UNIVERSITY OF CALIFORNIA,
IRVINE

Automated Detection and Repair of Text Accessibility Issues

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Abdulaziz Alshayban

2023

DEDICATION

To my beloved mother, who has been my constant source of inspiration and support.
And to my incredible wife, Atheer, and my precious children, Nasser and Joud, for filling my life with joy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

“ And say, ‘My Lord, increase me in knowledge.’ ” (Quran, 20:114)

First and foremost, I humbly express my deepest gratitude to Allah, the Most Merciful and Compassionate, for granting me the knowledge, fortitude, and determination needed to reach this significant milestone.

I extend my utmost appreciation to my advisor, Prof. Sam Malek, whose guidance and invaluable expertise have greatly contributed to my personal and professional growth. At the beginning of my Ph.D. journey, I had a limited understanding of how to conduct research and write papers. However, Sam was consistently supportive, offering guidance, inspiration, and a helping hand, and helped me navigate through this challenging process. I also want to thank my committee members, Prof. Iftekhar Ahmed and Prof. Josh Garcia, for their valuable feedback and direction. Their dedication and knowledge have played a pivotal role in my research progression and success.

I would like to express my heartfelt appreciation to my colleagues for their unwavering support throughout my journey. The generosity with which they shared their expertise, time, and skills has had a profoundly positive impact on my work. It has been a privilege to collaborate with such dedicated individuals.

To my friends, who have indeed become my second family, your consistent support and shared adventures have been priceless throughout this journey. My appreciation for each of you is beyond what words can convey.

A heartfelt thank you goes out to my family, especially my dear mother, showering me with unwavering support and encouragement throughout this process. Your love, understanding, and presence have made all the difference in my journey.

It’s hard to express how thankful I am to my loving wife, Atheer, who has been my support, my confidant, and my closest companion during this challenging journey. Her insights, constant encouragement, and love have been essential to my success. To my dear children, Nasser and Joud, your happiness and laughter have brightened my days, giving me the strength to face the most difficult times.

Lastly, I am thankful to King Saud University and the National Science Foundation for their generous funding and support, which have enabled the completion of this dissertation and the pursuit of my research goals.

VITA

Abdulaziz Alshayban

EDUCATION

Doctor of Philosophy in Software Engineering **2023**
University of California, Irvine *Irvine, CA*

Masters of Science in Software Engineering **2015**
George Mason University *Fairfax, VA*

Bachelor of Science in Software Engineering **2011**
King Saud University *Riyadh, Saudi Arabia*

RESEARCH EXPERIENCE

Graduate Research Assistant **2020–2023**
University of California, Irvine *Irvine, CA*

TEACHING EXPERIENCE

Teaching Assistant **2019–2020**
University of California, Irvine *Irvine, CA*

Teaching Assistant **2015–2017**
King Saud University *Riyadh, Saudi Arabia*

REFEREED CONFERENCE PUBLICATIONS

#A11yDev: Understanding Contemporary Software Accessibility Practices from Twitter Conversations — **April 2023**
In the CHI Conference on Human Factors in Computing Systems (CHI 2023)

AccessiText: Automatic detection of text accessibility issues in Android apps — **May 2022**
In the The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2022)

Latte: Use-case and assistive-service driven automated accessibility testing framework for android — **May 2021**
In the CHI Conference on Human Factors in Computing Systems (CHI 2021)

Accessibility issues in android apps: state of affairs, sentiments, and ways forward — **May 2020**
In the 42nd International Conference on Software Engineering (ICSE 2020)

ER catcher: a static analysis framework for accurate and scalable event-race detection in Android — **Sept 2020**
In the 35th International Conference on Automated Software Engineering (ASE 2020)

A benchmark for event-race analysis in android apps. — **June. 2020**
In the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys 2020)

ABSTRACT OF THE DISSERTATION

Automated Detection and Repair of Text Accessibility Issues

By

Abdulaziz Alshayban

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2023

Professor Sam Malek, Chair

Mobile technology has progressed beyond the scope of communication, increasingly influencing sectors such as education, health, and finance. For the 15% of the global population living with disabilities [97], accessibility is arguably the most crucial software quality attribute. Leading mobile operating systems, including iOS and Android, offer various built-in assistive services to enhance accessibility for users with disabilities [24, 17]. App accessibility relies on following guidelines, best practices, and extensive testing to confirm compatibility with assistive services. Failure to comply with theses requirements can lead to accessibility issues and barriers for users.

This dissertation aims to enhance mobile app accessibility by initially conducting a large-scale empirical study involving apps, developers, and users to determine the prevalence, categories, and characteristics of accessibility issues, along with development practices that might have contributed to these issues. Next, driven by insights from the study, the research focuses on improving app accessibility for low-vision users, especially those relying on the Text Scaling Assistive Service (TSAS). This is achieved by proposing practical methods and techniques to detect, localize, and automatically repair text accessibility issues stemming from incompatibility between apps and TSAS. The dissertation introduces AccessiText, an automated tool designed to accurately detect text accessibility issues by analyzing UI screenshots and metadata information collected through dynamic analysis. AccessiText employs various heuristics based on distinct types of text accessibility issues, discovered by examining user-reported feedback in Play Store reviews and Twitter data. Furthermore, this dissertation presents Artex, a search-based automatic repair technique utilizing a genetic algorithm to localize and automatically repair text accessibility issues, while minimizing layout distortion and preserving the app layout's consistency.

In our evaluation of the proposed techniques and tools, we conducted experiments and user studies on real-world commercial applications. The findings demonstrated the effectiveness, efficiency, and usefulness of these techniques in resolving text accessibility issues.

Chapter 1

Introduction

Mobile applications (apps) play a crucial role in billions of people's daily lives worldwide, offering a range of essential services, from personal banking to communication and health care. Ensuring ease of access to these services is vital for everyone, particularly for the approximately 15% of the world's population with disabilities [98]. Accessibility is the practice of making websites and applications usable by individuals with disabilities on various devices, such as smartphones and tablets [92]. Adhering to accessibility concepts and guidelines is not only a matter of ethical and social responsibility, but also a legal requirement as enforced by regulations, including Section 508 and 504 of the Rehabilitation Act, and the Americans with Disabilities Act (ADA) [6]. To meet the needs of individuals with disabilities, leading mobile operating systems, such as iOS and Android, have published platform-specific developer accessibility guidelines [26, 19], and integrated various assistive services such as screen readers, switches, and text accessibility features. Utilizing assistive services empowers individuals with disabilities to effectively use mobile devices and complete tasks that may be otherwise difficult. It is crucial for developers to adhere to relevant guidelines and integrate suitable best practices, ensuring accessible and optimal user experiences for users with disabilities.

A vital aspect of app accessibility for visually impaired users is text accessibility and readability, which can be enhanced by carefully considering elements such as contrast ratio, font selection, and text resizing. The Text Scaling Assistive Service (TSAS) is a service commonly utilized by low-vision individuals to increase the default text size for better readability, and it is one of the most popular assistive services used by mobile phone users [2]. However, incompatible apps, those designed without accessibility in mind, may lead to unexpected user interface behaviors, resulting in accessibility barriers. While some research has explored accessibility issues in mobile apps [80, 33, 40, 12], none have specifically addressed the unique challenges faced by low-vision users who rely on TSAS.

This dissertation seeks to bridge this gap by initially conducting a large empirical study on app accessibility issues, exploring their prevalence, categories, characteristics, and the development practices that contribute to these challenges. Next, driven by the insights of this study, the research focuses on improving app accessibility for low-vision users, especially those using TSAS, by proposing methods and techniques to detect, localize, and automatically repair text accessibility issues that arise due to app incompatibility with TSAS. To address these concerns, the dissertation presents AccessiText, an automated tool designed to identify text accessibility issues by analyzing UI screenshots and metadata information obtained through dynamic analysis. AccessiText employs heuristics based on multiple text accessibility issue types discovered via user-reported feedback from app reviews and Twitter data. Furthermore, this dissertation introduces Artex, a search-based automatic repair method utilizing a genetic algorithm to localize and automatically repair text accessibility issues while preserving layout consistency and minimizing distortion.

Evaluations and user studies on real-world commercial applications showcase the effectiveness, efficiency, and usefulness of the proposed techniques and tools. By addressing text accessibility challenges for low-vision users relying on TSAS, this dissertation advances app accessibility and enhances the mobile experience for disabled users.

1.1 Dissertation Structure

The rest of this dissertation is organized as follows: Chapter 2 presents background information. Chapter 3 discusses related work. Chapter 4 presents the research problem, three research hypotheses, and the scope of this thesis. Chapter 5 presents the results of a large-scale empirical study aimed at understanding the accessibility of Android apps from three complementary perspectives: apps, developers, and users. Chapter 6 presents AccessiText, an automated testing technique for text accessibility issues arising from incompatibility between apps and TSAS. Additionally, it describes the empirical study of identifying five different types of text accessibility issues by analyzing candidate issues reported by users in (i) app reviews for Android and iOS, and (ii) Twitter data. Chapter 7 describes Artex, a search-based automatic repair technique that aims to localize and provide fixes for text accessibility issues, followed by a conclusion in Chapter 8.

The following is the list of my research projects and publications:

- Syed Fatiul Huq, **Abdulaziz Alshayban**, Ziyao He, and Sam Malek. "*#A11yDev: Understanding Contemporary Software Accessibility Practices from Twitter Conversations*". In the CHI Conference on Human Factors in Computing Systems (CHI 2023).
- **Abdulaziz Alshayban**, Sam Malek. "*AccessiText: Automatic detection of text accessibility issues in Android apps*". In the The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2022).
- Navid Salehnamadi, **Abdulaziz Alshayban**, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, Sam Malek. "*Latte: Use-case and assistive-service driven automated accessibility testing framework for android*". In the CHI Conference on Human Factors in Computing Systems (CHI 2021).

- **Abdulaziz Alshayban**, Iftekhar Ahmed, Sam Malek. "*Accessibility issues in android apps: state of affairs, sentiments, and ways forward*". In the 42nd International Conference on Software Engineering (ICSE 2020).

- Navid Salehnamadi, **Abdulaziz Alshayban**, Iftekhar Ahmed, Sam Malek. "*ER catcher: a static analysis framework for accurate and scalable event-race detection in Android*". In the 35th International Conference on Automated Software Engineering (ASE 2020).

- Navid Salehnamadi, **Abdulaziz Alshayban**, Iftekhar Ahmed, Sam Malek. "*A benchmark for event-race analysis in android apps.*". In the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys 2020).

Chapter 2

Background

This chapter offers a brief overview of Android Graphical User Interfaces (GUIs) and software accessibility, with a particular focus on text scaling as a means to enhance accessibility. This discussion serves to provide context and clarify the subsequent analyses presented in this dissertation.

2.1 GUI in Android

In Android, an app is built around the concept of activities, which represent individual screens or user interfaces within an app. Each activity can have a distinct purpose, such as displaying information, gathering user input, or performing a specific task. The user interface (UI) for an Android app is composed of a series of View and ViewGroup elements, with each activity consisting of multiple instances of these elements. A View is simply a user interface element that can be used to create interactive UI views that the user can interact with such as TextView, ImageView, CheckBox, etc. A ViewGroup is a type of view that is capable of hosting and organizing other views, known as children. This class serves as an invisible container to hold other views and to define the layout properties that control how their child's views are positioned on the screen.

Designing the UI is an essential step in creating an app that is functional, visually appealing, and easy to use. There are several ways to define the UI in an Android app, but two of the most common and effective methods are XML layouts and Java/Kotlin code. XML-based layouts are the most popular approach used by developers to build the UI in Android apps. XML layouts enable developers to define the style, position, size, and appearance of UI elements. The use of XML provides a clear separation between the UI design and the code that defines app functionality, making it easier to manage over time. XML layouts are also easy to modify using any text editor, providing developers with the flexibility they need to create complex and interactive UI designs.

One important aspect of UI implementation in Android is that each UI can be defined in multiple XML layout files, where each subset of the UI, such as buttons, text fields, or images, can be designed and configured separately in its own XML file. This approach allows for greater modularity and flexibility in designing UIs, making it easier to maintain and update them. For instance, if an app has multiple screens that share some common UI components developers can create separate XML files for those components and reuse them across screens. This helps reduce code duplication and promotes consistency in the UI design.

2.2 Mobile Apps Accessibility

In this section, we provide a brief description of accessibility issues within Android apps, outlining the issue, impacted audiences, and some examples.

Content Labeling

Impacted audience: Individuals with visual impairment.

Description: Content labels are alternative texts to images/actions. Although they are invisible, they can be accessed and announced by a screen reader (e.g., *TalkBack* in Android).

They are intended to provide a clear description of images or actions of buttons for individuals with visual impairment. Below is a list of issues related to content labeling.

- Speakable Text: This issue indicates User Interface (UI) elements that are visible on the screen, but missing text labels. Presence of this accessibility issue means screen reader will be unable to convey these elements to visually impaired users. This issue can be fixed by providing certain attributes in the layout XML file, or dynamically in the code.
- Duplicate Speakable Text: This issue indicates UI elements with the same labels are visible in the same screen. Duplicate labels make it difficult for the user to separate and identify each UI element.
- Redundant Description: For native Android elements such as a `Button`, screen readers can access the type of UI element and announce it to the user along with the label. Repeating the type of element in the label is redundant and may confuse the user.

UI Implementation

Impacted audience: Individuals with mobile impairment.
Description: Developers are required to avoid certain implementations of UI elements that challenge users with mobile impairment. Below is a list of accessibility issues related to UI implementation.

- Clickable Span: UI elements such as `ClickableSpan` may not be compatible with accessibility services such as screen readers, i.e., hyperlinks within those elements may not be detectable and will not be activated. For these hyperlinks to be supported by accessibility services, the use of alternative UI elements such as `URLSpan` is encouraged

- Duplicate Clickable Bounds: This issue indicates two or more elements, such as `nested Views`, that share the same space and boundaries on the screen. When using alternative navigation approaches, duplicate views can cause the same area on the screen to be focused more than once.

- Editable Content Description: This issue highlights improper setting of properties when implementing `EditText` elements. Supporting accessibility for `EditText` requires implementing a label describing the field when it is empty. Moreover, once the user enters text, that text should be announced.

- Unsupported Class Name: Native Android `View` provides type information to screen readers and other accessibility services. For example, accessibility services can recognize an element as Button or Checkbox automatically and announce that to the user. However, developers sometimes create a custom `View`, but forget to provide this information to screen readers and other accessibility services. This issue highlights a custom `View` that does not provide the type of elements to screen reader.

- Traversal Order: Screen readers navigate the elements on a screen based on their hierarchical order. Developers can override this navigation order by using specific attributes for each UI element within the XML layout file, allowing them to specify the following and previous elements on the screen. This issue identifies cases where cyclic navigation is present, which may leave the user stuck at a certain element and unable to explore the remaining elements on the screen.

Touch Target Size

Impacted audience: Individuals with mobile impairment.

Description: Small targets are difficult to tap accurately. This requires more effort for the user. Failure to successfully tap on a button may impede using the app altogether.

- Touch target size: This issue identifies clickable UI elements with small touch areas that can be difficult to use.

Low Contrast

Impacted audience: Individuals with visual impairment.
Description: Insufficient contrast among an app's UI elements can affect how easily users can read, find, and comprehend those elements. Below is a list of accessibility issues related to contrast.

- Text Contrast: This issue corresponds to visible text, where there is a low contrast ratio between the text color and background color.
- Image Contrast: This issue identifies images that are visible on the screen but with a low contrast ratio between the foreground and background colors.

2.3 Supporting Text Scaling for Accessibility

In Android, it is fairly simple to initially enable resizable text views so that they become sensitive to the user's selected preferences. As outlined in Android documentation, the platform allows dimensional values to be specified in a variety of ways, however, when it comes to specifying the text sizes, the use of scale-independent pixels (`SP`) is recommended as they can be adjusted based on the users' preference. Listing 2.1 shows an example of a scaleable `Textview` UI view component in Android. By setting the `width` and `height` properties of the view to `wrap_content`, we ensure that the width or height can expand as needed to contain the text within it. In iOS, the process of supporting scalable text size, while still straightforward, requires additional work and is not enabled by default. Apple encourages the use of their existing `UIFontTextStyle` classes, and then enabling

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Scaleable Text!"
    android:textSize="26sp"
/>
```

Listing 2.1: A TextView that defines its size in terms of SP units. The text displayed will scale based on the user's preference.

the `adjustsFontForContentSizeCategory` properties of the UI view elements to have an automatic update based on the user selected text size. In case of developers using custom fonts, the process requires additional work by the developer.

At first, it may seem effortless to support app text scaling. Developers can simply follow the outlined steps in the platform documentation to enable that feature without much work. However, supporting this feature without considering proper layout design and running tests with larger text sizes, especially in rich and complex UIs, can result in many accessibility issues for users.

According to the Web Content Accessibility Guidelines (WCAG) [94], the recognized standard for digital accessibility, web and mobile apps should meet some minimum requirements called *success criteria*. The *Resizable Text* success criteria mandate that the app's textual content must be resizable (scaleable) up to double the default size without losing any of the app content or functionality.

Chapter 3

Related Work

The purpose of this chapter is to provide an overview of the related work which forms the basis of our proposed research. In particular, it discusses relevant accessibility standards, previous empirical studies on the accessibility of mobile apps, and related work on the testing and repair of accessibility issues.

3.1 Accessibility Standards and Guidelines

Accessibility standards and guidelines are vital for guaranteeing that digital content and technologies are usable by individuals of all abilities. By providing a set of established principles and best practices, these frameworks guide developers and designers in creating more inclusive and accessible apps and digital products.

Among the most globally utilized and recognized guidelines are the Web Content Accessibility Guidelines (WCAG) [94] developed by the World Wide Web Consortium (W3C). These guidelines provide a comprehensive framework covering various aspects of digital accessibility, such as perceivability, operability, understandability, and robustness. Additionally, both Android and iOS platforms have their own set of platform-specific accessibility guidelines

[24, 17], which aim to help developers create apps that are accessible and user-friendly for people with different abilities.

These standards and guidelines serve as essential tools to promote accessibility and inclusion in the digital world, thereby ensuring a seamless user experience for all.

3.2 Previous Empirical Studies

Empirical studies involve systematic data collection and analysis, often through observation and experimentation, to generate insights or validate theories. In the context of accessibility in mobile apps, empirical studies serve as valuable tools to better understand the extent of accessibility issues, identify specific problems, and assess the effectiveness of possible solutions. Recently, studies have started to focus and further investigate accessibility issues in mobile apps. However, the number of these studies is in comparison smaller than those for the web. Notably, most of the prior work in this space is small scale. Coelho et al. manually evaluated four government mobile apps using W3C Accessibility Guidelines [94] and found that accessibility issues are extensive in these cases [84]. Milne et al. investigated the accessibility of mobile health sensors for blind users [69]. Walker et al. evaluated weather apps and found them not to be universally accessible [93].

Vendome et al. [34] performed an empirical study to understand Android apps' accessibility issues. However, our study is answering a much wider and more comprehensive set of questions, as we take a broader approach by looking into the perspective of users, developers, and apps, while their study only investigates the developer's perspective. Furthermore, our study analyzes 10 additional types of accessibility issues. Finally, our approach employs dynamic analysis for detection of accessibility violations, while they use static analysis, enabling us to detect a wider variety of accessibility issues as several UI elements in Android are populated at runtime.

Researchers have also previously looked at specific accessibility issues, such as alternative text labels [78], alternative image labels [69, 73, 84], and missing labels [73]. These studies help characterize accessibility problems. However, the small scales at which they were performed make it difficult to more generally assess the state of accessibility in mobile apps. Additionally, none of these studies has considered developers and attempted to understand the development practices that lead to the occurrence of accessibility issues in apps. Our study aims to fill these gaps.

All in all, the aforementioned studies do not address one of the main objectives of our work: obtaining a holistic view of the prevalence of accessibility issues, understanding why developers create apps with accessibility problems, and comprehending the impact of these issues on user perception. This comprehensive perspective is crucial for an effective approach to enhancing accessibility in mobile apps.

3.3 Accessibility Testing and Repair

Accessibility analysis can be difficult and time-consuming, as it requires human expertise and judgment to determine what barriers may exist for people with disabilities. Researchers have investigated various ways of automating the accessibility analysis process [49, 75, 42, 40], which can be broadly categorized into two categories: static and dynamic accessibility analysis.

Lint [61] is an Android analysis tool for potential issues in various categories such as security, performance, and accessibility. However, it can only identify a limited set of accessibility issues including missing content descriptions and missing accessibility labels declared directly in the XML layout files. Moreover, as a static analysis tool, Lint requires access to app source code to find such issues.

In the context of accessibility, dynamic analysis has had more success in identifying and detecting issues [40]. Accessibility Scanner [83], the recommended tool from Google to test apps for accessibility, is based on the Accessibility Testing Framework [46], an open-source library of various automated checks for accessibility, and it can detect a wide set of accessibility issues. Alshayban et al. [12] proposed an automated accessibility testing technique by implementing a random crawler to simplify the process of accessibility testing. MATE [40] is another tool focused on improved and more efficient exploration process for accessibility testing. However, both of theses tools are limited to the same set of accessibility issues as scanner, as they are based on the same accessibility testing framework.

Latte [80] is an approach aimed at reusing existing tests written to evaluate an app's functional correctness to assess its accessibility as well. It executes the test cases with the help of two types of assistive services, screen readers and switches, to identify accessibility failures. A recent work [33] by Chiou et al. utilized a combination of static and dynamic analyses to detect keyboard accessibility traps in web apps when using a keyboard interface. Salehnamadi et al. [82] highlighted the limitations of current accessibility testing tools and proposed Groundhog, an automated accessibility crawler for mobile apps that detects accessibility issues without requiring developer input. Alotaibi et al. [11] presented a novel approach for automatically detecting TalkBack interactive accessibility failures in Android apps, addressing existing limitations in accessibility-related techniques. In another study, Salehnamadi et al. [81] introduced a record-and-replay technique that records touch interactions, replays them with Assistive Technologies (AT), and generates a visualized report, aiding developers in detecting complex accessibility issues.

While most previous research has focused on improving accessibility through the automated detection of issues, there has been a growing interest in automating the repair process. Several studies have aimed to enhance mobile app accessibility by utilizing automated solutions. Chen et al. [31] developed LabelDroid, a deep-learning-based model that automatically predicts labels for image-based buttons in Android apps, ensuring better compatibility with screen readers. Mehralian et al. [67] further advanced this approach with COALA, a context-aware label generation technique that significantly outperforms LabelDroid in both user studies and automatic evaluations. Additionally, Chen et al. [32] focused on the automatic recognition and classification of icon types in mobile applications, offering comprehensive coverage of icons found in various apps. Finally, Alotaibi et al. [10] presented an automated method to repair size-based accessibility issues, making them more user-friendly for people with motor disabilities.

Overall, none of the above-mentioned solutions investigate text accessibility issues, nor evaluate how the use of TSAS affects the app UI and introduces accessibility barriers for users.

3.4 GUI Testing

Our approach is also related to the area of GUI testing. Generally, GUI testing is a form of dynamic analysis to verify the UI functionality of the application under test. This type of testing aims to check whether the UI behaves correctly by executing various test inputs (e.g., clicking a button, typing in a text field). However, since manual GUI testing is costly and time-consuming, numerous automated GUI testing techniques and tools have been proposed to assist developers in automatically testing app UIs for potential issues and crashes. While the majority [79] of these tools [15, 20, 51, 64] focus on the functional aspect of the app by revealing crashes through testing the app UI with various inputs, some focus on specific issues that impact the non-functional aspects of the app.

Swearngin et al [87] proposed a deep learning based technique for uncovering potential usability issues in UI elements tappability. Seenomaly [101] is an automated technique for detecting GUI animations effects, such as card movement, menu slide in/out, snackbar display GUI animation, that degrade the app usability and violate the platform's UI design guidelines. Draw [45] helps developers optimize the UI rendering performance of their mobile apps performance by identifying the UI rendering delay problems. TAPIR [59] is a static analysis tool for identifying inefficient image displaying (IID), which can impact the app performance and user experience. UIS-Hunter [100] focuses on detecting UI design smells that violate Google Material Design Guidelines, for example, illegible buttons due to lack of contrast, or confirmation dialogs with only a single action that cannot be dismissed.

Chapter 4

Research Problem

4.1 Problem Statement

Due to the increased reliance of disabled people on their mobile devices, ensuring the accessibility of mobile apps, and the support of assistive services within apps has become more important than ever. Mobile platforms, such as Android and iOS, inclusion of various integrated assistive services to help disabled users can present a set of new challenges for app developers. Making apps accessible, and ensuring they are compatible with such services require following a set of best practices and accessibility guidelines, and more importantly, thoroughly testing the app under the various settings offered by the assistive service to identify any issues that can render apps inaccessible for users.

The challenges caused by the lack of understanding and tools to ensure accessibility of mobile apps can be summarized as follow:

"The growing dependence of users with a disability on mobile apps to complete their day-to-day tasks stresses the need for accessible software. However, currently there is a lack of support and compatibility in mobile apps for assistive services including the Text Scaling Assistive Service (TSAS), which is used by people with low-vision to increase the text size and make it accessible to them. The use of TSAS with incompatible apps may introduce text accessibility issues for users. As a result, there is a need for practical techniques to advance accessibility testing of TSAS, including various automated techniques for the detection, localization, and repair of text accessibility issues."

4.2 Research Hypothesis

Existing accessibility analysis tools rely on a set of predefined rules and violations in the app code to identify accessibility issues, while it is effective in detecting various types of issues, a major set of issues is not detectable via these tools, and can only be detected by testing the app under the various settings and configuration offered by the assistive service. For example, when using TSAS with incompatible apps, i.e., those implemented without accessibility in mind, can result in unforeseen behavior in the app user interface and layout, introducing various accessibility issues for users.

While several recent studies have investigated accessibility issues affecting mobile apps [80, 33, 40, 12], none has focused on studying mobile apps support for low-vision users that use TSAS. This is a rather surprising gap, since TSAS is one of the most widely used assistive services [2]. Additionally, the impact of text accessibility issues goes beyond aesthetics, and can, in addition to a reduced user experience, completely break some of the app's functionalities and make it inaccessible for a disabled user relying on TSAS.

Hypothesis 1: *Conducting a study of mobile apps users' feedback posted on Twitter can help in identifying a set of recurring text accessibility issues when using the text scaling assistive service.*

Hypothesis 2: *An automated technique for the detection of incompatible apps with text scaling assistive services can be developed by utilizing visual information from app UI screenshots and various metadata extracted using dynamic analysis.*

To test these hypotheses, we will identify different classes of text accessibility issues by analyzing candidate issues reported by users in (i) app reviews for Android and iOS, and (ii) discussions and issues reported by users on Twitter. Subsequently, utilizing the identified set of text accessibility issues, we will develop AccessiText, an automated technique for accurate detection of text accessibility issues. Finally, we will evaluate our tool on a set of real-world commercial apps to assess its effectiveness in detecting and addressing accessibility concerns.

Hypothesis 3: *A search-based automatic repair technique can be devised for localizing and fixing text accessibility issues in mobile apps.*

To verify the correctness of this hypothesis, I will develop Artex, a search-based automatic repair technique utilizing a genetic algorithm to localize and automatically repair text accessibility issues, while minimizing layout distortion and preserving the app layout's consistency.

Chapter 5

Understanding App Accessibility at Scale

Mobile apps are an integral component of our daily life. Ability to use mobile apps is important for everyone, but arguably even more so for approximately 15% of the world population with disabilities. This chapter presents the results of a large-scale empirical study aimed at understanding accessibility of Android apps from three complementary perspectives. First, we analyze the prevalence of accessibility issues in over $1,000$ Android apps. We find that almost all apps are riddled with accessibility issues, hindering their use by disabled people. We then investigate the developer sentiments through a survey aimed at understanding the root causes of so many accessibility issues. We find that in large part developers are unaware of accessibility design principles and analysis tools, and the organizations in which they are employed do not place a premium on accessibility. We finally investigate user ratings and comments on app stores. We find that due to the disproportionately small number of users with disabilities, user ratings and app popularity are not indicative of the extent of accessibility issues in apps. We conclude the chapter with several observations that form the foundation for future research and development.

5.1 Introduction

Mobile applications (apps) play an important role in the daily life of billions of people around the world, from personal banking to communication, to transportation, and more. Ability to access these necessary services with ease is important for everyone, especially for approximately 15% of the world population with disabilities [98]. As app usage has steadily increased over the years among the disabled people, so has their reliance on the accessibility features. In a survey conducted in 2017, it was found that 90.9% of visually impaired respondents used screen readers on a smartphone [95], which is substantially higher than prior years.

Due to the increased reliance of disabled people on their mobile devices, ensuring that app features are accessible has become more important than ever before. Awareness as to the accessibility of apps has been growing as well. Google and Apple (primary organizations facilitating the app marketplace) have released developer and design guidelines for accessibility [26, 19]. They also provide accessibility services and scanners as part of their platforms [18, 25, 83, 27]. Mandates from government regulations, such as Section 508 and 504 of the Rehabilitation Act and the Americans with Disabilities Act (ADA) [6], are bringing further attention to accessibility factors in apps. Not surprisingly, as a result of such legislation, accessibility-related lawsuits in US federal courts have been growing, e.g., by 180% in 2018 compared to 2017 [90].

Despite these accessibility-focused efforts, studies have found significant accessibility issues in apps [77]. This suggests a continuing need for increasing accessibility awareness among researchers, developers of mobile platforms (e.g., Apple, Google), and developers of individual apps. Although some researchers have studied the accessibility of mobile apps, those studies remain limited in terms of the number of subjects considered, or the number of accessibility issues examined [7, 38, 57, 96, 77, 34]. Furthermore, it is not clear to what extent developers utilize accessibility features in their apps. To the best of our knowledge, no prior work has investigated the development practices pertaining to accessibility of mobile apps, to answer questions such as: how prevalent are different categories of accessibility issues in mobile apps? why developers write apps with accessibility issues? what do the developers want from the accessibility analysis tools? etc.

In this chapter, we aim to cover this gap in research by providing a holistic view of Android accessibility from three complementary perspectives: apps, developers, and users. We investigate prevalence of accessibility issues (the apps), reasons why developers create apps with accessibility issues (the developers), and how accessibility issues impact user perception (the users). First, we conduct a mining study based on Android apps collected from AndroZoo [22] to investigate the extent to which accessibility issues are present. Next, we analyze the developers and organizations involved in the creation of these apps to determine their association with accessibility issues. We then conduct a survey with practitioners to gather a deeper understanding of the underlying reasons for creating apps with accessibility issues. Finally, we analyze user-provided reviews of the collected apps to understand potential associations between accessibility issues and users' perceptions.

Overall, the chapter makes the following contributions:

- We report on the first large-scale analysis of prevalence of a wide variety of accessibility issues (11 types) in over $1,000$ Android apps across 33 different application categories.
- We present the findings of a survey involving 66 practitioners, which shed light on the current practices and challenges pertaining to accessibility, as well as practitioners' perception regarding accessibility tools and guidelines.
- We discuss how the presence of accessibility issues, and their extent, impact users' perception of apps.

- Based on our results, we outline implications for developers and researchers, and provide suggestions for improving the existing tools to better support accessibility in mobile apps.

The chapter is structured as follows: we provide a background on accessibility issues in Android in Section 2.2, followed by a brief review of prior research efforts in Section **??**. In Section 5.2, we present our approach of mining Android apps and surveying developers for answering our intended research questions. In Section 5.3, we present our findings. Section 6.5 discusses the results and outlines implications for developers and researchers.

5.2 Methodology

Our study consisted of the following steps: (1) we first collected a large set of Android apps and filtered those that were not buildable; (2) we evaluated the accessibility of subject apps using a custom-build tool that we developed on top of popular accessibility libraries and testing frameworks; (3) we collected and analyzed developer and organization information pertaining to each app to identify their association with accessibility issues; (4) we then conducted a survey with practitioners to gather a deeper understanding of the underlying reasons for developing apps with accessibility issues; and finally (5) we manually analyzed user-provided reviews of the collected apps to understand potential associations between accessibility issues and users' perception. We now describe each of these steps in further detail.

5.2.1 Study Subjects

For our study, we selected Android as it is the most popular mobile platform [21]. We selected $1,500$ top free apps from Google Play Store. These apps belonged to 33 different categories such as health and fitness, music and audio, productivity, and etc. After identifying the apps, we downloaded their APKs from AndroZoo [9], which is a repository of Android apps with more than 9 million APKs. As part of our study, we wanted to investigate if same developers tend to create similar types of accessibility issues. We selected a random set of 60 developers and found that 52 of them had multiple apps among the initial $1,500$ apps. We then identified other apps from Google Play Store that these developers have published and are not already in our list of projects. This criteria added 200 more apps to our list.

Since one of our goals was to investigate how accessibility levels change over time, we needed to analyze multiple versions of apps. We selected the top 60 apps in terms of their *Activity coverage* (defined in Section 5.2.3) and obtained multiple versions for each app.

5.2.2 Accessibility Evaluation Tool

To evaluate the accessibility features of Android apps, we developed an accessibility evaluation tool that leverages the accessibility checks provided by Google's Accessibility Testing Framework [46], which is an open-source library that supports various accessibility-related checks and can be applied to various UI elements such as `TextView`, `ImageView`, and `Button`. Google's Accessibility Testing Framework is also the underlying engine that is used by Google Scanner [83], a Google recommended app for assessing the accessibility of Android apps. We did not use Google Scanner [83] for our study, as it requires the users to manually run the app, go through each screen and initiate the evaluation process, making it time-consuming and not scalable for a large-scale analysis. Since Google's Accessibility Testing Framework is open-source, it provided us with the opportunity to integrate it into our evaluation tool with ease and automate the entire process.

Our accessibility evaluation tool has two major parts. One part simulates user interactions and the other part monitors the device for `Accessibility Events`. We detail each part below.

Simulating User Interactions

We assess accessibility of apps dynamically, as several UI elements in Android are populated at runtime, making it rather difficult to detect them statically. To that end, our tool first installs the app on an emulator running on a laptop with Intel Core i7-8550U, 1.80GHz CPU, and 16GB of RAM. We used an Android image configured with Google services, API level 25 and 1080 by 1920 pixel display resolution.

After successfully installing an app on the emulator, our tool uses Android Monkey [20] to simulate user interaction. Android Monkey is a UI testing tool developed by Google. It generates pseudo-random gestures, such as clicks and touches, to simulate user interactions.

Our accessibility evaluation tool runs each app for a time limit of 30 minutes, during which the app is restarted multiple times to maximize the coverage of Activities and prevent Monkey from getting stuck on specific screens. In the case of a crash, the tool restarts the app and continues to crawl. Monkey takes a value as the seed to generate the random events. We feed Monkey with a different seed value for each run to maximize coverage. Additionally, at this step, we collect coverage metrics, such as the number of covered Activities and lines of code.

Monitoring Accessibility Events

We developed an Android app, called *Listener*, that was installed on the emulator as part of our accessibility evaluation tool. Listener has a Service running in the background that uses Android's `Accessibility API` to listen for `Accessibility Events`, as each app is crawled. `Accessibility API` is included in Android to support the implementation of accessibility services.

`Accessibility Events` are system-level signals that indicate state changes on the device, e.g., when a Button is clicked, or a new screen is opened. Every time an `Accessibility Event` is detected, the app takes a screenshot of the current screen, and retrieve the hierarchy of all the

UI elements (Views) that are visible to the user. It then invokes Google's Accessibility Testing Framework to perform the various accessibility checks [46]. Since there are no benchmarks for evaluating this kind of tool, we evaluated the tool by running it on several apps and manually verifying the results.

5.2.3 Data Collection and Analysis

We collected different types of accessibility issues for each app using our accessibility evaluation tool. We also collected Package name, Activity name, and number of user interface elements and lines-of-code for each app. We calculated the app `Activity coverage` by dividing the number of unique Activities that are explored by the total number of Activities in the app. We eliminated apps from our analysis with very low `Activity coverage`, i.e., apps for which our tool was not able to explore more than one Activity. We finally ended up with $1,135$ apps in our corpus. Figure 5.1 shows the distribution of apps in our final dataset.

For apps in our dataset that were obtained from Google Play Store, we crawled each app page and collected various meta-data including category, name of developer, number of installs, number of reviews and rating score. Since we collected accessibility issues from screens, and screens with larger number of elements are prone to more accessibility issues, we needed to normalize the data to avoid such bias. To that end, we used *inaccessibility rate*, a metric calculated by dividing the number of elements with accessibility issues on a screen over the total number of elements on the same screen that are prone to accessibility issues [99]. This ratio is calculated for each of the 11 types of accessibility issues. For example, the inaccessibility rate for *TextContrast* type would be the number of elements with *TextContrast* issues divided by all the *TextView* elements that are potential victims for this type of accessibility issue.

Game UIs are not built using native UI elements, instead they are built based on graphic libraries such as OpenGL [47] and Unity [91] [28], where interactive UI elements such as

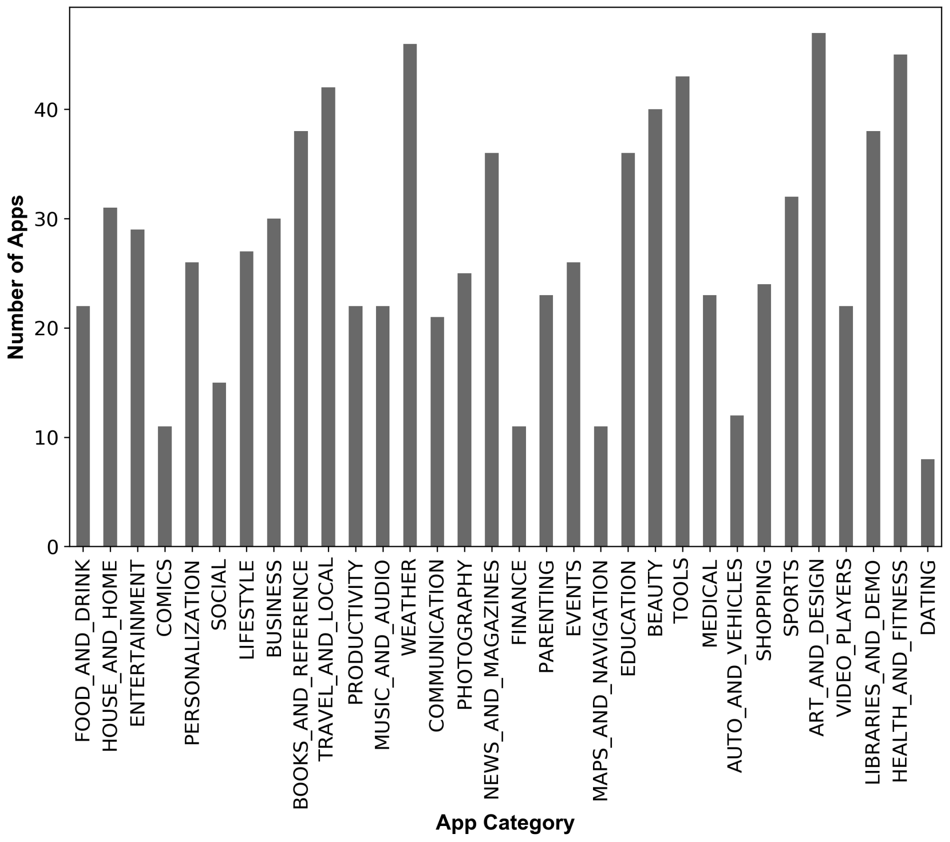


Figure 5.1: Number of apps for each category in the dataset

buttons are rendered as images in the background. Existing tools can neither evaluate these elements nor examine their properties, since these elements do not provide enough information to the accessibility framework. As a result, we excluded the Game category from our analysis.

An important concern with existing accessibility analysis tools is that they report all accessibility issues without assigning any ranking. Our goal was to identify fruitful ways in which the accessibility issues could be ranked, thus engineers can prioritize their effort in resolving such issues. To identify plausible approaches, we randomly selected 25 apps and 100 screens from our dataset, and manually analyzed the reported accessibility issues. Our manual analysis revealed three different cases: (1) Some accessibility issues make it difficult to use the app, while others make the app completely unusable. As an example the 5-star rating element of the Yelp app lacks accessibility support, leaving this core functionality inaccessible to many people. This led us to define *severity of impact on the user* as a ranking criterion. (2) Some accessibility issues are easily fixed, while others require redesigning the interface completely. This motivated us to define *ease of fix* as another ranking criterion. (3) Not all reported accessibility issues are in fact accessibility issues, as the tools sometimes produce false positives. Thus, we defined *certainty of the warning (true positive)* as another plausible ranking criterion.

We also investigated the impact of company culture on accessibility issues. We identified apps in our dataset that are developed by well-known companies. It is naturally the expectation that such companies would have better software development resources than others. The selection criteria is based on the Forbes Top 100 Digital Companies list [43]. The list contains companies such as Amazon, Google, and Microsoft. 23 apps met this criteria. We also analyzed the impact of accessibility issues on users perception. To answer this question, we crawled the Google Play Store and collected meta-data about each app including app rating score, and whether it was promoted as an Editors' Choice on the Store. Our analysis covered reviews written in English only. Prior to performing the review analysis, we pre-processed the text of the reviews using NLTK library [71]. We applied text tokenization, stemming, and lower-case letters conversion. We then searched the dataset using a set of accessibility-related keywords that are based on the different accessibility guidelines and tools, sample keywords include “accessibility”, “visual impairment”, “blind”, etc. We improved upon this set as we scanned through reviews that discussed accessibility. The search process flagged 704 reviews. Two authors independently read the reviews and assigned the accessibility concern types and whether the sentiment was positive or negative. We also had a high inter-rater reliability of 0.84. . We ended up with 150 verified accessibility-related reviews from 102 different apps.

5.2.4 Survey

To validate our findings, we performed an online survey of Android developers. In this section, we describe the survey design, participant selection criteria, pilot survey, data collection, and analysis.

Survey design

We designed an online survey to gather a deeper understanding of the underlying reasons for creating apps with accessibility issues. We asked demographic questions to understand the respondents' background (e.g., their number of years of professional experience). We then asked them about their current practice of using guidelines and tools for assessing accessibility (if any). We also asked them about the challenges of ensuring accessibility based on their experiences. We presented some of the accessibility challenges identified through our empirical analysis of apps and asked the respondents to rate each of them with one of the following ratings and to provide a rationale for their rating: *very important, important, neutral, unimportant, very unimportant.* A respondent can also specify that he/she prefers not to answer. We included this option to reduce the possibility of respondents providing arbitrary answers. During our app analysis, we noticed that none of the available accessibility analysis tools distinguish between the reported accessibility issues. We identified three potentially fruitful methods of ranking reported accessibility issues: *severity of impact on the user, certainty of the warning (true positive), and ease of fix.* We asked the respondents to rate each of the 3 ranking methods with one of the following ratings and to provide a rationale for their rating: *very important, important, neutral, unimportant, very unimportant.* A sample of the survey instrument can be found at the companion website [86].

Participant Selection

We recruited participants for the survey from the list of open-source app developers on F-Droid. In total, we identified 740 unique email addresses for our survey.

Pilot Survey

To help ensure the validity of the survey, we asked Computer Science professors and graduate students (two professors and two Ph.D. students) with experience in Android development and in survey design to review the survey to ensure the questions were clear and complete. We conducted several iterations of the survey and rephrased some questions according to the feedback. In this stage, we also focused on the time limit to ensure that the participants can finish the survey in 10 minutes. The responses from the pilot survey were used solely to improve the questions and were not included in the final results.

Data Collection

We used Qualtrics [76] to send a total of 740 targeted e-mail invites for the survey. 6 of those emails bounced and we received 9 automatic replies, leaving at most 725 potential participants, assuming all other emails actually reached their intended recipients. According to the Software Engineering Institute's guidelines for designing an effective survey [54], *"When the population is a manageable size and can be enumerated, simple random sampling is the most straightforward approach"*. This is the case for our study with a population of 740 software developers.

From the 740 sent emails, we received 66 responses (8.9% response rate). Previous studies in software engineering field have reported response rates between 5.7% [74] and 7.9% [66]. We disqualified 5 partial responses. Finally we considered 61 responses. We received responses from 18 countries across 5 continents. The top two countries where the respondents reside are Brazil and the United States. The professional experience of our respondents varies from 0.25 years to 6 years, with an average of 3.11 years.

Data Analysis

We collected the ratings our respondents provided for each accessibility issue, converted these ratings to Likert scores from 1 (Strongly Disagree) to 5 (Strongly Agree) and computed the average Likert score. We also extracted comments and texts from the "other" fields by the survey respondents explaining the reasons behind their choices. To further analyze the results, we applied Scott-Knott Effect Size Difference (ESD) test [89] to group the accessibility issues into statistically distinct ranks according to their Likert scores. We excluded responses that selected "I don't know" for our ESD test. Tantithamthavorn et al. [89] proposed ESD as it does not require the data to be normally distributed. ESD leverages hierarchical clustering to partition the set of treatment means (in our case: means of Likert scores) into statistically distinct groups with non-negligible effect sizes.

5.3 Results

In this section, we present the results of our study from three complementary perspectives: apps, developers, and users.

5.3.1 App Perspective

We start by looking into the prevalence of accessibility issues in the apps. More specifically, we answer the following research question.

RQ1: How prevalent are accessibility issues in Android apps?

We measure the *inaccessibility rate* of each app for the 11 types of accessibility issues explained earlier in Section 2.2. The inaccessibility rate is calculated by dividing the number of UI elements (such as *TextView* or *Button*) infected with accessibility issues by the total number of UI elements that are prone to such accessibility issues. We also use the *overall inaccessibility rate*, which is the average of all the inaccessibility rates for the different types of accessibility issues.

Figure 5.2 shows the distribution of overall inaccessibility rate for all the apps in our dataset. As shown in figure 5.2, a small number of apps have no accessibility issues, while most apps do. In our dataset, the mean inaccessibility rate for each app is 6.04% and the standard deviation is 2.42%.

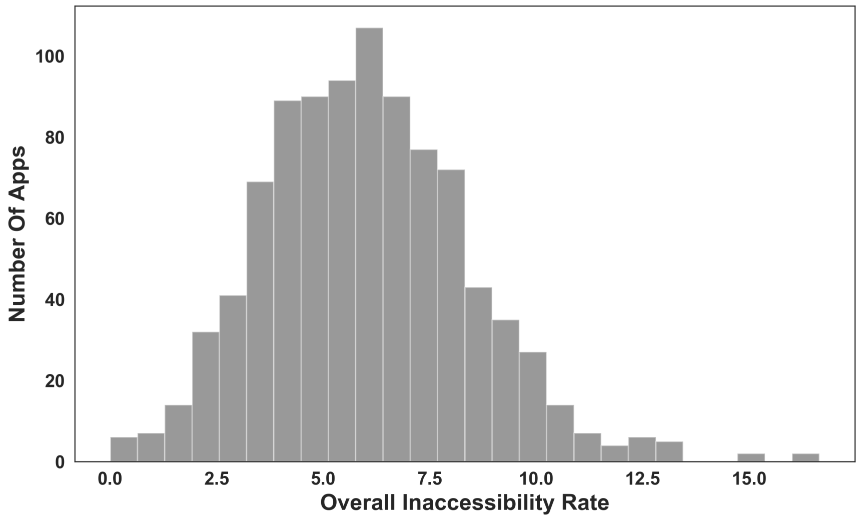


Figure 5.2: Distribution of inaccessibility rate among apps

Observation 1: Accessibility issues are prevalent across all categories of apps, and the mean *inaccessibility rate* is 6.04%.

RQ2: What are the most common types of accessibility issues? Are specific categories of apps more susceptible to accessibility issues than others?

Next, we take a closer look at the inaccessibility rate among the various types of accessibility issues. Table 5.1 shows that *Text Contrast*, *Touch Target*, *Image Contrast*, and *Speakable Text* are the most frequent and have a mean of 22.81%, 19.78%, 12.85%, and 11.08%, respectively. Almost a quarter of *TextView* elements reported a *Text Contrast* issue. None of the apps in our dataset had a *Traversal Order* accessibility issue. Since this is a very specific problem with app navigation approach that is optional to use by developers, zero occurrence of this accessibility issue is not surprising. We omit this issue from our analysis in the rest of the tables.

Table 5.1: The distribution of accessibility issues

Type of accessibility issue	**Mean**	**Std**	**Max**
TextContrast	22.81	11.61	65.10
TouchTargetSize	19.78	10.09	52.63
ImageContrast	12.85	11.99	50.0
SpeakableText	11.08	8.34	42.24
RedundantDescription	0.93	3.40	50.0
DuplicateSpeakableText	0.89	1.47	15.45
ClassName	0.68	1.96	19.64
DuplicateClickableBounds	0.55	0.92	8.33
EditableContentDesc	0.31	2.69	50.0
ClickableSpan	0.15	0.95	14.72
TraversalOrder	0.0	0.0	0.0
All accessibility types	6.04	2.42	16.64

Figure 5.3 shows inaccessibility rate of the apps grouped into 33 categories. We observe that despite slight variations, all categories have accessibility issues. The overall inaccessibility rate is between 4.2% and 7.3.%. Music and Audio category exhibits the highest inaccessibility rate of 7.3%. While different categories of apps have similar distribution of overall inaccessibility rates, certain types of accessibility issues are more frequent in some categories. For example, apps in the *Finance* category have the lowest *SpeakableText* inaccessibility rate at about 4.0%, while *Design and Beauty* has the highest inaccessibility rate of this type at around 16.0%.

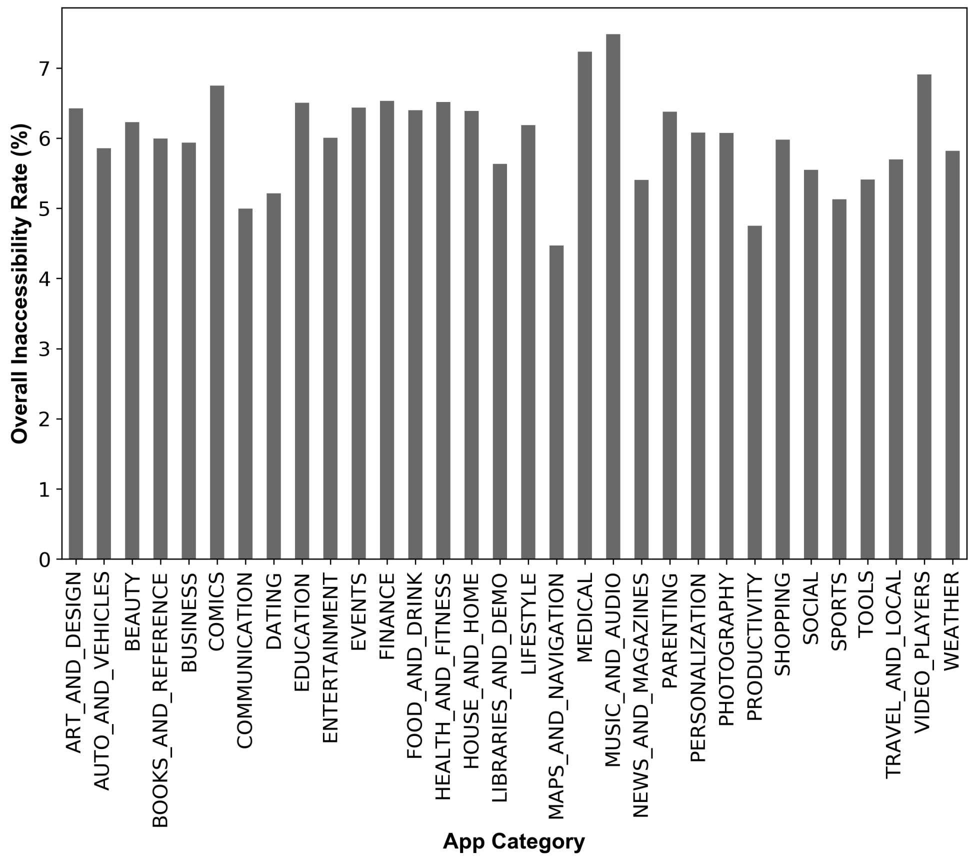


Figure 5.3: Distribution of inaccessibility rates across the different categories

Observation 2: 10 out of 11 types of accessibility issues evaluated in the study were present in the evaluated apps, but to varying degrees.

Since developers use templates provided by Android Studio to build their apps, we posit that the presence of accessibility issues in templates can contribute towards the prevalence of accessibility issues in apps. To that end, we analyzed the templates and found that 5 out of the 10 templates provided by Android Studio suffered from *Text Contrast* , *Touch Target* and *SpeakableText* accessibility issues. For instance, a screen built using *Tabbed Activity* template has *Text Contrast* issues for the titles of the different tabs (as shown in Figure 5.4).

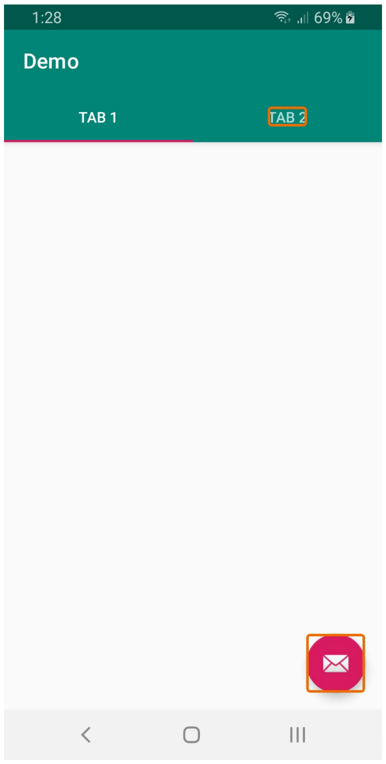


Figure 5.4: Two accessibility issues are identified in the *TabbedActivity* template. *TextContrast* for the title of inactive tab, and a missing *SpeakableText* For the *Button*

Observation 3: 50% of the templates provided by Android Studio, the most popular IDE for Android development, have accessibility issues.

RQ3: How does accessibility evolve over time in Android apps?

In order to answer this question, we used a subset of the apps with multiple versions (details in Section 5.2). We excluded apps with only one version from our analysis as our goal was to investigate the evolution of accessibility issues. In total, this analysis involved 60 apps with 181 versions. We then performed the accessibility evaluation and calculated the difference in inaccessibility rate among the subsequent versions. A positive difference indicates more accessibility issues than the prior version, and vice versa.

Since we are using inaccessibility rate instead of the total number of accessibility issues for our analysis, changes in the app user interface are less likely to impact the results. Figure 5.5 depicts the summary of changes in accessibility issue for 128 updates for 53 apps. Majority of the updates (47%) improved the app's overall accessibility, 28% of the updates impacted the overall accessibility negatively, and for the remaining 25% overall accessibility levels remained the same. Note that despite the use of inaccessibility rate, it is possible that the reduction in inaccessibility rate is not due to fixes but due to the addition of UI elements. However, we still consider it an improvement in the overall accessibility of a new version, as the new UI elements did not introduce any new accessibility issues.

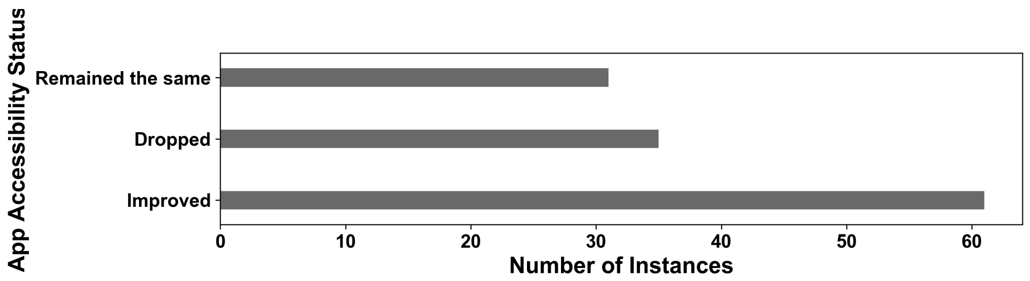


Figure 5.5: How Apps accessibility levels changed over time

Observation 4: Apps become more accessible over time, with 47% of app updates improving the overall accessibility.

5.3.2 Developer Perspective

We now explain our findings regarding the associations between developer/organization and accessibility issues.

RQ4: Do same developers tend to create similar types of accessibility issues?

We examine whether developers are creating apps with similar types of accessibility issues. To answer this question, we first identified a subset of developers who had contributed to multiple projects in our corpus. We then explored Google Play Store to identify all apps written by these developers, which yielded 200 new apps. Finally. we calculated the inaccessibility rate for 260 apps. Table 5.2 shows the average Standard Deviation (SD) of inaccessibility rate for apps developed by the same developer and apps developed by different developers. In the first column of Table 5.2, first we calculate the standard deviation for apps grouped by each developer who has multiple apps and then calculate the average. In the second column, we did a similar calculation but only for developers who contributed a single app. From table 5.2, we observe that the average standard deviation of inaccessibility rate in apps developed by the same developers is 1.70, whereas it is 2.42 for apps developed by different developers.

We performed a Two Sample t-test for each category of accessibility issues and found that for three of the categories (*SpeakableText*, *DuplicateSpeakableText* and *ClassName*) population means are statistically significant (Two Sample t-test, $p < 2.2e-16$) represented using an asterisk (*) in Table 5.2. Since we are performing multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction [39], which gives us an adjusted α value of 0.004 to be used as the significant level. We also report the Cohen's d value (effect size) for each accessibility issue. A Cohen's d value ≥ 0.8 indicates a large effects size, a value ≥ 0.5 and < 0.8 indicates a medium effects size, and a value ≥ 0.2 and < 0.5 indicates a small effects size. Although the observed effect is for the most part small, it is not negligible. This is reasonable, because other factors also impact the presence of accessibility issues.

Table 5.2: Comparison of SD values for apps made by the same developers, and different developers.

Accessibility issue	**Inaccessibility rate SD**		**Cohen's d**
	Same developer	**Different developers**	
TextContrast	9.35	11.61	0.02
TouchTargetSize	7.86	10.09	-0.14
ImageContrast	7.64	11.99	0.06
SpeakableText *	4.50	8.34	0.33
RedundantDescription	0.66	3.40	0.02
DuplicateSpeakableText *	0.69	1.47	-0.23
ClassName *	0.42	1.96	-0.31
DuplicateClickableBounds	0.40	0.92	0.14
EditableContentDesc	0.0	2.69	-0.18
ClickableSpan	0.76	0.95	0.09
Overall inaccessibility	1.70	2.42	0.01

Observation 5: App developers tend to create apps with similar types of accessibility issues.

RQ5: What are the underlying reasons for developing apps with accessibility issues?

We surveyed Android developers (See Section 5.2 for details) to get their opinion about the reasons behind accessibility issues in apps.

Table 5.3 represents the percentage of respondents selecting an option. It shows that *lack of awareness about accessibility and its importance* is identified as the top reason (48.53%). *Additional cost* and *lack of support from management* are the other top reasons.

Observation 6: Developers perceive lack of awareness as the top reason for introducing accessibility issues in apps.

Table 5.3: Reported challenges with ensuring accessibility

Challenges with Ensuring accessibility	**Percent**
Lack of awareness about accessibility and its importance	48.53
Additional cost of ensuring accessibility	16.50
Lack of support from management	15.53
Lack of tools	9.70
Lack of standards and guidelines	8.73
Not sure which standards to follow	0.97

RQ6: Do apps developed by large and well-known companies have better inaccessibility rate than other apps?

To answer this question, we identified apps that are developed by well-known companies such as Amazon, Google, Microsoft, and etc. The selection criteria is based on the Forbes Top 100 Digital Companies list [43]. We posit that these companies have access to more experienced developers and better development resources than others. 23 apps satisfied this criterion. From table 5.4, we observe that the mean for one type of accessibility issue is noticeably different. *SpeakableText* rate is 70% lower in the apps produced by top companies compared to the mean for all other apps in the dataset. Surprisingly, no major difference is observed among the other accessibility issues.

Table 5.4: Comparison of the mean for apps' inaccessibility rates developed by top companies against all other apps

Accessibility issue	**Inaccessibility rate**		**Cohen's d**
	Apps by top companies	**Other apps**	
TextContrast	20.63	22.75	-0.28
TouchTargetSize *	19.79	19.67	0.01
ImageContrast	12.60	12.86	0.02
SpeakableText *	3.39	11.21	-1.14
RedundantDescription *	2.49	0.90	0.36
DuplicateSpeakableText	1.10	0.89	0.12
ClassName *	1.68	0.68	0.47
DuplicateClickableBounds	0.33	0.55	-0.28
EditableContentDesc *	1.80	0.31	0.37
ClickableSpan *	0.76	0.14	0.41
Overall inaccessibility *	4.80	6.03	-0.48

We wanted to understand the reason behind this. In the survey, we asked the developers whether accessibility evaluation is part of their app development/testing process. Out of the 32 survey respondents that are paid developers, 23 did not have any accessibility evaluation as part of their app development process. We also asked the developers whether accessibility of their apps is treated with importance in their organization. 27 respondents mentioned that accessibility is not treated as importantly as other quality attributes, such as security, in their organization. These might be some of the reasons why apps developed by top companies are as susceptible to accessibility issues as apps developed by other companies.

Observation 7: The inaccessibility rates for apps developed by top companies are similar to inaccessibility rates for other apps, except for *SpeakableText* accessibility issue.

RQ7: Do developers perceive all accessibility issues equal?

Our goal was to understand how developers perceive different accessibility issues. To that end, we presented a list of some of the accessibility issues identified through the app analysis and asked the developers to rate them (See Section 5.2 for details). Table 5.5 presents the 6 accessibility issues ranked according to the Scott-Knott ESD test in terms of means of Likert scores for all the respondents. *Redundant Description* , *Text Contrast* and *Image Contrast* are the top two groups.

Table 5.5: Accessibility issues ranked according to the Scott-Knott ESD test (all respondents)

Group	**Accessibility issue category**
1	RedundantDescription
2	TextContrast
2	ImageContrast
3	TouchTargetSize
3	DuplicateSpeakableText
4	SpeakableText

In our study, we noticed that none of the existing tools rank the reported accessibility issues, thus do not provide any means of prioritizing accessibility issues that should be resolved by the developer. We asked the respondents to rate three ranking methods that we identified through a manual analysis and provide a rationale for their rating (See Section 5.2 for details). We used Scott-Knott ESD test to rank their responses. Respondents ranked *severity of impact on the user* as the primary criterion, and *certainty of the warning (true positive)* and *ease of fix* as the second and third criterion, respectively.

Observation 8: Developers believe *impact on user* should be the primary criterion for ranking (prioritizing) the accessibility issues.

5.3.3 User Perspective

Here, we report our findings regarding how users' perception about apps is affected by the presence of accessibility issues.

RQ8: What accessibility issues do users complain about?

To answer this question, we identified 704 reviews from 102 different apps using the process explained in Section 5.2, resulting in 150 accessibility-related reviews. Figure 5.6 summarizes the content of all the app reviews in terms of the type of accessibility concern discussed. Almost half of the reviews discussed accessibility without specifying the exact issues. Users in the other half of the reviews discussed mainly 3 concerns: (1) difficulties related to missing label or content description, (2) text size or color, and (3) image/icon contrast or size. We also found that users tend to use app review to communicate both positive and negative experiences with app accessibility. In some cases users gave a bad review and stated that they are going to delete the app as it is unusable.

Some of the accessibility issues that were discussed in the reviews are not detectable by automated tools and require manual evaluation. For example, users reported issues related to the grouping of UI elements on the screen. Users with visual impairment may rely on using Google TalkBack linear navigation service to understand what is shown on the screen (they swipe right and left to move from one element to another). In one app screen, there were too many elements visible, and navigating that screen linearly was a tedious task, and slowed down the reading experience. Alternatively, developers should hide elements that do not add value to the user, either by marking them as unimportant for the screen reader, or by grouping them with other UI elements under descriptive headings.

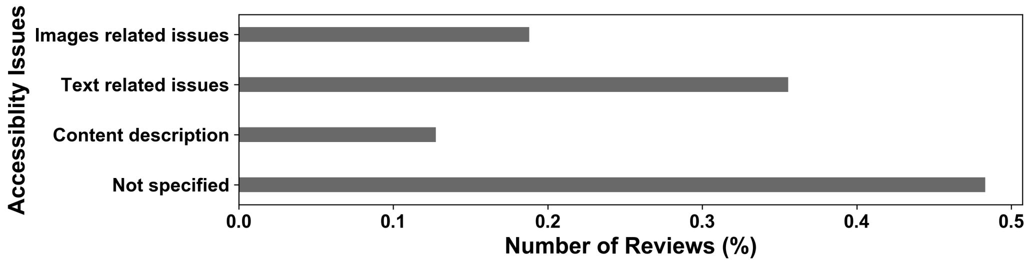


Figure 5.6: The rate of the different accessibility concerns discussed in app reviews

Observation 9: Almost half of the accessibility issues reported by users in app reviews are about difficulties related to missing label or content description, text size or color, and image/icon contrast or size.

RQ9: Do accessibility issues have any association with app ratings?

We explore the association, if any, between apps' inaccessibility rate and their user ratings on Google Play Store. We crawled the Google Play Store and collected meta-data about each app including user rating, and whether it was promoted as an Editors' Choice on the Store. Table 5.6 shows the details of the correlation analysis for apps' inaccessibility rates and user ratings. We used Pearson correlation coefficient since the data is normally distributed.

Observation 10: There is no strong association between the presence of accessibility issues and app ratings.

We also checked whether presence of inaccessibility issues has association with being promoted as Editors' Choice. Table 5.7 compares the inaccessibility rate for apps that were promoted as Editors' Choice (a total of 83 apps in our dataset) against all other apps that were not

Table 5.6: Inaccessibility rates correlation with app rating

Accessibility issue	Correlation value
TextContrast *	0.150
TouchTargetSize	-0.023
ImageContrast *	0.108
SpeakableText	-0.020
RedundantDescription	0.019
DuplicateSpeakableText	-0.059
ClassName	-0.028
DuplicateClickableBounds	0.012
EditableContentDesc	-0.020
ClickableSpan	0.022
Overall inaccessibility rate	0.050

promoted as Editors' Choice. Interestingly, apps that were selected as Editors' Choice had similar inaccessibility rate compared to those that were not selected.

Table 5.7: Comparison of inaccessibility rates for apps that were selected as Editors' Choice in Play store.

Accessibility issue	**Inaccessibility rate**		**Cohen's d**
	Editors Choice	**Other apps**	
TextContrast	25.30	22.71	0.23
TouchTargetSize	18.99	20.19	-0.12
ImageContrast *	15.41	12.43	0.25
SpeakableText	9.81	11.16	-0.16
RedundantDescription	1.12	0.97	0.04
DuplicateSpeakableText	0.71	0.91	-0.15
ClassName	1.04	0.66	0.18
DuplicateClickableBounds *	0.37	0.59	-0.26
EditableContentDesc	0.18	0.36	-0.07
ClickableSpan	0.05	0.17	-0.16
Overall inaccessibility	6.29	6.02	0.11

Observation 11: Presence of accessibility issues does not impact popularity of an app.

5.4 Discussion

Accessibility issues are widely prevalent. One goal of our study was to investigate and identify the most prominent accessibility issues. Our findings show *Text Contrast*, *Image Contrast*, and *Touch Target* are widely prevalent accessibility issues in all categories (33 in our dataset) of Android apps. Since the apps used in this study are the top free apps in the Google Play Store, our results represent the accessibility status of the most commonly used Android apps on the market.

Individuals with different kinds of disability are affected. To make things worse, identified accessibility issues affect individuals with different types of disabilities. As an example, *Touch Target* accessibility issue impedes the app use for individuals with mobile impairment. Color- and image-contrast related accessibility issues create difficulty for visually impaired users. Our results also highlight the fact that approximately 39 million blind users and 246 million low-vision users worldwide [29] are mostly affected by accessibility issues, since the most frequent issues identified in our analysis were *Text Contrast*, *Image Contrast*, and *Touch Target*. These findings call for action to the software engineering community for reducing accessibility issues and lowering the barrier for individuals with different kinds of disability.

Accessibility issues are found even in the Android templates. Our study provided us with useful insights as to the underlying reasons for why existing app are so riddled with accessibility issues. One such reason appears to be the presence of accessibility issues in the templates provided by IDEs. 5 out of the 10 templates from Android Studio suffer from *Text Contrast*, *Touch Target* and *SpeakableText* accessibility issues (two of the top three identified accessibility issues). It is a common practice for developers to build their apps based on these templates. Interestingly, Android Studio is not only the most popular IDE for Android development, but it is also from Google, the company that makes the most popular accessibility analysis tool for Android (i.e., Google Accessibility Scanner).

Developer are generally unaware of the accessibility principles. The pervasiveness of accessibility issues can also be attributed to the lack of awareness and training among the developers. This was identified as the top challenge (48.53%) among the survey respondents (Table 5.3). For example, all of our survey respondents mentioned using Lint. However, prior research shows that developers do not actually know how accessibility warnings from Lint impact the app's use by a disabled person [34]. Developers could benefit from training on accessibility design principles, yet such training is a rarity in most formal academic curricula.

Apps do not get worse over time. When we looked at the evolution of accessibility in apps, we found that 72% of the apps are either improving or remaining the same in terms of accessibility (Figure 5.5). This is surprising, for two reasons. First, conventional code quality attributes, e.g., code smells, tend to degrade over time as the complexity and size of apps increase [8], while with respect to accessibility we observe a reverse effect. Second, developers claim to be unaware of accessibility issues and standards (Table 5.3), thus the improvements in accessibility cannot be attributed to a conscious effort on their part. This warrants further investigation into why and how the accessibility of apps improve over time.

Tools have ways to go. Our results highlight the limitations of current accessibility analysis tools. Some of the accessibility issues are not easy to identify. One such example is the Dropbox app, which requires the user to navigate through all files in the current folder before accessing the"menu", "select", or "more" buttons. This makes it difficult, or even impossible, for individuals with mobile impairment to use the app. Identifying such accessibility issues is not straightforward and requires considering the context of interaction. None of the current accessibility analysis tools consider context. Existing tools are also unable to evaluate apps with non-native elements in their UIs such as Games. Attention from the research community is needed to investigate the accessibility issues that are context-specific or occur in non-native UI elements.

Furthermore, we found that the reported accessibility issues are not equally important. In some cases, a single inaccessible element undermines the entire purpose of an app. As an example, we found the rating UI widget of the Yelp app lacks accessibility support, leaving the core functionality of this app inaccessible to many users. We noticed that none of the analysis tools report severity of issues. Rather important issues are presented alongside of those that affect tangential functions of an app, e.g., text contrast on the About screen.

The high number of accessibility issues reported by the existing tools, reaching hundreds in some cases, overwhelm the developers. Therefore, research is needed into identifying effective means of prioritizing accessibility issues in terms of their importance. We took the first step and identified three potentially fruitful methods of ranking accessibility issues: *severity of impact on the user, certainty of the warning (true positive), and ease of fix.* When asked to rank these during the survey, developers ranked “severity of impact on the user” as the primary criterion for prioritizing the accessibility issues. However, further research is required to identify and compare different prioritization criteria.

Another limitation of the existing accessibility testing tools is the inability to consider all kinds of impairment. As an example, Google Accessibility Scanner has no support for testing hearing impairment related issues. Devising new robust tools for automatically testing accessibility for all kinds of impairment requires further attention from the research community.

An interesting observation is that while accessibility issues can render an app completely useless for a disabled person, the fixes to many accessibility issues are in fact quite easy. For instance, text contrast can be easily fixed through simple changes to the font and/or background colors. Given the recent advances in automated program repair in the software engineering research community, this appears to be a fruitful avenue of future research.

Finally, one approach for stemming accessibility issues is to plan for accessibility in the early design phase rather than handling it as an afterthought at the end of the development phase. Catching accessibility issues early on at the design stage allows the developer to adapt the UI without significant effort. Automated accessibility analysis tools are needed that can be applied to early design prototypes, e.g., UI sketches. Furthermore, we found that the most popular build-time code scanning tool for Android, called *Lint*, only provides support for detection of one type of accessibility issue. We believe incorporating more comprehensive checks in the build process of IDEs could drastically reduce the accessibility issues that creep into the final product, allowing the developers to resolve the issues early in the development.

Gaps between developer and user perception. The preliminary indication from our study is that developers and users have different perceptions as to the impact of accessibility issues on the usability of apps. Bridging this gap would help developers prioritize fixes for accessibility issues that are most critical for users first. However, in order to do a meaningful comparison and have a better understating of user perspective, a more extensive user study, involving disabled users, would be needed. This is outside the scope of our current study, given that property conducting such a study would require, among others, access to users with different types of disability (e.g., visual, hearing, mobility impairment). Nevertheless, we consider this to be an interesting avenue of future work.

Organizations need to pay attention. When we tested the relationship between app ratings and presence of accessibility issues, we did not find any strong association. We posit this has to do with the fact that disabled people are a small minority of app users, thus, reports by this category of users do not have a significant impact on the overall app ratings. This became further evident when we saw a lack of association between presence of accessibility issues and popularity of an app (Table 5.7). Unfortunately, this implies that disabled users cannot rely on app ratings to determine which apps to install, and accessibility-related criticism of apps tends to go unnoticed, as most users do not share the same concerns.

Our analysis revealed that inaccessibility rates for apps developed by top companies are relatively similar to inaccessibility rates for other apps. Moreover, out of the 32 survey respondents that are paid developers, 23 did not have any accessibility evaluation as part of their app development process. Respondents also mentioned that accessibility is not treated as importantly as other aspects of quality, such as security, in their organization. Lack of support from management was also identified as one of the challenges (15.53%) with regards to ensuring accessibility (Table 5.3). All of these indicate that even the top companies in most cases do not pay attention to accessibility. We believe training the app developers and increasing their general awareness of the accessibility issues could improve the state of affairs, as they become ambassadors of accessibility in their organizations.

5.5 Threats to validity

We have strived to eliminate bias and the effects of random noise in our study. However, it is possible that our mitigation strategies may not have been effective.

Bias due to sampling : To increase our confidence that the subject apps are representative, we used multiple sources (i.e., Google Play Store and F-Droid). We also used large number of projects belonging to multiple categories. Since we used only two sources, our findings may not be generalizable to all Android apps. However, we believe that the large number of projects sampled from multiple sources adequately addresses this concern.

Bias due to tools used: Any error in the tools used may affect our findings. To minimize this risk, we leverage Google Accessibility Testing Framework which is widely used by other researchers [77, 41] and practitioners. This is also the underlying framework for the state-of-art Google Scanner.

Moreover, it is possible that there are defects in the implementation of our tool. To that end, we have extensively tested our implementation, among others, verifying the results against a small set of apps for which we manually verified the accessibility issues. Additionally, we do not claim to identify all accessibility issues, as it is possible that certain accessibility issues cannot be identified using the existing tools.

Bias due to survey: It is possible that the survey participants misunderstood some of the survey questions. To mitigate this threat, we conducted a pilot study with Android developers with different experience levels from both open-source community and industry. We also conducted a pilot study with survey design experts. We updated the survey based on the findings of these pilot studies.

5.6 Conclusion

We presented the results of a large-scale empirical study aimed at understanding the state of accessibility support in Android apps. We found accessibility issues to be rampant among more than $1,000$ popular apps that were studied. We identified a number of culprits, including, among others, the observation that developers are generally unaware of the accessibility principles, and that existing analysis tools are not sufficiently sophisticated to be useful, e.g., are unable to prioritize accessibility issues. Due to the disproportionately small number of disabled users, apps with extensive accessibility issues are highly popular and have good ratings. Thus, disabled people have no way of determining which apps are suitable for their use based on the app ratings. Moreover, app developers appear to lack the incentives and backing of their organizations to make their apps usable by this small, yet important, segment of society.

Our ultimate goal is to help catalyze advances in mobile app accessibility by shedding light on the current state of affairs. Our findings can help practitioners by highlighting important skills to acquire, and educators by recommending important skills to include in the curriculum. The findings also highlight opportunities for researchers to address the limitations of existing tools.

Chapter 6

AccessiText: Automated Detection of Text Accessibility Issues in Android Apps

For 15% of the world population with disabilities, accessibility is arguably the most critical software quality attribute. The growing reliance of users with disability on mobile apps to complete their day-to-day tasks further stresses the need for accessible software. Mobile operating systems, such as iOS and Android, provide various integrated assistive services to help individuals with disabilities perform tasks that could otherwise be difficult or not possible. However, for these assistive services to work correctly, developers have to support them in their app by following a set of best practices and accessibility guidelines. Text Scaling Assistive Service (TSAS) is utilized by people with low vision, to increase the text size and make apps accessible to them. However, the use of TSAS with incompatible apps can result in unexpected behavior introducing accessibility barriers to users. This chapter presents AccessiText , an automated testing technique for text accessibility issues arising from incompatibility between apps and TSAS. As a first step, we identify five different types of text accessibility by analyzing more than 600 candidate issues reported by users in (i) app reviews for Android and iOS, and (ii) Twitter data collected from public Twitter accounts. To automatically detect such issues, AccessiText utilizes the UI screenshots and various metadata information extracted using dynamic analysis, and then applies various heuristics informed by the different types of text accessibility issues identified earlier. Evaluation of AccessiText on 30 real-world Android apps corroborates its effectiveness by achieving 87.92% precision and 95.3% recall on average in detecting text accessibility issues.

6.1 Introduction

Mobile technology has progressed beyond the scope of communication and has enabled areas like education, entertainment, and finance. For 15% of the world population with disabilities [97], accessibility is arguably the most critical software quality attribute. The growing reliance of users with disability on mobile apps to complete their day-to-day tasks further stresses the need for accessible software.

Popular mobile operating systems, such as iOS and Android, provide various integrated assistive services, such as TalkBack (a screen reader for users with visual impairment), SwitchAccess (a service for navigating an app via switches instead of the touchscreen), or Voice Access (a service for controlling the device with spoken commands) to help individuals with various disabilities (e.g., vision, motor) use their phones and perform tasks that could otherwise be difficult or not possible. However, for these assistive services to work correctly, developers have to support such services in their apps by following a set of best practices and accessibility guidelines [24, 17]. Disappointingly, several studies [12, 80, 77] have shown lack of accessibility and compatibility of mobile apps with assistive services.

App developers can significantly improve the accessibility and readability of text in their apps by considering factors such as contrast ratio, font selection, and text resizing. From an accessibility standpoint, in addition to satisfying the minimum text size requirement and providing larger text where possible, it is also essential to ensure that text can be adjusted according to users' specific needs. Users with a variety of visual impairments make this adjustment to improve their ability to read small text on a small screen. Once this setting is adjusted, the platform and any apps that have built-in support for this feature will resize the displayed text within the app.

One of the most poplar assistive services among mobile app users is the Text Scaling Assistive Service (TSAS) [2], which is utilized by people with low vision, to increase the default text size and make apps accessible to them. The web content accessibility guidelines (WCAG) [94], the recognized standard for digital accessibility, states the requirement that users must have the ability to adjust the text size, without losing any content or functionality. However, similar to other assistive services, the use of TSAS with incompatible apps, i.e., those implemented without accessibility in mind, can result in unforeseen behavior in the app user interface and layout, introducing various accessibility issues for users.

While several recent studies have investigated accessibility issues affecting mobile apps [80, 33, 40, 12], none has focused on studying mobile apps support for low-vision users that use TSAS. This is a rather surprising gap, since TSAS is one of the most widely used assistive services [2].

To facilitate a a proper understanding of text accessibility issues, this chapter presents a study towards characterizing text accessibility issues in mobile apps, as reported by users. We identify a set of text accessibility classes encountered by users by analyzing more than 600 candidate issues reported by users in (i) app reviews for Android and iOS, and (ii) discussion and issues reported by users on Twitter. Then, leveraging the identified set of text accessibility issues, we devise and propose AccessiText , an automated technique for accurate detection of text accessibility issues. We evaluate our tool on a set of 30 real-world apps from various categories. Additionally, We discuss how the different types of text accessibility issues impact users, and discuss the causes and provide suggestions on how developers can improve their apps to mitigate them.

Our findings highlight several important insights, including the presence of various types of text accessibility issues in mobile apps. Most importantly, the impact of text accessibility issues is not just limited to a reduced user experience due to a distorted and less appealing UI, but can also completely break some of the app functionalities and make it inaccessible for a disabled user relying on TSAS. For example, in some apps, the user is unable to navigate from one screen to another, as the UI view responsible for handling the user interaction becomes completely unreachable, rendering the corresponding functions inaccessible.

Overall, the chapter makes the following contributions:

- As a first step, we identify five different classes of text accessibility issues by analyzing more than 600 candidate issues reported by users in (i) app reviews for Android and iOS, and (ii) discussion and issues reported by users on Twitter.
- Then, leveraging the identified set of text accessibility issues, we devise and propose AccessiText , an automated technique for accurate detection of text accessibility issues. We evaluate our tool on a set of 30 real-world commercial apps.
- We discuss how the different types of text accessibility issues impact users, and discuss the causes and provide suggestions on how developers can improve their apps to mitigate them.

The chapter is structured as follows: In Section 6.2, we present our study on identifying the different types of text accessibility issues. In Section 6.3, we describe how our approach, AccessiText , works. In Section 6.4, we present our findings. Section 6.5 discusses the results and outline relevant insights.

6.2 An empirical study of text-based accessibility issues in mobile apps

As a first step to our study, we wanted to develop a deeper understanding of the types of accessibility problems that ensue when an app does not properly handle text scaling. In this section, we provide an overview of our findings, which set the foundation for our automated testing technique described later in this chapter.

6.2.1 Design and Data Collection

This section introduces the methodology of our study of users' feedback regarding the use of TSAS. We detail how we extracted and processed data. To determine the variety of accessibility issues that can result from text scaling, we manually analyzed two different sources of information described below:

- **App reviews.** These are posts by users of (i) Android apps on the Google Play store, and (ii) iOS apps on the App Store. App reviews have been identified as a prominent source of valuable feedback in mobile apps [52, 55, 60].They can provide information such bugs or issues [72], summary of user experience [48], request for features and enhancements [30]. Our Android reviews dataset includes reviews from 867 top apps. The App Store dataset includes reviews from 1,350 top apps.
- **Twitter data.** These are tweet messages collected from public Twitter accounts. It is common for users to utilize Twitter public platform to provide feedback and report issues to developers, as the majority of apps have a public presence on the platform. Additionally, feedback posted on Twitter has been found to sometimes be more relevant and informative to app developers than other sources [72, 70]. Thus, mining Twitter data provides significant valuable insights into the types of accessibility issues experienced by

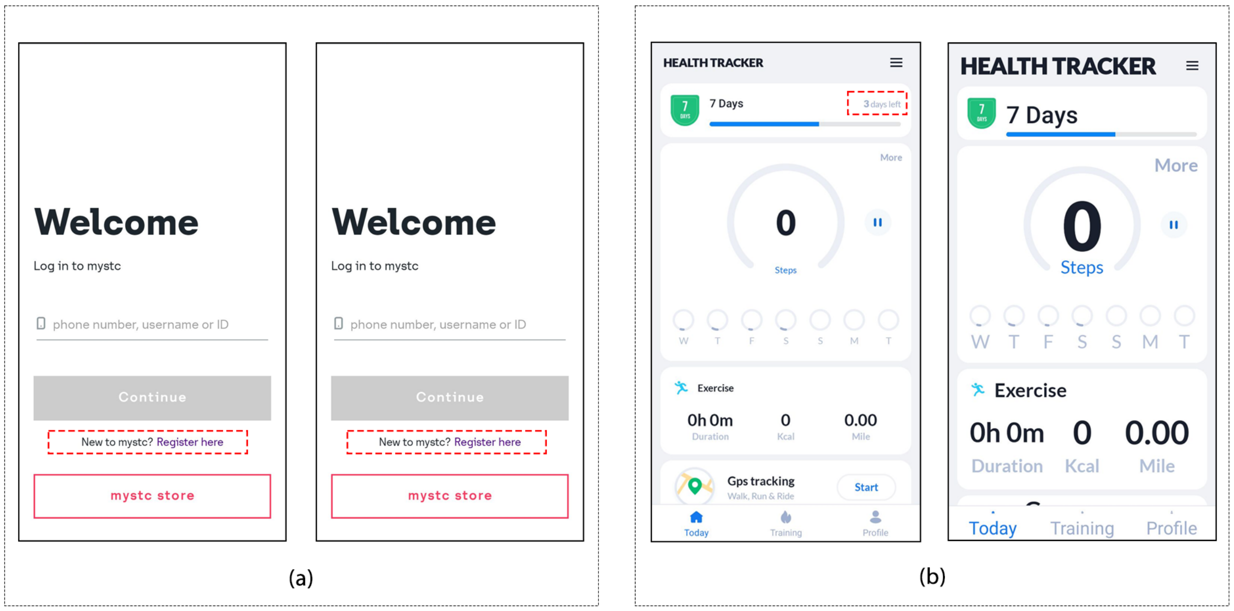


Figure 6.1: Examples of (a) unresponsive view issue, and (b) missing view issue

mobile apps users. We used the Twitter Academic API [3] to collect the public tweets.

For both the Twitter and app stores datasets, our analysis covered reviews and tweets in English only. We first collected candidate tweets and reviews by searching both datasets with keywords relevant to the use of TSAS. For Twitter data, we only consider tweets with images, which are typically screenshots of the app containing the issue. Sample queries included keywords such as "accessibility", "large text", "low vision", and "visually impaired". While some users mentioned the term "accessibility" when describing a text accessibility issue, others addressed and described such issues without mentioning the term. As using keywords to select user reviews and tweets related to accessibility may result in many false positives, in the first iteration, we manually analysed the content of all selected accessibility reviews and tweets to exclude those that are not related to accessibility issues. It is important to note that we did not consider data items tagged as false positive , i.e., discussions not related to text-based accessibility in mobile apps, in the count of the documents manually analyzed. At the end, we collected a set of 412 app reviews, and 235 tweets. Given the limited number, we considered all of them in our manual analysis.

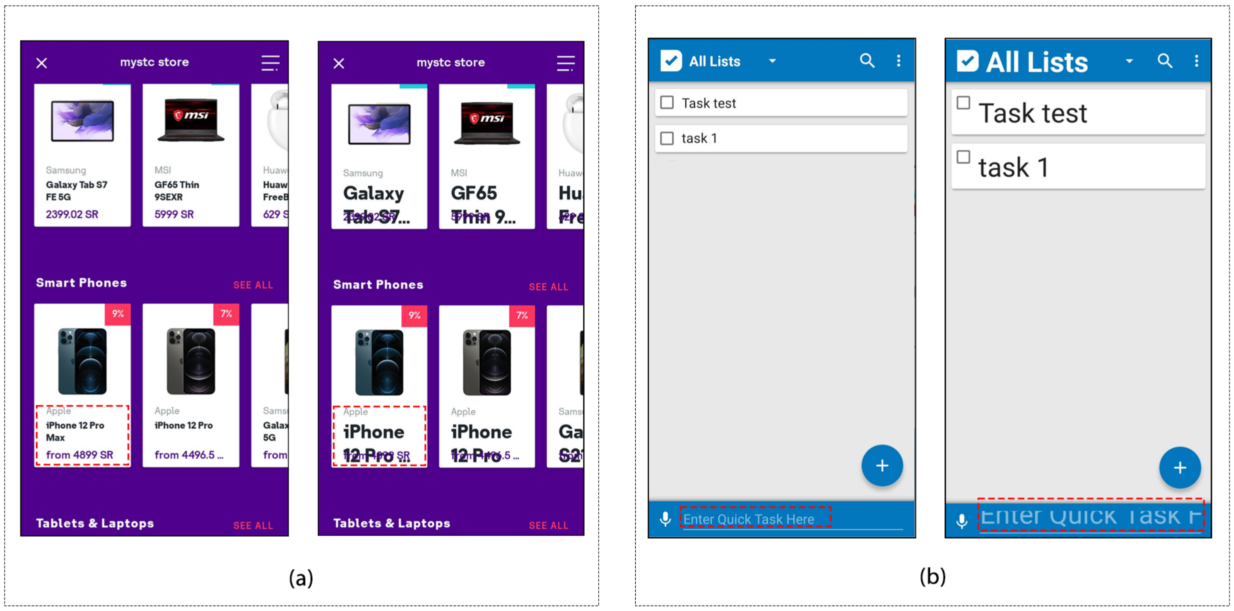


Figure 6.2: Examples of (a) overlapping views issue, and (b) cropped view issue

The data collected from the two sources listed above was manually analyzed following a procedure inspired by open coding [68]. Our goal was to identify and classify the type of accessibility issue reported by the user, by analysing the tweet/app review text and associated image, and extract any additional information provided.

We were able to classify the type of text accessibility issue reported by users in 135 data items. The remaining data broadly falls under two categories: (i) request for an additional feature from the developer to be able to adjust the font size, from which it was not clear whether it is because the app does not support TSAS or just because the user is not aware such an assistive service exists, or (ii) reported a text accessibility issue with the text scaling assistive service, but did not provide enough information for us to identify the type of issue, e.g., a user would describe the app UI to be distorted and the text unreadable without providing much details or a specific description.

Finally, the output of this step was a set of text accessibility issues for mobile apps, described in the following section.

6.2.2 Results

We list and describe a number of text accessibility issues that are the result of the manual coding process for Twitter and app stores data.

Unresponsive Views: Issues in this category describe textual views with a fixed size, that do not respond to text size adjustments by TSAS, making this assistive service useless to the user. Figure 6.1(a) shows an example of an unresponsive textual view (indicated by the red dashed line) in the login screen for `STC` app, an account management app with more than than 10 millions downloads in the Play Store. The main reason for this type of issue is the use of density independent pixels `(dp)` for text font sizes, which unlike scale-independent pixels `(sp)`, do not respond to font size preference specified by the user. Additionally, images of text can also lead to the same issue as they cannot be scaled up by users. Both of these options are sometimes used by developers to easily keep a consistent look and feel for the app across multiple devices and configuration, and unfortunately as a result, reducing the level of accessibility and compatibility with assistive services for the app.

An example of a user feedback on `MyVerizon` app for this type of issue: *The app itself seems fine but does not honor the larger text size accessibility option. This has been reported to them numerous times.*

Missing Views: When the text size increases, it is typical for views to be rearranged on the screen as other textual views occupy more space, and as a result, it is not uncommon for some views to disappear from the visible part of the screen, and become completely inaccessible by users. Figure 6.1(b) shows an example of a missing view in the main dashboard for `Health Tracker` app where the number of remaining days in the current challenge (delineated using the dashed red line on the top right) disappears when adjusting the text size. The impact of this issues is not just limited to distorting the UI and making it less appealing, but can also break some of the app functions and make it inaccessible for a disabled user relying on TSAS.

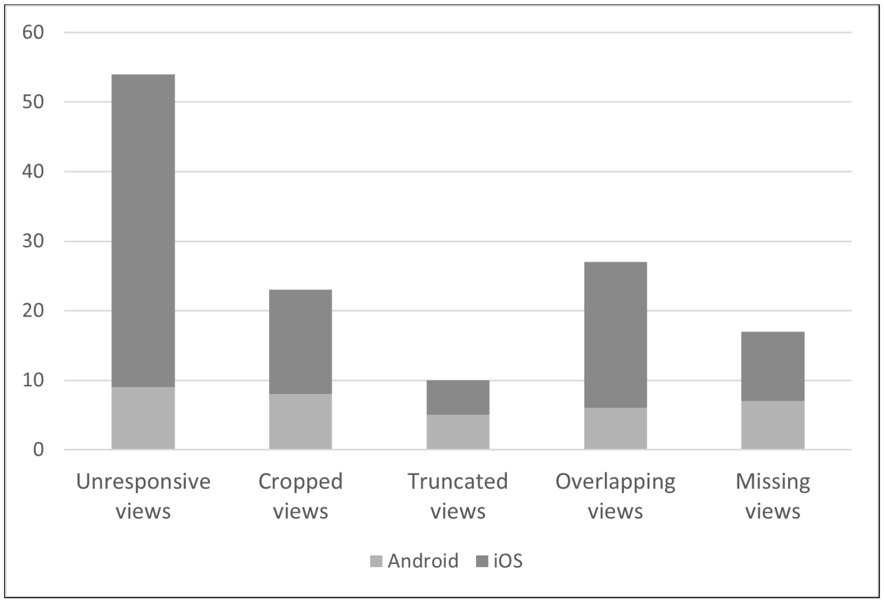


Figure 6.3: Number of text accessibility issues grouped by platform

An example of a user feedback on `Messenger` app for this type of issue: *Really disappointed that the app I use the most has been ruined in accessible large font. Pictures next to names gone, [...]*

Overlapping Views: Overlapping happens when two views on the same screen are rendered fully or partially over each other, resulting in one of the views covering the content of the other. Figure 6.2 (a) shows an example of two overlapping textual views in the `STC` app. We can observe how the product title is covering the price text, making it hardly readable. The common reason behind this category of issues is poorly defined constraints behind these views; both the start and end constraints need to be defined.

An example of a user feedback on `Discord` app, for this type of issue: *I have very poor eye sight due to a genetic condition. I rely on the accessibility options available on the iPhone and I'm very sad to see that the app doesn't play well with large text. All the text is over lapping making it hard to use the app.*

Cropped Views: This type of issues happens when the displayed text grows beyond the constrained height of the containing view, causing part of the text to be invisible. Figure 6.2

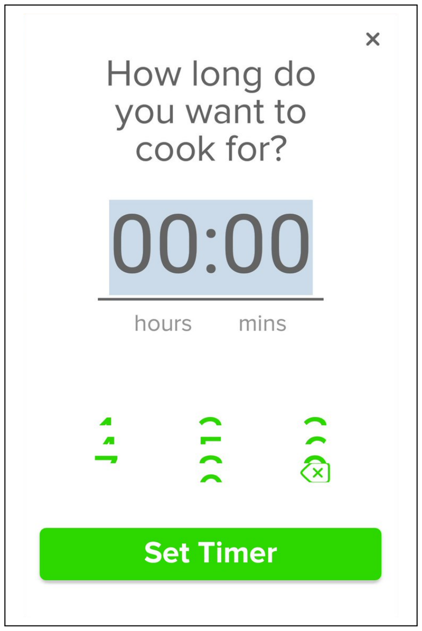


Figure 6.4: A cropped view accessibility issue for `AnovaCulinary` app as reported by a user.

(b) shows an example of a cropped view in the `Todo List` app. The impact of this issue can range from aesthetically unpleasant text to a completely unreadable and inaccessible one, depending on the severity of the cropping. Typically these kinds of issues are related to hard coding layout limits. This allows the content to scale to different lengths and sizes.

An example of a user feedback on `AnovaCulinary` app, as part of which the user also provided a screenshot of the app, shown in Figure 6.4, that clearly demonstrated the issue: *It's not easy to set the timer in your android app when it looks like this. I suspect this is caused by large settings of Accessibility.*

Truncated Views: Text truncation, i.e., shortening, typically happens when the text grows beyond the constrained width of the containing view. Truncated parts of a text are replaced by an ellipsis `(...)`. Figure 6.6 shows an example of a truncated view in the `Insight Timer` app. While text truncation is an effective way to hide additional details and keep the UI design consistent, it can negatively impact the UI accessibility by hiding important information from the user.

An example of a user feedback on `ANZ bank` app, for this type of issue: *Can you please test your Android app when a phone is using largest font [...] As when paying another person the bank account number gets trimmed & can't see all the numbers. You need to test applications with large font sizes & accessibility features enabled.*

Figure 6.3 shows the number of text accessibility issues grouped by Android and iOS, the two mobile platforms considered in our study. The lack of support for text scaling by apps is disappointing, given that both platforms provide facilities for aiding developers to avoid these issues. The identified five types of text accessibility issues are present in both platforms. Unresponsive views, overlapping views, and cropped views are the most common issues reported by users. The high number of unresponsive views in iOS is consistent with the results of a recent survey by Diamond [5], a technology consulting company. By default, Android development supports text resize, while iOS requires developers to use built-in fonts and enable a specific flag in the system, or modify their custom fonts to accommodate resizing. This difference between the two platforms may explain the significant increase in unresponsive issues in iOS compared to Android.

6.3 Approach

Given the insights from our empirical study, we set out to develop an automated tool for testing and detecting text accessibility issues in Android. Although iOS can also benefit from such a tool, our current implementation only supports Android. Extending our work to iOS will be an area of our future work. Figure 6.5 shows an overview of our approach, called AccessiText , consisting of two main components:

(I) Test Runner component that executes a given GUI test script for an app under two settings, first, with the default text size, and then, with larger text by activating TSAS. During the test execution, AccessiText captures a series of screenshots, and collects various metadata related to the UI view components present on each screen that was explored during

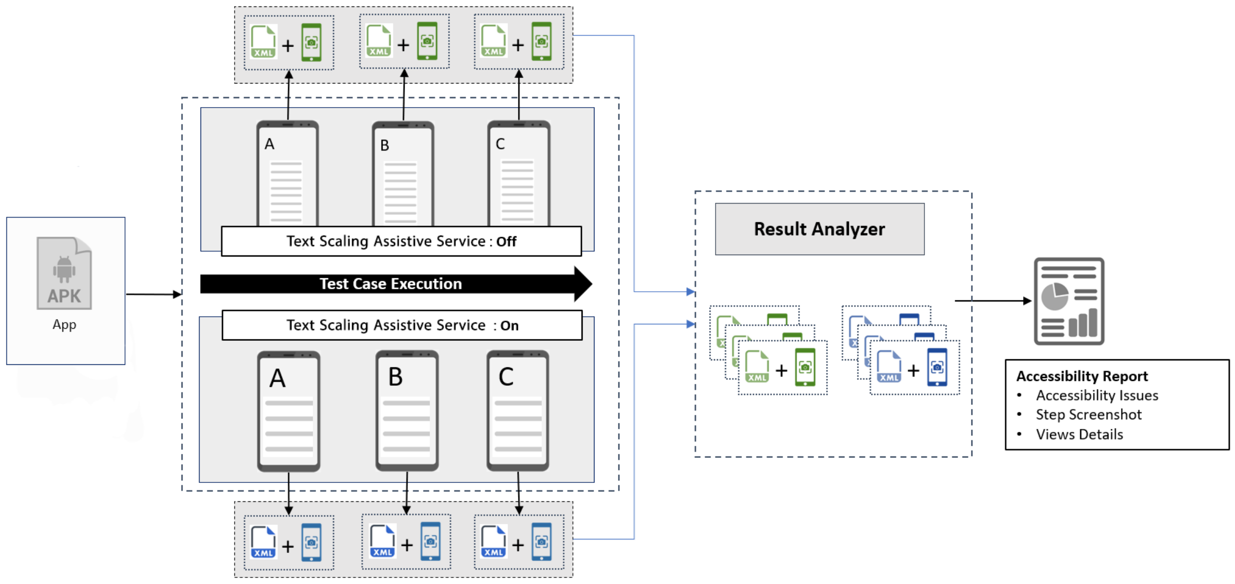


Figure 6.5: Overview of AccessiText

the test execution.

(II) Result Analyzer component that utilizes the information from the previous component, and applies various checks, i.e., predefined rules, to detect any text accessibility issues encountered. Finally, Result Analyzer generates an accessibility report that provides a detailed description of all the accessibility issues and their contextual information.

We implemented AccessiText using Python programming language and utilized Appium testing Framework [23]. In the remainder of this section, we describe AccessiText 's two components in detail.

6.3.1 Test Runner

Test Runner takes a GUI test script as input and executes it twice, first with device default text size, and then with the larger text size. AccessiText uses Android Debug Bridge (`adb`) tool to control the text size and activate/deactivate TSAS at each run. A GUI test case represents an actual use-case provided by the app, and consists of basically a sequence of steps, where each step typically identifies a particular UI view, i.e., `Button`, and specifies an action, e.g., `click` or `scroll`, that is performed on that view.

While executing each step in the test, AccessiText takes a screenshot, and extract an XML dump for the currently displayed screen. XML dump file is parsed to get hierarchical views and properties details of each UI view in the current screen. Properties details include information such `app-name`, `view-class`, `bounds`, and `text`. Listing 6.1 shows an example of the list of properties parsed from the XML dump for a UI view in the hierarchy. View metadata information will later be used by Result Analyzer component to compare the different UI views and identify various text accessibility issues.

AccessiText presumes the test cases are (1) written for each app using the default text size, and (2) actions in the test cases identify the UI views through either `resource-id` or `text` containment (i.e., static attributes). These types of tests are also expected to work with TSAS activated. AccessiText does not support tests cases in which UI views are identified using absolute coordinates on the screen. If a test uses absolute coordinates, it may not work when TSAS is activated, because the positions of views change due to the increase in text size. These assumptions are reasonable and widely applicable. Indeed, developers almost always write tests for their apps with the default text size. Developers typically do not write tests with absolute coordinates, because regardless of TSAS, tests using absolute coordinates cannot be executed on devices with different screen resolutions.

During the test execution, Test Runner component performs additional exploration steps that are not defined in the provided test. For example, when executing a test case with TSAS activated, after each step, AccessiText will try to identify whether the currently displayed screen is scrollable either horizontally or vertically. If so, it will perform a scrolling action, and collect the additional UI views displayed after scrolling. This step is critical to identifying additional views that were originally part of the screen under the original settings (default text size) but have been pushed down (due to increased text size) and became hidden. This list of additional views (after scrolling) will enable us to perform an accurate and complete

```
index="0"
text="Get started"
resource-id="com.google.android.apps.authenticator2:id/howitworks_button_get_started"
class="android.widget.Button"
package="com.google.android.apps.authenticator2"
content-desc=""
checkable="false"
checked="false"
clickable="true"
enabled="true" focusable="true"
focused="false"
scrollable="false"
long-clickable="false"
password="false"
selected="false"
bounds="[231,1176][488,1272]"
```

Listing 6.1: UI view properties parsed from the XML dump for a `Button` with the text: Get Started

comparison for all the views rendered with and without TSAS activated.

In some cases, Test Runner may not be able to execute certain steps with the TSAS activated. This is likely to happen when a view handling the action is missing or inaccessible (e.g., `clicking` on a missing `TextView`). In this case, the accessibility issue is flagged as a functionality failure. When this occurs, Test Runner component deactivates TSAS, falls back to the original setting (i.e., the default text size), executes the step, and then activates TSAS and continues with executing the remaining steps in the test case. This way AccessiText is able to identify all text accessibility issues in the use case exercised by the test.

6.3.2 Result Analyzer

The Result Analyzer utilizes the information collected by Test Runner, e.g., the list of UI views and their metadata along with the UI screenshot for each step, and performs a set of checks to detect the text accessibility issues described in Section 6.2.

Unresponsive Textual Views: This check identifies textual views that do not respond to text size changes by TSAS, making this assistive service useless to the user. To detect this issue, first, AccessiText filters textual views, i.e., views of type `Button`, `EditText`, and `TextView` based on `class` property from the metadata in the parsed XML. For each view that satisfies this selection criteria, AccessiText then obtains an image for the view by cropping the corresponding step screenshot based on `bounds` property from the XML. Finally, AccessiText utilizes Tesseract, an open-source OCR engine, to identify the bounding box for the text inside the selected view, and calculates the text height.

For the setting S with default text size, and setting S' with TSAS activated, we can conclude that a textual view is unresponsive if the view v_i under setting S, and ${v_i}'$ under setting S' have the same text height.

Figure 6.1(a) shows an example of an unresponsive text (indicated by the red dashed line) in the login screen for `STC` app. The main reason for this type of issue is the use of density-independent pixels `(dp)` for text font sizes, which unlike scale-independent pixels `(sp)`, do not respond to font size preference specified by the user. Additionally, images of text can also lead to the same issue as they cannot be scaled up by users. Both of these options are sometimes used by developers to easily keep a consistent look and feel for the app across multiple devices and configuration, and unfortunately as a result, reducing the level of accessibility for the app.

Missing Views: When the text size increases, it is typical for views to be rearranged on the screen as other textual views occupy more space. As a result, it is not uncommon for some views to disappear and become completely inaccessible by users. Figure 6.1(b) shows an example of a missing view in the main dashboard for `Health Tracker` app where the number of remaining days in the current challenge disappears when using ATAS.

To detect such issues, AccessiText ensures that each view v_i under setting S, is also present on the same screen under setting S'. It is worth noting that a view ${v_i}'$ is likely to have different coordinates than v_i and in some cases even not visible on the currently displayed part of the screen. However, it can still be found when a user scrolls down. AccessiText takes into consideration this scenario, and checks for ${v_i}'$ in the additional views after scrolling as

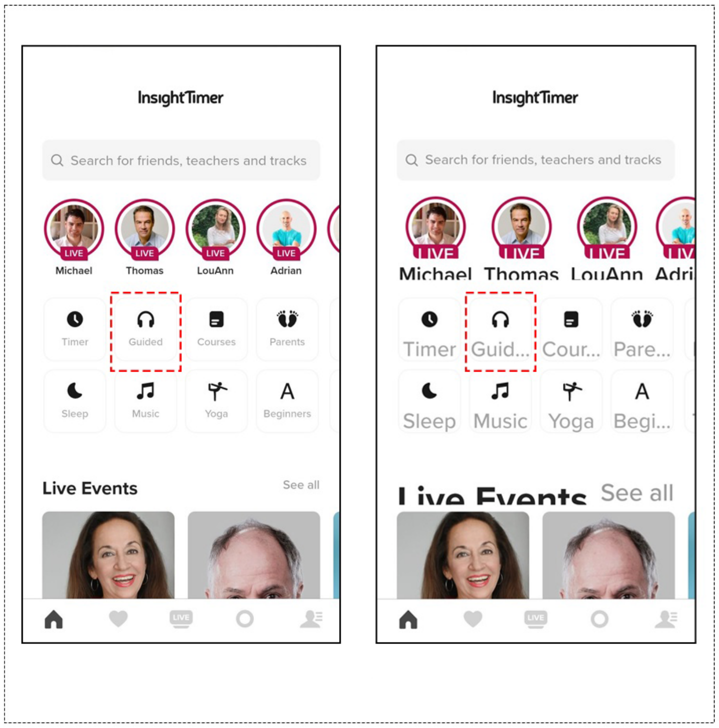


Figure 6.6: Example of a truncated view

provided by Test Runner component.

Overlapping Views: Overlapping happens when two views on the same screen are rendered fully or partially over each other, resulting in one of the views covering the content of the other. Figure 6.2(a) shows an example of overlapping views in the STC app. We can observe how the product title is covering the price text.

AccessiText obtains *(x, y)* coordinates of the upper left corner and the lower right corner of each view v_i from bounds property from the XML. Overlapping issue happens if two views, $v_i{}'$ and $v_j{}'$ in the same screen overlap each other under setting S' but not under S. Intentional overlapping elements such as Floating Action Button (FAB), or overlapping views that are part of the original design are ignored and not flagged as issues. The assumption here is that any unintended overlap between two elements under settings S' but not under S, is undesirable and likely to cause accessibility issues.

Cropped Views: This type of issue occurs when the text grows beyond the constrained height of the containing view, causing part of the text to be invisible. The impact of this issue can range from aesthetically unpleasant text to a completely unreadable and inaccessible one, depending on the severity of the cropping.

To detect this issue, first, AccessiText filters textual views, i.e., views of type `Button`, `EditText`, and `TextView`, based on `class` property from the metadata in the parsed XML. It then obtains an image for the view by cropping the corresponding step screenshot based on `bounds` property from the XML. Finally, AccessiText utilizes Tesseract to identify the bounding box for the text inside the view, and calculates the text height. Given the text height under setting S, we can easily calculate the expected text height under setting S' by multiplying default text height by the scale factor provided to TSAS.

For the same view v_i under setting S, and $v_i{}'$ under setting S', if the text height difference between v_i multiplied by the scale factor (expected height) and actual height of $v_i{}'$ is above a specific threshold, the text within view $v_i{}'$ is determined to be cropped. The above-mentioned threshold is configurable, allowing the user of AccessiText to select a threshold that best fits the desired trade-off between the number of false positives (when the threshold is set too low) and true negatives (when the threshold is set too high) reported by the tool.

Truncated Views: Text truncation, i.e, shortening, typically occurs when the text grows beyond the constrained width of the containing view. Truncated parts of a text are represented by an ellipsis `(...)`. At a minimum, AccessiText ensures that there is at least one word of non-truncated content in a truncated text. While this is the default setting, the minimum required number of non-truncated words is configurable, and would affect the rate of false positives and true negatives. AccessiText utilizes Tesseract to extract the text from view's image, and compares it with `text` property from the XML. If the first word is truncated, then that view is considered to have a truncated text issue.

6.4 Evaluation

We have evaluated AccessiText on real-world apps to answer the following research questions:

- RQ1. How effective is AccessiText for detection of text accessibility issues? What are the precision and recall for our approach?
- RQ2. How efficient is AccessiText in terms of its running time for detection of text accessibility issues?

6.4.1 Experimental Setup

We evaluated our proposed technique using 30 apps. 15 of these were selected from the set of apps reported by users to have text accessibility issues, identified in the empirical study in Section 6.2. We complemented our data set with another 15 apps randomly selected from different categories on Google Play (e.g., travel, productivity, communication).

We created one test case per app using Appium [23], which is an open-source testing framework. Each test case reflects a sample of an app's main use cases (e.g., register an account, add a task, view a product), as provided in its description. Our experiments were conducted on a laptop with Intel Core i7-8550U, 1.80GHz CPU, and 16GB of RAM. We used an Android device configured with API level 28 and 1440×2960 pixel display resolution. The text scaling factor was set to two, allowing TSAS to resize the text to double default text size. Although TSAS can be set higher, we believe doubling the text size is an appropriate choice as it follows the requirements specified by the accessibility guidelines outlined in WCAG [94], which requires that an app's textual content be resizable up to double the default size without losing content or functionality.

Table 6.1: The number of detected accessibility issues and running time for each app

	Unresponsive View	Missing View	Overlapping View	Cropped View	Truncated View	Total Issues	Running Time (seconds)
NZCovid Tracer *	-	2	1	-	-	3	58
Al-chan *	1	1	16	-	-	18	46
Accor All *	-	-	-	3	1	4	32
Instagram *	6	-	-	4	1	11	68
Uber *	-	1	-	3	-	4	47
AnovaCulinary *	8	-	2	10	-	20	90
ABC news *	2	1	4	-	-	7	51
CNET *	-	-	10	1	-	11	37
Chase *	1	1	1	1	-	4	49
MyQ *	1	-	3	-	-	4	71
Delta *	-	-	1	2	-	3	39
Allegiant *	8	-	4	-	-	12	50
Rush *	1	1	4	-	-	6	64
Pocket Casts *	-	-	4	4	2	10	36
Medium *	-	-	1	1	-	2	45
Zoom	-	-	1	8	-	9	47
StepTracker	-	1	3	2	-	6	60
Goal Tracker	-	-	4	1	-	5	48
GetUpside	-	1	5	2	-	8	56
STC	17	-	6	-	-	23	83
Insight Timer	1	-	-	3	2	6	31
To Do List	-	-	-	1	-	1	53
Vocabulary	-	6	5	2	-	13	92
Google Auth	1	-	2	2	-	5	59
Lose it	-	3	10	1	-	14	51
AllTrails	-	3	-	-	-	3	63
Roadie	-	-	5	4	-	9	95
Fedex	3	-	-	5	-	8	49
RecipeKeeper	-	1	4	-	-	5	80
Investment Portfolio	-	-	11	3	4	18	168

6.4.2 Effectiveness of AccessiText

To answer this question, we carefully checked each accessibility issue found by AccessiText to ensure their correctness. Table 6.1 shows, for each issue type, the number of accessibility issues detected. Apps with a star (*) after the app name are from the set of apps reported by users to have text accessibility issues, identified in the empirical study in Section 6.2. Table 6.2 demonstrates the effectiveness of AccessiText in terms of correctly detecting accessibility issues discussed earlier. These results demonstrate that on average, AccessiText has an overall 87.6% precision and 95.8% recall for the different types of issues. Thereby, AccessiText is

Table 6.2: Precision and recall of AccessiText

	# of Detected Issues	Precision	Recall
Unresponsive Views	50	98%	100%
Missing Views	28	82.14%	100%
Overlapping Views	109	88.07%	100%
Cropped Views	63	71.42%	95.83%
Truncated Views	10	100%	83.33%
Total	260	87.92%	95.83%

substantially effective at detecting accessibility issues.

The relatively lower precision score for issues of type *Cropped Views* is mainly caused by inaccurate results returned by the OCR tool. recall that AccessiText utilizes the tool to measure and compare the text height based the bounding boxes returned by the tool. Text that has low contrast with its background can be difficult to localize accurately. This also applies to the results of *Truncated Views*. A false positive *Missing View* can occur when the **Test Runner** component is unable to automatically scroll either horizontal or vertically to reach the view, due to a limitation in the `Accessiblity API` utilized by Appium framework for interacting with the app.

Overall, the results in this table show that AccessiText was able to find accessibility issues in all of the apps in our dataset. The number of issues detected in each app range from 1 to 23 with an average of 8.3 issues per app. We can also observe that all the apps, except two, suffer from two or more types of accessibility issues.

We can see that *Overlapping View*, *Cropped View*, and *Missing View* are the most common types of accessibility issues, and are present in 23, 21, and 12 apps, respectively, of the 30 apps in our dataset. Overlapping views has the highest average number of occurrences in each app.

Table 6.1 indicates that a few applications have accessibility issues of *Truncated View*. The low number of issues could be attributed to the conservative approach that AccessiText uses when checking for issues of type *Truncated View*, where the presence of only one word of the original text for the view is sufficient to not be flagged. Additionally, this issue can only occur in UI views that have their `ellipsize` property set to true by the developer (in the layout XML file), which is not the default option.

6.4.3 Performance of AccessiText

The last column of Table 6.1 shows, for each application, the total running time that AccessiText needed to execute the test case and produce its analysis results. The running time ranges from 31 seconds to 168 seconds (with an average of 1 minute and median of 51 seconds). Overall, the results for RQ2 show that AccessiText was able to detect accessibility issues within a short time, as the average running time for our approach is around 1 minute. The running time shown includes running the test case and interacting with the app, obtaining screenshots and xml dump data for the different screens, performing the various heuristics to check accessibility issues, and generating the final report with the list of accessibility issues.

Several factors affect the running time of our approach, including the number of screens and the complexity (i.e., number of UI views) of the UI layout. As the number of screens and the complexity of those screens grow, AccessiText needs to examine and validate more UI views for potential accessibility issues.

Another factor is the network delay due to the communication between AccessiText running on the laptop and Appium running on the mobile device. To improve the execution time, AccessiText minimizes the requests sent to the Appium server by fetching the UI screenshots and their XML layouts in one call, storing them locally on the laptop, and subsequently processing that information locally to determine the properties of UI views comprising each screen. This architecture allows for a faster analysis compared to sending separate requests to Appium for information about each UI view. Although not the setup we used in our experiment, running the test cases in parallel on two devices for the two settings (default and enlarged text) would further cut the running time by half.

6.5 Discussion

Here, we elaborate further on findings and observations drawn from both our empirical study of text accessibility issues and our experiments with AccessiText:

- *The impact of text accessibility issues goes beyond aesthetics.* The impact of text accessibility issues is not just limited to a reduced user experience due to a distorted and less appealing UI, but can also completely break some of the app's functionalities and make it inaccessible for a disabled user relying on TSAS. For example, in `AllTrails` app (recall Figure 6.1), the user is unable to navigate to the other tabs on the main on-boarding screen, as the UI view responsible for handling the swiping event is pushed off the screen and becomes completely unreachable, rendering this function inaccessible. Similarly, in cases when the screen has overlapping UI views, the impact can be very serious, especially if both UI views are interactive (clickable) with each view performing a different functionality, resulting in one of them to be inaccessible.

- *Various factors influence the severity of text accessibility issues.* For each type of text accessibility issue, there are factors that can influence its severity. For issues of type *Overlapping Views*, the level of overlap between the views is the main factor: the more area of overlap there is, the higher the chances that one or both UI views become unreadable or inaccessible. For issues of type *Cropped Views* and *Truncated Views*, the extent of cropping (or shortening) determines how they affect users. In cases where the cropping is high, the words can be completely unreadable, making the view containing the text inaccessible. For issues of type *Missing Views*, the type of view and its content, in addition to whether it is an interactive UI view or not, determine its impact. When a view goes missing, it is mainly due to the fact that it was pushed beyond the bounds of the current screen. Missing views can be a major issue, as the user is not even aware that an element on the screen is missing.

It is even more significant when the missing view is an interactive view, i.e., a button or a clickable text that performs some functionality in the app, as explained earlier.

- *Improperly designed layouts lead to text accessibility issues.* An important consideration when creating large and complex layouts is to use UI view components that are flexible and responsive, such that they can gracefully adapt to larger text size, and ensure that all the UI views are arranged according to the relationships between sibling views and the parent layout. Missing properly formulated constraints between neighboring UI views may cause various text accessibility issues when scaling an app's text. According to Android documentation, responsive layouts can be achieved through a number of best practices. These include (1) avoiding hard-coding specific value for any UI view components and alternatively using `wrap_content` or `match_parent`, which allow a view to set its size to whatever is necessary to fit the content within that view or expand as much as possible within the parent view, respectively, and (2) using `ConstraintLayout` to specify the position and size for each view according to spatial relationships with other views on the screen. This way, all the views can move and stretch together as the screen size changes.

- *Accessibility testing is a challenge for developers.* Previous studies [12, 44, 4] indicate a lack of awareness among developers about basic access principles. Further exacerbating this general lack of knowledge about accessibility, testing of software for accessibility is a difficult problem, challenged by the availability of numerous assistive services (e.g., screen reader, switch access, TSAS, etc.) and device models (e.g., devices with different screen sizes). Without proper tools and automated techniques, developers are simply overwhelmed with the number of settings under which they have to test accessibility properties of their apps.

- *Consistent design vs accessible design.* Many instances of text accessibility issues found in our study are caused by hard-coded UI view dimensions and font sizes. To ensure that the app looks and feels consistent, developers are tempted to use specific values for the `width` and `height` attributes when defining the UI views. These practices may result in apps that are not accessible or compatible with assistive services, including TSAS.

- *Certain lack of empathy.* Although it was not a goal of our study to report how developers respond to user feedback, we noticed that app developers responded differently to user feedback related to text accessibility issues. In numerous cases the developer response to the issue was to recommend that users go back to the default text size to solve the issue, considering this to be an unreasonable user expectation, instead of acknowledging this as an accessibility issue that needs to be fixed. For example, the following is an example response from a developer of `PulsePoint`, an app for requesting emergency assistance, to a user feedback: *If you're using a very large default font, the 'agree' button may be pushed off of the page. Reduce your font size and try again.*

- *Shifting accessibility to earlier stages of software development.* Accessibility can be better supported when it is deliberately considered in the early phases of the development life-cycle. User experience design teams should consider assistive-service users when drafting early artifacts, such as app UI mock-ups. This would allow developers to determine how the app UI layout should adjust and behave to variable text size preferences from the early stages of development.

6.6 Threats to validity

Our work is prone to several threats to validity:

- Threats to internal validity concern factors internal to our settings that could have influenced our results. This is, in particular, related to possible errors in the manual process of tagging the set of text accessibility issues from the various data sources. To reduce the threat, we followed the widely-adopted open coding approach [77] and validated all results for consistency. Additionally, to minimize the risk of bias due to implementation errors in our tool, we have extensively tested our implementation, verifying the results manually to confirm the accuracy of our approach at finding the accessibility issues.

- Threats to external validity concern the generalizability of our findings. To maximize the generalizability of the categories of text accessibility issues, we have considered two different data sources (app reviews and Twitter data), across two mobile platforms (iOS and Android). However, it is still possible that we could have missed some accessibility issue types available in sources we did not consider. Additionally, For experimental setup, we used apps that have been reported to have confirmed text accessibility issues by users. We also complemented our data set with additional apps from different categories like finance, communication, travel and shopping.

6.7 Conclusion

This chapter presents an automated testing technique, called AccessiText , for text accessibility issues when using text scaling assistive services. The design and implementation of our approach is informed by a large analysis of reported issues by users on mobile app stores and Twitter. Evaluation of AccessiText on real-world Android apps corroborates its effectiveness. Apart from the accessibility issue detection, we investigated and discussed possible causes of these issues, and how developers can improve their apps to mitigate such issues.

In our future work, in addition to extending our current implementation to support the detection of text accessibility issues in iOS, we will devise automated program repair techniques for text accessibility issues. We believe it is possible to leverage an approach similar to AccessiText to evaluate alternative, potentially automatically generated, UI designs that contain fixes to a variety of text accessibility issues. The challenge lies in ensuring such automatically generated designs conform to the original look and feel of the app. We also plan to integrate our approach into the development environments used by developers to support just-in-time analysis and detection of text accessibility issues and layout violations, allowing developers to immediately see the impact of their decisions and how they may render the app inaccessible for assistive-service users.

Chapter 7

ARTEX: Automated Repair of Text Accessibility Issues in Android Apps

7.1 Introduction

Mobile technology has evolved beyond improving communication and now plays a crucial role in fields like education, entertainment, and finance. With about 15% of the global population living with disabilities [97], and an increasing number of people relying on mobile applications to access essential services, accessibility features are becoming increasingly important in software development.

Mobile operating systems, such as iOS and Android, continue to enhance their assistive services to accommodate users with various disabilities, including those with visual, motor, and hearing impairments. These systems often provide built-in accessibility tools that can significantly improve the user experience for these individuals. However, for these tools to be effective, app developers need to ensure that their software adheres to accessibility guidelines and best practices [24, 17]. Unfortunately, recent studies have demonstrated that many mobile apps are still lacking in accessibility and compatibility with assistive services [12, 80, 77] , which can lead to a suboptimal experience or even exclude some users entirely.

Earlier research on the text scaling assistive service (TSAS), a tool meant to support low vision users by enlarging text for better readability, revealed that compatibility issues with certain apps can create numerous accessibility challenges [13]. Applications that are not designed with accessibility in mind can lead to text access problems that negatively impact users' experience and create barriers for those who rely on such assistive services.

In this chapter, our work builds upon previous findings and goes a step further by introducing an automated approach for repairing text accessibility issues. Our method aims to address the compatibility problems between assistive services like TSAS and applications that were not developed with accessibility in mind. By employing this automated solution, we hope to enhance the user experience for low-vision individuals and make mobile apps more inclusive for all users.

We derive our approach from the field of Automatic Program Repair (APR), which focuses on automatically detecting and repairing software faults. One significant subset of APR is search-based techniques, which treat repairs as search problems and optimize program modifications using Genetic Algorithms (e.g., GenProg [58]) or pre-established transformation rules (e.g., PAR [56]). These techniques have significantly contributed to the progression of automatic program repair. APR has been utilized in various domains and contexts, including functional correctness [37, 88], security [50, 62, 63], test repair [85, 36], and UI repair [65, 53, 14].

Our proposed approach not only aims to address and resolve text accessibility issues but also ensures that the fixes have minimal impact on the UI. By introducing only necessary modifications, we strive to maintain UI consistency and avoid unwanted changes that could disrupt the overall user experience. Moreover, we prioritize preserving the original UI when using the default text size. This ensures that our solution caters to the needs of low-vision

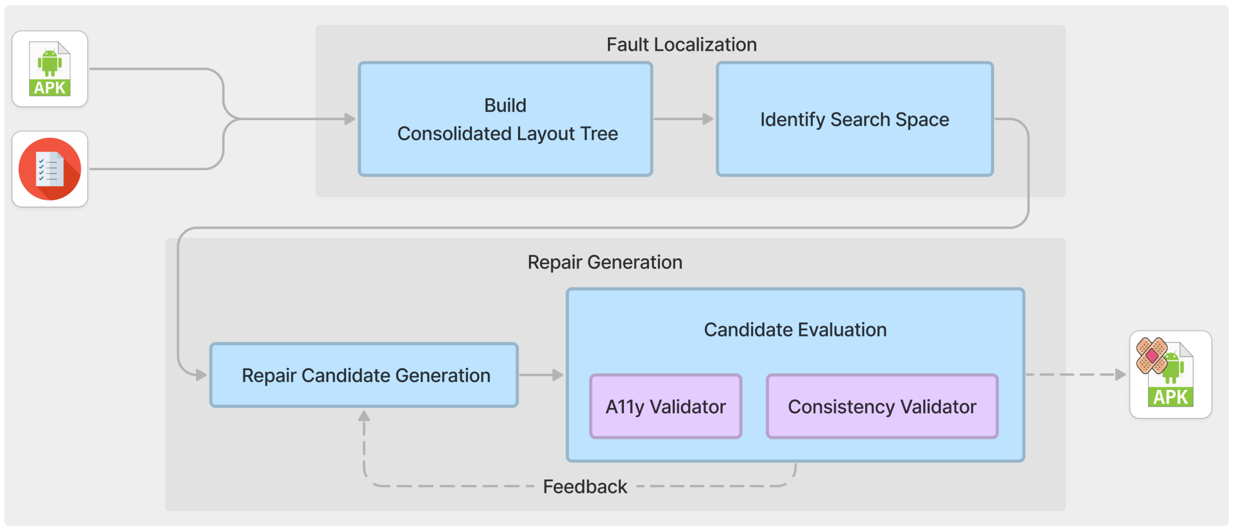


Figure 7.1: Overview of Artex

users without compromising the experience for those using the standard settings.

Overall, the chapter makes the following contributions:

- We present a novel technique that can automatically resolve text accessibility issues while preserving consistency and the original UI appearance.
- We provide a comprehensive evaluation of our proposed approach, including experiments demonstrating its effectiveness, efficacy, and user studies to show that users rate the resulting fixes positively in terms of both effectiveness and visual appeal.

7.2 Approach

We have developed an automated tool, Artex, which aims to automatically repair text accessibility issues in Android applications. The approach is composed of two main components, as illustrated in Figure 7.1:

(I) The Fault Localization component accepts an Android app Apk file, and a list of text accessibility issues as input. Its primary function is to identify the areas likely responsible for these issues. To achieve this, the component initially constructs a Consolidated Layout Tree representing all views (UI elements) and view groups present on the target screen when the text accessibility issue arises. This Layout Tree serves as a comprehensive representation of the app's user interface elements at that specific moment. Subsequently, the component analyzes the Layout Tree, and identifies the most relevant parts of the tree that, when modified, are likely to provide fixes for the text accessibility issues.

(II) The Candidate Solutions Generation component employs a genetic algorithm, a search-based technique, to create a set of candidate solutions by modifying the previously generated Consolidated Layout Tree. The evaluation of each solution uses a multi-objective fitness function that takes into account two aspects: accessibility and consistency. By considering consistency, the search process remains driven to create solutions that maintain both accessibility and consistency with the original design, preserving the overall look and feel. Eventually, the candidate solution with the highest fitness value is chosen, and a patched APK is generated based on this optimal solution.

We implemented Artex using Python programming language. In the remainder of this section, we provide a detailed description of the two components of our approach.

7.2.1 Fault Localization

The Fault Localization component involves two primary steps:

- Building the Consolidated Layout Tree
- Identifying the scope of the search space

Step 1: Building the Consolidated Layout Tree

In the initial step of this component, the focus lies on constructing a Consolidated Layout Tree for the screen displaying the accessibility issue. This Consolidated Layout Tree forms a hierarchical tree structure, capturing all visible UI elements in the current state of the user interface. The tree structure represents not only the parent-child relationships among the distinct UI elements but also captures their associated properties. These properties include dimensions (height and width), orientation, and resources like colors and string values.

Artex generates the Consolidated Layout Tree by effectively integrating data from two main sources: (1) static, declarative XML layout files, and (2) run-time view hierarchy. The XML layout files, created by developers, serve as the blueprint for the User Interface, specifying view types, layout styles, attributes, relationships, and constraints. On the other hand, the runtime view hierarchy reflects the actual structure of the app's UI while it is running and can be gathered using tools such as Android Debug Bridge (ADB). This combination of static and dynamic data results in a comprehensive representation of the app's UI.

To create the Consolidated Layout Tree, the approach follows a series of well-structured steps that ensure a comprehensive representation of the application's UI design and structure. Initially, the approach decompiles the APK file using APKTool [1] to access the application's source code and resources. Subsequently, it installs the application, launching the screen containing text accessibility issues.

Artex then utilizes Android Debug Bridge (ADB) [16] to acquire the runtime view hierarchy, which captures the app's UI structure as it appears during execution. By examining each view's unique ID within the runtime view hierarchy, the approach identifies the corresponding view in the layout files extracted from the APK. Next, it retrieves view properties from the layout file and integrates them into the consolidated layout structure. In the final stage, the approach iteratively traverses the view hierarchy, adding child views to their parent views within the consolidated layout. This process continues until all views in the XML view hierarch have been successfully incorporated into the Consolidated Layout Tree, resulting in a clear and informative representation of the app's UI.

Building a consolidated layout tree faces the challenge that not all views have unique IDs, as the ID attribute in Android is optional. To address this issue, a one-time preprocessing step is applied to the APK. In this step, IDs are added to all views in every layout file within the APK before the app starts running. This ensures that each node or view is mapped to its specific layout file where it is defined. Having IDs for all views is essential for analyzing the APK later, as it results in a more accurate representation of the app's structure and allows for a thorough examination of the app's UI components.

Step 2: Identifying the search space

The second step of the Fault Localization component focuses on identifying the scope of the search space. The search space refers to specific areas within the Consolidated Layout Tree where accessibility issues may be originating. Scoping the search space is an essential step to ensure that the search process is both efficient and effective, as it allows for a targeted and precise examination of potential problem areas in the app's UI.

Artex employs a tiered approach with three levels to identify the search space. The first level involves the specific UI element with the accessibility issue, known as the target element level. The second level, i.e, the immediate context level, encompasses UI elements related to the target element including siblings and the immediate parent. The third level, or the layout hierarchy level, includes the entire layout hierarchy of the target element, starting from the grandparent and ancestor nodes up to the root of the Consolidated Layout. By including the properties of these elements, Artex can include potential areas of the user interface that are likely impacting the accessibility issue.

While Artex three-level approach is comprehensive in identifying potential problem areas related to the accessibility issue, it intentionally excludes UI elements that are unrelated to the target element or its context, such as non-ancestor nodes that are not involved in the layout hierarchy of the target element, or elements without any connection to the accessibility issue, helps avoid unnecessary examination of the app's UI components. This exclusion ensures a more targeted approach, saving time and resources in the fault localization process.

7.2.2 Repair Generation Phase

In this section, we discuss the Repair Generation component, which is the second component of our proposed technique. We approach this problem as an optimization problem using a search-based method, specifically employing genetic algorithms to find optimal solutions.

As our approach follows the genetic algorithm, next, we'll describe some of the key steps and parts involved in applying the genetic algorithm to our problem, discussing how each step contributes to finding optimal solutions for addressing text accessibility issues within user interfaces.

Candidate Solution Representation: In our approach, a candidate solution is represented as an XML tree that outlines the UI layout. Within this representation, the genes correspond to individual views or UI elements in the tree. The population of candidate solutions consists of such XML trees. Utilizing a tree-like representation for the candidate solution is particularly suitable because it allows for a clear and structured way to represent UI elements and their hierarchical relationships. Additionally, with this approach, we can efficiently design genetic operators that manipulate the tree's genes and attributes, while also adhering to problem constraints. This ultimately results in a more effective search process for addressing text accessibility issues.

Selection: In our genetic algorithm, we employ tournament selection as the method for selecting individuals from the population. Tournament selection entails randomly choosing a small subset of individuals who then compete against one another, with the one demonstrating the highest fitness value being selected as a parent for the next generation. This method is advantageous for preserving population diversity and encouraging exploration. By occasionally selecting less fit individuals, the algorithm's sensitivity to changes in the fitness function is reduced, which ultimately contributes to a more effective optimization process.

Mutation Operators: To introduce variability within our search space, we have designed mutation operators that incorporate domain-specific knowledge and tackle the primary aspects of our problem: size, position, placement, and relationships between UI elements. By concentrating on these vital aspects, we have developed a variety of mutation operators specifically crafted to effectively address text accessibility issues. These include Dimensions Mutation, Orientation Mutation, Constraint Mutation, and Container Type Mutation.

- *Dimensions Mutation:* Adjusts the size of UI elements by modifying the `android:layout_width` and `android:layout_height` attributes. This operator changes the dimensions of UI elements.
- *Orientation Mutation:* Alters the orientation of a layout or group of views by modifying the `android:orientation` attribute, impacting the overall organization of the UI and allowing exploration of different orientations.
- *Constraint Mutation:* Modifies the position and placement of UI elements by altering their `app:layout_constraint` attributes, enabling exploration of various alignments and relationships among elements.
- *Container Type Mutation:* Alters the container view group for a UI element, experimenting with different layouts and arrangements.

Fitness Function: The fitness function is one of the most crucial components in determining the quality of candidate solutions and guiding the search process. In our approach, the evaluation of each candidate solution is carried out using a multi-objective fitness function that takes two aspects into account: accessibility and consistency. While our main goal is to improve the accessibility level, maintaining a balance between accessibility and design consistency is equally essential. By incorporating the consistency objective, the search process is motivated to generate solutions that seamlessly integrate accessibility fixes into the original design. This ensures that the modifications made retain the overall look and feel of the original design while addressing the identified accessibility issues.

To address the accessibility objective, our approach employs an improved version of AccessiText [13], an automated tool for detecting text accessibility issues. This tool evaluates the presence and severity of such issues in UI designs in order to enhance user experience. Overall, the fitness function will favor and reward candidate solutions that reduce the number of text accessibility issues, or reduce their severity.

The second objective focuses on maintaining consistency by maximizing the similarity between candidate layouts and the original base layout. In order to accomplish this, Artex measures the similarity between each candidate layout and the original layout by utilizing spatial graphs. Spatial graphs are data structures that represent the spatial relationships between objects in a two-dimensional or three-dimensional space [35]. In the context of UI layout optimization, spatial graphs capture various aspects of UI elements, such as their existence, area, distances between views, and angles between views. These graphs provide a way to quantify and analyze the geometric and topological relationships between elements in complex layouts.

A *spatial graph* for comparing layouts is formally defined as $G = (V, E, A)$, where:

- V: Set of vertices representing the UI elements in the layout.
- E: Set of edges indicating the spatial relationships between the pairs of vertices (UI elements).
- A: Set of attributes associated with the vertices and edges, which include the following properties:

    1. *View existence*: Represented by a node in the graph.

2. *View area*: The size of each UI element, and is determined from its bounds attribute.
3. *Distance between views*: The Euclidean distance between the centers of each pair of UI elements.
4. *Angle between views*: The angle formed by the line connecting the centers of each pair of UI elements relative to the positive x-axis.

By capturing these properties in a graph, spatial graphs allow for the quantitative comparison of two layouts, facilitating the identification of similarities and differences between them.

Artex utilizes the spatial graph comparison algorithm to compare two layouts by quantifying their similarities and differences. This algorithm helps measure how similar or dissimilar the layouts are.

To compare layout similarity, the algorithm first identifies shared nodes or UI elements, and edges, which represent spatial relationships, within the graphs of both layouts. It then proceeds to evaluate various attributes associated with the edges in the graphs. The next step involves calculating the normalized differences between the corresponding pairs of edge attributes, such as distance and angle, while considering the maximum possible difference for each attribute. This calculation leads to normalized values within a consistent range.

Subsequently, the algorithm adds up these normalized differences to determine the total accumulated difference between the graphs. Using this accumulated difference, a similarity score is generated, which indicates the degree of similarity between the two layouts. A higher similarity score suggests that the layouts are more alike, while a lower score indicates that they are significantly different from each other. A similarity score of zero means that the layouts share no nodes or edges, making them completely dissimilar. On the other hand, a similarity score of 100 implies that the layouts are identical in terms of their nodes, edges, and attributes.

Crossover: Crossover is a vital genetic operator that combines genetic materials of two parent solutions to generate offspring in the context of the genetic algorithm. In our approach, we use the uniform crossover technique for this purpose. This method functions by randomly selecting a gene from one of the parent solutions at each gene position, relying on a fixed crossover probability to determine the gene contribution to the offspring. Utilizing uniform crossover is advantageous, as it supports the preservation of good gene combinations through equally probable inheritance from either parent. This, in turn, contributes to a more efficient optimization process.

Managing the Search Space

In order to efficiently manage the search space, two key steps can be employed to improve the process: eliminating irrelevant search areas and excluding invalid candidates. By implementing pruning strategies, we effectively reduce unnecessary parts of the search tree that are unlikely to provide optimal solutions. This not only saves computational resources but also enhances overall efficiency. Moreover, it is essential to exclude any invalid candidates by removing those that violate the system constraints.

In summary, our strategy for the repair generation phase revolves around the utilization of genetic algorithms, incorporating tournament selection, uniform crossover, and custom-designed mutation operators that are based on domain-specific knowledge. We effectively manage the search space by performing pruning operations to eliminate irrelevant portions and exclude any invalid candidates.

7.3 Evaluation

We have evaluated Artex on real-world apps to answer the following research questions:

- RQ1. How effective is Artex for repairing text accessibility issues?
- RQ2. How efficient is Artex in terms of its running time for repairing text accessibility issues?
- RQ3. Does Artex provide fixes that are preferred by users?

7.3.1 Experimental Setup

We evaluated our proposed approach using a set of apps from prior work [13], which included apps with confirmed issues reported by users. After excluding non-native apps, we ended up with a final sample of 15 apps for our study.

Our experiments were conducted on a laptop with Intel Core i7-8550U, 1.80GHz CPU, and 16GB of RAM. We used an Android device configured with API level 28 and 1440×2960 pixel display resolution.

7.3.2 Effectiveness of Artex

In response to our first research question (RQ1), we analyzed the fix rates of the various text accessibility issues addressed by Artex. The fix rate is defined as the percentage of successfully repaired issues compared to the total number of identified issues for each specific type. Table 7.1 displays five different issue types: Unresponsive Views, Missing Views, Overlapping Views, Cropped Views, and Truncated Views.

The average fix rate of 85% across all issue types shows that Artex is effective in fixing a variety of text accessibility issues. Our approach performs exceptionally well in resolving

Table 7.1: Accessibility issue types and their respective fix rates using Artex

A11y Issue Type	Fix Rate
Unresponsive Views	100%
Missing Views	71%
Overlapping Views	73%
Cropped Views	85%
Truncated Views	100%
Average	85%

Unresponsive and Truncated Views, with a 100% fix rate for both, and in Cropped views with an 85% fix rate. Although the fix rates for Missing and Overlapping views (71% and 73%, respectively) are slightly lower, Artex still demonstrates promising results in addressing these issues.

In addition to the previously discussed aggregated results, we also present a detailed breakdown in Table 7.2, which displays the app IDs and their corresponding fix rates for each of the 15 applications included in the analysis. For the majority of the applications, Artex achieved a perfect fix rate, meaning that it was able to resolve all issues within the app. Nonetheless, Artex was not successful at finding a correct fix in 2 apps and achieved a lower fix rate (28% and 85%) in two other apps. These results suggest an overall high effectiveness of our approach but also highlight some limitations.

Figures 7.3 and 7.3 illustrate example fixes for Cropped View, and Overlapping View issues, respectively, with the left images displaying the apps before the fixes and the right images showing the apps after the issues have been resolved.

To better understand the limitations of our approach, we closely examined the cases where Artex was unable to effectively address text accessibility issues. This investigation allows us to identify specific challenges that our approach may struggle with and to suggest possible improvements.

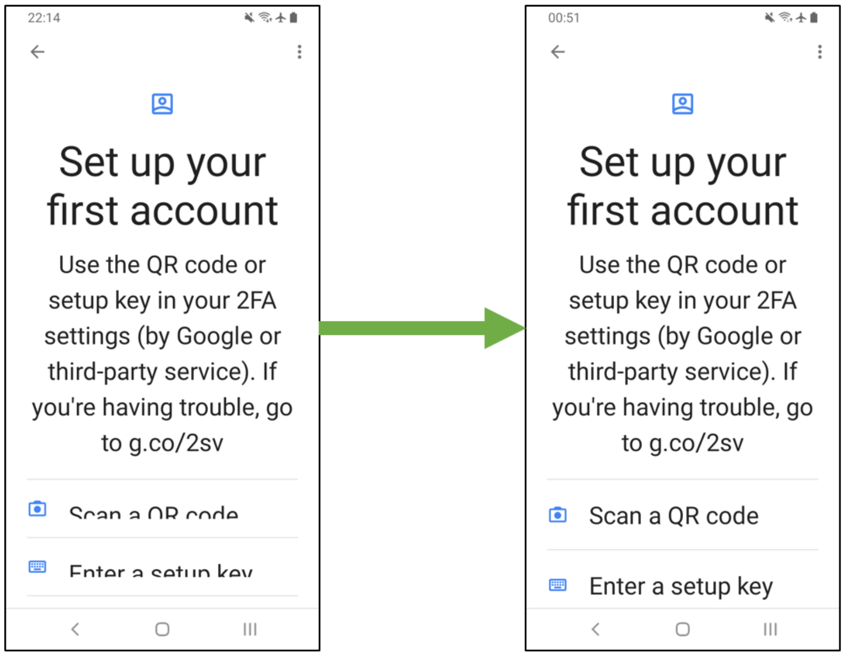
Some of the observed limitations include WebViews and overly complex constraint layouts.

Table 7.2: Running time and fix rates for Artex across different apps

App id	Fix Rate	Running Time (minutes)
com.splendapps.splendo	100%	19
nz.govt.health.covidtracer	100%	25
com.alltrails.alltrails	100%	39
com.google.android.apps.authenticator2	100%	35
com.fedex.ida.android	100%	373
com.accor.appli.hybrid	100%	196
au.com.shiftyjelly.pocketcasts	28%	58
com.lixar.allegiant	100%	25
com.chamberlain.myq.chamberlain	100%	41
com.cbsinteractive.cnet	100%	40
com.abc.abcnews	85%	55
com.finangeros.investgeros	100%	28
steptracker.healthandfitness.walkingtracker	0%	135
com.delta.mobile.android	0%	15
com.way4app.goalswizard	100%	44

Web views are components in Android that allow developers to display web content directly within the native app interface, effectively embedding web pages into the application. For example, in `delta.mobile` app, some UI elements are made entirely of WebView components, even though it is a native app. Artex is not designed to handle WebViews, as their layout is not part of the app and is fetched from the internet instead. Our approach mainly focuses on addressing layouts made using the default method, through XML layout declarations.

Additionally, overly complex cases of constraint layout can also pose challenges to our approach. Constraint layouts are essentially containers in Android that allow developers to create dynamic and flexible user interfaces by defining relationships between UI elements. While our dataset included many constraint layout cases that were successfully resolved by Artex, handling those with extreme complexity, involving various interdependent constraints, may prove difficult. In such situations, Artex might struggle to find a correct solution that can resolve the issue, as any changes in the layout have the potential to impact the entire UI.


22:14

Set up your first account

Use the QR code or setup key in your 2FA settings (by Google or third-party service). If you're having trouble, go to g.co/2sv

Scan a QR code

Enter a setup key

00:51

Set up your first account

Use the QR code or setup key in your 2FA settings (by Google or third-party service). If you're having trouble, go to g.co/2sv

Scan a QR code

Enter a setup key

Figure 7.2: A comparison of a screen displaying a Cropped View issue; the left image shows the app before the fix, while the right image demonstrates the app after the issue has been resolved.

7.3.3 Performance of Artex

To answer our second research question (RQ2), we assessed the performance of our approach by measuring the time taken (in minutes) to resolve accessibility issues for each app. The last column of Table 7.2 presents, for each application, the total running time Artex required to find a fix or terminate if no solution could be identified. The running time needed to resolve accessibility issues across various apps exhibits notable variation. On average, the entire process takes approximately 77 minutes to complete. The repair process includes multiple key tasks, which contribute to the overall time required. These tasks include the decompilation of APKs and generation of Constraint Layout Trees (CLTs), the rebuilding and regeneration of APKs (which is the most time-consuming task), and the evaluation of candidate solutions.

The running time of our approach is influenced by several factors, which can result in variability in the time taken to resolve accessibility issues for different apps. These factors encompass aspects such as app size, which involves the number of activities and resources;

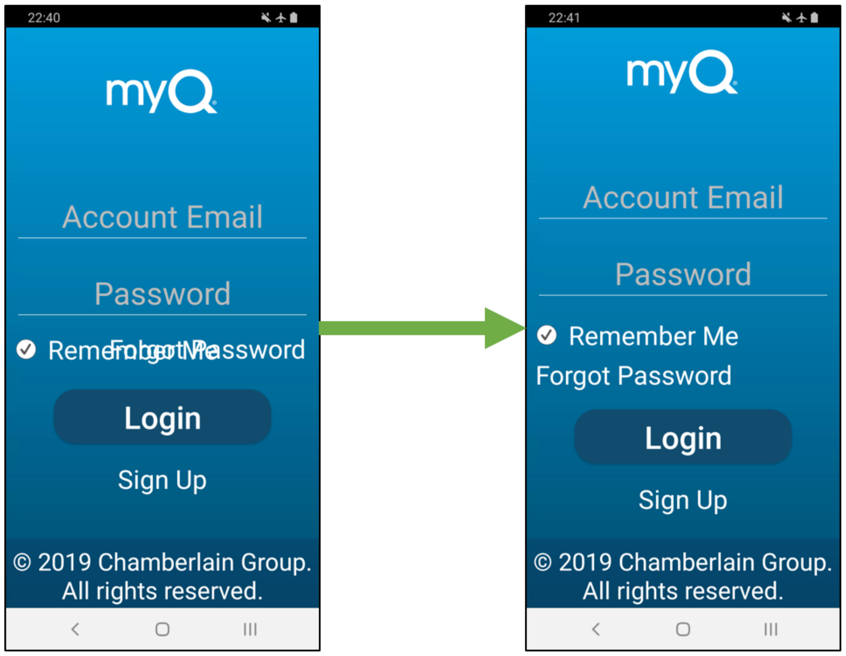


Figure 7.3: A comparison of a screen displaying an Overlapping View issue; the left image shows the app before the fix, while the right image demonstrates the app after the issue has been resolved.

the complexity of the UI layout, affecting the number of UI views to be managed; and the total number of text accessibility issues to be resolved. Each of these factors can individually or collectively contribute to the overall duration, potentially leading to variability in the time required for addressing accessibility issues in various apps.

7.3.4 User Preferences for Provided Fixes

In response to our third research question (RQ3), which focuses on a user study to evaluate users' preferences, we conducted a survey comparing the original UI and the fixed UI in terms of addressing accessibility issues. A total of 100 participants, recruited using Amazon Mechanical Turk, assessed 10 distinct screens. These participants were from the US (to ensure English proficiency) and were Master Workers (to ensure higher quality responses).

Participants were provided with two options — the original UI and the fixed UI — and were-asked to rate their experiences with the fixed UI in terms of effectiveness and aesthetics. The

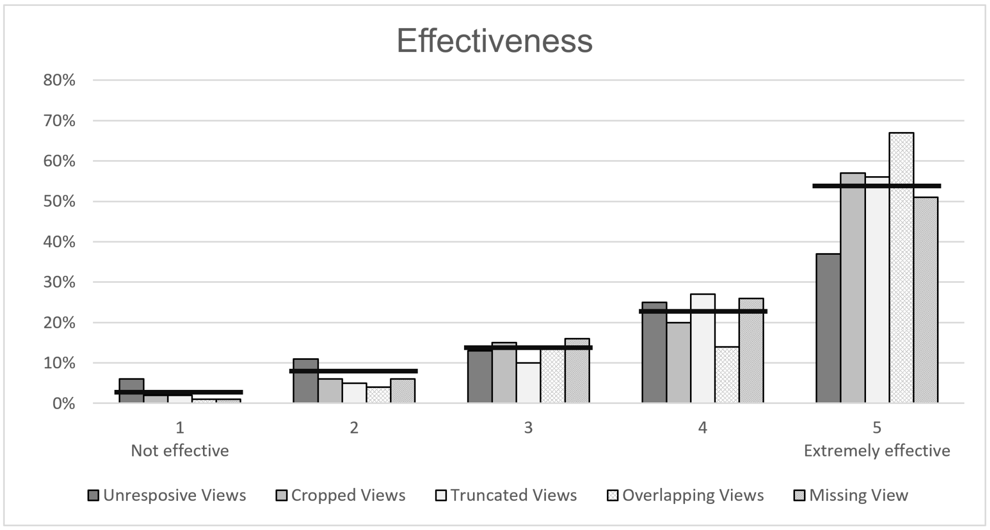


Figure 7.4: A graph summarizing how users rated Artex's proposed fixes for effectiveness.

following questions were presented to the participants:

1. How effective was the automated repair tool in resolving the accessibility issue?
2. How satisfied are you with the aesthetics of the fixed UI, in comparison with the original UI?
3. Were there any aspects of the fix that you did not like?

By conducting this user study, we aimed to gather valuable insights from users about the automated repair tool's performance in resolving accessibility issues and obtain feedback on the visual appeal of the fixed UI. Following the completion of the user study, we analyzed the participants' responses.

Figure 7.4 provides a summary of the effectiveness ratings given by the users. The y-axis represents the percentage of respondents, while the x-axis illustrates the ratings on a scale of 1 to 5, where 1 signifies "least effective" and 5 indicates "most effective." Overall, the average effectiveness rating is 4.2, which implies a generally favorable perception of the solution's

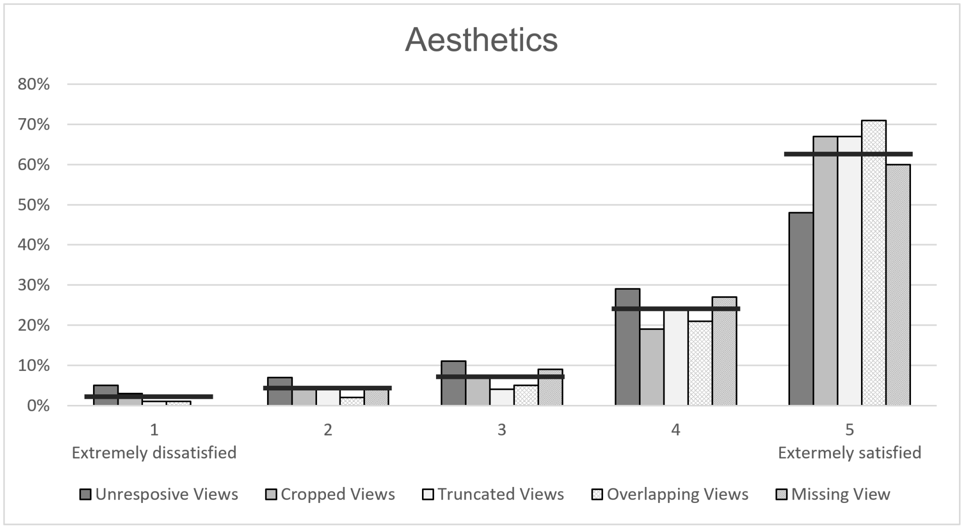


Figure 7.5: A graph summarizing how users rated Artex's proposed fixes for aesthetics.

effectiveness. A majority of the respondents rated our approach as ”extremely effective,” further emphasizing the positive reception of the solution.

Figure 7.5 displays the users' satisfaction ratings regarding the fixed UI aesthetics compared to the original UI. When rating their satisfaction on a scale from 1 (not satisfied) to 5 (extremely satisfied), participants reported an average score of 4.4. This outcome demonstrates that users generally found the aesthetics of the fixed UI to be satisfactory, indicating a positive perception of the solution's visual design.

Regarding the third question, which was an optional open-ended inquiry, participants were given the opportunity to provide their opinions on any aspects of the fix that they did not like. This feedback can help identify areas for further improvement of how Artex works. The majority of respondents reported that they were satisfied with the solution and had no suggestions. However, some participants did provide recommendations for improvement. For instance, one common theme involved optimizing UI space more effectively by adjusting the layout spacing to accommodate more information on a single screen without overwhelming the user or cluttering the interface.

In conclusion, the user study reveals that users generally found the automated repair tool effective in resolving accessibility issues and were satisfied with the aesthetics of the fixed UI.

7.4 Threats to validity

Our work is prone to several threats to validity:

- Threats to external validity refer to the factors that can limit the generalizability of our research findings to a broader context or other populations. In our evaluation and experimental setup, we assessed our approach on a dataset of apps that was utilized in prior work [13]. This specific dataset consists of apps that have been reported by users to exhibit confirmed text accessibility issues. Additionally, we applied multiple evaluation approaches and criteria, including quantitative experiments and qualitative user studies, in order to strengthen the external validity of our findings.
- Threats to internal validity relate to factors that affect the accuracy of cause-and-effect relationships within the study. To address these concerns, we implemented Artex using well-known libraries and tools, such as ADB and UIautomator, which have been extensively used in prior work. We also tried to minimize potential bugs by thoroughly testing our tool prototype to ensure reliability and robustness in our research findings.

7.5 Conclusion

In this chapter, we presented, Artex, an approach to automatically fix text accessibility issues in mobile apps. Our approach, which employs Genetic Algorithms (GA), focuses on resolving the accessibility issues while preserving the consistency of the original layout. We have discussed key GA components used in our approach, including representation, mutation, and fitness function. In addition, we explored the use of spatial graphs for layout comparison, and efficient ways utilized in our approach to manage the search space.

To evaluate the effectiveness of our approach, we used a dataset of apps with known text accessibility issues from prior work. Our approach demonstrates a promising solution to repair text accessibility issues in mobile apps, delivering both practical and aesthetically pleasing results.

Chapter 8

Conclusion

The dissertation focused on improving software accessibility for users with disabilities, specifically targeting users with low vision, and improving compatibility with Text Scaling Assistive Services (TSAS). The research began with a large empirical study of accessibility issues in Android apps, shedding light on common issues, developer challenges, and accessibility practices.

The contributions of this research include the development of AccessiText and Artex, automated tools to detect and repair text accessibility issues in Android apps. AccessiText efficiently detects text accessibility issues using heuristics backed by user feedback, with an average precision of 88% and recall of 95%. Furthermore, Artex repairs text accessibility issues employing the genetic algorithm as a search-based technique, on average fixing 83% of all identified text accessibility issues. User studies corroborate the overall effectiveness and usefulness of the generated fixes.

Future work involves enhancing our technique to support a wider range of app frameworks, particularly focusing on cross-platform app development tools such as React Native and Flutter. Apps built using these frameworks are rapidly gaining popularity; however, research suggests that they are often less accessible. Another research direction involves incorporating our detection tool into Integrated Development Environments (IDEs) and other development tools for just-in-time analysis. This real-time feedback will enable developers to identify and resolve accessibility issues within their code more efficiently than waiting to run the UI on a physical device or emulator. To achieve this, the detection method must be capable of identifying these accessibility issues based on XML code patterns or other static information. Developing a robust set of heuristics and rules that can effectively recognize potential problems in the source code will allow developers to create more accessible apps.

Bibliography

[1] Android Apktool. `http://code.google.com/p/android-apktool/`.

[2] Accessibility research mobile apps, 2021.

[3] Twitter api for academic research — products — twitter developer platform, 2021.

[4] Hayfa Y Abuaddous, Mohd Zalisham Jali, and Nurlida Basir. Web accessibility challenges. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 2016.

[5] Diamond Accessibility. 2021 state of accessibility report: Where do we stand today?, Dec 2021.

[6] ADAlaws. Adalaws, 2019.

[7] Gaurav Agrawal, Devendra Kumar, Mayank Singh, and Diksha Dani. Evaluating accessibility and usability of airline websites. In Mayank Singh, P.K. Gupta, Vipin Tyagi, Jan Flusser, Tuncer Ören, and Rekha Kashyap, editors, *Advances in Computing and Data Sciences*, pages 392–402, Singapore, 2019. Springer Singapore.

[8] Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. An empirical study of design degradation: How software projects get worse over time. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[9] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves 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. IEEE, 2016.

[10] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. Automated repair of size-based inaccessibility issues in mobile applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 730–742. IEEE, 2021.

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

[12] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam 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] Abdulaziz Alshayban and Sam 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.

[14] Ibrahim Althomali, Gregory M Kapfhammer, and Phil McMinn. Automated repair of responsive web page layouts. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 140–150. IEEE, 2022.

[15] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. IEEE, 2012.

[16] Android. Android debug bridge, 2020.

[17] Android. Build more accessible apps, 2020.

[18] AndroidAccessibility. Android accessibility overview. accessed april 12th, 2018, 2019.

[19] Androidguide. Android accessibility developer guidelines, 2019.

[20] androidmonkey. Application exerciser monkey:android developers, 2019.

[21] Androiduse. Android user worldwide, 2019.

[22] AndroZoo. Androzoo, 2019.

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

[24] Apple. Accessibility on ios, 2020.

[25] AppleAccessibility. Apple accessibility - iphone. accessed april 12th, 2018, 2018.

[26] Appleguide. Apple accessibility developer guidelines, 2018.

[27] AppleScanner. Apple accessibility scanner., 2019.

[28] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 238–249. ACM, 2016.

[29] BlindUserworldwide. Blind user worldwide, 2019.

[30] Laura V Galvis Carreno and Kristina Winbladh. Analysis of user comments: an approach for software requirements evolution. In *2013 35th international conference on software engineering (ICSE)*, pages 582–591. IEEE, 2013.

[31] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Guoqiang Li. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*, page 322–334, Virtual, 2020. ICSE.

[32] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Towards complete icon labeling in mobile applications. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2022.

[33] Paul T Chiou, Ali S Alotaibi, and William GJ Halfond. Detecting and localizing keyboard accessibility failures in web applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 855–867, 2021.

[34] Santiago Liñán Christopher Vendome, Diana Solano and Mario 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 (ICSME)*. IEEE, 2019.

[35] Mark R.T. Dale. *Spatial Graphs*, page 191–221. Cambridge University Press, 2017.

[36] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444. IEEE, 2009.

[37] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, pages 30–39, 2014.

[38] Trinidad Domínguez Vila, Elisa Alén González, and Simon Darcy. Website accessibility in the tourism industry: an analysis of official national tourism organization websites around the world. *Disability and rehabilitation*, 40(24):2895–2906, 2018.

[39] Werner Dubitzky, Olaf Wolkenhauer, Hiroki Yokota, and Kwang-Hyun Cho. *Encyclopedia of systems biology*. Springer Publishing Company, Incorporated, 2013.

[40] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon 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.

[41] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE, 2018.

[42] espresso. Espresso : Android developers, 2019.

[43] forbes. forbes, 2019.

[44] Andre P Freire, Cibele M Russo, and Renata PM Fortes. A survey on the accessibility awareness of people involved in web development projects in brazil. In *Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A)*, pages 87–96, 2008.

[45] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. Every pixel counts: Fine-grained ui rendering analysis for mobile applications. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[46] Google. google/accessibility-test-framework-for-android, Mar 2018.

[47] Khronos Group. The industry's foundation for high performance graphics, 2019.

[48] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *2014 IEEE 22nd international requirements engineering conference (RE)*, pages 153–162. Ieee, 2014.

[49] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh 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, 2014.

[50] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in neural information processing systems*, 31, 2018.

[51] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[52] Claudia Iacob and Rachel Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *2013 10th working conference on mining software repositories (MSR)*, pages 41–44. IEEE, 2013.

[53] Stéphane Jacquet, Xavier Chamberland-Thibeault, and Sylvain Hallé. Automated repair of layout bugs in web pages with linear programming. In *Web Engineering: 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, Proceedings*, pages 423–439. Springer, 2021.

[54] Mark Kasunic. Designing an effective survey. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2005.

[55] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE software*, 32(3):70–77, 2014.

[56] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[57] Royce Kimmons. Open to all? nationwide evaluation of high-priority web accessibility considerations among higher education websites. *Journal of Computing in Higher Education*, 29(3):434–450, 2017.

[58] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[59] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lü. Characterizing and detecting inefficient image displaying issues in android apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 355–365, 2019.

[60] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 352–361. IEEE, 2015.

[61] Lint. Improve your code with lint checks., 2019.

[62] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722, 2016.

[63] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*, pages 229–246. Springer, 2017.

[64] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[65] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. Xfix: an automated tool for the repair of layout cross browser issues. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 368–371, 2017.

[66] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2017.

[67] Forough Mehralian, Navid Salehnamadi, and Sam 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, 2021.

[68] Matthew B Miles, A Michael Huberman, and Johnny Saldaña. *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[69] Lauren R Milne, Cynthia L Bennett, and Richard E Ladner. The accessibility of mobile health sensors for blind users. 2014.

[70] Maleknaz Nayebi, Henry Cho, and Guenther Ruhe. App store mining is not enough for app improvement. *Empirical Software Engineering*, 23(5):2764–2794, 2018.

[71] nltk. Natural language toolkit¶, 2019.

[72] Dennis Pagano and Walid Maalej. User feedback in the appstore: An empirical study. In *2013 21st IEEE international requirements engineering conference (RE)*, pages 125–134. IEEE, 2013.

[73] Kyudong Park, Taedong Goh, and Hyo-Jeong So. Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of HCI Korea*, pages 31–38. Hanbit Media, Inc., 2014.

[74] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 2018.

[75] Neha Patil, Dhananjay Bhole, and Prasanna Shete. Enhanced ui automator viewer with improved android accessibility evaluation features. In *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*, pages 977–983. IEEE, 2016.

[76] qualtrics. qualtrics. `https://www.qualtrics.com/`, 2019. Accessed: 2019-08-17.

[77] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob 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. ACM, 2017.

[78] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob 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*, pages 119–130. ACM, 2018.

[79] Kabir S Said, Liming Nie, Adekunle A Ajibode, and Xueyi Zhou. Gui testing for mobile applications: objectives, approaches and challenges. In *12th Asia-Pacific Symposium on Internetware*, pages 51–60, 2020.

[80] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. Latte companion website. `https://github.com/seal-hub/Latte`, 2020.

[81] Navid Salehnamadi, Ziyao He, and Sam 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.

[82] Navid Salehnamadi, Forough Mehralian, and Sam Malek. Groundhog: An automated accessibility crawler for mobile apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[83] scanner. Accessibility scanner - apps on google play, 2019.

[84] Leandro Coelho Serra, Lucas Pedroso Carvalho, Lucas Pereira Ferreira, Jorge Belimar Silva Vaz, and André Pimenta Freire. Accessibility evaluation of e-government mobile applications in brazil. *Procedia Computer Science*, 67:348–357, 2015.

[85] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 545–555, 2019.

[86] Android Accessibility Study. Android accessibility study, 2019.

[87] Amanda Swearngin and Yang Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Artificial Intelligence for Human Computer Interaction: A Modern Approach*, pages 73–96. Springer, 2021.

[88] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 187–198, 2018.

[89] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[90] Jason Taylor. 2018 ada web accessibility lawsuit recap report [blog], 2018.

[91] Unity Technologies. Unity, 2019.

[92] w3c. w3c, 2019.

[93] Bruce N Walker, Brianna J Tomlinson, and Jonathan H Schuett. Universal design of mobile apps: Making weather information accessible. In *International Conference on Universal Access in Human-Computer Interaction*, pages 113–122. Springer, 2017.

[94] WCAG. Web content accessibility guidelines (wcag) overview, 2019.

[95] WebAIM. Screen reader user survey 7 results. retrieved may 10, 2018, 2018.

[96] Brian Wentz, Dung Pham, Erin Feaser, Dylan Smith, James Smith, and Allison Wilson. Documenting the accessibility of 100 us bank and finance websites. *Universal Access in the Information Society*, pages 1–10, 2018.

[97] WHO. World report on disability, 2011.

[98] who. World health organization. (2011). world report on disability, 2019.

[99] Shunguo Yan and PG Ramachandran. The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)*, 12(1):3, 2019.

[100] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. Don't do that! hunting down visual design smells in complex uis against design guidelines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 761–772, 2021.

[101] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1286–1297. IEEE, 2020.