Examining Image-Based Button Labeling for Accessibility in Android Apps through Large-Scale Analysis

Anne Spencer Ross[1], Xiaoyi Zhang[1], James Fogarty[1], Jacob O. Wobbrock[2]
[1]Paul G. Allen School of Computer Science & Engineering, [2]The Information School
DUB Group, University of Washington, Seattle, WA, USA
{ansross, xiaoyiz, jfogarty}@cs.washington.edu, wobbrock@uw.edu

ABSTRACT
We conduct the first large-scale analysis of the accessibility of mobile apps, examining what unique insights this can provide into the state of mobile app accessibility. We analyzed 5,753 free Android apps for label-based accessibility barriers in three classes of image-based buttons: Clickable Images, Image Buttons, and Floating Action Buttons. An epidemiology-inspired framework was used to structure the investigation. The *population* of free Android apps was assessed for label-based inaccessible button *diseases*. Three *determinants* of the *disease* were considered: missing labels, duplicate labels, and uninformative labels. The *prevalence*, or frequency of occurrences of barriers, was examined in apps and in classes of image-based buttons. In the app analysis, 35.9% of analyzed apps had 90% or more of their assessed image-based buttons labeled, 45.9% had less than 10% of assessed image-based buttons labeled, and the remaining apps were relatively uniformly distributed along the proportion of elements that were labeled. In the class analysis, 92.0% of Floating Action Buttons were found to have missing labels, compared to 54.7% of Image Buttons and 86.3% of Clickable Images. We discuss how these accessibility barriers are addressed in existing *treatments*, including accessibility development guidelines.

Author Keywords
Mobile app; large-scale analysis; accessibility; image-based buttons

ACM Classification Keywords
• **Human-centered computing~Empirical studies in accessibility** • Human-centered computing~Heuristic evaluations

INTRODUCTION
Mobile applications (apps) are becoming increasingly important in daily life, providing information and services in a range of settings that include banking, communication, education, entertainment and travel. It is important that these powerful capabilities are available to all people, regardless of their abilities or use of assistive technologies. However, prior work has shown there still exist important accessibility barriers within apps [20,21,37,40,42,43].

Awareness of the need to create more accessible apps is increasing. Google and Apple are the primary organizations that facilitate mobile technology and the app marketplace, through the Android and iOS platforms. Both have released developer and design guidelines for accessibility [16,30], provide accessibility services as part of their platforms [2,15], have app development libraries that include built-in compatibility with assistive technologies, and have released accessibility testing scanners [17,28] and suites [10]. Some companies creating popular apps have also made statements and taken actions to create more accessible apps [19,45]. Such approaches have included creating internal guidelines [45], having specialized accessibility teams [19], actively prioritizing accessibility [19,45], and working with people with disabilities during app development and testing [45].

Despite these accessibility-focused efforts, studies of relatively small groups of apps have found they still include significant accessibility barriers [20,21,37,42,43]. This suggests a continuing need for accessibility improvements, however, the field lacks a detailed understanding of the state of mobile app accessibility at a large-scale, "population" level.

Many design patterns in mobile apps are image and icon focused and use image-based buttons for main functionalities. One key component of accessibility for screen reader users is labeling image-based buttons. This need is parallel to the need for alt-text for images on the web. However, there is no large-scale understanding of the prevalence of unlabeled image-based buttons, how effective tools are at promoting labeling, nor the potential causes of failure to label. Ross *et al*. [42] present an epidemiology-based framework that suggests large-scale analyses can help answer some of these questions. The framework emphasizes that apps do not exist in isolation. It suggests that, in addition to testing individual apps, additional benefits and insights can be gained by exploring app accessibility at the population-level, situated within the richer ecosystem of influential factors. Such analyses can give unique insights into the state of app accessibility and opportunities for improvements. An epidemiologically-inspired analysis can also establish a baseline against which to measure the evolution of app accessibility over time.

ASSETS '18, October 22–24, 2018, Galway, Ireland

ACM ISBN 978-1-4503-5650-3/18/10…$15.00
https://doi.org/10.1145/3234695.3236364

This work provides the first large-scale analysis of image-based button labeling for accessibility in free Android apps. Using Ross *et al.*'s epidemiology-based framework [42], we look at three label-based accessibility errors: *missing label, duplicate label,* and *uninformative label*. We analyze the occurrence of these errors in three popular classes of interactive image-based buttons: Clickable Images, Image Buttons, and Floating Action Buttons (FABs). Elements are tested in a dataset of 5,753 apps. We explore patterns of error occurrences within apps as well as within the three classes of image-based button. We also discuss potential factors contributing to these patterns, such as whether and how these errors are presented in existing developer guides.

Our analysis demonstrates a concrete application of concepts from Ross *et al.'s* epidemiology-inspired framework [42]. We present a large-scale analysis of the labeling of image-based buttons. Additionally, we explore how accessibility barriers are presented and addressed in the tools and techniques developers may use to create and test Android apps. We discuss how that presentation may relate to the results of the app analysis. The insights gained through this research can guide data-driven enhancements to app accessibility. Our work additionally demonstrates the value of large-scale app accessibility analysis.

RELATED WORK

Although there has not been a prior large-scale analysis of mobile app accessibility, related work has been performed on apps and websites. Small-scale analyses have identified that app accessibility is still a problem and informed our choice to focus on label-based inaccessibility in buttons. The small-scale analyses highlighted the need for a more holistic understanding of app accessibility at a larger scale. Prior work performing large-scale analyses of apps for purposes other than accessibility has demonstrated that such analyses can provide unique views into the state of apps. One such study also provided the data analyzed in this work [24]. Finally, prior work in large-scale analyses of web accessibility exemplifies the types of insights that can be gained through large-scale and longitudinal study.

Small-Scale App Accessibility Analysis

App accessibility has been investigated in small-scale studies. These analyses reveal the continued existence of accessibility barriers within apps, notably label-based barriers.

Some accessibility analyses focused on specific categories of apps, as in health [37], smart cities [20], and government engagement [43]. Others took a more general sample of apps [21,40]. The number of apps analyzed ranged from 4 to 10. 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.

For accessibility metrics, these studies largely used adapted versions of the Web Content Accessibility Guidelines (WCAG) [20,21,22,43,44]. Industry-released guidelines [37] and Section 508 [37] were also used for creating assessment criteria. One of our tests for label-based accessibility barriers in image-based buttons is based on best labeling practices presented in guidelines. The other two tests we apply are based on the Accessibility Test Framework for Android [29]. This testing framework covers the same concepts for image labeling as the metrics of prior work. It is also specifically designed for testing Android apps.

Image labeling has been explicitly noted as a significant problem [20,37,40,43]. Park *et al.* [40] rated the *severity* of errors as well as frequency. Missing labels was rated as the highest severity of the ten errors they tested, at 6.5 out of 7. These results show that accessibility barriers exist within apps and label-based errors are a worthwhile initial focus.

Large-Scale App Analysis

Large-scale app analyses have been used to explore app characteristics other than accessibility. These studies demonstrate the richness of insights that can be gained from large population-level analyses.

App sample sizes for these large-scale analyses were on the order of thousands of apps [18,24,25,26,34,39]. Another study collected app usage data from 77 participants over nine months [38] instead of focusing on a specific set of apps.

A range of topics was explored, including security [1,26,34], design patterns [24,25], and code reuse trends [39]. App usage has also been explored, including usage contextualized by time and location [18,26] or social context [38]. Ours is the first population-scale exploration of app accessibility.

Large datasets of Android apps have been released for analysis. Deka *et al.* [24] crawled ~10,000 free Android apps and collected metadata, screen shots, and View hierarchies, as discussed in the Data section below. The data is available in the Rico repository [23]. Deka *et al.* [24] analyzed the dataset for common app design patterns. We use the Rico dataset for our analysis due to its size and detail.

Alli *et al.*'s Androzoo [1] project has collected over 5 million app APKs[1] and makes them available for academic use. App APKs have been regularly added to the dataset since 2011. Refining methods to capture data from app APKs is an opportunity for future work that could allow leveraging this dataset for large-scale, longitudinal accessibility analysis.

Large-Scale Web Accessibility

The web has a long history of automated accessibility analysis, including at a large scale. Insights gained from these large-scale analyses further motivate the need for similar studies of app accessibility. Hanson *et al*. [33] performed a longitudinal study of 100 top government and commercial websites over 14 years. Findings include that accessibility overall improved over time. In follow-up work,

[1]APKs are Android's file package for the installation and running of apps. It is similar to a .exe file in Windows.

Richards *et al.* [41] discussed potential factors that contribute to these trends, such as changes in overall web coding practices. Kane *et al.* [35] analyzed 100 websites for accessibility, including for missing labels. They found "[o]n average, 77% of significant (non-decorative) images were labeled on each page." (p153). This again shows that unlabeled images are an important problem.

ANDROID BACKGROUND

Understanding the context in which apps are created and tested is imperative to understanding how that context impacts app accessibility. Android is a large, open, and diverse ecosystem. There are many styles, methods, guides, and tools for creating apps. We focus our analysis on approaches to creating image-based buttons that are part of the core Android API. We chose image-based buttons because they are a key component of interactivity in apps. The image-based nature of these buttons makes them particularly susceptible to label-based accessibility barriers.

The components of the app ecosystem we focus on are: (1) three Android element classes for creating image-based buttons; (2) Android-released design and development tools; and (3) testing tools. In this section, we define important concepts in each component to establish a foundational understanding of the Android app environment. Each of the concepts described plays a key factor in our analysis.

Relevant Android Classes of Image-Based Buttons

Our analysis is performed on three commonly-used classes of Android image-based button: Clickable Image, Image Button, and Floating Action Button (Figure 1). We present details on the usage of each class of button as well as how the three classes relate to one another, including the date each class of button was added to the Android platform. Knowing how long the class has been available provides context for later discussions on how well the element has been integrated into the larger Android app environment.

The image-based nature of the classes of button we analyze make them susceptible to label-based errors. In order to properly label an image-based button such that it interacts properly with screen readers, alternative text descriptions must be added in the button's *content description* field.

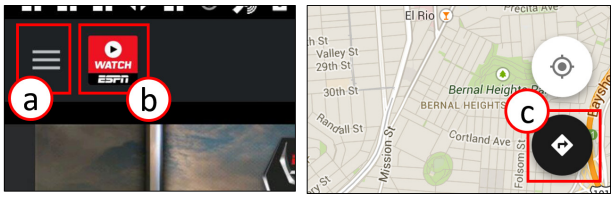

Figure 1: Screens from apps with examples of image-based buttons using (a) an Image Button, (b) a Clickable Image, and (c) a Floating Action Button.

Clickable Images

Images can be rendered in an app using elements from the Android API class `android.widget.ImageView` [9]. If the *clickable* property is set to true, the image functions as a button (Figure 1b). We call such elements Clickable Images. Clickable Images have slightly different defaults and rendering than Image Buttons (which are discussed below). Non-decorative images should be labeled with a *content description*. The `ImageView` class has been in the Android API since Android 1.0, released in September 2008.

Some images may be decorative and therefore should be labeled with a null string to properly be avoided by screen readers. The interactivity of *clickable* images indicates a non-decorative functionality. We therefore treat all assessed Clickable Images as non-decorative, and a null string label as a *missing label* accessibility barrier (discussed in the Label-Based Inaccessible Button Disease subsection below).

Image Button

Image Buttons are from the Android API base class `android.widget.ImageButton` [8]. This is a sub-class of the Clickable Image's `ImageView` class. As the name suggests, Image Buttons are buttons that visually present an image rather than text (Figure 1a). Image Buttons were part of the Android 1.0 release in September 2008.

Floating Action Button (FAB)

Floating Action Buttons (FABs) are visually prominent buttons that "float" above an underlying interface (Figure 1c). According to Google's Material Design guidelines [27], "a floating action button represents the primary action in an application." Simple icons, such as a heart or pencil, are usually the visual label. FABs rarely have visual text labels. Given the importance of FABs for key functionality and their image-based style, proper labeling with *content descriptions* is imperative for screen reader compatibility.

FABs are from the class `android.support.design.widget.FloatingActionButton` [6]. It is a sub-class of `ImageButton` and a sub-sub-class of `ImageView`. FABs were added in Android 22.2.0 in May 2015 [6], much later than the other two classes of image-based button.

Current Android Tools

The explicit or implicit emphasis that app development and testing tools put on accessibility (e.g., in their default settings, in tests they perform, in the guidance they give) can impact the accessibility of apps created with that tool. If a tool is widely used, that can amount to a large influence. We present here a suite of Android development and design tools and testing suites. Some of these tools are focused on enhancing accessibility. In the Analysis section below, we analyze these tools as potential factors that influence app accessibility. We focus heavily on Android authored tools, due to their impact, availability, and because the classes of image-based buttons we focus on are part of Android's base API. The analysis approaches used and the insights gained are likely transferrable to other tools.

Android Design and Development Tools

Android publishes many resources to aid designers and developers in creating Android apps. These resources include guidelines and example code. Guides address good practices for creating and testing apps [12,28,32,36]. The

design guidelines are called Android Material Design [36]. A subset of these guides focus on accessibility [3,30]. Android also has Quality Guidelines [12] to articulate what key components an app should have and what fundamental tests an app should pass to be ready for distribution. The Quality Guidelines do not explicitly mention accessibility.

Android additionally provides example source code to aid in app development. Within the collection of code samples are specific example projects for using FABs [7] and for accessibility-minded development [5]. In the Analysis section below, we discuss how the content of the Material Design and developer guidelines might impact app accessibility.

Android Testing Tools

Android has a set of testing tools to run while developing Android apps. We focus on the general testing tool Android Lint (v23.0.0) [10] and the accessibility-specific Accessibility Test Framework for Android [29]. We describe these tools here and the sections below explore details of these tests that may impact the app accessibility trends seen in our analysis.

Android Lint is a code scanning tool that can help "identify and correct problems with the structural quality of your code without your having to execute the app or write test cases" [10]. Bugs detected by Lint include some instances of label-based button accessibility barriers. The details of which barriers are caught in what context are explored in the Analysis section. Android Lint is both available as a standalone tool and also integrated into Android Studio v3.0.1 [14], Google's development environment for apps.

The Accessibility Test Framework for Android [29] is a testing suite released by Google. It tests for accessibility barriers in apps, including missing labels and duplicate labels on image-based buttons. It can be integrated into unit or automated testing. This test framework is the basis for two of our three error tests, as explained in the Method section.

APPLYING AN EPIDEMIOLOGY-INSPIRED FRAMEWORK

We structured our analysis using an epidemiology-inspired framework [42]. In the framework, accessibility barriers are cast as *inaccessibility diseases* within a *population* of apps. This framing puts the onus of accessibility on the app. The epidemiology framework instigates an analytical approach by which we investigate the *label-based inaccessible button disease* in the *population* of free Android apps. The framework provides motivation and structure for large-scale, multi-factor analyses. We focus on the key concepts of *disease, population, diagnostic criteria, prevalence, risk factors,* and *treatments,* and how they relate to the higher-level objectives of *determining the extent of a disease,* and *evaluating treatments.* Further detail on these definitions and their role in the larger framework can be found in [42].

Key Terminology

The following key terminology provides a language with which to discuss the data and phenomena that we analyzed.

Label-Based Inaccessible Button Disease: a barrier to using an image-based button with a screen reader due to the button not having an appropriate alternative label. The *determinants*, or causes, of this disease that we test for are: missing a label, having a label that is a duplicate of other labels on the screen, or having an uninformative label.

Populations: a group of individual units of measurement that we wish to better understand. We consider the population of apps that have at least one element of any of the three classes of image-based button: Clickable Image, Image Button, or Floating Action Button. We additionally analyze the population of image-based buttons, grouped by class.

Diagnostic Criteria: the metrics used to determine if an app or image-based button has a *label-based inaccessible button disease*. Our diagnostic criteria are captured in our tests for missing, duplicate, or uninformative labels. Implementation of these tests is detailed in the Method section below.

Prevalence: the count or proportion of units (i.e., apps or image-based buttons grouped by class) with a specific *determinant* (i.e. missing, duplicate, or uninformative label) of the *label-based inaccessible button disease.*

Risk Factors: a characteristic of an app or image-based button that affects how likely it is to have the *disease*. An app's rating is an example app characteristic. That a class of image-based buttons is frequently used in sample code is another example factor.

Treatment: a technique used to prevent (i.e., preventative) or repair (i.e., therapeutic) an *inaccessibility disease*. *Treatments* include design and development guidelines and testing tools.

Objectives of Our Analysis

The epidemiology-inspired framework [42] is centered around a set of primary objectives. In addition to terminology, we focused on two objectives to structure our analysis: determining the extent of a *disease* in a population and evaluating existing and new *treatments.*

Determining the Extent of a Disease

Understanding how often different accessibility barriers occur is one metric for understanding the impact of those barriers. It can also inform how to allocate resources for enhancing app accessibility or how to develop new interventions. The power of these types of insights is captured in the epidemiology-inspired objective of *determining the extent of a disease in a population*. The main metric we use for measurement is *prevalence.*

Measuring the *extent of a disease* over time can provide new insights. For example, longitudinal comparisons can indicate if the accessibility of apps is improving over time. Analyses can additionally provide guidance regarding significantly impactful factors. For example, if a wide-spread decrease in accessibility is seen after a major operating system update, it may indicate that the update should be investigated for its impact on accessibility. The analysis in this paper offers a baseline with which to compare future analyses of *label-based inaccessible button diseases* on Clickable Images, Image Buttons, or FABs.

Evaluating Existing and New Treatments

Tools, techniques, and tests (collectively called *treatments*), aimed at enhancing the accessibility of apps exist, such as those described in the above section. Evaluating the effectiveness of these *treatments* aids in creating tools that have the largest impact in the most efficient way. Large-scale analysis can guide such evaluations. Patterns of *prevalence* within a population can indicate what elements or apps are more or less likely to be inaccessible.

Motivated by this objective, we explore the existing *treatments* listed in the above section as potentially impacting accessibility. A discrepancy in the *prevalence* of missing labels between the three classes of image-based buttons informed our *treatment* exploration. We compare how the three types of elements are represented in the *treatments.*

Large-scale analysis alone cannot prove causation between treatments and the resulting accessibility of apps. However, it is a powerful component to guide complementary work such as interviews and user studies.

METHOD

The app data used in this work was a subset of data from the Rico repository [23]. We executed tests for three *determinants,* or causes, of the *label-based inaccessibility button disease*, checking for (1) missing labels, (2) duplicate labels, and (3) uninformative labels. The tests for missing and duplicate labels are based on the Accessibility Test Framework for Android [29]. The test for uninformative labels is based on a list of labels in our data that obviously violated good labeling practices, as determined by the first author. The dataset and the test definitions and implementations are detailed in the following sections.

Ratios are the primary metric used to present the *prevalence* findings. Ratios were chosen for our analysis to compare apps that had a range of number of elements used. The range of elements used within the extreme *prevalence* groups mirrors that of the overall apps, as described in the Analysis sections. This suggests that the insights gained apply across a variety of apps. Prior work has explored multiple methods for accessing accessibility on the web and compared the strengths and weaknesses of different approaches [46]. Such a comparison of the trade-offs in methods for assessing app accessibility is an opportunity for future work.

Data

Our dataset is a subset of the Rico repository [23]. We focus on the app metadata and View hierarchies. The metadata provides characteristics of each app, including its rating. The View hierarchies contain all captured elements of the screen in a nested, hierarchical structure. Each element has a set of characteristics including text, content description, class, ancestor classes, and children elements. Details of this dataset can be found in the paper published by Deka *et al.* [24].

We obtained our dataset by filtering the 9,772 free Android apps collected in the Rico repository [23] by the exclusion criteria described below. Our subset of the data contained 5,753 apps. The captured image-based buttons from these apps include 134,506 Clickable Images, 137,665 Image Buttons, and 6,579 FABs.

Exclusion Criteria

If a View hierarchy file was *null*, the screen it represented was ignored. In the dataset, every app is identified by its `package_name`. Each View hierarchy has an `activity_name` field of the form <package name>/<activity name> that indicates which app was in focus when the screen was captured. If the package name in the `activity_name` field did not match the `package_name` of the app being assessed, that specific screen was ignored. This rule eliminated screens captured that were outside of the app, such as the Android home screen, the lock screen, or a redirection to a website. If an app had no valid screens, and therefore zero captured elements, the entire app was ignored.

Because this analysis focuses on three image-based buttons (i.e., Clickable Images, Image Buttons, and FABs), only apps which had at least one such button are considered. The class of an element was determined by the `class` field in the View hierarchy. It is possible that other class names represented widgets from one of our classes of image-based buttons of interest, such as through class name mutations (e.g., for obfuscation or minification [13]). Because we did not have knowledge of what mutation algorithms may have been used, we did not attempt to use any nodes whose `class` field did not exactly match our class names of interest.

Limitations

Due to challenges in collecting such a large dataset of Android apps, some sets of captured screens were not representative of the meaningful screens and functionality within the apps. For example, the data collected for the WhatsApp Messenger app, which has over 185 in-app screens in the dataset, represents only the country selection and phone number verification screens within the app. This results from a limitation of current data collection techniques (i.e., the Rico crawler became "stuck" in these screens). Future large-scale analyses of accessibility will benefit from improved methods for data collection.

The Rico repository was originally collected for analysis of app design patterns, not for accessibility assessment. Using a dataset outside of its intended purpose adds limitations. For example, some characteristics of the screens that would allow for a better accessibility assessment, such as the "checkable" attribute or the "important for accessibility" flag, were not captured in the View hierarchies.

Despite the limitations of the Rico repository, it contains a significant amount of useful information that is otherwise difficult to collect. We believe this data is a solid foundation for this analysis and supports meaningful insights into the state of image-based button accessibility in apps.

Talkback Focusable

Not all interactive image-based buttons of an Android screen View hierarchy are useful for a screen reader. For example, if an element is in a hidden tab then it should not be focused. We isolated elements of interest according to the heuristics used by Android's TalkBack screen reader. Specifically, we translated the `isAccessibilityFocuasable` function from the TalkBack 6.0 source code [31] from Java to Python, using the element characteristics available in the captured View hierarchies. The `checkable` property of an element was not available in the dataset and that heuristic was skipped. Although this approach likely misses some important elements or includes some elements that are not of interest, it is the approach used by the Accessibility Test Framework for Android and is a good first iteration on the analysis.

Label-Based Inaccessibility Disease

Much like alt-text for images on the web, image-based elements in Android must be labeled with meaningful information for screen readers. For the three classes of buttons we consider, this information is added in the *content description* field of the button itself or inherited from a non-interactive child with a label. We test for three ways in which a button can fail to be appropriately labeled: missing label, duplicate label, and uninformative label.

The *diagnostic criteria* used to detect if an element had a missing or duplicate label was translated from the Google-released Accessibility Test Framework for Android 2.1 [29]. The *criteria* for the uninformative label test was developed by comparing labeling best practices to a preliminary manual exploration of labels seen within the dataset.

Missing Label

A major labeling error is the complete absence of a label. In such a case, a screen reader will speak an unhelpful label such as "unlabeled button" or just "button," if it announces any label at all. The *missing label* test was translated into Python from the `SpeakableTextPresent` test in the Accessibility Test Framework for Android 2.1 [29].

Duplicate Label

Having multiple clickable elements on a screen with the exact same label may be confusing to screen reader users. Examples of how this can be problematic are presented in Figure 4. This problem is tested by comparing the labels of all clickable, TalkBack-focusable elements on a single screen. If two or more elements have the exact same label, they are all flagged as having a *duplicate label error.* This criterion is based on the `DuplicateSpeakableTextViewHierarchyCheck` from the Accessibility Testing Framework for Android 2.1, looking only at clickable elements.

Uninformative Label

If a developer adds labels to elements, it is crucial the labels are meaningful. We did not test whether labels were accurate, such as whether a button labeled "back" actually functioned as a back button. However, a list of "uninformative labels" was constructed by the first author. The first author read over the set of all labels of the captured image-based buttons and noted labels whose content was only a reflection of the class or the field (i.e., the label was composed of only the words: button, image, content, description or "desc" for short, icon, and view). The resultant set of "uninformative labels" is: alt image, button, Button, contentDescription, desc, Desc, Description, Description Image, icon desc, [image], image, Image, images, Images, image description, Image Des, image description default, Icon, Image Content, ImageView, and View. Other labels may be equally uninformative and the identification of such labels is an opportunity for future work (e.g. through crowdsourced judgements).

ANALYSIS

The analysis was structured using the epidemiology-inspired concepts of *determining the extent of a disease*, identifying potential *risk factors*, and *evaluating existing treatments.* These objectives were applied to the population of apps as well as the population of classes of image-based buttons (Clickable Image, Image Button, or FAB).

Prevalence of Missing Label in Apps

All 5,753 apps in our dataset were tested for missing labels on their captured, TalkBack-focusable, image-based buttons. The number of image-based buttons captured per app in this group has mean=28, median=2, and range=0-5,536.

The distribution of the proportion of buttons in an app missing labels is bimodal (Figure 2). At the positive extreme, 2,067 apps (35.9%) have less than 10% of their image-based buttons missing labels. The distribution of image-based buttons captured per app in this group of apps has mean=26, median=12, range=1-1,315.

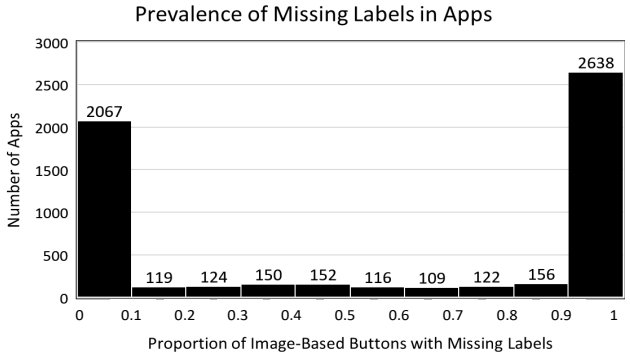

Figure 2: The distribution of the proportion of image-based buttons within an app with a missing label. A total of 5,753 apps were tested. A higher proportion is an app with more errors. The high number of apps at the extremes along with the uniform, non-zero distribution between the extremes hints at a rich ecosystem of factors influencing if an app's image-based buttons are labeled.

On the negative extreme, 2,638 apps (45.9%) have at least 90% of their image-based buttons missing labels. The distribution of the number of image-based buttons captured in apps at this extreme has mean=69, median=24, range=1-5,536. The remaining 1,048 apps (18.2%) are relatively

uniformly distributed between the two extremes (i.e., have 10-90% of their elements missing labels).

The bimodal nature of the distribution reflects two prominent groups of apps. These groupings may capture that some apps exist in environments with knowledge about and interest in accessibility and other apps do not. Additionally, the 18.2% of apps between the two extremes indicate a third interesting group of apps which are *sometimes* developed with accessibility features. The existence of this group points to the richer ecosystem encompassing apps, as articulated by Ross *et al*.'s epidemiology framework [42]. Factors such as different developers, different tools, competing priorities, or the evolution of new features may explain these sometimes-accessible apps. More work is needed to uncover what interactions between factors exist.

Prevalence of Missing Label in Classes

The first step of analysis considered apps as individual units in the population. We now analyze the *prevalence* of accessibility barriers within our three classes of image-based buttons (Clickable Image, Image Button, or FAB). This analysis gives insight into factors about the class itself that may affect the likelihood it is labeled.

Clickable Images, Image Buttons, and FABs are extremely similar in functionality. The steps for adding labels to these elements are the same. We explore if the impact of factors that differ between the classes negate the influence of their code-level and functional similarity by comparing the proportion of errors within each class of elements.

Our analysis shows discrepancies in the *prevalence* of the *missing label determinant* of the *label-based inaccessible button disease* between the classes of buttons (Table 1). The total number of errors and the number of apps that use an element of each class (regardless of whether it has an error) are also listed in Table 1. These results support the rich ecosystem of factors that differ between the classes heavily influences the button's likelihood of being labeled. Addressing those factors can help decrease the *prevalence* of *disease* in these commonly used classes; this can have a large impact on app accessibility over the whole population.

Table 1: The number of apps that have image-buttons of each class is shown in the # *Apps* column. The number of image-based buttons with a missing label and the percent out of all tested image-based buttons is presented per class in the # *Error* and *% Error* columns.

Missing Label

Class	# Apps	# Error	% Error
Clickable Image	2858	116124	86.3%
Image Button	4063	75303	54.7%
FAB	590	6055	92.0%

Evaluating Treatments for Missing Labels

The app ecosystem contains myriad factors that may impact a button's accessibility. *Treatments*, or tools that aim to reduce inaccessibility *diseases* in apps, are one group of factors. We explore a set of existing *treatments*, including general Android development and design guides [3,30], guides that are specific to ensuring accessibility [30], and accessibility testing tools [10]. The details of these tools are discussed in the Android Background section above.

We evaluated occurrences of Clickable Images, Image Buttons, and FABs in these tools for how they may affect accessibility. This evaluation provides insight into potential *factors* that contribute to the discrepancy in the *prevalence* of *label-based inaccessible button diseases* between the classes of buttons. It also exemplifies how this analysis can be used as a guide in the evaluation of tools.

The Accessibility subsection of Android Material Design's Usability Guide [3] indicates the necessity of supporting screen readers by "add[ing] audible descriptions to input controls and other elements." This necessity is echoed in the Android Accessibility Development Guidelines "Labeling UI Elements" section [30]. Although the guidelines discuss the need to label *all* graphical elements, they explicitly use Image Views (the class encompassing Non-Clickable and Clickable Images) and Image Buttons as examples. Image Buttons are further used in the code sample demonstrating the addition of a content description to an element. FABs are not explicitly mentioned in the accessibility development guides.

Android Lint v23.0.0 [10] has a set of accessibility warnings. Some unlabeled elements trigger warnings such as "Image without contentDescription." This warning will trigger for an unlabeled Clickable Image or Image Button. It will not trigger for an unlabeled FAB. The Android Test Framework for Accessibility [29] and the Android Accessibility Scanner v1.1.2 [28] based on that framework detect missing content descriptions in Clickable Images, Image Buttons, and FABs.

Android provides sample code demonstrating the use of different elements or techniques. The "Basic Accessibility" code sample [5] has an Image Button element and a non-clickable Image View element with content description labels. A comment within the code describes the need for content descriptions. Another comment notes that if the Image is decorative, it does not need a content description. FABs are not represented in the "Basic Accessibility" code sample. The copyright date on the sample is 2013.

FABs have a dedicated developer guide page [4] and a code sample `FloatingActionButtonBasic` [7]. Neither resource mentions the need to label FABs. Content descriptions are also missing for the FAB examples in both resources. The FAB developer guide was updated in March 2018. The copyright date on the FAB code sample is 2014.

We cannot conclude with certainty that the discrepancies in representation of the three classes within these *treatment* tools cause, or even impact, the discrepancy in error *prevalence* between classes of button. However, the omissions within the guidelines combined with the high *prevalence*, suggest an opportunity to test the success of guidelines. This analysis presents ways in which the guidelines may be improved. Implementing these improvements and then testing the impact of guideline usage

on app accessibility could give insight into the effectiveness of guidelines Such testing may include comparing the *disease prevalence* of apps known to have been made by consulting the guidelines against the *disease prevalence* within the general population.

The error percentages of all classes of image-based buttons, (i.e., including the Image Button class, which is well represented in the guidelines), also suggests these *treatments* are not sufficient to prevent all label-based accessibility barriers. Further investigation is needed to determine opportunities for improving existing or developing novel *treatments.* This analysis can help focus those efforts by revealing patterns in errors and best- and worst-case example apps. These examples may reveal good and bad practices within the richer ecosystem, such as common tools used by developers.

Prevalence of Duplicate Label in Apps

Duplicating labels among multiple elements on the same screen can cause confusion (Figure 4). To avoid an overpowering effect of missing label errors, we perform duplicate label analysis only on the 3,398 apps that have at least one *labeled* image-based button. Figure 3 shows the distribution of proportion of labeled buttons with duplicate labels per app. In 2,961 of those apps (87.1%), less than 10% of their captured, image-based buttons have duplicate labels. On the negative end of the distribution, 174 apps (5.1%) have over 90% of their buttons with duplicate labels. The remaining 263 apps (7.7%) are distributed relatively uniformly over the remaining spectrum of 10-90% error rates.

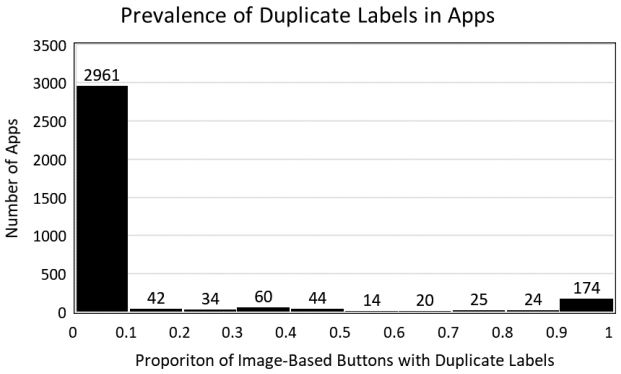

Figure 3: The distribution of the proportion of labeled image-based button elements within an app that have a duplicate label. A total of 3,398 apps were tested. Most apps have a very low proportion of their image-based buttons with the error. The more negative extreme of having 90%-100% of elements with the error has a small spike as well.

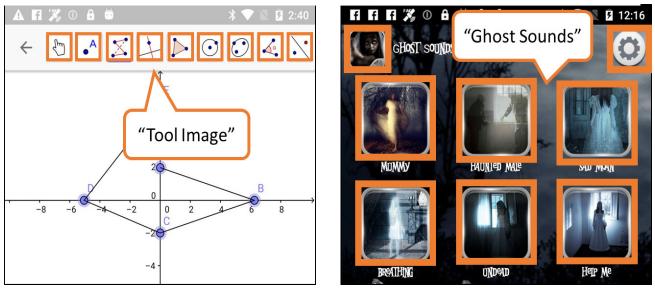

Figure 4: Two example app interfaces with duplicate label errors on image-based buttons. (left) The Clickable Image buttons for drawing different types of figures in a graphing app are all labeled "Tool Image." (right) In the Ghost Sounds app, all of the Clickable Image buttons for playing different ghost sounds as well as the settings and home button are labeled only "Ghost Sounds."

Prevalence of Duplicate Labels in Classes

We also analyzed the prevalence of duplicate labels within the considered classes of image-based buttons. Table 2 presents the percentage of elements of each class of button with duplicate labels out of all captured labeled buttons of that class. We exclude buttons missing labels. We present the number of elements of each class of button with a duplicate label. There is a discrepancy in *prevalence* between the classes of button.

Table 2: The number of labeled image-based buttons with a duplicate label and the percent out of all tested elements are presented per class in the *# Error* and *% Error of Labeled* columns. There is a notable discrepancy between the percentage of errors between the different classes image-based buttons.

Duplicate Label:

Class	# Error	% Error of Labeled
Clickable Image View	8599	46.8%
Image Button	5542	8.9%
FAB	104	19.9%

We performed a preliminary manual inspection of the labels of buttons with duplicate labels and discuss two noted patterns. One pattern was inappropriately having elements used for layout, not functionality, be clickable. This error causes the layout elements to be TalkBack-focusable and inherit a label from a contained button element.

Another noted pattern was apps that had a large proportion of Clickable Images, Image Buttons, or FABs with the same duplicate label on the same screen. Figure 4 shows examples of apps in which the duplicated label on many buttons contains no useful information related to their functionality. There are also cases in which duplicate labeling is not an error. An example of such an app from our dataset is a music app with a list of songs and artists. The artist for all of the songs was unknown, so labeling all of those elements "Unknown" was appropriate. More nuanced evaluations are needed to distinguish between valid and invalid duplicate labels.

This preliminary manual label inspection suggests types of errors developers make that can cause erroneous duplicate labels. More sophisticated methods are needed to gain a richer understanding of labeling practices that may further lead to or prevent duplicate labeling errors.

Prevalence of Uninformative Label in Apps and Classes

Uninformative labels occur when an image-based button has a non-null label that provides no helpful information to the element's functionality. For analyzing uninformative labels, we considered only apps with at least one labeled element to reduce the overpowering effect of elements with missing labels. Out of 3,396 total apps with at least one labeled element, 3,342 apps (98.4%) have less than 10% of their buttons with

uninformative labels. The low *prevalence* of uninformative labels is mirrored in the class analysis (see Table 3).

Table 3: The total number of labeled image-based buttons tested, the number of labeled image-based buttons with an uninformative label, and the percent out of all tested elements are presented per class in the *Total, # Error* and *% Error of Labeled* columns. Uninformative labels are much less *prevalent* compared to missing and duplicate labels.

Uninformative Label:

Class	Total	# Errors	% Error of Labeled
Clickable Image	18,372	1,802	9.8%
Image Button	62,306	1,278	2.1%
FAB	524	0	0%

Without a point of comparison, it is hard to say if the percentage of uninformative labels is "good," "acceptable," or "still problematic." However, the lower *prevalence* of uninformative labels compared to the *prevalence* of missing labels indicates that this problem is not as worrisome. It suggests that, if a label is added, it tends to at least contain an attempt at informative content. Further work is needed to do a more nuanced analysis of labels to determine their quality, perhaps using crowdsourcing to judge label usefulness.

Evaluating Treatments of Poor Labeling

Duplicate and uninformative labels are examples of poor labeling techniques in which a content description is added but not useful. The Material Design's Accessibility Guidelines [3] and the "Making Apps More Accessible" developer guidelines [11] include labeling image-based elements as key steps. The guides provide some indication of what a description should be, such as "provide useful and descriptive labels that explain the meaning and purpose of each interactive element to users" [11]. There is also guidance on practices to avoid when labeling, such as "Note: Many accessibility services, such as TalkBack and BrailleBack, automatically announce an element's type after announcing its label, you shouldn't include element types in your labels. For example, 'submit' is a good label for a Button object, but 'submitButton' isn't a good label" [11]. Avoiding duplicate labels is not explicitly mentioned in the guides.

Within the "Basic Accessibility" Android code sample [5], each of the elements with a content description has a unique label. Comments within that code sample offer guidance on what a label should be: "Since the `contentDescription` is read verbatim, you may want to be a bit more descriptive than usual, such as adding 'button' to the end of your description, if appropriate." Note this advice is counter to current best practices; adding the element type of "button" to a content description will cause a redundant label because TalkBack automatically announces the element type. Duplicate labels are not mentioned in the code sample.

Android Lint v23.0.0 scans do not warn about duplicate labels. The Accessibility Testing Framework for Android [29], and the Accessibility Scanner v1.1.2 [28] based on it, do have a test for duplicate labels. No tests within these tools cover any type of uninformative labels.

Relationship Between Rating and Missing Label

App accessibility relates to other factors in the ecosystem. Without detailed analyses, we do not know which factors are important or in what way they may impact accessibility. Identifying relationships between app accessibility and environmental factors can guide accessibility improvement efforts. Large-scale analyses give insight into such relationships.

For example, app ratings can inform an app creator of people's satisfaction with an app and can help other users find "good" apps. Accessibility is an important component of apps that needs to be expressed. Understanding the relationship between rating and *disease prevalence*, if one exists, can help us understand whether the current rating systems capture app accessibility. We focus on the missing label error because it had the highest prevalence and most varied distribution out of the three errors we analyzed.

We conducted a Spearman rank-order correlation test between an app's rating and the app's proportion of image-based buttons that were missing labels. We found a statistically significant relationship (ρ = -0.05, p = .001). Although statistically significant, the correlation coefficient is very low, suggesting that if a relationship exists, it is extremely weak. Looking at the distributions of proportion of missing labels by rating (Figure 5), we note there is high variability of missing labels over the entire range of app ratings.

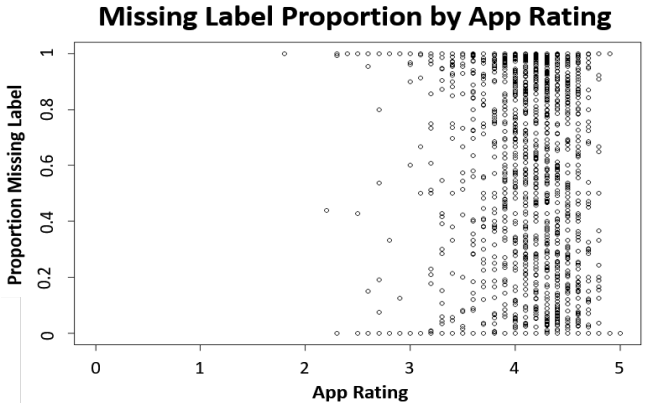

Figure 5: There is high variability in the relationship between an app's rating and its proportion of image-based buttons with missing labels. A statistically significant, but very weak correlation exists between the two factors (ρ = -0.05, p = .001). The weakness of the relationship suggests current ratings do not reflect the missing labels component of app accessibility.

The weakness of the relationship between app rating and missing label error rates suggest that the current app rating system may not sufficiently capture the missing labels component of accessibility. Given the importance of capturing and presenting end-user satisfaction with an app's accessibility, it may be beneficial to give apps an additional accessibility rating, better factor labeling into existing ratings, or otherwise highlight missing label barriers which

may be encountered in that app. Having access to a rating that reflected how well labeled an app is could allow an individual to more easily find apps that support their assistive technologies. Presenting these accessibility ratings as prominently as existing ratings may also draw broader attention to the state and importance of accessibility, potentially inspiring public pressure to improve labeling practices. Finally, accessibility ratings provide another avenue to inform app developers on the state of labels in their app in a form that emphasizes its importance.

DISCUSSION AND CONCLUSION

Applying Ross *et al.*'s [42] epidemiology-inspired framework provided a valuable lens for understanding the state of inaccessibility and its causes. Our large-scale analysis, the first of its kind, gives novel insight into the state of image-based button accessibility in free Android applications. We show *label-based inaccessible button disease* is still a large problem. Of the three *determinants*, or causes, for which we tested, missing labels was the most prevalent, uninformative labels was the least prevalent, and duplicate labels fell in between.

The tested classes of image-based buttons (Clickable Image, Image Button, and Floating Action Button (FAB)) have strong similarities in function and steps needed to prevent label-based errors (i.e. add an appropriate label in the `contentDescription` field). Despite these similarities, our analysis revealed discrepancies between the classes in the prevalence of missing labels.

The *multi-factor* analysis guided additional insight into what may have impacted the difference in *disease prevalence* between the classes. Looking to existing *treatments*, we see differences in how the three classes are represented in tools for design, development, and testing. These differences may account for some *disease prevalence* discrepancies. For example, FABs are neither explicitly mentioned in accessibility guides nor are the FABs in general example code labeled. Contrastingly, Image Buttons are labeled and appear often throughout the accessibility guidelines.

Investigating if the current app rating system captures the missing label accessibility barrier, we tested the relationship between app ratings and missing label prevalence. Based on the weakness of the correlation coefficient, we conclude current ratings may not adequately capture missing labels.

Label-based accessibility barriers in image-based buttons are only one *inaccessibility disease* that impacts app accessibility. Our future work will expand the analysis of this dataset to encompass a broader range of *inaccessibility diseases*, such as ensuring interactive elements are large enough. Considering a larger set of classes will also enhance the analysis. Such further analysis will provide insight into the susceptibility of different classes to a larger range of *inaccessibility diseases*.

The Rico repository represents the state of free Android apps at a single snapshot in time. Developing tools to support collecting large-scale mobile app data would allow for continued and more complex analysis, including longitudinal analyses. Such analyses would support ongoing investigation into techniques to improve app accessibility.

This analysis demonstrates that label-based image-based button accessibility barriers are still a prominent and widespread problem. In addition, utilizing the epidemiology-inspired framework [42] provided a useful structure to examine and understand the state of these barriers in the context of the app ecosystem of contributing factors. Continuing to collect and analyze large-scale data on app accessibility will help enable data-informed progress in enhancing the accessibility of apps.

ACKNOWLEDGEMENTS

This work was funded in part by the National Science Foundation under award IIS-1702751 and a Graduate Research Fellowship, by a Google Faculty Award, and by the Mani Charitable Foundation.

REFERENCES

1. Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. (2016). AndroZoo: Collecting Millions of Android Apps for the Research Community. *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, 468–471. http://doi.org/10.1145/2901739.2903508
2. Android Accessibility Overview. Accessed April 12th, 2018. https://support.google.com/accessibility/android/answer/6006564.
3. "Android Open Source Project." Accessibility - Usability - Material Design. Accessed April 12th, 2018. https://material.io/guidelines/usability/accessibility.html#accessibility-writing.
4. "Android Open Source Project." Add a Floating Action Button | Android Developers. Accessed April 12th, 2018. https://developer.android.com/guide/topics/ui/floating-action-button.html.
5. "Android Open Source Project." BasicAccessibility | Android Developers. Accessed April 12th, 2018. https://developer.android.com/samples/BasicAccessibility/res/layout/sample_main.html.
6. "Android Open Source Project." FloatingActionButton | Android Developers. Accesed April 12th, 2018. https://developer.android.com/reference/android/support/design/widget/FloatingActionButton.html
7. "Android Open Source Project." FloatingActionButtonBasic | Android Developers. Accessed April 12th, 2018. https://developer.android.com/samples/FloatingActionButtonBasic/res/layout/fab_layout.html

8. “Android Open Source Project.” ImageButton | Android Developers. Accessed April 12th, 2018. https://developer.android.com/reference/android/widget/ImageButton.html
9. “Android Open Source Project.” ImageView | Android Developers. Accessed April 12th, 2018. https://developer.android.com/reference/android/widget/ImageView.html
10. Android Open Source Project. Improve Your Code with Lint. v23.0.0. Accessed April 12th, 2018. https://developer.android.com/studio/write/lint.html
11. “Android Open Source Project.” Making Apps More Accessible | Android Developers. Accessed April 12th, 2018. https://developer.android.com/guide/topics/ui/accessibility/apps.html
12. “Android Open Source project.” Quality Guidelines | Android Developers. Accessed April 12th, 2018. https://developer.android.com/develop/quality-guidelines/index.html
13. “Android Open Source Project.” Shrink Your Code and Resources | Android Studio. Accessed April 12th, 2018. https://developer.android.com/studio/build/shrink-code.html
14. Android Studio. https://developer.android.com/studio/index.html
15. Apple. Accessibility - iPhone. Accessed April 12th, 2018. https://www.apple.com/accessibility/iphone/
16. Apple Accessibility Developer Guidelines. Accessed April 12th, 2018. https://developer.apple.com/accessibility/ios/
17. Apple Accessibility Scanner. Accessed April 12th, 2018. https://developer.apple.com/library/content/documentation/Accsssibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html
18. Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. (2011). Falling asleep with Angry Birds, Facebook and Kindle – A Large Scale Study on Mobile Application Usage. *Proceedings of the Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI ’11)*, 47–56. http://doi.org/10.1145/2037373.2037383
19. “Business Wire.” (2017). Wells Fargo Launches Enterprise Accessibility Program Office. https://www.businesswire.com/news/home/20171130005201/en/Wells-Fargo-Launches-Enterprise-Accessibility-Program-Office
20. Lucas Pedroso Carvalho, Bruno Piovesan Melchiori Peruzza, Flávia Santos, Lucas Pereira Ferreira, and André Pimenta Freire. (2016). Accessible Smart Cities?: Inspecting the Accessibility of Brazilian Municipalities’ Mobile Applications. *Proceedings of the 15th Brazilian Symposium on Human Factors in Computer Systems - IHC ’16*. http://doi.org/10.1145/3033701.3033718
21. Raphael Clegg-Vinell, Christopher Bailey, and Voula Gkatzidou. (2014). Investigating the Appropriateness and Relevance of Mobile Web Accessibility Guidelines. *Proceedings of the Web for All Conference (W4A ’14)*, 1–4. http://doi.org/10.1145/2596695.2596717
22. Michael Cooper, Peter Korn, Andi Snow-Weaver, Gregg Vanderheiden, Loïc Martínez Normand, and Mike Pluke. (2013). *Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT)*. http://www.w3.org/TR/wcag2ict/
23. “Data Driven Design Group.” Rico: A Mobile App Dataset of Building Data-Driven Design Applications. Accessed http://interactionmining.org/rico
24. Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. (2017). Rico: A Mobile App Dataset for Building Data-Driven Design Applications. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology - UIST ’17*, 845–854. http://doi.org/10.1145/3126594.3126651
25. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. (2016). ERICA: Interaction Mining Mobile Apps. *Proceedings of the Symposium on User Interface Software and Technology (UIST ’16)*, 767–776. http://doi.org/10.1145/2984511.2984581
26. Trinh-Minh-Tri Do and Daniel Gatica-Perez. (2010). By their apps you shall understand them: mining large-scale patterns of mobile phone usage. *Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia - MUM ’10*, 1–10. http://doi.org/10.1145/1899475.1899502
27. Floating Action Button Usage Guidelines. Accessed April 12, 2018. https://www.material.io/guidelines/components/buttons-floating-action-button.html%0A
28. Google. (2016). Accessibility Scanner. v1.1.2. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor
29. Google. (2015). Accessibility Test Framework for Android. Accessed February 25, 2018. https://github.com/google/Accessibility-Test-Framework-for-Android

30. Google. Android Accessibility Developer Guidelines. Accessed April 12, 2018. https://developer.android.com/guide/topics/ui/accessibility

31. Google. (2018). TalkBack Source Code. Accesed Febrary 14, 2018. https://github.com/google/talkback

32. "Google Open Source Project." Develop Apps | Android Developers. Accessed April 12th, 2018. https://developer.android.com/develop/index.html

33. Vicki L. Hanson and John T. Richards. (2013). Progress on Website Accessibility? *ACM Transactions on the Web*, *7*(1), 1–30. http://doi.org/10.1145/2435215.2435217

34. Shuai Hao, Bin Liu, Suman Nath, William G J Halfond, and Ramesh Govindan. (2014). PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. http://doi.org/10.1145/2594368.2594390

35. Shaun K. Kane, Jessie A. Shulman, Timothy J. Shockley, and Richard E. Ladner. (2007). A Web Accessibility Report Card for Top International University Web Sites. *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A) - W4A '07*, 148. http://doi.org/10.1145/1243441.1243472

36. Material Design. Accessed April 12th, 2018. https://material.io/guidelines/

37. Lauren R. Milne, Cynthia L. Bennett, and Richard E. Ladner. (2014). The Accessibility of Mobile Health Sensors for Blind Users. *International Technology and Persons with Disabilities Conference Scientific/Research Proceedings (CSUN 2014)*, 166–175. http://doi.org/10211.3/133384

38. Trinh Minh, Tri Do, Jan Blom, and Daniel Gatica-perez. (2011). Smartphone Usage in the Wild : a Large-Scale Analysis of Applications and Context. *Proceedings of the Conference on Multimodal Interfaces (ICMI '11)*, 353–360. http://doi.org/10.1145/2070481.2070550

39. Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. (2014). A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software*, *31*(2), 78–86. http://doi.org/10.1109/MS.2013.142

40. Kyudong Park, Taedong Goh, Hyo-Jeong So, Hyo-Jeong Association for Computing Machinery., HCI Society of Korea, and Hanbit Media (Firm). (2014). Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. *Proceedings of HCI Korea -- HCIK '15*, 478. https://dl.acm.org/citation.cfm?id=2729491

41. John T. Richards, Kyle Montague, and Vicki L. Hanson. (2012). Web Accessibility as a Side Effect. *Proc. ASSETS 2012*, 79. http://doi.org/10.1145/2384916.2384931

42. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. (2017). Epidemiology as a Framework for Large-Scale Mobile Application Accessibility Assessment. *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '17*, 2–11. http://doi.org/10.1145/3132525.3132547

43. Leandro Coelho Serra, Lucas Pedroso Carvalho, Lucas Pereira Ferreira, Jorge Belimar Silva Vaz, and André Pimenta Freire. (2015). Accessibility Evaluation of E-Government Mobile Applications in Brazil. *Procedia Computer Science*, *67*, 348–357. http://doi.org/10.1016/J.PROCS.2015.09.279

44. Clauirton Siebra, Tatiana Gouveia, Jefte Macedo, Walter Correia, Marcelo Penha, Fabio Silva, Andre Santos, Marcelo Anjos, and Fabiana Florentin. (2015). Usability requirements for mobile accessibility. *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia - MUM '15*, 384–389. http://doi.org/10.1145/2836041.2841213

45. "Starbucks Newsroom." (2015). Global Accessibility Awareness Day: Starbucks Celebrates Digital Inclusion. Accessed April 12th, 2018 . https://news.starbucks.com/news/digital-accessibility-in-starbucks-stores?hootPostID=0df1827b8efbc8223734e48ae2b64f43

46. Markel Vigo and Giorgio Brajnik. (2011). Automatic Web Accessibility Metrics: Where We Are and Where We Can Go. *Interacting with Computers*, *23*(2), 137–155. http://doi.org/10.1016/j.intcom.2011.01.001