A Large-Scale Longitudinal Analysis of Missing Label Accessibility Failures in Android Apps

Raymond Fok
rayfok@cs.washington.edu
University of Washington
Seattle, Washington, USA

Mingyuan Zhong
myzhong@cs.washington.edu
University of Washington
Seattle, Washington, USA

Anne Spencer Ross
ansross@cs.washington.edu
Bucknell University
Lewisburg, Pennsylvania, USA

James Fogarty
jfogarty@cs.washington.edu
University of Washington
Seattle, Washington, USA

Jacob O. Wobbrock
wobbrock@uw.edu
University of Washington
Seattle, Washington, USA

ABSTRACT

We present the first large-scale longitudinal analysis of missing label accessibility failures in Android apps. We developed a crawler and collected monthly snapshots of 312 apps over 16 months. We use this unique dataset in empirical examinations of accessibility not possible in prior datasets. Key large-scale findings include missing label failures in 55.6% of unique image-based elements, longitudinal improvement in *ImageButton* elements but not in more prevalent *ImageView* elements, that 8.8% of unique screens are unreachable without navigating at least one missing label failure, that app failure rate does not improve with number of downloads, and that effective labeling is neither limited to nor guaranteed by large software organizations. We then examine longitudinal data in individual apps, presenting illustrative examples of accessibility impacts of systematic improvements, incomplete improvements, interface redesigns, and accessibility regressions. We discuss these findings and potential opportunities for tools and practices to improve label-based accessibility.

CCS CONCEPTS

• **Human-centered computing** → **Accessibility systems and tools**; **Empirical studies in accessibility**.

KEYWORDS

mobile app accessibility; large-scale longitudinal analysis

ACM Reference Format:
Raymond Fok, Mingyuan Zhong, Anne Spencer Ross, James Fogarty, and Jacob O. Wobbrock. 2022. A Large-Scale Longitudinal Analysis of Missing Label Accessibility Failures in Android Apps. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3491102.3502143

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-9157-3/22/04…$15.00
https://doi.org/10.1145/3491102.3502143

1 INTRODUCTION

Mobile applications (apps) have become indispensable tools for access and participation across a wide variety of contexts (e.g., entertainment, financial services, food and groceries, transportation). However, the frequent failure of app developers to implement accessibility standards means that many app capabilities are inaccessible to people with disabilities. Recent analyses of the Android ecosystem have found many apps do not properly expose data required by platform accessibility services [39, 40, 50], undermining the functionality of these services and the accessibility of affected apps. For example, a person using a screen reader (e.g., Android TalkBack, iOS VoiceOver) who encounters an image button will expect a useful description (e.g., "Login, Button"), but a developer failure to provide a label will instead result in an unhelpful description (e.g., "Unlabeled, Button").

Many factors can contribute to the prevalence of app accessibility failures [38]. For example, an organization may not prioritize accessibility, a developer may lack accessibility awareness or expertise, or platform resources and tools may be inconsistent in their support. Accessibility efforts target such factors, including improvements to platform developer guidelines [6, 19, 20], scanners to support developers in inspecting app accessibility [5, 18], and organizational statements promising more accessible apps [3, 16, 43, 49]. Given the many factors that can contribute to an app's accessibility, it is generally difficult to gain an understanding of the impact of such accessibility efforts.

With a goal to better understand and ultimately improve the ecosystem of apps, research has examined large-scale analyses of app accessibility [39, 40, 50]. As a complement to analyses of individual apps, large-scale analyses offer the potential to reveal patterns across many apps. For example, prior large-scale analyses of app accessibility have measured the prevalence of specific accessibility failures and identified patterns that suggest contributors to those failures (e.g., inconsistent documentation and tooling across different classes of image-based elements) [39, 40, 50]. A limitation of these prior analyses has been that they are largely based on a single snapshot of each app (i.e., analysis of each app is limited to data collected from that app at a single point in time). In contrast, large-scale analyses of web accessibility have highlighted the potential for additional insight through longitudinal analyses (e.g., evolution of web accessibility over time, factors that have contributed to that evolution) [22, 34, 48]. We see a similar opportunity to examine

Figure 1: Two *failure plots* enabled by our unique longitudinal dataset, each visualizing a 16-month period of missing label failures in an app published by Zillow (i.e., a home buying app (top), and an apartment rental app (bottom)). An empty orange square indicates a unique element observed to have a missing label failure, a solid blue circle indicates a unique element observed as labeled, and a black point indicates the element was not observed in that snapshot. The June 2020 snapshot clearly shows repairs to longstanding missing label accessibility failures. The two apps also share four *ViewIdResourceName* values, suggesting these repairs may have been the result of a systematic accessibility improvement in shared code.

the evolution of app accessibility to gain new understanding of accessibility failures and to inform opportunities for improving the accessibility of the app ecosystem.

This research therefore presents a large-scale longitudinal analysis of the accessibility of Android apps, based in data crawled from 312 popular Android apps over the course of 16 months. Focusing on the prevalent and well-understood need to provide labels for image-based elements, we ask the following large-scale quantitative research questions:

RQ1: How prevalent are missing label failures in unique image-based elements?
RQ2: How frequently do changes in image-based elements introduce or repair missing label failures?
RQ3: How do changes in image-based elements impact the overall prevalence of missing label failures over time?
RQ4: How do changes in image-based elements impact per-app missing label failure rates over time?
RQ5: How do missing label failures impact navigation within apps?
RQ6: Do apps become more accessible as they become more popular?
RQ7: Are large and mature software organizations more effective at labeling image-based elements?

Key large-scale findings include missing label failures in 55.6% of unique image-based elements, longitudinal improvement in *ImageButton* elements but not in more prevalent *ImageView* elements, that 8.8% of unique screens are unreachable without navigating at least one missing label failure, that app failure rate does not improve with number of downloads, and that effective labeling is neither limited to nor guaranteed by large software organizations. We complement our large-scale quantitative findings with an examination of longitudinal failure plots in individual apps, presenting illustrative real-world examples of the accessibility impacts of systematic improvements, incomplete improvements, interface redesigns, and accessibility regressions. Finally, we discuss these findings and potential opportunities for tools and practices to improve label-based accessibility.

The specific contributions of this work include:

- A crawler to collect a unique large-scale longitudinal dataset on the accessibility of Android apps. This dataset contains a total of 3,775 crawls capturing the evolution of accessibility in 312 Android apps over 16 months. [1]
- Definitions of *screen equivalence* and *element equivalence* for analyses of large-scale longitudinal data, developed to support longitudinal tracking of the accessibility of elements across multiple crawls.
- An examination of missing-label accessibility failures in our large-scale longitudinal dataset. Labeling of image-based elements is a prevalent and well-understood accessibility need that focuses our 7 quantitative research questions while revealing broader implications.
- An examination of longitudinal failure plots, presenting illustrative real-world examples of the accessibility impacts of systematic improvements, incomplete improvements, interface redesigns, and accessibility regressions.
- The identification of opportunities for developer tools to improve support for systematic accessibility improvements, to reduce failures introduced through code duplication, and to support adoption of accessibility practices.

[1] Code and dataset available at https://github.com/appaccess/LAMA-CHI2022.

2 RELATED WORK

Our large-scale longitudinal research is primarily situated within prior work on mobile app accessibility, including analyses of factors impacting mobile app accessibility. Our motivation for pursuing large-scale longitudinal analyses is then based in part on prior examples of longitudinal analyses in web accessibility.

2.1 Analyses of App Accessibility

As mobile apps have become integrated into more aspects of everyday life, research has examined accessibility within specific classes of mobile apps, such as health apps [31, 51], smart city apps [8], and government services apps [42]. Other research has analyzed mobile apps generally to evaluate and extend existing accessibility guidelines for mobile form factors [11, 33]. These studies have been conducted through manual inspection of a small number of apps, generally finding those apps fail to implement effective accessibility. Such results have helped motivate large-scale studies that seek to understand the state of accessibility in the Android ecosystem [2, 39, 40, 50]. For example, an accessibility analysis of the Rico dataset [13] found more than 45% of evaluated apps had missing label accessibility failures in more than 90% of their image-based elements [39]. Additional large-scale analyses have similarly found high prevalence of various accessibility failures (e.g., missing labels, insufficient text or image contrast, insufficient touch target size) [2, 40, 50].

However, prior large-scale studies have largely focused on examining a single snapshot of each app (i.e., the accessibility of each app at a single moment in time). In the only known example of examining the accessibility of apps over time, Alshayban et al. present a small component of their overall analysis that considers a total of 181 crawls from 60 apps [2]. They do not report when analyzed versions were released, they note their methods cannot differentiate true accessibility improvements from other changes to apps, and the only reported result of their analysis is a high-level claim that accessibility improved in 47% of app updates. Our research therefore improves upon prior large-scale studies by collecting a unique large-scale longitudinal dataset to allow us to examine mobile app accessibility over time. Our design of data collection explicitly for the purpose of longitudinal analyses provides a larger-scale dataset of much greater fidelity than that examined by Alshayban et al. [2], and our findings are correspondingly more thorough (e.g., including a lack of support for overall improvement in missing label failures for image-based elements over the 16 months of our data collection).

2.2 Improving App Accessibility

Many efforts in research and practice aim to improve developer implementation of accessible apps. As part of promoting developer awareness and education, leading technology and accessibility organizations publish guidelines on best practices [6, 19, 20, 30, 47]. Major mobile platforms provide accessibility tests in their development environments and interactive scanners for inspecting app accessibility (e.g., Google's Accessibility Scanner [18], Apple's Accessibility Inspector [5]). Third-party accessibility organizations also offer products with similar goals (e.g., axe [15]). Research in related tools and platforms has explored higher-fidelity accessibility testing [41], the accessibility of mobile design and prototyping tools [28], and automated classification of app reviews to determine which might address accessibility [1]. Research in runtime enhancement of mobile app accessibility includes techniques for assistive macros [35], for accessibility services based on system-level content and events [36], for pointing enhancement [55], for personalizable accessibility overlays [37], and for creating accessible screenshots by embedding accessibility data directly in images [32]. Closer to our focus on missing label accessibility failures, research has demonstrated runtime repair of missing labels using interaction proxies [53] together with third-party annotation [54]. Other work has explored how search or machine learning might automatically generate some accessibility data [10, 29, 52]. Amidst such research, our large-scale longitudinal analyses are motivated in part by opportunities to inform new tools and practices for improving mobile app accessibility.

2.3 Longitudinal Analyses of Web Accessibility

Large-scale longitudinal analysis of mobile app accessibility is motivated in part by prior analyses of web accessibility. Several studies have analyzed web archives over a period of 1 to 5 years [7, 12, 21, 26], generally finding the prevalence of web accessibility failures increased over time. A longer-term analysis of web accessibility between 1999 and 2012 found many violations of accessibility standards, but observed that web accessibility overall showed slight improvement [22], likely driven by improvements in web technologies and tools rather than developer prioritization and implementation of accessibility [34]. Large-scale analyses in 2019 and 2020 found that web accessibility failures remain highly prevalent, even becoming more prevalent from 2019 to 2020 [48]. Mobile app accessibility has lacked any similar large-scale longitudinal analyses, perhaps in part due to the greater difficulty of collecting meaningful mobile app data over time. Our current data collection and analyses therefore both provide initial results and help to motivate additional large-scale longitudinal analyses of mobile app accessibility.

3 ANDROID ACCESSIBILITY BACKGROUND

We first provide a brief technical overview of Android terminology and screen reader functionality. This background helps support brevity and clarity in terms used throughout our analyses.

Android apps are composed of *elements* for layout and for interactive content (e.g., buttons, images, text), with each type of element defined in a *class*. For example, the *android.widget.ImageView* class in the Android API provides a commonly-used implementation for displaying images. A *view hierarchy* is the hierarchical representation of elements in an app at any given moment (i.e., analogous to the DOM in a web browser). Each element has a set of attributes defining its visual, functional, and accessibility data (e.g., its *ClassName*, whether it is clickable, its location on the screen, a *ContentDescription*).

Screen readers are an example of an assistive technology that rely on access to the view hierarchy through Android's Accessibility Service API. Screen readers such as Android's native TalkBack are often preferred by people who are blind, have low vision, or otherwise prefer audio feedback when using an app. Such assistive technologies use the view hierarchy to determine which elements to *focus* (e.g., text to read, interactive buttons) and which to ignore

(e.g., elements used solely for layout). As further detailed in Section 4.4, our analyses consider only *focusable* elements and therefore consider only elements that would be visited by a screen reader.

If an app's accessibility is not implemented correctly, the view hierarchy fails to provide the data required by assistive technologies. For example, if an app developer does not provide a label for an *ImageView*, TalkBack will announce a generic description (e.g., "Unlabeled, Button"). Such an accessibility failure can make it difficult for a person to find and interact with an app's functionality.

4 METHOD

This section presents our collection of a unique dataset for examining large-scale longitudinal mobile app accessibility. We detail the selection criteria for apps included in data collection, our development of a crawler for automated data collection, our development of equivalence metrics for interpreting resulting large-scale data, and our methods for assessing and analyzing missing label accessibility failures.

4.1 App Inclusion and Exclusion Criteria

To define a set of apps for our analyses, we began by considering the 25 most-downloaded free Android apps from each of the Google Play Store's 31 app categories (i.e., an initial set of 775 apps). We chose this initial inclusion criterion to: (1) prioritize most-downloaded apps because such apps are likely to be used or desired by people who use screen readers or other accessibility services, (2) select uniformly from each app category to provide diversity of purpose in included apps. Although analyses on additional platforms would be valuable (e.g., Apple's iOS), Android's global market share (i.e., 70% of the global market in 2020 [44]) provides significance to Android-specific results. Android findings may also generalize to or at least warrant examination on other platforms. Similarly, inclusion of only free apps is a common practical limitation of large-scale data collection (e.g., also a limitation of [2, 13, 50]).

From this initial set, we then excluded apps that did not expose a usable view hierarchy (i.e., as required for accessibility services and for our automated crawling and assessment). We made this determination by manually inspecting view hierarchy data in each app, obtained using an initial version of our automated crawler, excluding apps that lacked a view hierarchy exposing the majority of app content. Common classes of excluded apps were: (1) apps implemented using a gaming engine that did not expose a view hierarchy, (2) apps that presented their primary content in a *WebView* that did not expose a view hierarchy for that content, and (3) apps defined largely or entirely by multimedia content. Prior research has noted such apps are generally inaccessible due to their lack of a usable view hierarchy (e.g., [39, 40]), so our current focus is on additional insights enabled by analyses of apps that expose a usable view hierarchy. Exclusion of apps without a usable view hierarchy means our data and analyses are expected to underestimate the overall prevalence of accessibility failures.

Due to limitations of our automated crawling (i.e., as detailed in Section 5.1), we further excluded classes of apps for reasons unrelated to accessibility. Also identified through manual inspection, these were: (1) apps that operated only in a landscape orientation, because our crawling executed on a phone locked in portrait orientation, (2) apps that required a SIM card, because our crawling executed on a phone without a SIM, (3) apps that required a login credential we could not effectively mock (e.g., a social security number, a company-issued code, a driver's license number) or a verification we could not effectively mock (e.g., several dating apps required a verification photo in a specific pose), and (4) apps with primary content based on an external device (e.g., a virtual reality headset, a watch). Any portion of these apps we could have crawled (e.g., a login or verification screen) would have been unlikely to correspond to their overall accessibility. Expanding data collection to additional apps is therefore an opportunity for future research. Applying our exclusion criteria yielded a set of 391 apps that we included in initial data collection.

4.2 Data Collection Using Automated Crawling

We implemented an automated crawler informed by strategies from prior research in security [9, 23], privacy [4], and interface design [13, 14]. The crawler programmatically explores an app and captures accessibility data for each visited state. We collected longitudinal data by executing a crawl of each app on a monthly basis. This section describes our crawler, and Section 5.1 describes results of crawling from December 2019 to March 2021.

4.2.1 Crawling Devices. We collected app data using 4 identical Google Pixel 3a devices. Crawls were initially conducted with devices running Android 10, then upgraded to Android 11 beginning with the October 2020 crawl. We selected physical devices because we found them faster, more responsive, and more consistent than emulators.

4.2.2 Pre-Crawl Configuration. Before initiating each monthly crawl, we manually updated each app to its current version in the Google Play Store. We then disabled auto-update functionality (i.e., ensuring all data in a single crawl was from the same version). We reset each app's language to English (i.e., if it had been changed in the previous crawl), but did not otherwise clear app storage or reset any settings. For each app, we recorded its version and number of downloads from the Google Play Store.

4.2.3 Capturing Accessibility Data. We implemented data capture using a custom Android accessibility service, as in [54]. This service captures a view hierarchy with standard attributes (e.g., each element's location, *ClassName*), then augments the view hierarchy with accessibility data (e.g., *ContentDescription*, *IsImportantForAccessibility*) and flags for determining whether each element was *actionable* (e.g., clickable, long-clickable, scrollable). The service was installed before data collection, ran in the background, and was activated by the crawler using a software button. When activated, the service stored a screenshot and the augmented view hierarchy. Devices were locked in portrait orientation, for consistency in both captured data and crawler access to the data capture software button.

4.2.4 Crawler Strategy. Our crawler operated on a Linux workstation, connected to the physical devices via USB, communicating with each device using low-level Android Debug Bridge commands

for programmatic interaction and for retrieving captured accessibility data. The crawler controlled devices to explore apps using a modified depth-first search. At each step, the crawler selected a next actionable element from the current state, performed the action, waited for an interface update, activated the data capture service, then retrieved and parsed the new view hierarchy. We found Android's app state change indicators frequently did not correspond to complete availability of a new interface state, so we implemented a fixed 5-second delay after each programmatic interaction. This strategy was conservative (i.e., we found the delay sufficient to ensure the crawler did not capture in-progress rendering of state changes).

To support more thorough exploration, the crawler maintained a graph representation of its search. This is challenging because apps do not provide a well-defined representation of a *screen* (i.e., an app state that can be revisited and semantically corresponds to a node in a search). We approximate this using *crawl-time equivalence* heuristics, described in Section 4.3.1. This enables several optimizations for a more thorough search that would not be available in a random walk. First, upon exploring all actionable elements of a current screen, the crawler activated the system *Back* button to return to a previous screen. Second, upon revisiting a screen, the crawler prioritized actionable elements that were as-yet unexplored or led to regions of the graph that were as-yet unexplored.

We also designed the crawler to be robust to relaunching the app during a crawl, including preservation of the crawl graph. Relaunching was sometimes necessary when interaction exited the app (e.g., the crawler backed all the way out of an app, the crawler activated an actionable element that navigated to outside the app). Because any screen equivalence heuristic is only approximate, we also relaunched the app whenever the current view hierarchy was found inconsistent with the current expected node in the crawl graph (i.e., re-centering the crawl to the well-defined root of the app). Upon a restart, the crawler used the crawl graph to generate a simple action plan for reaching elements that had been encountered but not yet explored. Crawling terminated after activating all actionable elements in all discovered screens, or after reaching a timeout of two hours per app.

4.2.5 Crawler Optimizations. We observed crawler performance on a smaller set of test apps for 2 months prior to data collection and then in the first several months of data collection, implementing several optimizations to consistency and coverage. First, we implemented an *action prioritization* strategy for actionable elements within each screen, calculated based on the element's class (e.g., clicking an *ImageButton* is prioritized over text entry), element purpose determined from context (e.g., back buttons are given a lower priority), and the number of times an element has previously been activated. Second, we enhanced the crawler to support dictionary-based text entry (e.g., inputting a valid address as required in a food delivery app). Third, we limited the crawler to consider only the first three actionable children within certain Android classes (e.g., *android.widget.DatePicker*, *android.widget.TimePicker*), a heuristic we found effective for avoiding unproductive exploration of these types of elements. We implemented other minor optimizations whenever inspection of crawling data suggested an adjustment could improve consistency and coverage.

4.2.6 Human-Guided Versus Automated Exploration. Because other data collection efforts have sometimes emphasized human-guided exploration (e.g., [14, 52]), we note that we did leverage our own manual navigation of app login processes, allowing us to mock necessary login credentials as part of exposing app content. We did not otherwise find human guidance necessary, and our data collection results in Section 5.1 demonstrate our automated crawling was sufficiently thorough and consistent for our analyses (e.g., yielding more unique screens per app than prior crawls).

4.3 Defining Screen and Element Equivalence

Data collected in any exploration will include multiple captures of the same elements (e.g., captures of a screen before and after manipulating an element within that screen, captures of a screen encountered multiple times as part of exploration). Android apps do not provide a well-defined representation of a *screen* (e.g., analogous to a URL in well-defined web apps), so data collection and analyses require some heuristic for determining equivalence of two screens (e.g., matching 99.8% of pixels or all but 1 *ViewIdResourceName* in [13], a sequence of structure-based screen transformation and comparison heuristics in [54]). Our novel emphasis on longitudinal analyses introduces the additional challenge of relating elements across multiple explorations (e.g., the potential that elements of an app may be modified between explorations). This section introduces heuristics we developed for addressing these challenges in data collection and analyses, specifically in *crawl-time equivalence*, in *screen equivalence*, and in *element equivalence*.

4.3.1 Crawl-Time Equivalence. As described in Section 4.2.4, our crawler maintains a graph of visited screens to support automated exploration. We implemented *crawl-time equivalence* based on heuristics defined in [54]. Each heuristic tests for the presence of a common interface structure (e.g., a floating dialog, a navigation drawer, a tab layout), then potentially transforms the view hierarchy to normalize for comparison (e.g., transforming a view hierarchy to consider only the contents of a navigation drawer). Final equivalence is then based on two view hierarchies having the same set of *ViewIdResourceName* values and the same set of *ClassName* values (i.e., equivalent structure agnostic to content within that structure). These heuristics were previously validated through inspection of 2038 screens manually sampled from 50 apps [54], and our inspection of preliminary crawling data found them sufficient to support crawling. As part of adapting the heuristics to crawling, we modified the final comparison of *ViewIdResourceName* and *ClassName* values to use a hash-based lookup (i.e., improving performance over executing many pairwise comparisons).

4.3.2 Screen Equivalence. As described in Section 5.1, our crawling collected several orders of magnitude more data than was originally considered when defining the screen equivalence heuristics in [54] (i.e., we collected data from a larger number of screens in a larger number of apps). Inspecting collected data in our analyses, we found otherwise equivalent screens that were considered different only because of one or more *ViewIdResourceName* values that were themselves hashes or random strings. These were generally due to advertising libraries and other external packages.

Our screen-based analyses are therefore based on an enhanced definition of *screen equivalence*. After applying heuristic transformations of the view hierarchy (i.e., the first 6 heuristics in [54]), we additionally filter any element with a *ViewIdResourceName* that does not contain *android* (i.e., a native Android resource) or the app package name. As before, equivalence is then determined based on the set of *ViewIdResourceName* values and *ClassName* values. This strictly reduces the number of unique screens in each crawl (i.e., all screens considered equivalent by our enhanced heuristics are also equivalent according to the original heuristics), and our inspection of data in our analyses found this yielded more meaningful grouping of crawl data into screens for additional analyses.

4.3.3 Element Equivalence. Most of our analyses examine interface elements after applying a transformation for *element equivalence*, motivated by two challenges in examining large-scale longitudinal accessibility data. First, code re-use and templating make it unclear how to count occurrences of accessibility failures. For example, a "Back" arrow button might appear on multiple screens or a "Favorite" star button might appear next to every item in a list. Such recurring elements may be implemented in a single piece of code (i.e., a single *ContentDescription* may determine whether all are accessible) or may be similar in appearance but implemented in different code (e.g., each may require its own *ContentDescription*). Second, our desire to track the accessibility of individual interface elements over multiple crawls is complicated by any change to other elements of an interface between those crawls. Defining element equivalence across crawls to first find an equivalent screen and then the equivalent element within that screen will fail if any element in the screen has changed (i.e., the screen will not be equivalent between crawls).

Our *element equivalence* transformation therefore considers two elements equivalent if they have the same *ViewIdResourceName* and *ClassName*. Within a single crawl of an app, this effectively ignores where an element occurs in the app, treating all occurrences as a single instance. Similarly, application of this heuristic across multiple crawls effectively assumes any occurrences are equivalent. The transformation is therefore similar to assuming any elements with the same *ClassName* and *ViewIdResourceName* share the same underlying implementation. The next section discusses implications of this heuristic for our accessibility assessment.

4.4 Accessibility Assessment and Analyses

In this first large-scale longitudinal analysis of Android app accessibility failures, we focus on the prevalent and well-understood need to provide labels for image-based elements, as emphasized in prominent accessibility guidelines [6, 19, 46] and prior large-scale analyses [2, 39, 40, 50]. More specifically we focus on *ImageButton* and *ImageView*, which we collectively refer to as *image-based elements*. These are core Android components, implemented by the classes *android.widget.ImageButton* and *android.widget.ImageView*. App accessibility encompasses a broad range of design and implementation decisions to ensure access for people with different needs, so labeling image-based elements is not by itself *sufficient* for ensuring app accessibility. However, providing labels is *necessary* and this well-understood accessibility need gives focus to our analyses while aiming to reveal broader implications.

4.4.1 Missing Label Failures. We implemented assessment of missing label failures by adapting logic from Google's open-source Accessibility Testing Framework for Android [17], which is used throughout Google's accessibility technologies and tools. Specifically, we adapted its logic to execute offline against crawl data. We identified *ImageButton* and *ImageView* elements using class information in the captured view hierarchy, then limited analyses to *focusable* image-based elements (i.e., ensuring we analyzed only elements that would be visited by a screen reader). Elements can be labeled directly (e.g., using a *ContentDescription* attribute) or can inherit a label from other elements (e.g., using a *LabelFor* attribute). We therefore defined a *missing label failure* to be any case where a focusable image-based element does not have a direct or inherited label, as identified using Google's platform-standard logic.

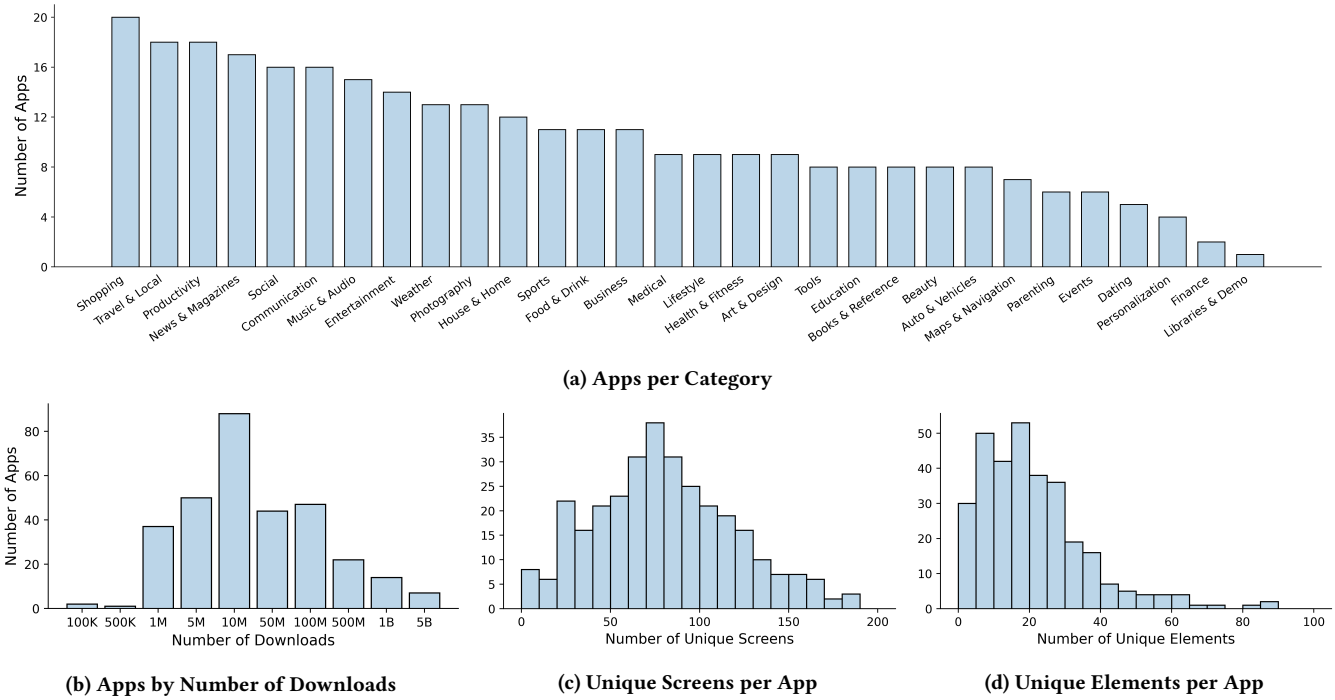
4.4.2 Failure Rate. We defined the *failure rate* of an app to be the proportion of unique image-based elements that contain a missing label failure. Because of *element equivalence*, this does not necessarily correspond to the prevalence of inaccessible elements a person would encounter when using an app, as each unique element may have a varying number of occurrences throughout an app. A unique element can also correspond to multiple occurrences, only some of which are accessible (i.e., because the underlying implementation does not share a label across occurrences). We define a unique element as containing an accessibility failure if *any* associated occurrence contains an accessibility failure. The failure rate therefore corresponds to proportion of unique *ImageButton* or *ImageView* elements that require at least 1 repair by a developer to provide an appropriate label.

4.4.3 Snapshots. For clarity and brevity, we defined a *snapshot* to be all data collected across all apps within a single set of monthly crawls. Each app is crawled once per snapshot (i.e., as described in Section 4.2.4), but the constraints of crawling mean these crawls happen neither instantaneously nor simultaneously.

4.4.4 Statistical Modeling for Analyses of Variance. Section 5 includes several analyses of variance. For brevity and readability of results, this section consolidates details of the modeling in each analysis of variance.

Section 5.2 analyzes longitudinal change in the prevalence of missing label failures across all snapshots. The dependent variable was *Missing Label Failure*, binary for each unique image-based element. We modeled *Crawl Month* as the number of months since the first snapshot (i.e., numeric between 0 and 14). Because each app was crawled multiple times throughout data collection, we accounted for non-independence by modeling *App* with a random intercept and *Crawl Month* with a random slope. We then analyzed *Missing Label Failure* using mixed logistic regression [45].

Section 5.3 analyzes longitudinal change in per-app missing label failure rate across all snapshots, while Section 5.5 analyzes per-app missing label failure rate according to number of downloads. Because effects of these factors are not necessarily independent, we modeled them together [27]. The dependent variable was *Failure Rate*, as defined in Section 4.4.2. We modeled *Downloads* as ordinal with 6 bins corresponding to a minimum number of downloads (i.e., 100 thousand, 5 million, 10 million, 50 million, 100 million, and 500 million downloads). Bins were derived from those provided



(a) Apps per Category

(b) Apps by Number of Downloads

(c) Unique Screens per App

(d) Unique Elements per App

Figure 2: An overview of our collected dataset, including a total of 3,775 crawls as part of 13 snapshots of 312 apps over 16 months. Summary distribution data shows that apps: (a) came from 30 categories of the Google Play app store, (b) varied in their number of downloads, (c) averaged 82.6 unique screens in each snapshot, and (d) averaged 20.2 unique image-based elements in each snapshot. Additional details are included in the Supplemental Materials.

by the Google Play Store to ensure each bin contained at least 40 apps (i.e., by combining bins for 100 thousand, 500 thousand, and 1 million and by combining bins for 500 million, 1 billion, and 5 billion). We again modeled *Crawl Month* as numeric between 0 and 14, *App* with a random intercept, and *Crawl Month* with a random slope. We then analyzed *Failure Rate* using mixed logistic regression, which is suitable for modeling proportions [25].

Section 5.6 analyzes per-app missing label failure rates across multiple apps released by the same organizations. The dependent variable was again *Failure Rate*. We modeled *Organization* as nominal (i.e., the 6 organizations described in Section 5.6). We again modeled *Crawl Month* as numeric between 0 and 14. We then analyzed *Failure Rate* using logistic regression [25]. We choose this simpler model because mixed logistic regression appeared unstable, likely due to the consistently low failure rates of apps by Google and Microsoft (i.e., consistently near 0, creating instability in a model that considered both *App* and *Organization*).

5 RESULTS AND ANALYSES

This section first presents the result of our data collection, then results from a series of quantitative analyses of that data. Quantitative analyses are organized around a set of research questions, and Section 6 will then complement these analyses through qualitative consideration of illustrative examples of the evolution of the accessibility of specific apps.

5.1 A Large-Scale Longitudinal Dataset on Android App Accessibility

We collected monthly snapshots of selected apps from December 2019 to March 2021. Collection of each snapshot began on the 1st of each month, and required about 12 days to complete. Restrictions on physical access amidst the COVID-19 pandemic meant that we were unable to initiate snapshots in July 2020, September 2020, and January 2021. We therefore collected a total of 13 snapshots during the 16-month data collection period.

We monitored the set of included apps between snapshots, finding it necessary to exclude some apps from additional crawling. These included apps that shifted from a free to a paid model, added new requirements for credentials that were difficult to mock, were no longer supported on our crawl devices, or were removed from the Google Play Store. After starting with the 391 apps described in Section 4.1, the final snapshot included 340 apps.

After collecting all snapshots, we manually inspected apps that had fewer than 6 unique screens in any snapshot. We excluded individual crawls that had failed to capture most of an app (i.e., by comparison to the same app in other snapshots). This excluded 210 individual crawls, including 53 where an app failed to launch or load content and 100 where the crawler failed to navigate past a blocking screen (e.g., an ad, a pop-up). After excluding individual crawls, we excluded 43 apps that were not successfully crawled at least once in both: (1) the first 4 snapshots, and (2) the final 4 snapshots. Given our focus on longitudinal analyses, we wanted apps that

Table 1: Number of unique image-based elements observed in each snapshot and the prevalence of missing label accessibility failures in observed unique elements. Over 16 months of data collection, we found a significant decline in missing label failures in unique *ImageButton* elements, but no detectable difference in unique *ImageView* elements or in combined unique image-based elements.

Crawl	Unique ImageButtons			Unique ImageViews			Combined Unique Image-Based Elements		
Month	*Failures*	*Elements*	*Failure %*	*Failures*	*Elements*	*Failure %*	*Failures*	*Elements*	*Failure %*
2019.12	648	1,370	47.3	1,760	2,961	59.4	2,408	4,331	55.6
2020.01	684	1,511	45.3	1,969	3,290	59.8	2,653	4,801	55.3
2020.02	716	1,654	43.3	2,269	3,721	61.0	2,985	5,375	55.5
2020.03	812	1,769	45.9	2,405	3,843	62.6	3,217	5,612	57.3
2020.04	751	1,768	42.5	2,547	4,085	62.4	3,298	5,853	56.3
2020.05	851	1,903	44.7	2,689	4,291	62.7	3,540	6,194	57.2
2020.06	781	1,867	41.8	2,625	4,132	63.5	3,406	5,999	56.8
2020.08	861	1,998	43.1	2,796	4,338	64.5	3,657	6,336	57.7
2020.10	939	2,143	43.8	2,773	4,534	61.2	3,712	6,677	55.6
2020.11	868	2,002	43.4	2,646	4,358	60.7	3,514	6,360	55.3
2020.12	764	1,874	40.8	2,565	4,249	60.4	3,329	6,123	54.4
2021.02	816	1,942	42.0	2,484	4,298	57.8	3,300	6,240	52.9
2021.03	793	2,030	39.1	2,514	4,256	59.1	3,307	6,286	52.6
Average	791	1,833	43.3%	2,465	4,027	61.2%	3,256	5,860	55.6%

were consistently crawled throughout data collection. Finally, we excluded 3 apps that did not include any instances of *ImageButton* or *ImageView*. Although our collected data could support other longitudinal analyses of these apps, they were not relevant to our current analyses.

Our analyses are therefore conducted with data from 13 snapshots of 312 apps including a total of 3,775 crawls. We maintained good crawl consistency, with 304 apps included in at least 10 snapshots. Apps also evolved throughout data collection, with 2,836 of 3,463 re-crawls (81.9%) executed with an app that had a different version than in the previous snapshot. Figure 2 summarizes the collected data, with collected data drawn from across 30 categories of the Google Play Store, the majority of apps having between 10 million and 50 million downloads, and snapshots including an average of 82.6 (sd = 48.3) unique screens and 20.2 (sd = 16.4) unique image-based elements for each app.

Summary statistics cannot be directly compared to prior data, due differences in factors such as data collection methods (e.g., automated crawling, manual collection) and definitions of screen equivalence. However, prior analyses [39, 40] based on the Rico dataset [13] were limited to a single snapshot averaging 7 unique screens per app. Yan et al. [50] present analyses based on manual collection of a single snapshot averaging 29 screens per app. Zhang et al. [52] describe manual collection of a single snapshot averaging 18.3 screens per app. Our data is therefore sufficient for our claims regarding large-scale analyses, and our data is further unique in focusing on longitudinal analyses across multiple snapshots. As part of supporting continued research enabled by our large-scale longitudinal data, we will release this data together with the publication of this research.

5.2 Longitudinal Analysis of Missing Label Failures

RQ1: How prevalent are missing label failures in unique image-based elements?

Analyzing unique elements in each snapshot, we found a high prevalence of missing label failures in both *ImageButton* and *ImageView* elements. Table 1 summarizes failures across all apps in each snapshot. We observed:

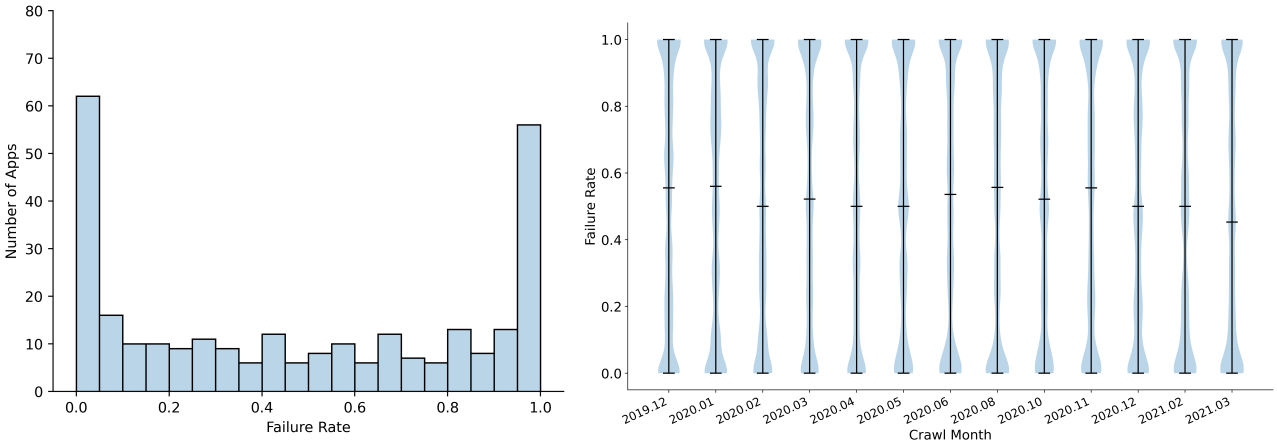
- An average of 1,833 unique *ImageButton* elements, of which 791 (43.3%) were inaccessible.
- An average of 4,027 unique *ImageView* elements, of which 2,464 (61.2%) were inaccessible.
- A combined average of 5,860 unique image-based elements, of which 3,256 (55.6%) were inaccessible.

This high prevalence is consistent with prior results and motivates a continued need to improve labeling.

RQ2: How frequently do changes in image-based elements introduce or repair missing label failures?

Our dataset is unique in its ability to track changes in the accessibility of image-based elements across snapshots. We applied *element equivalence* to examine such changes in unique elements between snapshots. We observed 5 apps with unusually high rates of new unique elements in each snapshot. Upon inspection, these apps assigned automatically-generated unique values of *ViewIdResourceName* (e.g., Airbnb assigned a unique *ViewIdResourceName* to the image of each listing). This prevents tracking across snapshots, so we excluded these apps from this analysis.

We found 4,705 examples of unique image-based elements that were newly observed after not being observed in any of at least 3 previous successful crawls of an app (i.e., suggesting these were new image-based elements). Conversely, we found 3,809 examples



(a) March 2021 Snapshot Per-App Failure Rate

(b) Longitudinal Per-App Failure Rate

Figure 3: We observed a strongly bi-modal distribution of *Failure Rate*: (a) in the March 2021 snapshot, 19.7% of apps had missing label failures in more than 95% of their unique image-based elements and 21.4% of apps had missing label failures in fewer than 5% of their unique image-based elements, and (b) there was no detectable difference in the distribution over 16 months of data collection.

of unique image-based elements that were observed in a crawl and then not observed in any of at least 3 additional successful crawls of that app (i.e., suggesting these image-based elements were removed).

Of newly observed unique image-based elements, 2,498 (53.1%) contained missing label failures. This was not significantly different from the 53.9% prevalence in other elements that had been previously observed at least once ($\chi^2(1,\ N{=}73,785) = 1.19,\ p = .28$). We therefore do not find evidence that *new* unique image-based elements are more likely to have missing label failures than *existing* unique image-based elements.

We found 169 examples of unique image-based elements that were observed with a missing label failure and then later observed to have corrected that failure (i.e., suggesting an accessibility repair). We also found 38 examples of unique image-based elements that were observed without a missing label failure and then later observed to have introduced a missing label failure (i.e., suggesting an accessibility regression).

RQ3: How do changes in image-based elements impact the overall prevalence of missing label failures over time?

To examine whether such changes led to an overall improvement in missing label failures, we conducted an analysis of variance, modeled as described in Section 4.4.4. We found no detectable improvement in *Missing Label Failure* in image-based elements according to *Crawl Month* ($\chi^2(1,\ N{=}70,108) = 0.53,\ p = .47$) in 16 months of data collection. Separate analyses of *ImageButton* and *ImageView* found significant improvement according to *Crawl Month* for *ImageButton* ($\chi^2(1,\ N{=}21,841) = 11.38,\ p < .001$) but no detectable improvement according to *Crawl Month* for *ImageView* ($\chi^2(1,\ N{=}48,267) = 0.18,\ p = .67$). The lack of an overall improvement is consistent with the much greater number of unique *ImageView* elements observed in our data.

5.3 Longitudinal Analysis of Per-App Missing Label Failure Rate

RQ4: How do changes in image-based elements impact per-app missing label failure rates over time?

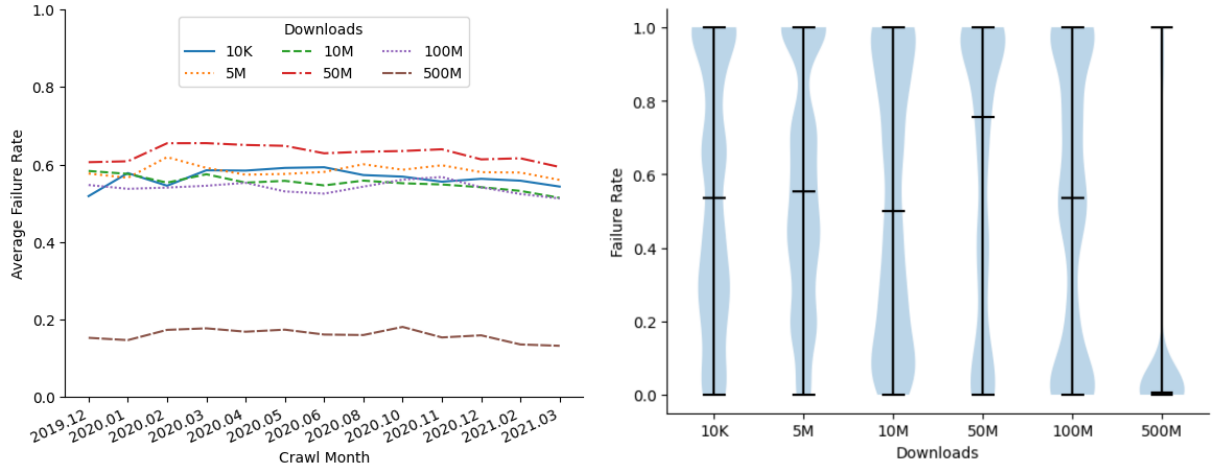
Although the analyses of missing label failures in Section 4.4.4 characterize our overall dataset, individual failures are generally not independent but instead clustered in apps that are more or less accessible. We therefore also analyzed per-app missing label failure rates (i.e., as defined in Section 4.4.2) in image-based elements, finding a strong bimodal distribution consistent with prior large-scale analyses [39, 40]. As illustrated in Figure 3a for the March 2021 snapshot, 57 apps (19.7%) had missing label failures in more than 95% of their unique image-based elements, 62 apps (21.4%) had missing label failures in fewer than 5% of their unique image-based elements, and the remainder of apps were approximately uniformly distributed in their failure rate.

Our longitudinal analysis found this bimodal distribution consistent across all snapshots, with no clear indication of change according to *Crawl Month* (Figure 3b). An analysis of variance, modeled as described in Section 4.4.4, found no detectable change in *Failure Rate* according to *Crawl Month* ($\chi^2(1,\ N{=}3,775) = 1.18,\ p = .27$). Consistent with Section 4.4.4, separate analyses of *ImageButton* and *ImageView* found significant improvement according to *Crawl Month* for *ImageButton* ($\chi^2(1,\ N{=}3,169) = 6.14,\ p = .013$) but no detectable improvement according to *Crawl Month* for *ImageView* ($\chi^2(1,\ N{=}3,691) = 0.03,\ p = .86$).

5.4 Impact of Missing Label Failures on App Navigation

RQ5: How do missing label failures impact navigation within apps?

Apps commonly use image-based elements as a primary or even exclusive method of navigation among screens (e.g., a shopping cart



(a) Longitudinal Per-App Failure Rate by Number of Downloads

(b) March 2021 Snapshot Per-App Failure Rate by Number of Downloads

Figure 4: We found no evidence that missing label failure rate incrementally improved as apps became more popular. Apps with more than 500 million downloads had lower failure rates, but most apps will never reach that point. There was no detectable difference among apps with fewer downloads.

icon for checkout, a gear icon for app settings). If such a navigation element is inaccessible, it becomes a barrier for all content that is accessed through that navigation element. Current accessibility heuristics (i.e., as in the Accessibility Testing Framework for Android [17]) and prior large-scale analyses (e.g., [39, 40, 50]) consider only the accessibility of individual elements. We examined how our collected crawl graph data could support additional analysis of the accessibility implications of image-based navigation elements.

We applied *screen equivalence* to our crawl graph data and identified screens that were reached only through an actionable image-based element. We performed this analysis on the 10 snapshots from March 2020 onward, as prior snapshots did not retain the crawl graph. As described in Section 4.2.4, a crawl graph consists of nodes corresponding to unique screens connected by edges corresponding to the action required to transition between screens. Due to crawler restarts and other artifacts of data collection, there may be disconnected components in the graph. Across analyzed snapshots, the largest component in each crawl of each app contained an average of 84.0 (sd = 48.9) of 88.1 (sd = 50.3) unique screens (95.3%). This further supports validity of our crawl data, and we proceeded by analyzing only the largest connected component.

Analyzing the largest connected component in each crawl, apps contained an average of 84.0 (sd = 48.9) unique screens. Of these, an average of 21.9 (sd = 22.3) (26.1%) could be reached only by navigating through at least one image-based element. Because some of these navigational image-based elements had missing label failures, an average of 7.4 (sd = 13.7) unique screens could not be reached without navigating through at least one image-based element with a missing label failure. This was 33.8% of average unique screens reachable only through navigational image-based elements, and overall meant that 8.8% of average unique screens were unreachable without navigating through at least one missing label failure.

5.5 Impact of Number of Downloads on Accessibility

RQ6: Do apps become more accessible as they become more popular?

When advocating for early commitment to and adoption of practices to promote accessibility, a common pressure is to defer accessibility in order to prioritize other design and implementation efforts. Such deferral is sometimes based on an argument that accessibility can be incrementally added later as an app becomes more popular. Although such deferral is problematic in multiple regards, our data also allows directly examining this argument in practice.

Figure 4 shows: (a) the average *Failure Rate* for apps grouped by *Downloads* in each monthly snapshot, and (b) the distribution of *Failure Rate* for apps grouped by *Downloads* in the March 2021 snapshot. We examined the relationship between *Downloads* and *Failure Rate* using an analysis of variance, modeled as described in Section 4.4.4, finding a significant effect of *Downloads* ($\chi^2(5, N{=}3775) = 63.60$, $p < .001$). We conducted post-hoc pairwise comparisons using Z-tests, corrected with Holm's sequential Bonferroni procedure [24], finding apps with more than 500 million downloads had significantly lower *Failure Rate* than apps with less than 500 million downloads ($p < .001$ for all pairs comparing 500 million downloads to other bins). There were no detectable differences in *Failure Rate* among pairs with fewer than 500 million downloads.

Although apps with more than 500 million downloads have lower *Failure Rate*, most apps will never reach that point. Our inclusion criteria emphasized the Google Play Store's most-downloaded apps, but only 43 apps had more than 500 million downloads. We found no detectable differences in *Failure Rate* for apps with fewer than 500 million downloads, providing no support for suggestions that apps will in practice be made incrementally more accessible as they become more popular.

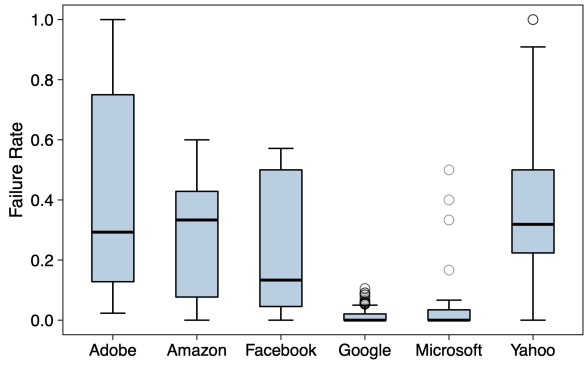


Figure 5: A box plot of per-app failure rate in 6 large and mature software organizations that published at least 4 apps included in our analyses. We found Google and Microsoft published apps with consistently lower failure rates than the other 4 organizations, and we found examples of low failure rates in apps published by another 30 different organizations. These results suggest that being one of the largest and most mature software organizations does not by itself guarantee accessible apps. Conversely, accessible apps in our dataset were not limited to these large and mature software organizations.

5.6 Impact of Organization on Accessibility

RQ7: Are large and mature software organizations more effective at labeling image-based elements?

Among highly downloaded apps in our data, we observed that many were published by large and mature software organizations. For example, of the 43 apps in our data that had more than 500 million downloads, 18 (41.9%) were developed by Google. We therefore examined how accessibility might be shaped by factors related to these organizations, analyzing missing label failure rates across multiple apps published by the same organizations. We identified 6 organizations that had published at least 4 apps in our data: Adobe (4), Amazon (4), Facebook (6), Google (22), Microsoft (7), and Yahoo (6). This resulted in 49 apps, with an average of 18.5 unique image-based elements in each crawl of each app.

Figure 5 plots the missing label failure rate for these apps according to *Organization*. Although each organization published at least one app with a low missing label failure rate, several organizations also published apps with much higher missing label failure rates. An analysis of variance, modeled as described in Section 4.4.4, found a significant difference in *Failure Rate* according to *Organization* ($\chi^2(5,\ N{=}583) = 136.4,\ p < .001$). We conducted post-hoc pairwise comparisons using Z-tests, corrected with Holm's sequential Bonferroni procedure [24], finding that apps published by Google or Microsoft had significantly lower *Failure Rate* than apps published by the other 4 organizations ($p < .01$ for all pairs comparing Google or Microsoft to other organizations). Organizational factors that can shape accessibility warrant additional investigation, but our data suggests that access to expertise and resources at large and mature software organizations does not by itself guarantee accessible apps.

Furthermore, while the above 6 organizations account for 32 of the 62 apps with less than 5% missing label failure rate in our March 2021 snapshot, the remaining 30 were published by 30 different organizations. Though our data does not otherwise characterize those different organizations (e.g., a large and mature organization may have only 1 app in our data), our results suggest the development of accessible apps is also not limited to the above 6 largest and most mature software organizations.

6 ILLUSTRATIVE EXAMPLES OF EVOLUTION IN APP ACCESSIBILITY

Prior large-scale analyses of mobile app accessibility have been conducted within a single snapshot and therefore were unable to observe specific changes in the accessibility of apps over time. We identified illustrative examples of evolution in the accessibility of specific apps by generating longitudinal *failure plots* for each app, each of which show the unique elements observed in an app and the accessibility of those elements across collected snapshots (e.g., as in Figure 1 and Figure 6). We highlighted plots for trends that seemed compelling or informative, then inspected the data behind each plot to ensure appropriate interpretation. Observations in these qualitative examples are intended to complement and provide additional nuance to our large-scale quantitative analyses.

6.1 Systematic Accessibility Improvement

Although our large-scale quantitative analyses did not find an overall longitudinal improvement of missing label failures in image-based elements, we did observe individual apps making systematic improvements to their accessibility. For example, Figure 1 shows failure plots for 2 apps published by Zillow: a home buying app and an apartment rental app. Across multiple snapshots, both apps included multiple unique elements with missing label failures. The June 2020 snapshot then repaired most of these failures across both apps. These repairs suggest a relatively systematic approach to improving accessibility, and the visual similarity of the apps together with 4 identical *ViewIdResourceName* values suggests they likely share underlying code that was improved. We did not observe the later introduction of new unique elements, so our data provides no insight into whether June 2020 improvements might have resulted from a one-time accessibility audit or from a more sustained organizational approach to ensuring accessibility. However, this relatively systematic improvement does provide an example of effective improvement and a contrast to our observations in other examples.

6.2 Incomplete or Opportunistic Accessibility Improvement

In contrast to such systematic improvement, we more commonly observed apps that made incomplete or opportunistic accessibility improvements. Such improvements were characterized by repairing one or more missing label failures while other elements in the same app or even the same screen remained unlabeled. For example, Figure 6 shows a failure plot for the FOX Now app. The March 2020 snapshot repaired 4 unique missing label failures (i.e., "Home", "Live", "Search", "My Account"), all within the app's navigation toolbar. Several other missing label failures in the same screens

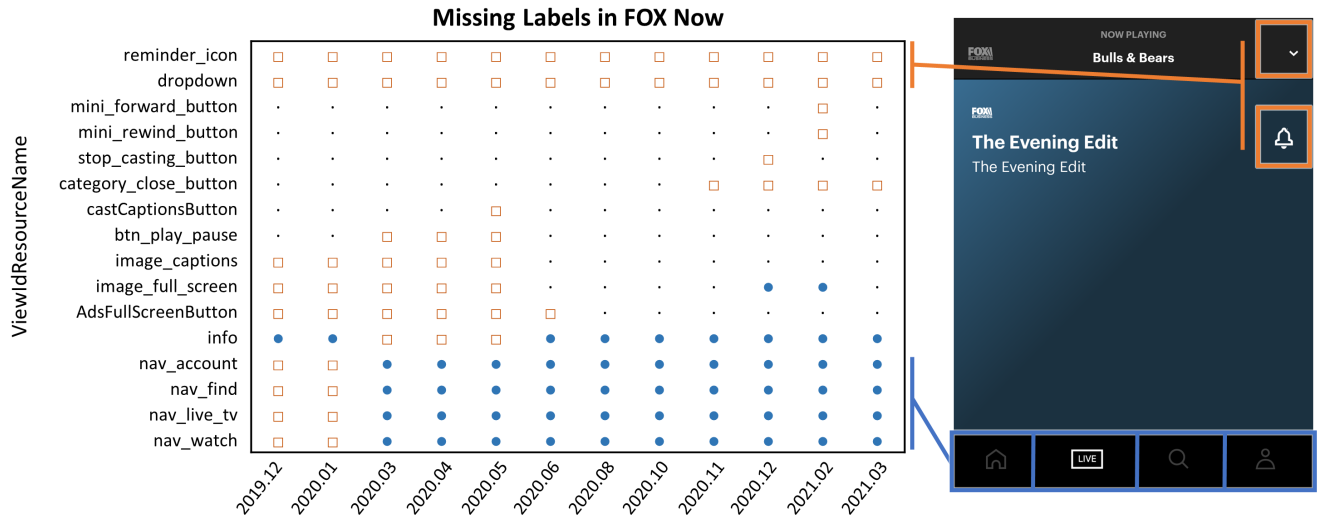


Figure 6: Failure plot for the FOX Now app. As in Figure 1, an empty orange square indicates a unique element observed to have a missing label failure, a solid blue circle indicates a unique element observed as labeled, and a black point indicates the element was not observed in that snapshot. The March 2020 snapshot repaired 4 unique missing label failures in the app's navigation bar, highlighted in blue in the accompanying screenshot. However, 2 other missing label failures in the same screen were not repaired, highlighted in orange. This incomplete repair suggests the app developers had an awareness of app accessibility and how to repair failures, but other factors prevented more complete implementation of accessibility repairs.

were not repaired (e.g., "Collapse", "Alert"), and the failure plot shows later snapshots introduced newly observed elements that contained new missing label failures. As another example, our original snapshot of the SHEIN-Fashion Shopping Online app included 36 unique elements with missing label failures. The December 2020 snapshot added labels to least 16 previously inaccessible elements across 7 unique screens. The February 2021 snapshot added labels to another 5 elements across 2 unique screens. Despite such repairs indicating awareness and knowledge regarding image labeling, the March 2021 snapshot still included 43 unique elements with missing label failures. In contrast to systematic repair in Section 6.1, these patterns suggest accessibility improvements due to specific limited reports (e.g., a complaint about a specific element may prompt repair of that element) or variations in development practice (e.g., one developer in a team may have better accessibility practices or may opportunistically repair elements they encounter). The developer of such an app has at least some awareness and knowledge regarding accessibility, but other factors seem to prevent them from implementing more complete accessibility support.

In addition to such incomplete labeling across snapshots, we also observed inconsistent labeling within snapshots. We examined 80 apps that had at least one instance of inconsistent labeling of equivalent elements (i.e., according to our definition of *element equivalence* in Section 4.3). For example, Figure 7a shows the Postmates app in the October 2020 snapshot. It contained multiple screens with a "Close" button, but only some properly labeled the button. Similarly, Figure 7b shows the Roomster app in the February 2021 snapshot. It contained screens with an "Edit Profile" *ImageButton* that was properly labeled on a *My Info* screen but not on a *Listings* screen. Our inspection found inconsistent labeling was most common among "Back", "Close", and other navigation actions. In contrast to the evidence that code reuse provided consistency even across apps in Section 6.1, such inconsistencies likely resulted from code repetition that failed to consistently include accessibility data.

6.3 Accessibility in Interface Redesigns

We observed that interface redesigns can have additional impact on accessibility. For example, the October 2020 snapshot of the DoorDash app introduced several changes relative to the prior August 2020 snapshot. We detected 17 unique image-based elements with missing label errors in the August 2020 snapshot that were then not detected in or after the October 2020 snapshot. Inspection of this accessibility-related event found these elements had generally been removed or re-implemented with a different *ViewIdResourceName*. For example, Figure 8 shows an *ImageView* with a "plus" icon was that added, allowing selection of the quantity of an item before adding it to a cart. Unfortunately, this and many other new elements also had missing label failures.

Although Section 5.2 found no detectable difference in the prevalence of missing label failures for newly-observed versus previously-observed unique elements, this example highlights that new accessibility failures can come together with changes in an interface. A person may have learned to work around existing failures (e.g., by memorizing paths through an app, by using Android support for custom labeling based on *ViewIdResourceName*), so new accessibility failures together with changes that invalidate a prior workaround could be additionally problematic.

(a) Postmates app, October 2020 snapshot

(b) Roomster app, February 2021 snapshot

Figure 7: Examples of inconsistent labeling of the same interface element: (a) 2 different screens from the Postmates app contain the same "Close" button, but only 1 is labeled, and (b) 2 different screens from the Roomster app contain the same "Edit Profile" button, but only 1 is labeled. Such inconsistent labeling suggests code repetition is contributing to accessibility failures.

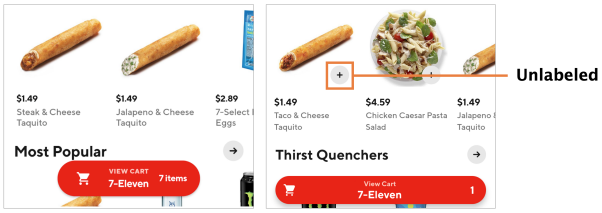


Figure 8: An example of a missing label failure in a new element introduced in a redesign, observed in the October 2020 snapshot of the Doordash app. The update added a new button for selecting the quantity of an item before adding it to the cart, but this element was not labeled. Accessibility failures introduced during redesigns may be additionally problematic if the redesign invalidates any workarounds that people have developed with a prior interface.

6.4 Accessibility Regressions

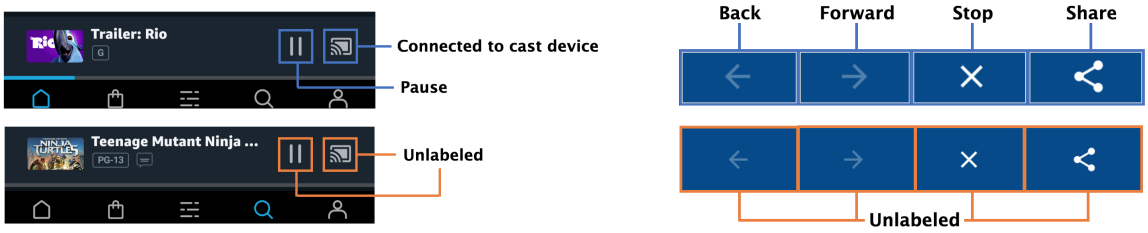
Our longitudinal tracking of elements across snapshots found 27 examples that were observed without a missing label failure, then later observed to have introduced a missing label failure. Inspection confirmed these were generally accessibility regressions, as illustrated in Figure 9. For example, the March 2020 snapshot of the Amazon Prime Video app included two *ImageView*-based buttons in a playback control toolbar that were properly labeled as "Pause" and "Connected to cast device", but these same elements then contained missing label failures in the May 2020 snapshot. Similarly, the August 2020 snapshot of the CBS Sports app included browser controls for an embedded *WebView* that were properly labeled (i.e., as "Back", "Forward", "Stop", and "Share"), but these same buttons then had missing label failures in the October 2020 snapshot. In both examples, the icons and arrangement of the image-based elements were unmodified and we saw no evidence of larger changes in the design. Other than the accessibility regression, the only detectable change in these elements was their height (i.e., the Amazon Prime Video buttons changed height from 204 pixels to 215 pixels; the CBS Sports buttons changed height from 132 pixels to 154 pixels, the icons within the buttons appeared identical but scaled slightly smaller). Our data does not clearly suggest how these regressions were introduced, but does illustrate a lack of testing or other mechanisms for preventing such regressions.

7 DISCUSSION

We have presented a set of initial analyses focused on missing label accessibility failures within a unique large-scale longitudinal dataset of app accessibility. Labeling image-based elements is a critical and necessary component of accessibility, as emphasized in accessibility guidelines and prior analyses. Prior large-scale analyses largely focused on a single snapshot of each app and therefore were unable to explore change in accessibility over time [2, 39, 40, 50]. We examined changes in accessibility over time using both large-scale quantitative analyses and the qualitative consideration of specific illustrative examples. In addition to providing a new understanding of current app accessibility, our analyses suggest potential design opportunities for improved labeling of image-based elements.

7.1 Supporting Systematic Accessibility Improvements

One set of opportunities is to help developers be more systematic in their labeling of image-based elements. Incomplete labeling in Section 6.2 demonstrates that developers have at least some awareness and knowledge, yet they do not systematically apply that knowledge. Section 5.3's bimodal distribution of per-app failure rate is also consistent with this finding, as the uniform middle portion of the distribution collectively accounts for approximately 60% of apps that provide labels for some but not all image-based elements. Future research could examine which elements are labeled versus not and how any such differences might suggest improvements in development and accessibility testing tools. One possibility is that developers are simply unaware of additional missing label failures. A tool based on automatically collected data like our crawls could help surface failures (e.g., complementing manual exploration using tools like the Accessibility Scanner [18]). Another possibility is that failures are in third-party components (e.g., a library used by an app, in which the app developer cannot directly repair an accessibility failure). Large-scale analyses might identify such library-based failures across different apps, help prompt repairs in such libraries, and support app developers in early decision-making around which



(a) Amazon Prime Video app, May 2020 regression

(b) CBS Sports app, October 2020 regression

Figure 9: Examples of accessibility regressions: (a) 2 playback control elements in the Amazon Prime Video app that were previously labeled but then observed with missing label failures in the May 2020 crawl, and (b) 4 browser control buttons in the CBS Sports app that were previously labeled but then observed with missing label failures in the October 2020 snapshot.

libraries to use. Our data overall suggests an audience of developers who are already motivated to improve accessibility and would benefit from tools that supported them in being more systematic.

7.2 Reducing Failures Introduced Through Code Duplication

Another set of opportunities is to reduce the number of distinct labels app developers must provide. Our definition of element equivalence and the inconsistent labeling in Section 6.2 highlight that apps contain many equivalent elements, some of which are inaccessible due to inconsistent labeling. Implementation of equivalent elements in separate code is straightforward (e.g., instantiating an *ImageButton* parameterized with an icon), but resulting code repetition contributes to accessibility failures. For example, the interface redesign in Section 6.3 reduced the number of unique elements with missing label failures by repairing or removing several elements with similar *ViewIdResourceName* (e.g., "close", "closeButton", "close_button"), although a new "btn_close" element then also introduced a new missing label failure. It is possible for developers to create and reuse components, and the systematic examples in Section 6.1 apparently benefited from code reuse. However, developer tools could also better promote label reuse as an example of the *Don't Repeat Yourself* principle. For example, modern development environments can detect code repetition as potentially problematic, and our analyses suggest repetition when instantiating image-based elements might be especially problematic (e.g., development environments might present a warning or support automated refactoring). Because we found many inconsistent labels in common components (e.g., "Back"), libraries that automatically pair common icons and appropriate labels could reduce labeling required of app developers. Implementation patterns for centralization and reuse of images and their labels, as already common when centralizing text into tables for localization, could support both reuse and more systematic coverage in labeling.

7.3 Supporting Adoption of Accessibility Practices

Additional opportunities to promote accessibility through developer tools and practices are suggested by considering the accessibility of an app in terms of its longitudinal evolution (i.e., rather than as a static property of an app). Accessibility advocates have long argued for the importance of starting with accessibility, and Section 5.5's analysis of missing label failure rate according to number of downloads demonstrates that suggestions an app will be made incrementally more accessible as it becomes more popular are generally not born out in practice. Such data can and should be used to advocate for stronger accessibility practices in organizations and stronger accessibility defaults in developer tools. At a smaller scale, developer tools might also target moments of change corresponding to new interface elements or significant interface redesigns. Section 5.4 found that image-based elements often provide the primary or exclusive path for navigating to portions of an app, so detecting the introduction of such an element could support stronger interventions to ensure its accessibility. Significant interface redesigns like in Section 6.3 provide a similar opportunity. The fact that a portion of an interface is already undergoing significant change provides an opportunity for stronger interventions to ensure accessibility in that change (e.g., organizations could require accessibility in new functionality as part of allowing developers to publish that functionality). Finally, our observation of specific accessibility regressions in Section 6.4 warrants further investigation and potential improvements in associated tools.

7.4 Limitations and Future Work

Crawls were conducted at monthly intervals since we found most popular Android apps like those included in our dataset were often updated. We found that 81.9% of sequential crawls were conducted on different app versions, suggesting that most apps were frequently updated between snapshots. Monthly sampling also offered several technical advantages, such as ensuring sufficient time for all apps to be crawled thoroughly, and providing a window to prepare the infrastructure for the next snapshot of crawls. Furthermore, our 16-month study is comparable in timeframe to some insightful longitudinal studies conducted on web accessibility [12, 26], and we also believe 16 months is an adequate amount of time for screen reader users to expect to see accessibility improvements within their commonly used mobile apps.

While our results suggest that monthly sampling over 16 months is sufficient to detect a small evolution of app accessibility, there are potential analyses that it does not support. Future longitudinal studies of app accessibility may consider other sampling frequencies to complement our analyses. For example, apps could be crawled at

every update to provide a fine-grained lens into each app's accessibility evolution. Less frequent crawls (e.g., sampling every three months) over a longer period of time could provide a broader perspective of accessibility changes, such as enabling the macroscopic analysis of the relationship between evolving developer tools and resources (e.g., developer documentation, mobile frameworks, and common app libraries) and accessibility failures. Finally, our study does not currently differentiate between accessibility failures of different image-based elements. One avenue for future work is to distinguish between missing labels that were caused during the initial design and development of an interface element, or those that were caused by missing label data when populating dynamic image elements from a database.

8 CONCLUSION

We have presented the first large-scale longitudinal analysis of missing label accessibility failures in Android apps, and created a unique dataset of monthly snapshots of 312 apps over 16 months. Our large-scale quantitative analyses of missing label failures in this dataset found missing label failures in 55.6% of unique image-based elements, longitudinal improvement in *ImageButton* elements but not in more prevalent *ImageView* elements, that 8.8% of unique screens are unreachable without navigating at least one missing label failure, that app failure rate does not improve with number of downloads, and that effective labeling is neither limited to nor guaranteed by large software organizations. We complemented this with qualitative examination of longitudinal failure plots in individual apps, presenting illustrative real-world examples of the accessibility impacts of systematic improvements, incomplete improvements, interface redesigns, and accessibility regressions. Finally, we suggested opportunities for better supporting systematic accessibility improvements, in reducing failures introduced through code duplication, and in supporting adoption of accessibility practices. As accessibility research in current and emerging technologies defines best practices for supporting individual accessibility needs and preferences, our research demonstrates the potential value of large-scale longitudinal data in examining how to ensure developer implementation of those practices.

ACKNOWLEDGMENTS

This work was supported in part by Google, by the University of Washington Center for Research and Education on Accessible Technology and Experiences (CREATE), and by National Science Foundation grant #IIS-1702751. We also thank our anonymous reviewers for their helpful feedback. Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any supporter.

REFERENCES

[1] Eman Alomar, Wajdi Aljedaani, Murtaza Tamjeed, William Catzin, Mohamed Wiem Mkaouer, and Yasmine Elglaly. 2021. Finding the Needle in a Haystack: On the Automatic Identification of Accessibility User Reviews. In *Proceedings of the 2021 Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA.

[2] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1323–1334.

[3] Amazon. 2021. Amazon Accessibility. https://www.amazon.com/b?ie=UTF8&node=15701038011

[4] Shahriyar Amini. 2018. Analyzing Mobile App Privacy Using Computation and Crowdsourcing.

[5] Apple. 2021. Accessibility Programming Guide for OS X. https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html

[6] Apple. 2021. Human Interface Guidelines - Accessibility. https://developer.apple.com/design/human-interface-guidelines/accessibility

[7] Ibtehal S. Baazeem and Hend Suliman Al-Khalifa. 2015. Advancements in web accessibility evaluation methods: how far are we?. In *Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services* (Brussels, Belgium) *(iiWAS '15)*. Association for Computing Machinery, New York, NY, USA, Article 90, 5 pages.

[8] 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. In *Proceedings of the 15th Brazilian Symposium on Human Factors in Computing Systems* (São Paulo, Brazil) *(IHC '16)*. Association for Computing Machinery, New York, NY, USA, Article 17, 10 pages.

[9] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 665–676.

[10] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 322–334.

[11] Raphael Clegg-Vinell, Christopher Bailey, and Voula Gkatzidou. 2014. Investigating the Appropriateness and Relevance of Mobile Web Accessibility Guidelines. In *Proceedings of the 11th Web for All Conference* (Seoul, Korea) *(W4A '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 4 pages.

[12] Angela L. Curl and Deborah D. Bowers. 2009. A Longitudinal Study of Website Accessibility: Have Social Work Education Websites Become More Accessible? *Journal of Technology in Human Services* 27, 2 (2009), 93–105.

[13] 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. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854.

[14] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual ACM Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 767–776.

[15] deque. 2021. axe: Accessibility Testing Tools and Software. https://www.deque.com/axe/

[16] Facebook. 2021. Facebook Accessibility. https://www.facebook.com/help/accessibility

[17] Google. 2021. Accessibility Test Framework for Android. https://github.com/google/Accessibility-Test-Framework-for-Android.

[18] Google. 2021. Get started with Accessibility Scanner. https://support.google.com/accessibility/android/faq/6376582

[19] Google. 2021. Google Accessibility Guidelines. https://developer.android.com/guide/topics/ui/accessibility/apps

[20] Google. 2021. Material Design - Accessibility. https://material.io/design/usability/accessibility.html#understanding-accessibility

[21] Stephanie Hackett, Bambang Parmanto, and Xiaoming Zeng. 2005. A retrospective look at website accessibility over time. *Behaviour & Information Technology* 24, 6 (2005), 407–417.

[22] Vicki L. Hanson and John T. Richards. 2013. Progress on Website Accessibility? *ACM Trans. Web* 7, 1, Article 2 (March 2013), 30 pages.

[23] 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. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. Association for Computing Machinery, New York, NY, USA, 204–217.

[24] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70.

[25] Peter B. Imrey. 2000. Poisson Regression, Logistic Regression, and Loglinear Models for Random Counts. In *Handbook of Applied Multivariate Statistics and Mathematical Modeling*, Howard E.A. Tinsley and Steven D. Brown (Eds.). Academic Press, San Diego, 391–437.

[26] Jonathan Lazar and Kisha-Dawn Greenidge. 2006. One Year Older, but Not Necessarily Wiser: An Evaluation of Homepage Accessibility Problems over Time. *Univers. Access Inf. Soc.* 4, 4 (May 2006), 285–291.

[27] Lung-Fei Lee. 1982. Specification error in multinomial logit models: Analysis of the omitted variable bias. *Journal of Econometrics* 20, 2 (1982), 197–209.

[28] Junchen Li, Garreth Tigwell, and Kristen Shinohara. 2021. Accessibility of High-Fidelity Prototyping Tools. In *Proceedings of the 2021 Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA.

[29] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-Driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 107–118.

[30] Microsoft. 2021. Accessibility overview. https://docs.microsoft.com/en-us/windows/uwp/design/accessibility/accessibility-overview

[31] Lauren R. Milne, C. Bennett, and R. Ladner. 2014. The Accessibility of Mobile Health Sensors for Blind Users. *Journal on Technology & Persons with Disabilities* 2 (2014), 10 pages.

[32] Sujeath Pareddy, Anhong Guo, and Jeffrey P. Bigham. 2019. X-Ray: Screenshot Accessibility via Embedded Metadata. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility* (Pittsburgh, PA, USA) *(ASSETS '19)*. Association for Computing Machinery, New York, NY, USA, 389–395.

[33] Kyudong Park, Taedong Goh, and Hyo-Jeong So. 2014. Toward Accessible Mobile Application Design: Developing Mobile Application Accessibility Guidelines for People with Visual Impairment. In *Proceedings of HCI Korea* (Seoul, Korea) *(HCIK '15)*. Hanbit Media, Inc., Seoul, KOR, 31–38.

[34] John T. Richards, Kyle Montague, and Vicki L. Hanson. 2012. Web Accessibility as a Side Effect. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility* (Boulder, Colorado, USA) *(ASSETS '12)*. Association for Computing Machinery, New York, NY, USA, 79–86.

[35] André Rodrigues. 2015. Breaking Barriers with Assistive Macros. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility* (Lisbon, Portugal) *(ASSETS '15)*. Association for Computing Machinery, New York, NY, USA, 351–352.

[36] André Rodrigues and Tiago Guerreiro. 2014. SWAT: Mobile System-Wide Assistive Technologies. In *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014 - Sand, Sea and Sky - Holiday HCI* (Southport, UK) *(BCS-HCI '14)*. BCS, Swindon, GBR, 341–346.

[37] André Rodrigues, André Santos, Kyle Montague, and Tiago Guerreiro. 2017. Improving Smartphone Accessibility with Personalizable Static Overlays. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility* (Baltimore, Maryland, USA) *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 37–41.

[38] 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* (Baltimore, Maryland, USA) *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 2–11.

[39] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2018. Examining Image-Based Button Labeling for Accessibility in Android Apps through Large-Scale Analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility* (Galway, Ireland) *(ASSETS '18)*. Association for Computing Machinery, New York, NY, USA, 119–130.

[40] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2020. An Epidemiology-Inspired Large-Scale Analysis of Android App Accessibility. *ACM Trans. Access. Comput.* 13, 1, Article 4 (April 2020), 36 pages.

[41] 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* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 274, 11 pages.

[42] 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 (2015), 348–357. Proceedings of the 6th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion.

[43] Starbucks. 2015. Global Accessibility Awareness Day: Starbucks Supports Digital Inclusion. https://stories.starbucks.com/stories/2015/digital-accessibility-in-starbucks-stores/

[44] StatCounter. 2021. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide

[45] Robert Stiratelli, Nan Laird, and James H. Ware. 1984. Random-Effects Models for Serial Observations with Binary Response. *Biometrics* 40, 4 (1984), 961–971.

[46] W3C. 2021. How to Meet WCAG (Quick Reference). https://www.w3.org/WAI/WCAG21/quickref/

[47] W3C. 2021. Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile. https://www.w3.org/TR/mobile-accessibility-mapping/

[48] WebAIM. 2021. The WebAIM Million. https://webaim.org/projects/million/

[49] 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

[50] Shunguo Yan and P. G. Ramachandran. 2019. The Current Status of Accessibility in Mobile Apps. *ACM Trans. Access. Comput.* 12, 1, Article 3 (Feb. 2019), 31 pages.

[51] Daihua Yu, Bambang Parmanto, Brad Dicianno, and Gede Pramana. 2015. Accessibility of mHealth Self-Care Apps for Individuals with Spina Bifida. *Perspectives in Health Information Management* 12 (04 2015), 19 pages.

[52] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P. Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA.

[53] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 6024–6037.

[54] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 609–621.

[55] Yu Zhong, Astrid Weber, Casey Burkhardt, Phil Weaver, and Jeffrey P. Bigham. 2015. Enhancing Android Accessibility for Users with Hand Tremor by Reducing Fine Pointing and Steady Tapping. In *Proceedings of the 12th International Web for All Conference* (Florence, Italy) *(W4A '15)*. Association for Computing Machinery, New York, NY, USA, Article 29, 10 pages.