AccessiText: Automated Detection of Text Accessibility Issues in Android Apps

Abdulaziz Alshayban
School of Information and Computer Sciences
University of California, Irvine , USA
aalshayb@uci.edu

Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
malek@uci.edu

ABSTRACT

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 paper 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 88.27% precision and 95.76% recall on average in detecting text accessibility issues.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Human-centered computing** → **Accessibility design and evaluation methods**.

KEYWORDS

Accessibility, Automated Testing, Mobile Application

ACM Reference Format:
Abdulaziz Alshayban and Sam Malek . 2022. 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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549118

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549118

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 [41], 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 [11, 16]. Disappointingly, several studies [8, 36, 38] 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) [1], 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) [40], 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

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

Listing 1: A TextView that defines its size in terms of SP units. The text displayed will scale based on the user's preference.

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 [8, 18, 19, 38], 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 [1].

To facilitate a proper understanding of text accessibility issues, this paper 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 paper 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 paper is structured as follows: In Section 2 we provide a brief background. In Section 3, we present our study on identifying the different types of text accessibility issues. In Section 4, we describe how our approach, AccessiText, works. In Section 5, we present our findings. Section 6 discusses the results and outline relevant insights. We provide a a brief review of prior research efforts in Section 8.

2 BACKGROUND

The user interface (UI) for an Android app is made up of a series of `View` and `ViewGroup` elements. Generally, an Android app contain one or more activities (i.e., screens), with each activity consisting of multiple instances of `View` and `ViewGroup`. The `ViewGroup` class is a subclass of the `View` class, and acts as a base class for layouts and views containers. The role of a `ViewGroup` is to provide an invisible container to hold other views and to define the layout properties that control how their child views are positioned on the screen. A `View` is defined as the user interface element which is used to create interactive UI views such as `TextView`, `ImageView`, etc., and is responsible for drawing and event handling.

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 the 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 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 properties such as `adjustsFontForContentSizeCategory` for 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) [40], 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. This requirement is also outlined in Apple Human Interface Guidelines [4] and Google Design Guidelines [11].

3 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 paper.

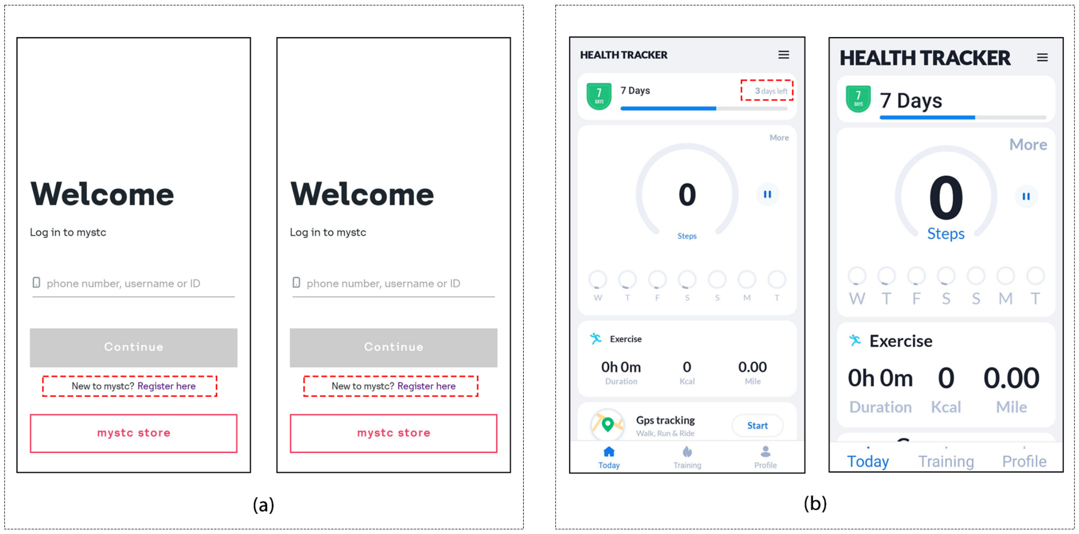

Figure 1: Examples of (a) unresponsive view issue, and (b) missing view issue

3.1 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 [26, 27, 29].They can provide information such as bugs or issues [34], summary of user experience [23], request for features and enhancements [17]. 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 [33, 34]. Thus, mining Twitter data provides significant valuable insights into the types of accessibility issues experienced by mobile apps users. We used the Twitter Academic API [2] 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.

The data collected from the two sources listed above was manually analyzed following a procedure inspired by open coding [32]. 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 extracts 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 detail 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.

3.2 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

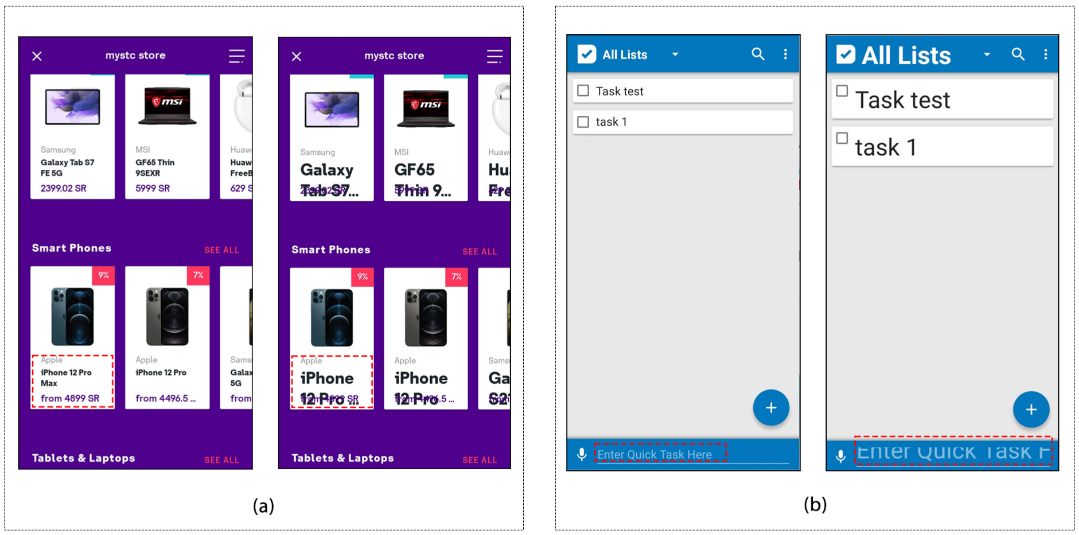
mystc store
Samsung Galaxy Tab S7 FE 5G 2399.02 SR
MSI GF65 Thin 9SEXR 5999 SR
Smart Phones
SEE ALL
9%
7%
Apple iPhone 12 Pro Max from 4899 SR
Apple iPhone 12 Pro from 4496.5
Tablets & Laptops
All Lists
Task test
task 1
Enter Quick Task Here
(a)
(b)

Figure 2: Examples of (a) overlapping views issue, and (b) cropped view issue

by TSAS, making this assistive service useless to the user. Figure 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 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 these 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.

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,

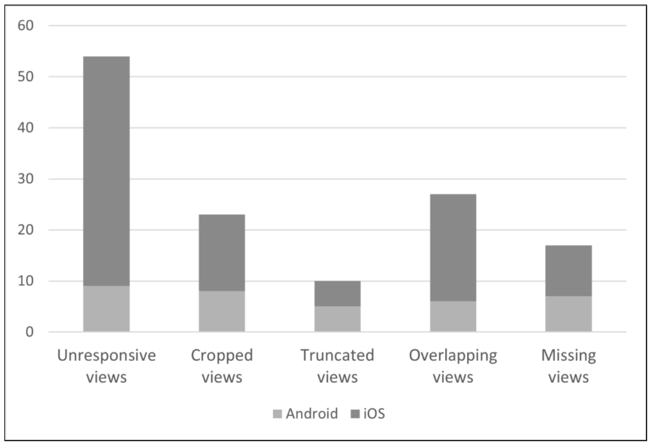

Figure 3: Number of text accessibility issues grouped by platform

resulting in one of the views covering the content of the other. Figure 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 the limited space and poorly defined constraints behind these views.

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 issue happens when the displayed text grows beyond the constrained height of the containing view, causing part of the text to be invisible. Figure 2 (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.

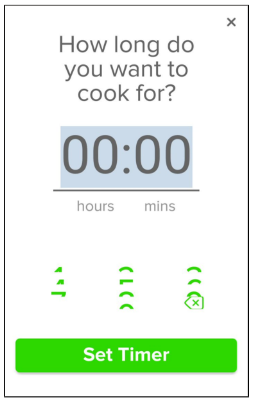

Figure 4: A cropped view accessibility issue for `AnovaCulinary` app as reported by a user.

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 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 5 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 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 [6], 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.

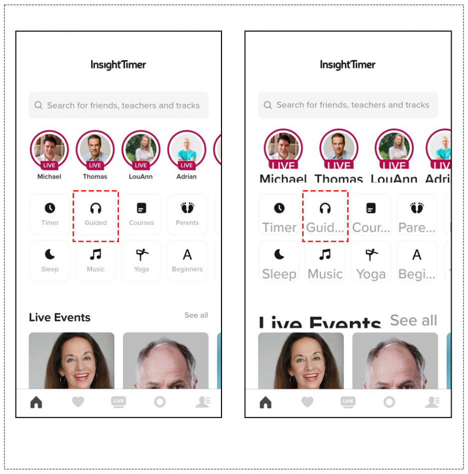

Figure 5: Example of a truncated view

4 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 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 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 [15]. In the remainder of this section, we describe AccessiText's two components in detail.

4.1 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

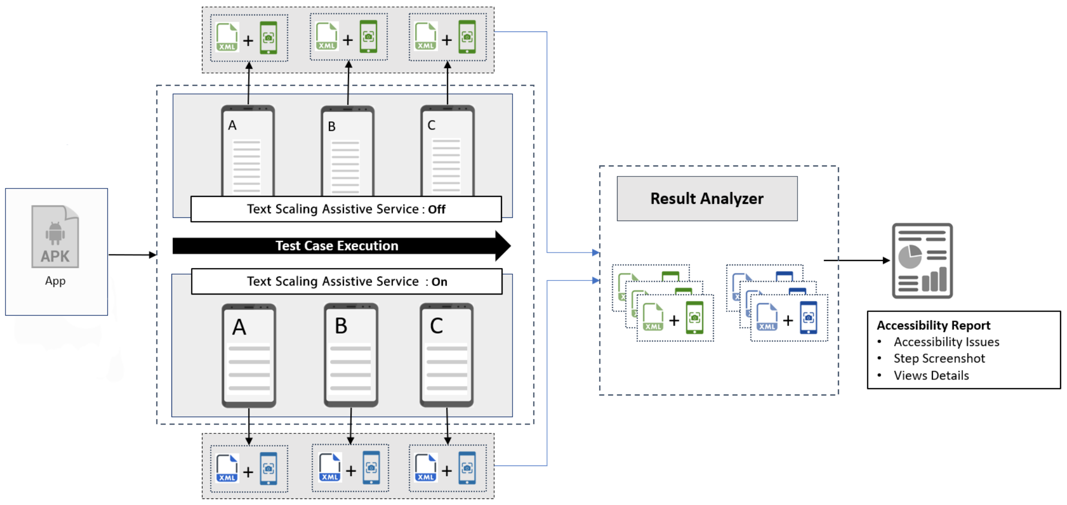

Figure 6: Overview of AccessiText

details of each UI view in the current screen. Properties details include information such `app-name`, `view-class`, `bounds`, and `text`. Listing 2 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 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

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

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.

4.2 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 3.

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.

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.

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 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.

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 and true negatives 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.

Finally, AccessiText generates an accessibility report that provides a description of all the accessibility issues and their contextual information. Moreover, the report provides additional information such as the level of overlap between the UI elements for issues of type *Overlapping Views*, and the extent of cropping for issues of type *Cropped Views*, which developers may utilize to prioritize and sort the provided accessibility issues based on their severity and impact.

5 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?

5.1 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 3. 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 [15], 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 (Galaxy S8) 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 [40], which requires that an app's textual content be resizable up to double the default size without losing content or functionality.

5.2 Effectiveness of AccessiText

To answer this question, we carefully checked each accessibility issue found by AccessiText to ensure their correctness. Table 1

Table 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 *	9	-	2	11	-	22	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	10	1	12	47
StepTracker	-	1	3	2	-	6	60
Goal Tracker	-	-	6	1	-	7	48
GetUpside	-	1	5	2	-	8	56
STC	24	-	6	-	-	30	83
Insight Timer	1	-	-	3	2	6	31
To Do List	-	-	-	2	-	2	53
Vocabulary	-	12	8	2	-	22	92
Google Auth	1	-	2	2	-	5	59
Lose it	-	3	18	1	-	22	51
AllTrails	-	3	-	-	-	3	63
Roadie	-	1	6	4	-	11	95
Fedex	3	-	-	5	-	8	49
RecipeKeeper	-	1	4	-	-	5	80
Investment Portfolio	-	2	12	3	4	21	168

Table 2: Precision and recall of AccessiText

	# of Detected Issues	Precision	Recall
Unresponsive Views	58	98%	100%
Missing Views	31	80.64%	100%
Overlapping Views	122	89.34%	100%
Cropped Views	67	73.13%	94.23%
Truncated Views	11	100%	84.61%
Total	289	88.27%	95.76%

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 3. Table 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 88.27% precision and 95.76% recall for the different types of issues. Thereby, AccessiText is 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 2 to 30 with an average of 9.5 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 13 apps, respectively, of the 30 apps in our dataset. Overlapping views has the highest average number of occurrences in each app.

Table 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.

5.3 Performance of AccessiText

The last column of Table 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 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 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 [5, 8, 20] 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.

7 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 [36] 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.

8 RELATED WORK

8.1 Accessibility Testing

Accessibility analysis can be difficult and time-consuming, as it requires human expertise and judgement to determine what barriers may exist for people with disabilities. Researchers have investigated various ways of automating the accessibility analysis process [12, 19, 24, 35], which can be broadly categorized into two categories: static and dynamic accessibility analysis.

Lint [13] 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 [19]. Accessibility Scanner [10], the recommended tool from Google to test apps for accessibility, is based on the Accessibility Testing Framework [22], an open-source library of various automated checks for accessibility, and it can detect a wide set of accessibility issues. Alshayban et al. [8] proposed an automated accessibility testing technique by implementing a random crawler to simplify the process of accessibility testing. MATE [19] 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 [38] 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 [18] 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.

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

8.2 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 [37] of these tools [9, 14, 25, 30] 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 [39] proposed a deep learning based technique for uncovering potential usability issues in UI elements tappability. Seenomaly [43] 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 [21] helps developers optimize the UI rendering performance of their mobile apps performance by identifying the UI rendering delay problems. TAPIR [28] is a static analysis tool for identifying inefficient image displaying (IID), which can impact the app performance and user experience. UIS-Hunter [42] 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. Additionally, various studies [7, 31] focused on web apps, specifically, the the detection and repair of presentation issues that are the result of internationalization or cross-browser failures.

Overall, none of the above-mentioned solutions investigate the use of TSAS in mobile apps, and how it can introduce accessibility barriers for users.

9 CONCLUSION

This paper 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.

The research artifacts for this study are available publicly at the companion website [3].

ACKNOWLEDGMENT

This work was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] 2021. Accessibility Research Mobile Apps. https://accessibility.q42.nl/

[2] 2021. Twitter API for academic research | products | twitter developer platform. https://developer.twitter.com/en/products/twitter-api/academic-research

[3] 2022. Accessitext. https://sites.google.com/view/accessitext/home

[4] 2022. human interface guidelines. https://developer.apple.com/design/human-interface-guidelines/accessibility/overview/text-size-and-weight/

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

[6] Diamond Accessibility. 2021. 2021 state of Accessibility report: Where do we stand Today? https://blog.diamond.la/the-state-of-accessibility-report-where-do-we-stand-today

[7] Abdulmajeed Alameer, Paul T Chiou, and William GJ Halfond. 2019. Efficiently repairing internationalization presentation failures by solving layout constraints. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 172–182.

[8] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. ICSE, Virtual, 1323–1334.

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

[10] Android. 2020. Accessibility Scanner - Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US.

[11] Android. 2020. *Build more accessible apps.* Retrieved August 20, 2020 from https://developer.android.com/guide/topics/ui/accessibility

[12] Android. 2020. *Espresso : Android Developers.* Google. Retrieved August 20, 2020 from https://developer.android.com/training/testing/espresso

[13] Android. 2020. *Improve your code with lint checks.* Google. Retrieved August 20, 2020 from https://developer.android.com/studio/write/lint?hl=en

[14] androidmonkey. 2019. Application Exerciser Monkey:Android Developers. https://developer.android.com/studio/test/monkey.html

[15] Appium. 2020. Mobile App Automation Made Awesome. http://appium.io/.

[16] Apple. 2020. *Accessibility on iOS.* Retrieved August 20, 2020 from https://developer.apple.com/accessibility/ios/

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

[18] Paul T Chiou, Ali S Alotaibi, and William GJ Halfond. 2021. 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*. 855–867.

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

[20] Andre P Freire, Cibele M Russo, and Renata PM Fortes. 2008. 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)*. 87–96.

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

[22] Google. 2018. google/Accessibility-Test-Framework-for-Android. https://github.com/google/Accessibility-Test-Framework-for-Android

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

[24] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. 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*. 204–217.

[25] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 77–83.

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

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

[28] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lü. 2019. Characterizing and Detecting Inefficient Image Displaying Issues in Android Apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 355–365. https://doi.org/10.1109/SANER.2019.8668030

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

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

[31] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 249–260.

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

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

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

[35] Neha Patil, Dhananjay Bhole, and Prasanna Shete. 2016. Enhanced UI Automator Viewer with improved Android accessibility evaluation features. In *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*. IEEE, 977–983.

[36] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2017. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*. ASSETS, Baltimore, MD, USA, 2–11.

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

[38] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–11.

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

[40] W3. 2020. *Web Content Accessibility Guidelines (WCAG) Overview.* World Wide Web Consortium. Retrieved August 20, 2020 from https://www.w3.org/WAI/standards-guidelines/wcag/

[41] WHO. 2011. World report on disability. Retrieved August 20, 2020 from https://www.who.int/disabilities/world_report/2011/report/en/

[42] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. 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)*. 761–772. https://doi.org/10.1109/ICSE43902.2021.00075

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