathur, G. Acar, M. J. Friedman, E. Lucherini, J. Mayer, M. Chetty, and A. Narayanan. Dark patterns at scale: Findings from a crawl of 11k shopping websites. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–32, 2019.

[111] 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.

[112] 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.

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

[114] 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.

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

[116] Monkey Taps LLC. I am - Daily Affirmations. https://play.google.com/store/apps/details?id=com.hrd.iam&hl=en_US&gl=US, 2024. Accessed: 2024-02-12.

[117] H. Mora, V. Gilart-Iglesias, R. Pérez-del Hoyo, and M. D. Andújar-Montoya. A comprehensive system for monitoring urban accessibility in smart cities. *Sensors*, 2017.

[118] M. Naseri, N. P. Borges Jr, A. Zeller, and R. Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. 2019.

[119] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen. Suggesting natural method names to check name consistencies. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. IEEE, 2020.

[120] U. D. of Justice. Americans with disabilities act. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEXLast Accessed: February 12, 2024.

[121] OverSight. Oversight. https://github.com/seal-hub/Oversight, 2022.

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

[123] L. Paninski. Estimation of entropy and mutual information. *Neural computation*, 2003.

[124] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002.

[125] K. Park, T. Goh, and H.-J. So. Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. *HCI Korea*, 2014.

[126] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[127] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing*, 2014.

[128] 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.

[129] 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, Austin Texas USA, May 2012. ACM.

[130] 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.

[131] PyTorch. Softmax. `https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html`, 2021.

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

[133] L. Richardson. Beautiful soup documentation. *April*, 2007.

[134] Robolectric. robolectric/robolectric. `https://github.com/robolectric/robolectric`, 2021.

[135] 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.

[136] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. Examining image-based button labeling for accessibility in android apps through large-scale analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, 2018.

[137] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. An epidemiology-inspired large-scale analysis of android app accessibility. *ACM Transactions on Accessible Computing*, 13(1):1–36, 2020.

[138] P. Royston. Approximating the shapiro-wilk w-test for non-normality. *Statistics and computing*, 1992.

[139] 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.

[140] 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.

[141] N. Salehnamadi, Z. He, and S. Malek. Assistive-technology aided manual accessibility testing in mobile apps, powered by record-and-replay. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2023.

[142] N. Salehnamadi, F. Mehralian, and S. Malek. COALA. `https://github.com/fmehralian/COALA`, 2021.

[143] 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.

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

[145] L. Sensiate, H. Lidio Antonelli, W. Massami Watanabe, and R. Pontin de Mattos Fortes. A mechanism for identifying dynamic components in rich internet applications. In *Proceedings of the 38th ACM International Conference on Design of Communication*, SIGDOC '20, pages 1–8, New York, NY, USA, 2020. Association for Computing Machinery.

[146] L. C. Serra, L. P. Carvalho, L. P. Ferreira, J. B. S. Vaz, and A. P. Freire. Accessibility evaluation of e-government mobile applications in brazil. *Procedia Computer Science*, 2015.

[147] S. Sharma, L. El Asri, H. Schulz, and J. Zumer. Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation. 2017.

[148] 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.

[149] 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.

[150] M. Taeb, A. Swearngin, E. School, R. Cheng, Y. Jiang, and J. Nichols. Axnav: Replaying accessibility tests from natural language. *arXiv preprint arXiv:2310.02424*, 2023.

[151] M. Tafreshipour, A. Deshpande, F. Mehralian, I. Ahmed, and S. Malek. Ma11y: A mutation framework for web accessibility testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[152] H. Takagi, C. Asakawa, K. Fukuda, and J. Maeda. Accessibility designer: visualizing usability for the blind. *ACM SIGACCESS accessibility and computing*, (77-78):177–184, 2003.

[153] M. Van Someren, Y. F. Barnard, and J. Sandberg. The think aloud method: a practical approach to modelling cognitive. *London: AcademicPress*, 11:29–41, 1994.

[154] R. Vedantam, C. Lawrence Zitnick, and D. Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[155] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez. Can everyone use my app? an empirical study on accessibility in android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution*, pages 41–52, Cleveland, OH, USA, 2019. IEEE, IEEE.

[156] D. Venkatesan. “malware may abuse androids accessibility service to bypass security enhancements, 2016.

[157] M. Vigo, J. Brown, and V. Conway. Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility*, pages 1–10, Rio de Janeiro Brazil, May 2013. ACM.

[158] W3. Web accessibility tutorials: Images concepts. https://www.w3.org/WAI/tutorials/images/, 2021.

[159] W3. Web Content Accessibility Guidelines (WCAG). `https://www.w3.org/WAI/standards-guidelines/#wcag`, 2021.

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

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

[162] W3. Understanding the four principles of accessibility. `https://www.w3.org/TR/UNDERSTANDING-WCAG20/intro.html#introduction-fourprincs-head`, 2022. Last Accessed: April 12, 2022.

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

[164] W3C. Accessible rich internet applications (wai-aria) 1.2. Technical report, World Wide Web Consortium (W3C), 2014.

[165] W3C. Inserting dynamic content into the document object model immediately following its trigger element. https://www.w3.org/WAI/WCAG22/Techniques/client-side-script/SCR26.html/, 2023. Last Accessed: December 7, 2023.

[166] W3C. Reordering page sections using the document object model. https://www.w3.org/WAI/WCAG22/Techniques/client-side-script/SCR27.html, 2023. Last Accessed: December 7, 2023.

[167] W3C. Understanding success criterion 2.2.2 — understanding wcag 2.0. https://www.w3.org/TR/UNDERSTANDING-WCAG20/time-limits-pause.html, 2023. Last Accessed: December 7, 2023.

[168] WAI. Understanding sc 3.3.1: Error identification (level a). `https://www.w3.org/WAI/WCAG21/Understanding/error-identification.html`, 2024. Last Accessed: March 5, 2024.

[169] WAI. Using aria role=alert or live regions to identify errors. `https://www.w3.org/WAI/WCAG21/Techniques/aria/ARIA19`, 2024. Last Accessed: March 5, 2024.

[170] F. Wang. Measurement, optimization, and impact of health care accessibility: a methodological review. *Annals of the Association of American Geographers*, 2012.

[171] W. M. Watanabe, R. P. Fortes, and A. L. Dias. Acceptance tests for validating aria requirements in widgets. *Universal Access in the Information Society*, 16:3–27, 2017.

[172] WebAIM. Alternative text. `https://webaim.org/techniques/alttext/`, 2021.

[173] WebAIM. Wave web accessibility evaluation tool. https://wave.webaim.org/, 2023. Last Accessed: December 5, 2023.

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

[175] Wikipedia. Camel case. `https://en.wikipedia.org/wiki/Camel_case`, 2021.

[176] Wikipedia. Snake case. `https://en.wikipedia.org/wiki/Snake_case`, 2021.

[177] F. Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[178] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1989.

[179] J. Wu, X. Zhang, J. Nichols, and J. P. Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483, 2021.

[180] L. Wu, B. Brandt, X. Du, and B. Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63, 2016.

[181] S. Yan and P. Ramachandran. The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing*, 2019.

[182] M. Zhang. Android ransomware variant uses clickjacking to become device administrator, 2016.

[183] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach, et al. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, Virtual, Okohama, Japan, 2021. ACM New York, NY, USA.

[184] X. Zhang, A. S. Ross, A. Caspi, J. Fogarty, and J. O. Wobbrock. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, 2017.

[185] 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.