An Epidemiology-inspired Large-scale Analysis of Android App Accessibility

ANNE SPENCER ROSS, XIAOYI ZHANG, and JAMES FOGARTY, Paul G. Allen School of Computer Science & Engineering, DUB Group, University of Washington, Seattle, WA, USA
JACOB O. WOBBROCK, The Information School, DUB Group, University of Washington, Seattle, WA, USA

4

Accessibility barriers in mobile applications (apps) can make it challenging for people who have impairments or use assistive technology to use those apps. Ross et al.'s epidemiology-inspired framework emphasizes that a wide variety of factors may influence an app's accessibility and presents large-scale analysis as a powerful tool for understanding the prevalence of accessibility barriers (i.e., *inaccessibility diseases*). Drawing on this framework, we performed a large-scale analysis of free Android apps, exploring the frequency of accessibility barriers and factors that may have contributed to barrier prevalence. We tested a population of 9,999 apps for seven accessibility barriers: few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextViews with *contentDescription*, fully overlapping clickable elements, and undersized elements. We began by measuring the prevalence of each accessibility barrier across all relevant element classes and apps. Missing labels and undersized elements were the most prevalent barriers. As a measure of the spread of barriers across apps, we assessed the five most reused classes of elements for missing labels and undersized elements. The Image Button class was among the most barrier-prone of the high reuse element classes; 53% of Image Button elements were missing labels and 40% were undersized. We also investigated factors that may have contributed to the high barrier prevalence in certain classes of elements, selecting examples based on prior knowledge, our analyses, and metrics of reuse and barrier-proneness. These case studies explore: (1) how the *few TalkBack-focusable elements* accessibility barrier relates to app category (e.g., Education, Entertainment) and the tools used to implement an app, (2) the prevalence of label-based barriers in image-based buttons, (3) design patterns that affect the labeling and size of Radio Buttons and Checkboxes, and (4) accessibility implications of the sizing of third-party plug-in elements. Our work characterizes the current state of Android accessibility, suggests improvements to the app ecosystem, and demonstrates analysis techniques that can be applied in further app accessibility assessments.

CCS Concepts: • **Human-centered computing → Empirical studies in accessibility**; **Accessibility design and evaluation methods**;

Additional Key Words and Phrases: Mobile applications, large-scale analyses, accessibility

X. Zhang is now at Apple, Inc.
This work was funded in part by the National Science Foundation under award IIS-1702751 and a Graduate Research Fellowship, by a Google Faculty Award, and by the Mani Charitable Foundation.
Authors addresses: A. S. Ross, X. Zhang, and J. Fogarty, Paul G. Allen School of Computer Science & Engineering, DUB Group, University of Washington, Seattle, WA, USA; emails: {ansross, xiaoyiz, jfogarty}@cs.washington.edu; J. O. Wobbrock, The Information School, DUB Group, University of Washington, Seattle, WA, USA; email: wobbrock@uw.edu.

1936-7228/2020/04-ART4 $15.00
https://doi.org/10.1145/3348797

ACM Reference format:
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.
https://doi.org/10.1145/3348797

1 INTRODUCTION

Mobile applications (apps) facilitate a range of important activities including communication, travel, finance, education, and entertainment. Unfortunately, accessibility barriers can prevent people who have impairments and/or use assistive technology from making full use of apps [14, 15, 29, 33, 35]. One such assistive technology is a screen reader, which announces screen content through audio and is primarily used by people who are blind or have low vision. An image-based button without a label[1] is one accessibility barrier that a person using a screen reader could encounter. In such cases, many screen readers unhelpfully announce "unlabeled button," leaving the image-based button's associated functionality difficult or impossible to access. Although such accessibility barriers are known to exist, the prevalence of these barriers is not well documented.

Ross et al.'s epidemiology-inspired framework [33] provided inspiration and structure for assessing the state of app accessibility. Epidemiology aims to understand and enhance the health of a population, working in concert with parts of the health system that focus on individual health. Drawing from this perspective, Ross et al.'s framework considers a *populatio*n of apps and the rich ecosystem in which they exist. The framework poses accessibility barriers as a *disease* within this population of apps; a *healthy* app is one that is accessible to all people. The framework's language and measurements focus on the apps themselves, rather than the people using those apps, and the onus of accessibility lies within apps. The health of an app population is influenced by *factors* such as developer education, reuse of classes of elements, and company policy. The framework's ecosystem-based approach to enhancing app accessibility works together with existing individual-based techniques such as development guidelines [13, 24] or testing suites [8, 12, 22]. Population-level analyses can complement such techniques by providing information on what the most common accessibility barriers are. Knowing the extent of different accessibility barriers in the app population can guide investigations into factors that may influence when and how often barriers occur.

Drawing on the epidemiology-inspired framework, we performed a large-scale analysis of accessibility barriers in free Android apps. We present our prior work [34] that assessed three classes of image-based buttons for label-based accessibility barriers in 5,753 apps. We then extend that work by including 9,999 apps and examining approximately 14K[2] classes of screen elements for seven accessibility barriers: *few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextViews with contentDescriptions*, *fully overlapping clickable elements*, and *undersized elements*.

Our analyses identify prevalent accessibility barriers as well as potential factors that affect an element's likelihood of having those barriers. Key findings in our prior work [34] include:

- We found a high prevalence of the *missing labels* barrier in image-based buttons. Of the 5,753 apps tested, 46% were missing labels on at least 90% of their image-based buttons.
- We identified features of the Android Lint v23.0.0 [8] testing suite that may impact the prevalence of *missing labels*. Android Lint is an automated testing suite that ships on the

[1]Section 3 provides definitions of technical components of Android apps that are discussed throughout this work.
[2]Due to code obfuscation (Section 3.1), some of the 14K unique class names may represent the same actual class.

Android Studio developer tool and scans source code for elements *missing labels*. One potential shortcoming of the Lint tests is that failing the *missing labels* test triggers a warning, not an error, so apps still compile. Additionally, elements from the clickable Images and Image Button classes triggered the Lint warning while Floating Action Button elements did not.

- Android Accessibility Guidelines [9] have explicit documentation and example code for creating some image-based buttons. However, we found some code examples from Android's non-accessibility-specific developer guides created image-based buttons that were *missing labels*. A developer would need additional knowledge or tools to create accessible apps from those examples. This could have a widespread impact on app accessibility if those code examples are often used.

In this extended work, key additional findings include:

- Some highly reused classes of elements have high barrier prevalence. For example, 82% of Checkboxes and 62% of Radio Buttons tested were too small based on Android's minimum size recommendation [4]. The design pattern in which a Checkbox or Radio Button was used appeared to affect the likelihood it would be too small. For example, Radio Buttons used for implementing page tabs were more likely to be large enough while Radio Buttons used as page indicators (e.g., in a tutorial sequence) were more likely to be too small. The size of Radio Buttons used for selecting among a set of items was affected by what components of the Radio Button were included in the clickable region (e.g., whether only the circular button is clickable or whether the associated text label is also clickable). Figure 16 shows examples of different clickable regions and the impact on element accessibility.
- If an element is not focusable, then many assistive technologies cannot interact with it. Apps implemented with cross-platform tools (e.g., Adobe Air) or with game engines (e.g., Unity) commonly had the accessibility barrier of *few TalkBack-focusable elements*. Apps in Android's Education category were disproportionately likely to have the *few TalkBack-focusable elements* barrier.
- Third parties may influence the accessibility of their plug-in elements through defaults or usage guides. For example, the Google+ and Facebook Login elements were similarly sized and styled across apps. Google+'s visually smaller icon was *too small in both* height and width for 43% of uses and *too short only* for an additional 56% of uses. The Facebook Login element was *too small in both* dimensions for only 0.3% of uses but was *too short only* in 82% of tested uses.

In addition to these findings, we present our methods for filtering this large-scale app data to help surface potential trends in accessibility barriers. Specifically, we choose classes of elements based on prior knowledge about common accessibility barriers, the reuse of classes across apps, and the barrier proneness of the classes. These filtered analyses highlight patterns in accessibility barriers.

As a high-level overview of Android app accessibility, we first present the prevalence of accessibility barriers over all apps and all relevant classes of elements appearing in those apps. Accessibility of apps throughout the population is heavily impacted by the accessibility of classes with widespread use. We therefore further present the prevalence of the most common accessibility barriers within the five element classes that are most reused among apps. Finally, we perform scoped assessments on a few key classes of elements. The choice of element classes was based on the results of the broader-scoped analyses, on metrics such as class reuse, and on existing knowledge of common accessibility barriers.

In our discussion (see Section 9), we apply Ross et al.'s epidemiology-inspired framework to our findings. This framing allows us to situate prior large-scale accessibility research [41], our prevalence findings, and our case studies into the rich ecosystem of factors that impact app accessibility at scale. We then use the framework and our results to discuss possible opportunities for enhancing app accessibility at a population level.

2 RELATED WORK

This article extends our work performing the first large-scale analysis of mobile app accessibility [34]. We present prior work on app accessibility analysis, app analysis beyond accessibility, and large-scale longitudinal studies of website accessibility.

2.1 App Accessibility Analyses

Prior work has explored accessibility barriers in apps. Some work focused on specific types of apps such as health [29], smart cities [14], and government engagement [35]. Other work used more generalized samples of apps [15, 31]. Sample sizes for these studies primarily ranged from 4 to 10 apps [14, 29, 35] with one study analyzing 479 apps [41]. Many accessibility barriers were found to be frequent and problematic, including elements missing labels [14, 35, 41], incorrectly labeled [29], not focusable [14, 31, 41], too small in size [41], and too low in contrast [14, 35, 41]. However, it is difficult to establish a general understanding of the state of app accessibility with these studies due to the narrow scope of their assessments. These prior studies of app accessibility were performed on smaller populations of apps (one to two orders of magnitude fewer apps than the 9K we tested), emphasizing an opportunity to use large-scale analysis to develop a more general, holistic understanding of the state of app accessibility.

Ross et al.'s analyses of 100 apps [33] closely parallel the analyses performed in this work. This similarity is due to their usage of the Google Accessibility Scanner, a tool for manually testing apps that is implemented using the same Accessibility Testing Framework for Android on which we based our tests. Differences in tests performed in our work versus prior work highlight the value of multiple approaches to assessment to construct a richer characterization of app accessibility.

In work most similar to ours, Yan and Ramachandran [41] assessed 479 free Android apps for a set of accessibility barriers. Yan and Ramachandran's analysis of the current state of app accessibility contributes a detailed description of an app assessment tool, metrics for capturing the state of app accessibility, and the frequency of a set of accessibility barriers in 479 apps. Our analyses complement Yan and Ramachandran's work with some variation in accessibility barrier tests, different metrics, a focus on looking at each barrier independently, a deeper exploration of element classes, and a significantly larger dataset of over 9K apps.

In their work, Yan and Ramachandran [41] propose two main measures of app accessibility: inaccessible element rate (i.e., a measure of the proportion of elements with *any* accessibility barrier out of the total number of tested elements) and accessibility issue rate (i.e., a measure of the number of accessibility barriers in an app out of all possible accessibility barriers in an app). Their metrics are similar to our prevalence measure, which is the proportion of elements that have a *specific* accessibility barrier out of all elements tested for that barrier. These normalized measures allow for comparison among apps that have different number of elements.

Yan and Ramachandran used the IBM Mobile Accessibility Checker to identify accessibility issues based on 22 accessibility rules grouped by violation, potential violation, and warning, depending on the accuracy of the automated test. A subset of these rules was explained in detail in the article. Missing label, element too small, lack of element focus, text too small, low text contrast, and lack of sufficient spacing between elements were identified as the main causes of accessibility barriers in apps.

The differences in testing techniques and reported statistics do not allow direct comparison of our results to those of Yan and Ramachandran [41]. However, the analyses from our work and theirs indicate similar underlying problems and highly prevalent accessibility barriers. We discuss the connection between our results and theirs in more detail in Section 9.

2.2 Web Accessibility Analyses

The web has a long history of accessibility analyses. Hanson et al. [26] performed a longitudinal study of 100 top government and commercial websites over 14 years. Findings include that, overall, accessibility improved over time. In follow-up work, Richards et al. [32] discussed potential contributing factors such as changes in web coding practices. Kane et al. [28] performed manual and automated accessibility analyses of 100 university websites. Their results indicate the continued impact of inaccessibility on the web as well as potential contributing factors that include the university's country and legislation. Such work highlights the characterizations of accessibility that can be developed through large-scale and longitudinal analyses, leveraging manual and automated methods, and exploring influential factors in accessibility through data.

2.3 Population-level App Analyses

Prior large-scale app analyses have focused on a range of topics, including security [2, 21, 27], design patterns [19, 20], and code reuse trends [30]. These studies help characterize the richer ecosystem in which apps exist. For example, Mojica et al. [30] found 84% percent of class signatures in the 208,601 apps analyzed were reused among apps. This finding supports our focus on class reuse as a factor that can influence app accessibility at a large scale. Deka et al.'s [19] analysis of ~9K apps for visual design allowed them to group and categorize similar interface designs. We similarly investigated the visual design choices associated with having certain classes of elements with and without accessibility barriers (see Section 8.3).

A limited number of large datasets of Android apps have been released for analysis. Deka et al. [19] released the Rico repository [18], a collection of ~10K free Android apps containing data such as app metadata, screenshots, and view hierarchies. We use the Rico dataset for our analysis due to its size and detail, as discussed in Section 4.1. Alli et al.'s Androzoo [2] project has collected over 5M app APKs[3] and makes them available for academic use. App APKs have been regularly added to the dataset since 2011. Refining the capture of data directly from APKs is an opportunity for future work that could allow leveraging this dataset for large-scale and longitudinal accessibility analysis.

3 ANDROID BACKGROUND

This section presents a brief technical background on how Android apps are implemented and how they interact with assistive technologies. We define a key set of terms used throughout our analyses and discussions: app category, element, class, view hierarchy, and TalkBack-focusable. When uploading an app to the Google Play Store, an app creator can choose one *category* from a pre-established list such as Education, Communication, or Weather.

3.1 Elements, Classes, and View Hierarchies

Elements are the building blocks of app interfaces. Some types of elements are used to define the visual structure (i.e., layout) of a screen, such as the arrangement of child elements into rows or columns. Other types of elements are the primary visual and interactive content of a screen, such as its buttons, text boxes, and images.

[3]APKs are Android's file package for the installing and running apps. It is similar to an .exe file on Microsoft Windows.

The *class* of an element refers to a packaged implementation of the element that can be re-used. For example, the Android API class `android.widget.ImageButton` provides an implementation of the basic visuals and skeleton functionality of an image-based button. The Android API provides many classes for commonly used layout and visual elements (e.g., a standard button class `android.widget.Button`). Third parties can also supply classes for creating elements, such as Facebook's Login button (discussed in Section 8.3).

Screens are usually composed of many elements, often nested within one another. A *view hierarchy* is a representation of all of a screen's elements and their nesting, named after the base Android class `android.view.View`. The view hierarchy also captures *attributes* about each element. These attributes include the class, the location on the screen, if it is clickable, if it is visible, and its `contentDescription` (needed for labeling, as discussed in Section 5.2).

Minification techniques [36] reduce the size of an app by shortening the names of classes, functions, and other code. This practice obfuscates information captured in the view hierarchy. For example, a button element from the class `android.widget.Button` could have its class attribute obfuscated to a value `z` in the view hierarchy. The mapping of obfuscated class names to original class names depends on obfuscation technique and cannot be determined from the app APK alone.

3.2 TalkBack-focusable

Assistive technologies must determine which elements should receive focus (e.g., text to read, interactive buttons) versus which should be ignored (e.g., used for layout, in a hidden tab, behind a pop-up dialog box). We labeled elements that are visited by Android's TalkBack screen reader as *TalkBack-focusable*. Most of our tests considered only elements that were Talkback-focusable (i.e., most of our tests ignore elements that would also be ignored by the screen reader). The exception is our *few TalkBack-focusable elements* test, which considers a lack of focusable elements as an indication of a larger accessibility issue.

We used the logic from TalkBack's implementation to determine which elements in a screen's view hierarchies were TalkBack-focusable (see Appendix A for detail). TalkBack uses heuristics to determine if it will focus on an element, such as whether the element is clickable. We adapted the heuristics to use the information available from the Rico view hierarchies. Not all element attributes used by TalkBack were available in the dataset, such as the `checkable` property, so those heuristics were skipped. The Google-released Accessibility Test Framework for Android [23], one of the state-of-the-art accessibility testing tools for Android, similarly reimplements TalkBack's logic to determine which elements to test. Although this approach likely misses some important elements or includes some elements that are not of interest, it mirrors the Accessibility Test Framework and provides a meaningful characterization of app accessibility.

4 METHOD

This section presents our dataset and the development of our filtering metrics. Applying filters to our dataset allowed us to highlight barriers with widespread impact. The filters we applied and our method for choosing filters offers an approach for future analyses.

4.1 Data

Our app dataset is a subset of the Rico repository, downloaded April 17, 2018 [18]. We filtered the 10,477 free Android apps in the repository based on exclusion criteria described below. We analyzed 9,999 apps that contained elements from approximately 14K distinct element class names. For each app, the dataset included the app's metadata, a set of screenshots, and the view hierarchies associated with those screenshots. The metadata captured app attributes such as its *category*.

A developer can choose one Android-defined category for their app (e.g., Education, Communication).

The view hierarchies were JSON files that represented the screen structures and the characteristics of all captured elements on those screens. Element attributes included class, text, `contentDescription`, bounds, ancestor classes, and children elements. Details of this dataset and its collection can be found in the paper by Deka et al. [19].

4.1.1 Exclusion Criteria. Every app in the dataset has a unique `package_name`. Each view hierarchy file indicated which app was in focus when the screen was captured with an `activity_name` field in the form `<package_name>/<activity_name>`. If the package in the `activity_name` field of a captured view hierarchy did not match that of the app being assessed, that specific screen was ignored as invalid. This criterion eliminated screens captured outside of the app, such as the Android home screen, the lock screen, or a redirection to a web browser. If a view hierarchy file was `null`, the associated screen was discarded as invalid. If an app had no valid screens, and therefore zero captured elements, then the entire app was ignored.

A portion of our analyses focused on subsets of element classes. In such cases, we considered only apps with at least one element from a class of interest. The class of an element was determined by its `class` field in the view hierarchy. Some apps had been obfuscated (Section 3.1). For example, the class `ztu` and `yei` were both mutated names for the same Google+ button shown in Figure 19. As is typically the case with obfuscation, we did not know the mapping from original class name to obfuscated name. We therefore treated each class name in the dataset as distinct. The mapping from these class names to the Google + button was discovered by manually viewing the screenshots that contained `ztu` and `yei` class elements, a non-scalable approach for all potentially obfuscated class names.

4.1.2 Limitations. The Rico repository was collected by Deka et al. [19] to analyze design patterns; it was not originally motivated by accessibility analyses. Using a dataset outside of its intended purpose often adds limitations. For example, the captured view hierarchies did not include some element attributes that would have improved accessibility assessment (see Appendix A for details on missing attributes).

The Rico dataset was also limited to Android apps. Prior research on iOS apps [29, 35] identified many similar accessibility barriers, such as missing labels. However, the prevalence of those barriers found in this work may not reflect the state of iOS app accessibility. A future comparison of iOS and Android apps could yield insight into how factors that differ between the two platforms ecosystems may have a strong impact on app accessibility. For example, the Apple App Store for iOS and the Google Play store for Android have different processes a developer must go through to publish on their marketplace. Differences in those processes could impact the accessibility of those app populations.

Large datasets of Android app screens are difficult to collect, especially when collection tools do not have access to the source code. Determining if two screenshots are of the same screen is one challenge, because app screens have no robust, unique identifier (i.e., no equivalent to a URL in the web). Due to this challenge, some apps had very little of their functionality captured. For example, the dataset contained over 185 screens for the WhatsApp Messenger app. However, those screens cover a very narrow range of its functionality. Specifically, all of the captured screens are of the country selection and phone number verification steps of registration. This lack of coverage reflects limitations in capture techniques. The Rico crawler may have failed to identify these as duplicate screens because of pixel-level differences in the screenshots resulting from the visible country names changing due to scrolling through the country list. Barriers to logging into apps (e.g., needing a bank account) or the need for human input (e.g., a login screen) are additional

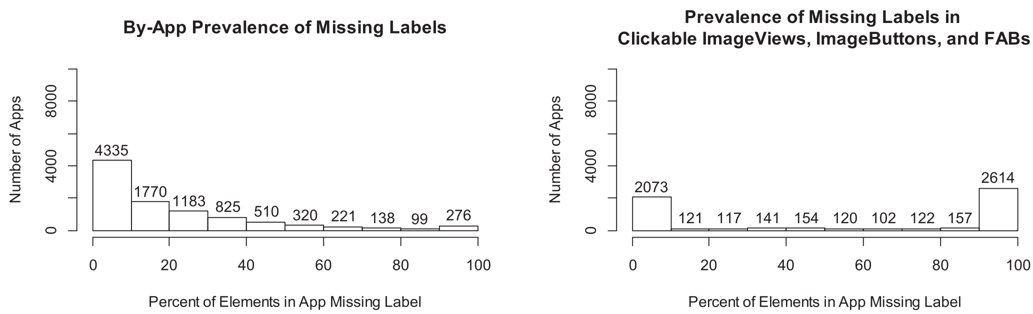

Fig. 1. A comparison of the distribution of percent of elements within an app that are missing labels between (left) considering all tested classes of elements in the apps (9,677 apps total) and (right) focusing on Clickable Images, Image Buttons, and Floating Action Buttons in apps (5,721 apps total). Considering only the more relevant elements in an app highlights a significant problem that is not apparent from the analysis of all elements.

data collection challenges. Further work in improving data collection methods would allow for improved large-scale app assessment.

Despite the limitations of the Rico repository, it contains a significant amount of useful information that is otherwise difficult to collect. We believe these data give meaningful insights into the state of app accessibility.

4.2 Filtering

We filtered data based on existing knowledge of accessibility barriers (i.e., *knowledge-based filters*) and based on the reuse of classes and the likelihood of accessibility barriers in those classes (i.e., *metric-based filters*). Analyses of focused subsets of data can reveal trends in app accessibility that may be obscured in analyses of the full dataset.

Our analysis of label-based accessibility barriers in image-based buttons (Section 8.2) is one example of a knowledge-based filtered analysis. Prior work and existing accessibility guides [14, 29, 31, 35] identified elements with missing labels as an important app accessibility barrier. However, the significance of missing labels is not immediately clear from a prevalence analyses on the full dataset (Figure 1, left). Apps predominantly fall in the low prevalence region of the distribution, with only 2.9% of apps having over 90% of their elements missing labels. The infrequency of the barrier suggests missing labels might be best addressed by focusing on specific, poor-performing apps.

Image-based elements need the developer to provide a `contentDescription` attribute for use as a label, whereas text-based elements do not (see Section 5.2 for details). Using that knowledge, we filtered the data to focus on three popular classes of image-based buttons. In contrast to the unfiltered data analysis, missing labels were highly prevalent in image-based buttons; 46% of apps had over 90% of their image-based buttons missing labels (Figure 1, right). This filtered distribution suggests a more systematic problem involving factors beyond specific apps. Section 8.2 presents details of our image-based button analysis.

Metric-based filters complement knowledge-based filters by revealing trends that may not be apparent from current knowledge (e.g., Section 8.3 on Checkboxes and Radio Buttons). We used two metrics to support filtering: (1) class *reuse* (i.e., the number of different apps that have at least one element of that class captured and tested in the dataset) and (2) class *barrier-proneness* (i.e., the number of elements of that class with a given accessibility barrier out of all tested elements of that class).

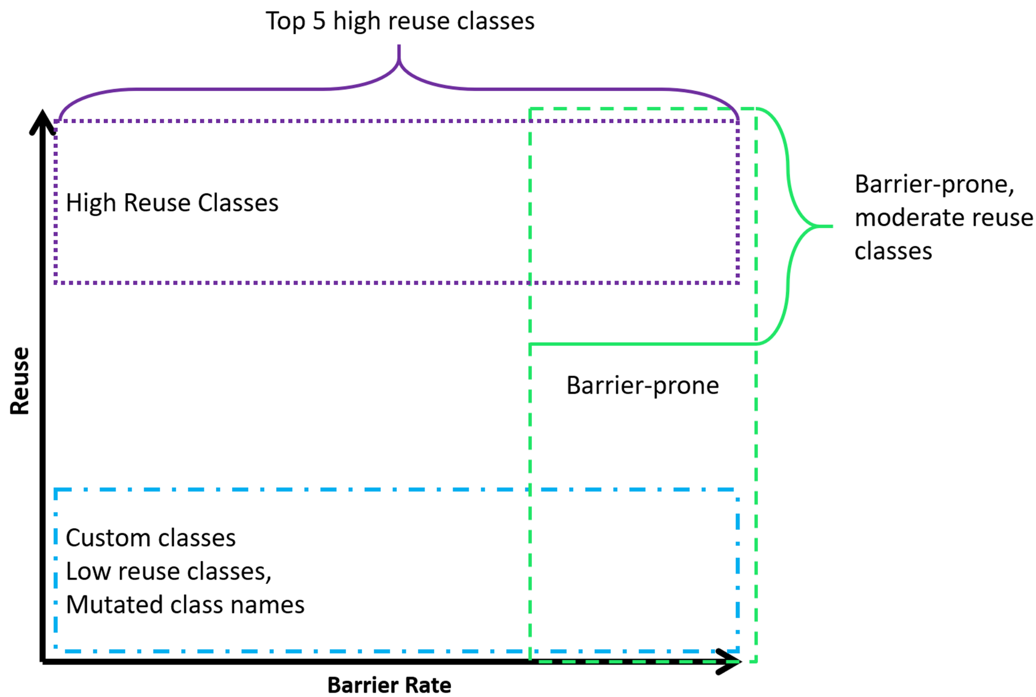

Fig. 2. Two dimensions of impact when considering classes of elements. Interface class reuse was defined as the number of apps that contained at least one tested element of that class. Barrier proneness was defined as (the total number of elements of a class across all apps that had a given accessibility barrier) / (the total number of elements of a class tested for that barrier). Parts of the space are labeled with the types of classes the region likely contains. Our analyses focused on two sections of the space indicated with brackets.

Analyzing any region of the space defined by these two metrics (as visualized in Figure 2) may yield different types of app accessibility insights. For example, classes with low reuse may include more custom classes developed for a specific app or small set of apps. Trends in low-reuse classes may reflect risk factors associated with creating and using custom or uncommon classes. Such insights could suggest opportunities to improve documentation or tools for creating custom classes. In our dataset, however, some of the low reuse class names were due to code obfuscation. Deeper investigations of the accessibility of low reuse classes are opportunities for future work.

We focused on two groups of classes: (1) *high reuse classes* and (2) *barrier-prone moderate reuse classes*, as indicated by the brackets in Figure 2. For *high reuse classes*, we look at the top five most reused classes, regardless of rate of accessibility barriers. The accessibility of high reuse classes can have a widespread impact on the overall accessibility of the population of apps. For *barrier-prone moderate reuse classes*, we look at classes with high barrier rates and moderate reuse. We define moderate reuse as classes in the top 1% of reuse out of the approximately 14K total unique class names. This definition filters out classes that are used in very few apps, including the approximately 9K class names that appeared in only one app. The top 1% captures a range of reuse. For example, for label-based barrier, the top 1% of reused classes contains 140 classes that are each used in 36–6,233 apps. High reuse classes will be included in the moderate reuse subset but will not necessarily have high barrier rates. A table of classes in the top 1% of reuse for each accessibility barrier can be found in Appendix B. For clarity and brevity, in this work, we present analyses of a subset of high reuse classes and barrier-prone moderate reuse classes.

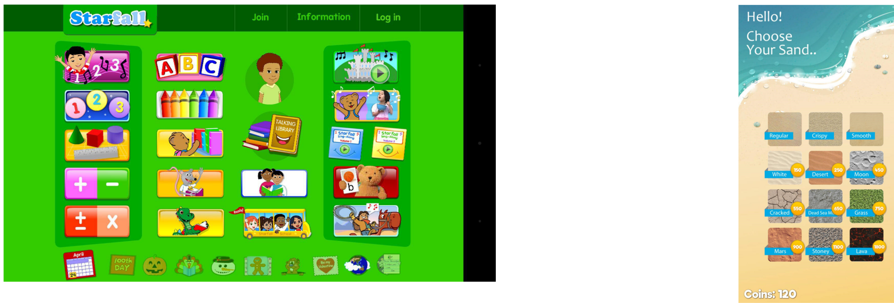

Fig. 3. Example apps with the *few TalkBack-focusable elements* barrier. (left) The Starfall app has zero focusable elements per screen despite being full of evident targets and information. (right) The Sand Draw app has one focusable element per screen despite showing text boxes and a grid of 12 targets.

The *barrier-prone moderate reuse classes* filter captures classes that may not necessarily be the most reused but are more likely to have an accessibility barrier when they are used. Investigating extreme cases of classes with high barrier prevalence can guide and complement other investigations into what creates a worst-case class (e.g., lack of documentation, lack of tool support, lack of awareness or education). Although understanding barrier-prone classes regardless of reuse (i.e., the entire right side of Figure 2) might also provide valuable insights, we focus on classes with at least moderate reuse due to the impact of their accessibility across many apps in the population. We apply the knowledge-based, *high reuse,* and *barrier-prone moderate reuse* class filters to our analyses in Sections 7 and 8.

5 ACCESSIBILITY BARRIERS

This section defines our test for each accessibility barrier together with the criteria for elements that were tested for each barrier. Most prior studies [14, 15, 17, 35, 37] constructed their accessibility tests using adapted versions of the Web Content Accessibility Guidelines (WCAG). Tests were also based on industry-released guidelines [29] and Section 508 [29], legislation in the United States that legally mandates certain government-related technology must be accessible to people with disabilities. The majority of the accessibility barriers we test for are based on similar guidelines.

We tested apps for seven accessibility barriers: *few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextViews with contentDescriptions, fully overlapping clickable elements,* and *undersized elements.* All tests except the tests for *uninformative labels* and *few TalkBack-focusable elements* were based on the Google-released Accessibility Test Framework for Android [23]. Details on the use of the Accessibility Test Framework for Android to implement the tests can be found in Appendix C. Tests for *uninformative labels* and *few TalkBack-focusable elements* were operationalized by the first author based on prior work and known accessibility barriers. As with all automated accessibility evaluation tools, our accessibility tests have limitations in their coverage and accuracy. We note these limitations for each test below.

5.1 Few TalkBack-focusable Elements

Having one or fewer TalkBack-focusable elements on a screen is likely an instance of the screen not properly exposing its elements to assistive technologies. An app screen with one or no TalkBack-focusable elements is functionally unusable with many assistive technologies, equivalent to interacting with an unresponsive blank screen. Examples of screens with *few TalkBack-focusable elements* are presented in Figure 3.

Apps with one or fewer focusable elements per screen, averaged over all captured screens, tested positive for the *few TalkBack-focusable elements* barrier. We tested all 9,999 valid apps (i.e., apps that had at least one element captured, even if it was not TalkBack-focusable) for *few TalkBack-focusable elements*.

One limitation of this test is its inability to distinguish between problematic screens and screens that may legitimately have only one TalkBack-focusable element, such as a loading screen with only a progress bar or an empty list. However, most designs have more than one element and screens with one or fewer TalkBack-focusable elements likely contain a set of non-focusable but essential elements. Future work could explore more robust testing techniques (e.g., computer vision or crowdsourcing) to more accurately identify apps that do not properly expose their screens.

5.2 Label-based Inaccessibility

Screen readers use element labels to announce what an element says or represents. For text-based elements, the TalkBack screen reader can directly use that text for the label. For image-based elements, an additional label source is needed, akin to alt-text for images on the web. For most image-based elements, this label comes from the `contentDescription` attribute or is inherited from a labeled child element. We used the logic from TalkBack and the Accessibility Test Framework for Android to determine the label, or lack thereof, for each element. Our dataset did not include the `labelFor` element attribute, which allows a developer to explicitly use one element as a label for another (e.g., using a text box with visible text as the announced label for an editable text field). Thus, there may be some elements labeled using the `labelFor` technique that our tests classify as unlabeled. However, the accessible development documentation [3, 24] and Android Studio Lint test recommendations [8] do not cover using the labelFor attribute—they only mention the use of a `contentDescription`. We therefore believe our element label identification technique provided a reasonable evaluation.

We tested for four *label-based* accessibility barriers: *missing label, duplicate label, uninformative label,* and *editable TextView with contentDescription*. Labels are primarily used by screen readers, therefore, we only tested TalkBack-focusable elements. Because TalkBack uses different logic for Web View elements than it uses for native Android elements, we did not test Web Views for label-based errors. We did not test editable TextViews for *missing, duplicate,* or *uninformative* labels, since those elements have unique label requirements, as captured in the *editable TextView with contentDescription* barrier. The testing criteria for each *label-based* accessibility barrier are detailed below.

5.2.1 Missing Label. Unlabeled elements create major accessibility barriers in apps. When focusing on elements with *missing labels*, a screen reader may announce an unhelpfully vague label (e.g., "unlabeled button," "button") or nothing at all. We tested the 9,677 apps that had at least one TalkBack-focusable element.

5.2.2 Duplicate Label. The presence of multiple clickable elements on a screen with the exact same label may be confusing to people using a screen reader. Examples of problematic duplicate labels are presented in Figure 13. We tested for duplicate labels by comparing the labels of all clickable, TalkBack-focusable elements on a single screen. A clickable element tested positive for *duplicate label* if it had the exact same label as another clickable element on the same screen. There are instances of legitimate duplicate labels in apps (e.g., a list of songs in a music app where all the authors are labeled "Unknown"). Our techniques do not distinguish between legitimate and problematic duplicate labels. To avoid an overpowering effect of missing labels, we performed duplicate label analyses only on labeled elements. We tested the 8,869 apps that had at least one screen with at least two labeled clickable elements.

5.2.3 Uninformative Label. It is crucial for element labels to be meaningful; an element labeled "image" can be as significant of an accessibility barrier as an unlabeled element. To develop a list of uninformative labels, the first author reviewed the data for all labels of clickable Image Views, Image Buttons, and Floating Actions Buttons. The first author noted labels whose content was only a reflection of the class or the `contentDescription` attribute name (i.e., the label was composed of only these words or their abbreviations: button, image, content, description, icon, or view[4]). Note that we did not test whether labels were accurate (e.g., whether a button labeled "back" actually functioned as a back button). Future work could expand uninformative label detection (e.g., using crowdsourcing to expand and assign severity to the list of uninformative labels). We tested the 9,650 apps that had at least one labeled clickable element, excluding elements with *missing labels*.

5.2.4 Editable TextView with ContentDescription. Most non-text elements are labeled using the `contentDescription` attribute. However, the `contentDescription` attribute should *not* be used with editable TextView elements. Editable TextView elements allow a person to enter text (e.g., typing text into a search bar) and TalkBack should announce the entered text. A `contentDescription` can interfere with that functionality; TalkBack may announce the `contentDescription` instead of any entered text. Therefore, adding a `contentDescription` for an editable TextView is an accessibility barrier. Instead, the `hint` attribute can be used to add a label to an empty editable TextView, akin to a visual text prompt appearing in a textbox before a person begins entering text. Our test defined editable TextViews as elements that are TalkBack-focusable and have an ancestor class of `android.widget.EditText`. Because the `hint` attribute was not available in the Rico view hierarchies, we tested that editable TextViews did not provide a `contentDescription` but could not test whether they appropriately provided a `hint`. We tested the 2,919 apps that had at least one editable TextView element.

5.3 Fully Overlapping Clickable Elements

Clickable elements that fully overlap make it challenging to activate an occluded target. If the fully overlapping elements perform different actions when clicked, then it is impossible to use both functionalities. However, even fully overlapping clickable elements that perform the same functionality can cause problems for assistive technologies such as Google's VoiceAccess, which supports hands-free interaction. VoiceAccess first visually labels each clickable element with a number (Figure 4). People can then speak a number aloud to interact with the corresponding element. For example, in the left app in Figure 4, someone could say "tap 24" to go to the photo view. If two clickable elements are fully overlapping, then they will both receive a distinct number. The extraneous number labels add visual clutter and confusion to the screen. In the left example in Figure 4, someone could say "tap 21" or "tap 22" to go to the map view. The visual clutter and confusion are further elevated when more than two clickable elements fully overlap. For example, in the right app in Figure 4, someone has seven numbers they can verbally "tap" to activate the same image.

Fully overlapping elements with the same functionality can occur when a developer accidentally codes an element and its parent element to be clickable with the same action. The Accessibility Test Framework for Android tests if elements that have the `clickable` and `importantForAccessibility` attributes set to `True` are *fully overlapping*. The Rico dataset, however, did not include the `importantForAccessibility` element attribute. We therefore approximated the Accessibility Test Framework's approach by testing whether each clickable,

[4]The resultant set of "uninformative labels" was: alt image, button, Button, contentDescription, desc, Desc, Description, Description Image, icon desc, [image], image, Image, images, Images, image description, Image Des, image description default, Icon, Image Content, ImageView, and View.

Fig. 4. Apps with VoiceAccess turned on. Each number represents an interactive element. Left: Two fully overlapping clickable elements with the same functionality are notated as 21 and 22. Right: Many overlapping clickable elements add substantial visual clutter and confusion.

TalkBack-focusable element fully overlapped with another clickable, TalkBack-focusable element on the same screen. We determined an element's location on the screen using its `bounds` attribute. We tested the 9,171 apps with at least two clickable elements on a screen.

5.4 Size-based Inaccessibility

Interactive elements can be too small for people to accurately touch. This barrier can be particularly significant for people with motor impairments or for people using an explore-by-touch screen reader technique that announces elements as a person moves a finger around the screen. Following the Google Accessibility Test Framework for Android [23], we tested elements that were clickable and TalkBack-focusable for size-based accessibility barriers. Our test used the Google Accessibility Guidelines for Android [9] suggestion of a minimum size of $48dp \times 48dp$.

The Android guideline size suggestion uses the density-independent pixels unit (*dp*) to account for varying screen resolutions (which are measured in dots per inch, or *dpi*). We calculated the size of an element in the dataset using the bounds attribute from the view hierarchy, measured in pixels (*px*). We then converted the pixel-unit measurement (*px*) to density-independent pixels (*dp*) using the formula $dp = (px \times 160)/dpi$ [38].

The pixel density (*dpi*) value is determined by device resolution, but the Rico publication [19] does not specify the capture device. We therefore reverse-engineered a likely *dpi* for our calculations, deciding on a value of 560 *dpi*. We based this value on the Google Nexus 6P, and we additionally verified that it yielded behaviors consistent with the Rico dataset. For example, we compared the size of a variety of elements from the Rico dataset to the size of that same element in the current version of the same app running live on a Nexus 6P. We also used the Google Accessibility Scanner [22], based on the Accessibility Test Framework for Android [23], to test the minimum size of several elements in the live version of an app, comparing results to those obtained for the associated element in the Rico dataset. The consistency of our results with these measures of live apps confirmed that our pixel density assumption was reasonable.

We define our overall *size-based* barrier (1) *too small in either* as elements that are not tall enough, not wide enough, or both. To explore nuances between failures to make elements large enough in different dimensions, we then break down the barrier by dimension into (2) *too small in both:* elements that are not tall enough and not wide enough; (3) *too short only:* elements that are only not tall enough; and (4) *too narrow only:* elements that are only not wide enough. The prevalence of *too small in either* barrier for a given class is the sum of the other three dimension-specific barrier prevalence measures.

Developers can use several techniques to create elements that meet the size recommendations. The most obvious approach is creating elements that are themselves large enough. However, enlarging elements may conflict with a desired visual appearance. An alternative is to place a visible element inside of a larger, invisible container and to set the larger container as the clickable element [9]. In these cases, the invisible container is TalkBack-focusable, whereas the contained visible element would not be. Our tests, which were applied to TalkBack-focusable elements, capture both implementation techniques. We tested the 9,650 apps with at least one clickable TalkBack-focusable element.

6 ALL-CLASS, BY-APP ANALYSES

We present the prevalence of each accessibility barrier over all classes of tested elements in all apps. This measure is one indication of the spread and impact of each accessibility barrier. It is important to note that prevalence alone cannot fully characterize app accessibility. Even a few barriers within an app can create significant accessibility problems if those barriers impact core app functionality. Conversely, a relatively large number of barriers isolated in a rarely used part

Table 1. Prevalence of Apps with Few TalkBack-focusable Elements, Broken Down by How Many Elements Per Screen on Average the App Had

average # TalkBack-focusable elements per screen	# apps	% apps
0	253	2.5%
(0,1)	98	1.0%
1	440	4.4%
[0,1]	791	7.9%

of an app may have less impact. We focus on prevalence of barriers within apps as one valuable and informative component of characterizing the accessibility of the app population.

6.1 Few TalkBack-focusable Elements

We examined the prevalence of *few TalkBack-focusable elements,* considering populations of apps with an average of exactly 0, exactly 1, and between 0–1 focusable elements per screen (Table 1). The experience of using a screen that has zero TalkBack-focusable elements is likely similar to that of using a screen with one focusable element (e.g., a screen reader cannot access any screen content on the screen or give any indication of screen functionality). However, the factors influencing when an app has an average of zero versus up to one focusable element per screen may be different (e.g., different developer tools or libraries).

Of all 9,999 tested apps with at least one element (i.e., focusable or not), 791 (7.9%) had an average of no more than 1 focusable element per screen. This included 253 apps (2.5%) with an average of exactly 0, 98 apps (1.0%) with an average between 0 and 1, and 440 apps (4.4%) with an average of exactly 1 element per screen.

Although the prevalence of *few TalkBack-focusable elements* is lower than some other accessibility barriers, the severity of the barrier may be notably higher (i.e., not being able to interact with any of an app's functionality). We discuss a subset of apps with the *few TalkBack-focusable elements* and potential causes of that barrier in Section 8.1.

6.2 Label-based Inaccessibility

We examined the prevalence of four types of *label-bas*ed accessibility barriers: *missing labels, duplicate labels, uninformative labels*, and *editable TextViews with contentDescriptions.*

6.2.1 Missing Labels. We tested 9,677 apps with at least one TalkBack-focusable element for *missing labels*. There was a median of 124 TalkBack-focusable elements tested per app (*M*: 232, *SD*: 338, *R*: 1–7,916[5]). In the distribution of the *missing labels* barriers over all tested elements in all apps, most apps were in the lower prevalence ranges (Figure 5, left). In the low prevalence range, 4,335 apps (45%) were missing less than 10% of their element labels. At the lowest prevalence, 2,191 (22%) of apps had exactly zero tested elements missing labels. The apps with exactly zero elements missing labels had a median of 28 tested elements (*M*: 65, *SD*: 113, *R*: 1–1,830).

Looking at apps with a high prevalence of missing labels, 276 apps (3%) had 90% or more of their elements missing labels. Within the rest of the distribution, 4,288 apps (44%) were missing 10%–50% of their element labels and 778 apps (8%) were missing 50%–90% of their element labels.

Screen readers can use text that elements present visually, as labels are automatically generated based on that same text. In general, to label elements without text, developers must add a `contentDescription`. We explored whether the additional effort needed to label non-text

[5]In the statistical reporting, *"M"* stands for arithmetic mean, "*SD*" stands for standard deviation, and "*R*" stands for range.

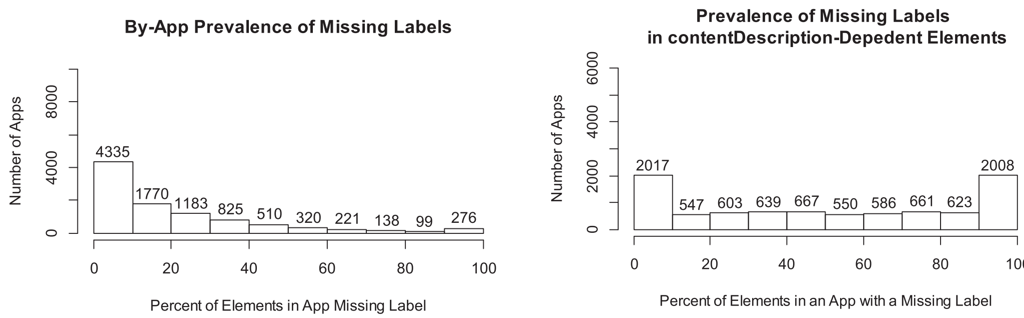

Fig. 5. Distribution of the *missing label* accessibility barrier per app. Left: considering all TalkBack-focusable elements, showing 9,677 apps. Right: considering all elements that depended on a `contentDescription`, showing 8,901 apps. Focusing only on elements that use `contentDescriptions` highlights a high prevalence of missing labels.

elements put those elements at higher risk for having *missing labels* by filtering the data to exclude elements that had a label but did not have a `contentDescription`. This knowledge-based filter approximated excluding elements that automatically have labels because of their visible text.

The resulting distribution of `contentDescription`-dependent elements was bimodal, with peaks at the left and right extremes (Figure 5, right). There was a median of 50 tested `contentDescription`-dependent elements per app (M : 99, SD : 161, R : 1–5, 666). In the left end of the distribution, 2,017 apps (23%) had 0%–10% of their elements missing labels. In that 0%–10% interval, 1,415 apps (16%) had exactly zero elements missing labels. The apps that had exactly 0% of their elements missing labels had a median of 18 `contentDescription`-dependent elements tested (M : 36, SD : 56, R : 1–656).

At the right end of the distribution, 2,008 apps (23%) had 90%–100% of their elements missing labels with 1,535 apps (17%) at exactly 100%. The apps with exactly 100% had a median of 19 `contentDescription`-dependent elements tested for missing labels (M : 55, SD : 129, R : 1–1,942).

The remaining 4,876 apps (55%) were uniformly distributed, having between 10%–90% of their `contentDescription`-dependent elements missing labels. This mid-range prevalence suggests that even when labels were applied to some elements in an app, they were not applied to all elements. Section 8.2 examines an additional analysis of the *missing labels* barrier in image-based elements.

6.2.2 Duplicate Labels. We tested the screens of 8,869 apps with at least two labeled clickable elements for the *duplicate labels* accessibility barrier. There was a median of 99 tested elements per app (M : 185, SD : 279, R : 1–7,714). The distribution shows low prevalence of *duplicate labels* across apps (Figure 6). In the left end of the distribution, 7,902 apps (89%) had 0%–10% of their clickable elements with *duplicate labels*. There were 6,812 (77%) apps with exactly 0% of tested elements with *duplicate labels*. These apps at 0% had a median of 70 tested elements (M : 132, SD : 200, R : 1–6,777). The remaining apps tended to have lower prevalence of *duplicate labels*; 829 apps (9.3%) had 10%–50% of tested clickable elements with the *duplicate labels* accessibility barrier. We discuss patterns of duplicate labels in image-based buttons in Section 8.2.2.

6.2.3 Uninformative Labels. We tested 9,121 apps with at least one labeled TalkBack-focusable element for *uninformative labels*. Our app population had a low prevalence of this barrier (Figure 7). There was a median of 98 elements tested per app for *uninformative labels* (M: 184, SD: 277, R: 1–7,714). Out of 9,121 apps tested, 9,003 (99%) had 0%–10% of their elements with uninformative

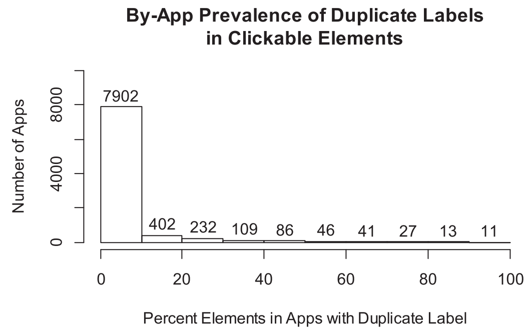

Fig. 6. Distribution of clickable elements with duplicate labels per app out of 8,869 apps.

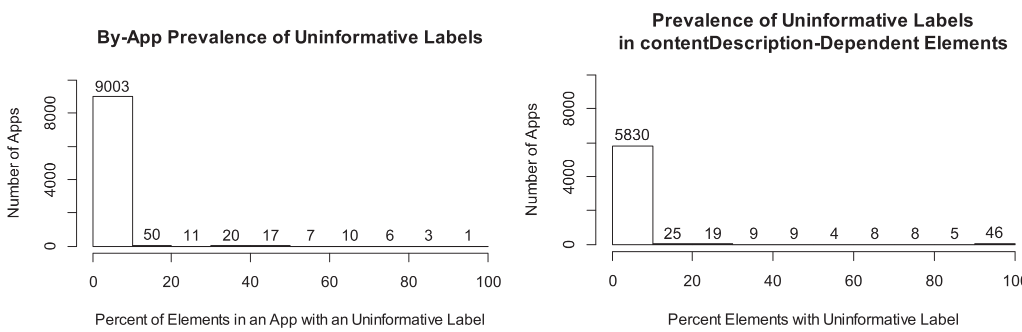

Fig. 7. Distribution of percent of uninformative element labels per app over (left) all tested elements per app, out of 9,121 apps; (right) `contentDescription`-dependent elements per app, out of 5,963 apps.

labels. The 8,796 apps (96%) with exactly 0% of their elements with uninformative labels had a median of 94 elements tested (*M*: 178, *SD*: 273, *R*:1–7,714).

Informative labels may be automatically obtained from the visible text of an element; therefore, we again filtered to elements that required a `contentDescription` (i.e., as in Section 6.2.1's examination of missing labels). Tested apps had a median of 23 elements that obtain their label from a `contentDescription` (M : 48, SD : 79, R : 1–1,455). Within this population, *uninformative labels* still had low prevalence; 5,787 apps (97%) had exactly 0% of their tested elements with uninformative elements. Apps with 0% had a median of 23 elements with labels from a `contentDescription` tested (M : 47, SD : 77, R : 1–1,455). Only 133 apps (2%) had more than 10% of their `contentDescription`-dependent elements with uninformative labels.

6.2.4 Editable TextView with ContentDescription. We tested 2,919 apps with at least one editable TextView. The *editable TextView with contentDescription barrier* had a low prevalence. Apps had a median of 9 editable TextView elements tested (M : 20, SD : 33, R : 1–596). A total of 2,806 apps (96%) had 0%–10% of their editable TextViews with the barrier (Figure 8). The 2,800 apps (96%) with exactly 0% of their elements with this accessibility barrier had a median of 9 elements tested (M : 20, SD : 33, R : 1–596). There was a small spike in the right-most extreme of the distribution with 65 apps (2%) having 90%–100% of their *editable TextViews with contentDescriptions.*

6.3 Fully Overlapping Clickable Elements

We tested 8,886 apps with at least two clickable TalkBack-focuable elements for the *fully overlapping clickable elements* barrier. This accessibility barrier had a low prevalence across apps

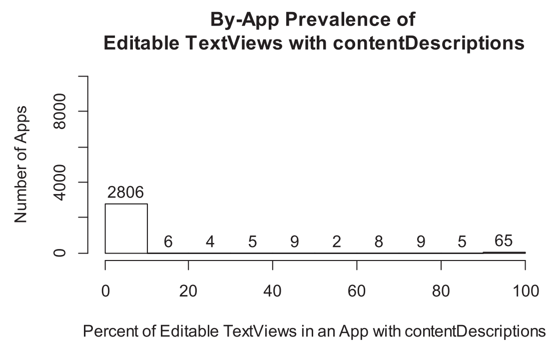

Fig. 8. Distribution of percent of editable TextViews per app that incorrectly had a `contentDescription`, out of 2,919 apps.

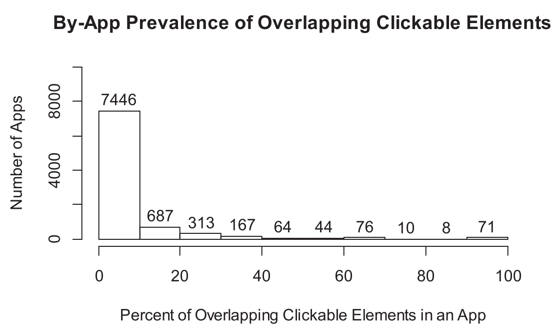

Fig. 9. Distribution of percent of fully overlapping clickable elements per app, out of 8,886 apps.

(Figure 9). There was a median of 110 elements tested per app (M : 200, SD : 297, R : 2–7,792). At the left end of the distribution, 7,446 apps (83.8%) had 0%–10% of their clickable elements fully overlapping; 6,007 apps (68%) had exactly 0%. Apps with exactly 0% of tested elements with the barrier had a median of 88 elements tested (M : 164, SD : 249, R : 2–6,711). The remaining apps tended to have a lower prevalence of this accessibility barrier; 1,231 apps (13.9%) had 10%–50% of their clickable elements fully overlapping.

6.4 Size-based Inaccessibility

The 9,650 apps tested for *size-based* accessibility barriers had a median of 98 elements tested (M: 188, SD: 290, R: 1–7,792). Figure 10 presents the distributions for the four types of the *size-based* accessibility barriers. The *too small in either* accessibility barrier captures elements that are undersized in their height, width, or both. We further break down the barriers by dimension into *too short only, too narrow only*, and *too small in both.* An element that is wide enough and tall enough is *big enough.* Apps had a notable prevalence of undersized elements, regardless of dimensions (i.e., *too small in either*); 6,875 apps (71%) had more than 10% of their elements *too small in either.* Decomposing size-based barriers by dimension, apps seemed most at risk of having elements that were *too short only*; 5,289 apps (55%) had more than 10% of their elements *too short only. Too small in both* errors were the next most prevalent; 2,753 apps (29%) had more than 10% of their elements *too small in both* dimensions. Apps appeared least likely to have elements that were *too narrow only* with only 706 apps (7%) having more than 10% of their elements with this accessibility barrier.

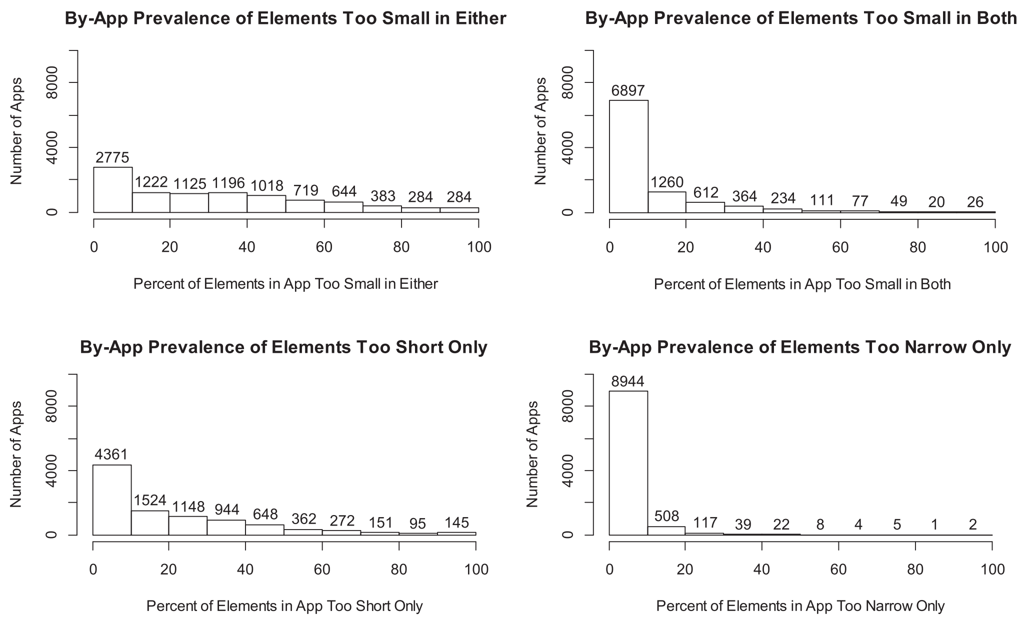

Fig. 10. Distributions of the percent of elements in each app with (top left) the size-based accessibility barrier *too small in either* and the dimension-based subcategories of: (top right) *too small in both*, (bottom left) *too short only*, and (bottom right) *too narrow only*. 9,650 apps were tested.

At the left end of the distribution, 1,925 apps (20%) had exactly 0% of their elements *too small in either*, with a median of 19 elements tested per app in that extreme ($M:$ 42, $SD:$ 83, $R:$ 1–1,853); 4,486 apps (46%) had exactly 0% of their elements *too small in both*, with a median of 41 elements tested per app ($M:$ 88, $SD:$ 139, $R:$ 1–1,853); 2,627 apps (27%) had exactly 0% of their elements *too short only*, with a median of 26 elements tested per app ($M:$ 60, $SD:$ 119, $R:$ 1–2,483); 6,901 apps (72%) had exactly 0% of their elements *too narrow only*, with a median of 66 elements tested per app in that extreme group ($M:$ 138, $SD:$ 227, $R:$ 1–7,792).

The number of apps within the mid-range of undersized elements (10%–90%) indicates there is within-app variation on element sizing. Section 8.3 explores how the context of use of different classes of elements may affect sizing and contribute to within-app sizing variation.

7 HIGH REUSE CLASSES

The previous section's by-app distributions contribute high-level insights into how frequently apps are affected by the accessibility barriers. We now explore the prevalence of accessibility barriers in different classes, collected across apps. Enhancing the accessibility of classes that are more susceptible to barriers can improve accessibility *across* apps. To focus on this breadth of impact, we analyzed the top five most reused classes for the two most prevalent accessibility barriers: *missing label* and *size-based barriers*. We measure *class reuse* as the number of apps that had at least one element of a given class.

7.1 Missing Label

The top five most reused classes of elements tested for *missing labels* are presented in Table 2. The list includes three layout or container classes (i.e., Linear Layout, Scroll View, and List View) with very low barrier prevalence. The low prevalence (1.0%) of *missing labels* in the Text View

Table 2. Prevalence of Missing Labels in the Top Five Most Reused Element Classes

Class	**# apps using element**	**# elements**	**# missing label**	**% missing label**
`android.widget.LinearLayout`	6,233	208,614	2,304	1.1%
`android.widget.TextView`	5,597	186,161	1,925	1.0%
`android.widget.ScrollView`	5,299	43,943	612	1.4%
`android.widget.ListView`	4,189	41,446	2,293	5.5%
`android.widget.ImageButton`	4,038	126,675	66,458	52.5%

All Elements That Were TalkBack-focusable Were Considered.

Table 3. Prevalence of Missing Labels in `contentDescription`-dependent Elements of Top Five Most Reused Classes

Class	**# apps using element**	**# elements**	**# missing label**	**% missing label**
`android.widget.ImageButton`	4,038	247,015	163,008	66.0%
`android.widget.ImageView`	2,976	635,228	580,975	91.5%
`android.support.v7.widget.AppCompatImageView`	1,845	157,827	133,346	84.5%
`android.support.v7.view.menu.ActionMenuItemView`	1,842	49,529	2,537	5.1%
`android.support.v7.widget.AppCompatImageButton`	1,725	108,049	73,175	67.7%

Elements that had a label but not a `contentDescription` were excluded.

text-based class is not surprising, since labels can be created directly from visible text. The only high prevalence among the top five most reused classes of elements tested for missing labels is the Image Button class at 53%. The low prevalence in the majority of these high reuse classes reflects the need for filters to capture the impact of missing labels. Layout and container elements can inherit labels from their contained children elements; Text View elements can obtain labels from the visible text.

Consistent with the by-app analyses of *missing labels* (Section 6.2.1), we applied a knowledge-based filter to exclude elements that can use labels automatically provided by visible text, focusing on elements that are directly dependent on a `contentDescription`. The top five most reused classes that are dependent on a `contentDescription` are primarily for images (Table 3). The `android.widget.ImageView` and `android.support.widget.AppCompatImageView` classes are nearly synonymous; `AppComptImageView` subclasses `ImageView` to support older versions of Android [11]. The same relationship of subclassing for legacy support applies to `android.widget.ImageButton` and `android.support.v7.widget.AppCompatImageButton` classes. These high reuse, image-based elements have high prevalence of *missing labels*. Approximately 67% of elements from both classes of Image Button were missing labels, and approximately 90% of tested elements of both Image View classes had the barrier. We explore missing labels in image-based buttons in more detail in Sections 8.2 and 8.3.

7.2 Size-based Inaccessibility

The top five most reused classes tested for *size-based* accessibility barriers included three layout classes (Tables 4 and 5). Of those, Linear Layout and Relative Layout had notable prevalence of size-based accessibility barriers, being *too small in either* for 27% and 29% of their uses. Considering

Table 4. The Number of Elements in Each of the Top Five Most Reused Classes with Size-based Accessibility Barriers

Class	# apps using element	# elements	# too small in either	# too small in both	# too short only	# too narrow only
`android.widget.LinearLayout`	4,342	150,631	40,636	1,898	36,479	2,259
`android.widget.ListView`	4,185	41,355	346	0	254	92
`android.widget.ImageButton`	4,030	124,605	49,631	30,424	16,124	3,083
`android.widget.Button`	3,912	129,054	49,321	16,060	30,022	3,239
`android.widget.RelativeLayout`	2,989	97,608	27,843	3,076	21,997	2,770

Table 5. The Percentage of Elements of Each of the Top Five Most Reused Classes with Size-based Accessibility Barriers

Class	# apps using element	# elements	% too small in either	% too small in both	% too short only	% too narrow only
`android.widget.LinearLayout`	4,342	150,631	27.0%	1.3%	24.2%	1.5
`android.widget.ListView`	4,185	41,355	0.8%	0%	0.6%	0.2%
`android.widget.ImageButton`	4,030	124,605	39.8%	24.4%	12.9%	2.5%
`android.widget.Button`	3,912	129,054	38.2%	12.4%	23.3%	2.5%
`android.widget.RelativeLayout`	2,989	97,608	28.5%	3.2%	22.5%	2.8%

the dimension-specific barrier breakdown, the Linear Layout and Relative Layout elements were predominantly *too short only*.

Both button classes appearing in the top five most reused classes (i.e., Button and Image Button) have high barrier prevalence with almost 40% of tested elements being *too small in either*. However, the distribution of the dimensions in which the elements were undersized differs. Image Buttons were about twice as likely to be *too small in both* (24%) as compared to *too short only* (13%). Buttons had the reverse distribution, being twice as likely to be *too short only* (23%) than *too small in both* (12%). We further explore size-based accessibility barriers in the Button and Image Button classes in Section 8.3.2.

8 CASE STUDIES

Our previous analyses by app and of specific filtered classes give insights into which accessibility barriers were most prevalent. We now explore *why* and *under what conditions* some apps and classes of elements might have been more susceptible to accessibility barriers than others.

To gain this insight, we investigated patterns in the usage and occurrence of accessibility barriers among similar classes of elements. Our case studies cover: (1) apps with the *few TalkBack-focusable elements* barrier, exploring their category and the tools used to create the apps;

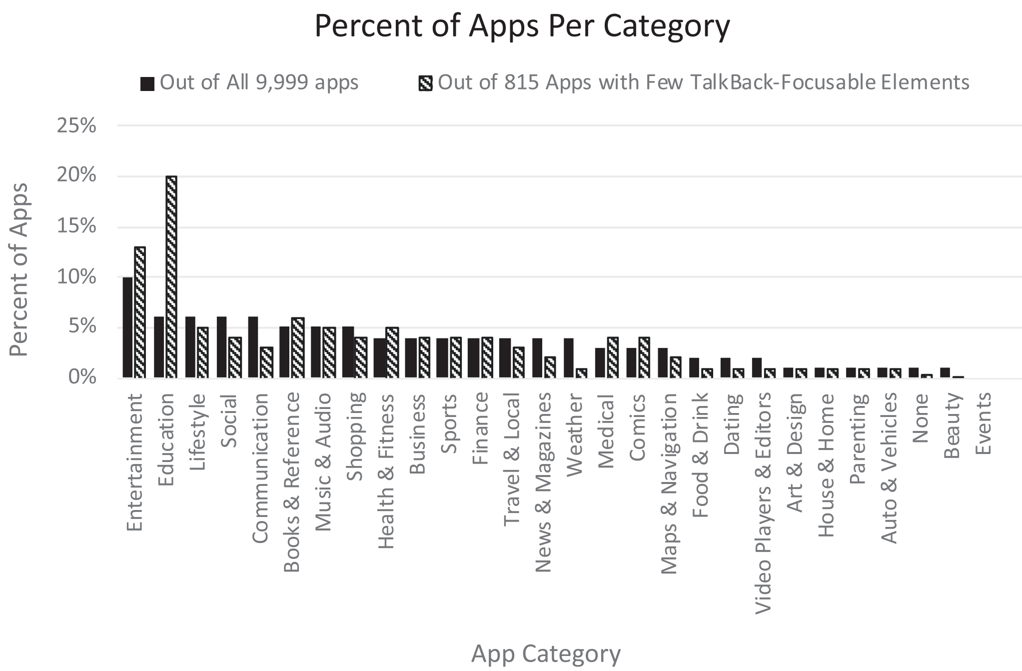

Fig. 11. The distribution of the categories of the 815 apps with the *few TalkBack-focusable elements* barrier and of the 9,999 apps. Education apps are disproportionately likely to have *few TalkBack-focusable elements.*

(2) labeling practices in image-based buttons; and (3) visual design patterns and the prevalence of size-based and label-based barriers in Buttons, Image Buttons, Checkboxes, Radio Buttons, and third-party plug-in elements. These case studies were chosen using our previously discussed knowledge-based filters (i.e., prior work, existing guidelines, and experience working with TalkBack) and metric-based filters (i.e., reuse and barrier-proneness).

Understanding these case studies can help inform efforts to improve these specific, impactful cases. The insights gained within and across these case studies may also extend to broader improvement efforts, such as more systematic problems in app development tools. Additionally, the techniques used to scope and assess these case studies can be applied to find other patterns of interest.

8.1 Apps with Few TalkBack-focusable Elements

We consider what categories of apps are most susceptible to the *few TalkBack-focusable elements* barrier. The app category is defined by the app developer and found in the app metadata of our dataset.

The distribution of percent of apps per category overall and of apps with the *few TalkBack-focusable elements* barrier per category is presented in Figure 11. Of the population of 815 apps with *few TalkBack-focusable elements*, 162 apps (19%) are in the Education category; this is disproportionate to the 645 Education apps (7%) in the overall population of 9,999 apps tested. However, Communication apps and Weather apps seem to be underrepresented in apps with *few TalkBack-focusable elements.*

We investigated potential contributing factors by examining what classes of elements frequently appeared in apps with the *few TalkBack-focusable elements* barrier. Two notable factors were: (1) the use of multi-platform and hybrid web/native app creation tools (e.g., Adobe Air), and (2) the use of

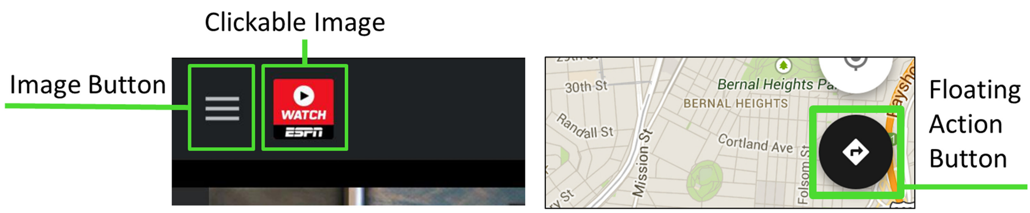

Fig. 12. Examples of elements from the (left) Image Button, (center) Clickable Image, and (right) Floating Action Button classes. In practice, Image Buttons and Clickable Images are used interchangeably. Floating Action Buttons are primarily icon-based and denote the most important action on the screen.

game engines (e.g., Unity). We determined tool and plug-in usage by the specific class names within the view hierarchies. For example, the class `com.unity3d.player.UnityPlayer` indicated use of the Unity game engine [40]. Appendix D presents the frequency at which some of the tool classes that we identified in apps with *few TalkBack-focusable elements* occur in apps with and without the barrier. The table presents the tool associated with the classes, the number of elements of each class captured, the number of apps each class appears in, and the frequency of the barrier occurring in apps with an average of exactly zero and (0,1] TalkBack-focusable elements per screen.

Multi-platform and hybrid web/native Android app plug-ins allow developers to create apps that re-use code across desktop, web, and mobile implementations. In our examination of apps with *few TalkBack-focusable elements*, we found common multi-platform plug-ins including Apache Cordova [10], Adobe Air [1], and Crosswalk [39]. These plug-ins support app creators in authoring apps using HTML, CSS, Javascript, and Flash, which the tools then implement for Android using an underlying alternative to Android's WebView. These classes of elements often occurred in apps with *few TalkBack-focusable elements*. As detailed in Appendix D, 64% of identified Adobe Air elements, 69% of Crosswalk elements, and 75% of Apache Cordova elements occurred in apps with *few TalkBack-focusable elements*.

We also identified game engine classes in apps with *few TalkBack-focusable elements*, including Unity [40] and Cocos2d-x [16]. Game engines can be used to create cross-platform experiences in a manner not unlike hybrid web/native tools. Game engines also enable more expressive styles and functionality (e.g., placing a greater emphasis on 3D graphics and animation) as compared to general app creation tools like Android Studio. Game engine classes were highly associated with the barrier; 53% of identified Cocos2d-x elements and 100% of identified Unity elements were found in apps with *few TalkBack-focusable elements* (see Appendix D).

8.2 Image-based Buttons

Based on the established impact of missing labels (e.g., as noted in prior work [14, 29, 31, 35] and in Android development guidelines [24]), we assessed image-based buttons for *label-based* accessibility barriers. We focused on three classes of image-based buttons: Clickable Images, Image Buttons, and Floating Action Buttons (FABs). We defined Clickable Images as elements from the `android.widget.ImageView` class that had the `clickable` attribute set to `True` in the view hierarchy. Image Buttons were from the class `android.widget.Image Button`, which is a subclass the `ImageView` class. FABs were from the class `android.support.design.widget.FloatingActionButton`, a subclass the `ImageButton` class. All classes needed a `contentDescription` if they did not have an inherited label or a visible-text-based label. Figure 12 shows examples of each type of button.

The reuse of each class is presented in Table 6. Clickable Images and Image Buttons were released in Android 1.0 in September 2008 [6, 7]. The lower number of apps using FABs may be due to the

Table 6. The Number of Apps from Our Population of 9,999 That Have Image-based Buttons of Each Class

Class	# apps	% of app population
Clickable Image	2,976	29.8%
Image Button	4,038	40.4%
FAB	590	5.9%

Table 7. Frequency of Image-Based Buttons Missing Labels

Class	# elements	# missing label	% missing label
Clickable Image	115,667	99,825	86.3%
Image Button	126,675	66,458	52.5%
FAB	6,389	5,932	92.8%

relatively recent introduction of the FAB class in Android 22.2.0 in May 2015 [5]. The lower total number of FAB elements may also reflect guidelines suggesting using at most one FAB per screen.

We chose Clickable Images and Image Buttons because they were mentioned in Android's element labeling guides, were in the top five most reused classes that depended on `contentDescriptions` (Section 7.1), and had a high prevalence of *missing labels*. The FAB class was included to explore how *label-based* barriers affected a less reused, more recently released class. We tested image-based button for *missing labels, duplicate labels*, and *uninformative labels*. See our prior work [34] for a more extensive analysis of this case study.

8.2.1 Missing Label. All three of the tested image-based button classes had a high prevalence of the *missing label* barrier, as shown in Table 7. The reuse, number of elements with the barrier, and prevalence varied (Tables 6 and 7) despite their technical similarities. Each measure provides a different lens on the impact of the barrier over the population. Image Buttons were found in the greatest number of apps with 40% of apps having at least one Image Button. This high class reuse indicates a high likelihood that people would encounter Image Buttons in many apps and about half of those elements were missing labels. While Clickable Images were found in fewer apps, the class had the greatest number of elements with the *missing label* barrier; this class produced the most problematic elements in the population that someone may encounter. FABs were fewer in number of apps using and number of elements missing labels but had the highest prevalence at 93% of FABs missing a label. Thus, although FABs are less frequently encountered, when they are used, they are extremely likely to be unlabeled. The differences in prevalence of missing labels in the classes despite their technical similarities and release dates suggest other factors impact the likelihood of missing labels. Potential factors include Android guidelines and testing suites, as they can be used in the creation of many applications. Our prior work [34] explored how these tools addressed or failed to address the need to label image-based buttons.

8.2.2 Duplicate Labels. We examined prevalence of the *duplicate label* barrier in each of the three classes of image-based buttons, displayed in Table 8. We tested screens with at least two image-based buttons that had been labeled, excluding buttons that were missing labels. Clickable Images had the highest prevalence of and the most elements with duplicate labels; of the 15,531 labeled Clickable Image elements tested, 7,250 (47%) had a *duplicate label*. Image Buttons and FABs had lower prevalence and number of labeled image-based buttons; 4,536 out of 55,617 (8%) labeled Image Buttons elements and 36 out of 454 (8%) labeled FABs tested had a *duplicate label*.

Table 8. The Number of Labeled Image-based Buttons with a Duplicate Label and the Percent out of All Tested Elements Are Presented Per Class in the *# duplicate Label* and *% Duplicate Label (of labeled)* Columns

Class	# labeled elements	# duplicate label	% duplicate label (of labeled)
Clickable Image	15,531	7,250	46.7%
Image Button	55,617	4,536	8.2%
FAB	454	36	7.9%

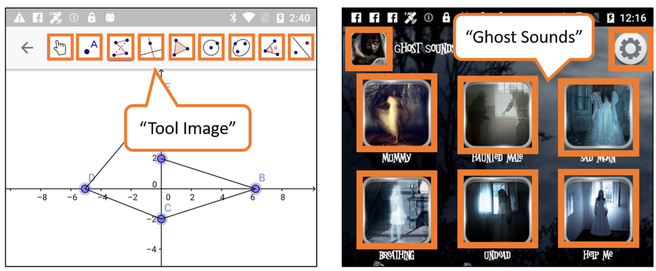

Fig. 13. Two example app interfaces with *duplicate labels* on image-based buttons. Left: The Clickable Image Buttons for drawing different types of figures in a graphing app are all labeled "Tool Image." Right: In the Ghost Sounds app, all of the Clickable Image Buttons for playing different ghost sounds as well as the settings and home button are labeled "Ghost Sounds."

Table 9. The Total Number of Labeled Image-based Buttons Tested (*# Labeled Elements*), the Number of Labeled Image-based Buttons with an Uninformative Label (*# Uninformative Label*), and the Percent of All Tested Elements with the Barrier (*% Uninformative Label (of labeled)*) Are Presented Per Class

Class	# labeled elements	# uninformative label	% uninformative label (of labeled)
Clickable Image	18,372	1,802	9.8%
Image Button	62,306	1,278	2.1%
FAB	524	0	0.0%

Uninformative labels are much less prevalent compared to missing and duplicate labels.

Through manual inspection of apps with the barrier, we noted apps with a large proportion of image-based buttons on a given screen whose duplicate labels were general text, such as "Tool Image" (Figure 13). Possible contributors to this pattern are copy-and-paste errors or a lack of understanding of the function of a `contentDescription`.

Not all screens with multiple elements with the same label are erroneous. For example, our dataset contained a music app with a list of songs in which all of the artists were "Unknown." In this case, labeling all artist information buttons with "Unknown" was appropriate. As this example illustrates, quite nuanced methods are needed to distinguish between valid and invalid duplicate labels.

8.2.3 Uninformative Label. For analyzing the *uninformative label* barrier, we only considered labeled image-based buttons. Out of 3,396 total apps with at least one labeled image-based button, 3,342 apps (98.4%) have fewer than 10% of their image-based buttons with *uninformative labels*. The low prevalence of the *uninformative label* barrier is mirrored in the by-class analysis presented in Table 9.

Compared to *missing labels, uninformative labels* were not as prevalent. This comparison suggests that if an image-based button is labeled, it tends not to be a blatantly uninformative label. Further work analyzing labels is needed to determine the quality of labels at a more nuanced level

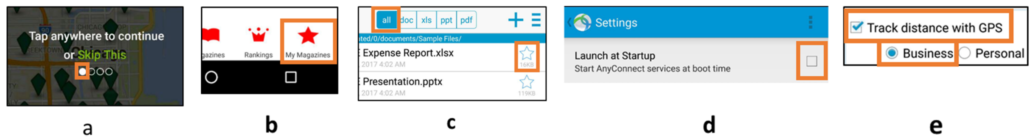

Fig. 14. Example uses of Radio Buttons and Checkboxes. (a) Radio Buttons used as pagination indicators, (b) Radio Buttons used as tabs, (c) a Radio Button tab and Checkbox favorite button, (d) a settings Checkbox, (e) a settings Checkbox and Radio Button option selection.

(e.g., the "Tool Image" label discussed as a duplicate label in Figure 13 is also an uninformative label, but not captured by our analysis).

8.3 Design and Usage Patterns in Sizing and Labeling

As a complement to our quantitative analyses of accessibility barriers and specific classes of elements, we examined patterns in the visual design and uses of classes of elements to gain insight into how design may influence accessibility. We focus on *missing labels* and *size-based* accessibility barriers in three groups of interactive element classes: high reuse buttons; barrier-prone moderate reuse Checkboxes and Radio Buttons; and barrier-prone moderate reuse third-party social media plug-ins for Facebook and Google+. Sampling from these different groups can yield insights into how patterns of accessibility in element classes may be influenced by barrier proneness, reuse, or origin from the Android API or a third party. To find patterns in usage, the first author visually examined elements from each class cropped from the full screenshots.

The high reuse Button and Image Button classes had notable prevalence of the *missing label* and *size-based* accessibility barriers (see Tables 4, 5, and 10). The Button class is intended to be used for text-based buttons, while the Image Button class is optimized for image-based content (e.g., the cog icon button commonly used for "Settings"). In practice, both button classes were used to create image-based buttons. Using an Image Button versus a Button for image-based button content may reflect differences in developer experience or education, which may be another factor impacting app accessibility.

Checkboxes and Radio Buttons were *barrier-prone moderate reuse* classes, identified among the highest in prevalence for *size-based* accessibility barriers and in the top 1% of reused classes (see Appendix B). Checkboxes and Radio Buttons were often used for selecting between options, choosing a setting, favoriting, implementing a tabbing interface, or implementing paging indicators (Figure 14).

The Facebook Login and Google+ element classes are examples of custom elements from third-party companies released for app creators to integrate with their services. The Facebook Login and Google+ elements were identified through metric-based filtering; both classes appeared on the list of *barrier-prone moderate reuse* classes for *size-based* accessibility barriers (see Appendix B). We noted that classes `ztu` and `yei` were obfuscated class names for the Google+ element shown in Figure 19.

Differences in occurrences of accessibility barriers between these element groups could yield insights into how reuse or third-party implementation may affect element accessibility. Similarities within the presentation of accessibility barriers in all of these classes could indicate more universal implementation decisions that affect app accessibility across classes.

8.3.1 Missing Labels in Design and Usage Patterns. The social network third-party plug-in buttons for Facebook Login and Google+ had very low prevalence of *missing labels* (Table 10). Only four of the Facebook Login buttons (1%) and zero Google+ buttons had the *missing label* barrier.

Table 10. Prevalence of *Missing Label* Barriers in Image Buttons, Buttons, Checkboxes, Radio Buttons, Google+, and Facebook Login elements

Class	# apps using class	# elements	# missing label	% missing label
`android.widget.ImageButton`	4,038	126,675	66,458	52.5%
`android.widget.Button`	3,916	130,195	34,140	26.2%
`android.widget.CheckBox`	711	10,636	6,907	64.9%
`android.widget.RadioButton`	402	11,315	3,024	26.7%
Google+ (`ztu` and `yei` combined)	90	687	0	0.0%
`com.facebook.login.widget.LoginButton`	133	309	4	1.3%

Moreover, the labels on these elements were consistent across apps and of high quality (see Appendix E for full label list). There were only four unique labels for Google+ buttons, two phrasings of "Touch to add to plus one" and two labels indicating the number of +1's an app had. Across all 309 Facebook Login buttons, there were 28 unique labels that were slight variations of "sign in with Facebook" in different languages. The consistency of labels across apps suggests that these labels may have been added by the third-party element creators and rarely modified by individual app developers that used that social media plug-in. These components exemplify how the accessibility of a well-created element that is reused by many apps can promote greater app accessibility across the population. Due to the lack of the *missing label* barrier in these third-party social plug-ins, we focus on Buttons, Image Buttons, Checkboxes, and Radio Buttons for the remainder of the section.

The Android API Button, Image Button, Checkbox, and Radio Button classes had a notable prevalence of the *missing label* barrier (Table 10). The Image Button class had 52% of tested elements *missing labels*, and the Button class had 26% of elements *missing labels*. Checkboxes had the highest prevalence of missing labels out of these classes at 64%, and Radio Buttons had 26%.

Stand-alone icons (i.e., without accompanying visible text) appeared in Button, Image Button, Radio Button, and Checkbox elements that had the *missing label* barrier. Similar icons appeared on elements from the button classes that were *labeled* and *missing labels* such as the back arrow and the menu icon as displayed in Figure 15. This similarity suggests the icon itself may not be a primary factor affecting whether a button class will be labeled.

The traditional circular Radio Button and square Checkbox icons occurred in elements of those classes that were *labeled* and *missing labels* (Figure 15). However, unlike the button classes, there was a difference in visual design in labeled and unlabeled Checkbox and Radio Button. When the circle or square icon was stand-alone, the element tended to be unlabeled. When a circle or square icon was grouped with a visible text-based label, it did not need a `contentDescription` and tended to be *labeled*. There is a similar trend in *labeled* Buttons having visible text grouped with an icon (e.g., the "I Agree" Button in Figure 15, bottom left).

Image-based text visually looks like text but is implemented only in pixels. Some elements from the button classes, Radio Buttons, and Checkboxes that were *missing labels* used image-based text. For example, Figure 15 shows image-based text on the "Translat" and "send" Buttons, the "SHOW" Image Button, and the "Save" Radio Button. Image-based text cannot be automatically read as a label by TalkBack and instead needs a `contentDescription`. Figure 15 shows an example Image Button of a keypad "8" implemented as an image-based text element that was appropriately labeled using the `contentDescription`.

8.3.2 Size-based Inaccessibility in Design and Usage Patterns. For the *size-based* accessibility barriers, both the Button and Image Button classes had almost 40% of tested elements *too small*

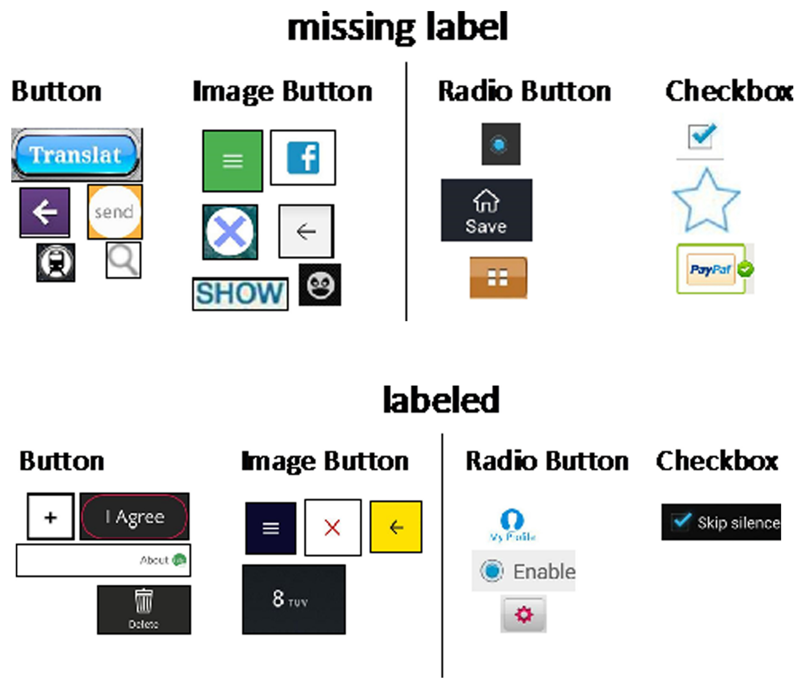

Fig. 15. Example Buttons, Image Buttons, Radio Buttons, and Checkboxes that were: (top) *missing labels* or (bottom) *labeled*. Most unlabeled elements use only icons or image-based text. Created labeled element. Labeled elements with image-based text or no text had `contentDescriptions`.

Table 11. Prevalence of size-based errors in Checkbox and Radio Button classes.

Class	# elements	# apps using class	% too small in either	% too small in both	% too short only	% too narrow only
`android.widget.CheckBox`	10,176	686	82.1%	46.8%	27.6%	7.7%
`android.widget.RadioButton`	11,124	393	62.0%	21.2%	40.5%	0.3%

in either dimension (Tables 4 and 5). Image Buttons were more likely to be *too small in both* (24%) than *too short only* (13%). Button elements had the reverse trend, being more likely to be *too short only* (23%) than *too small in both* (12%).

Sized-based barriers were more prevalent in Checkboxes (82%) and Radio Button (62%) than the other button classes, as presented in Table 11. Checkboxes were more likely to be *too small in both* (47%) than *too short only* (28%). Radio Buttons were the opposite, more frequently being *too short only* (41%) than *too small in both* (21%). Neither element class was prone to having elements that were *too narrow only*.

Interface elements from the Button, Image Button, Radio Button, and Checkbox classes with the same *size-based* accessibility barriers had similar visual design and use patterns (Figures 17 and 18). Interface elements of these classes that were *too small in both* tended to be composed of a tightly cropped icon. A few *too small in both* elements had short visible text (e.g., a single word, a number). Among elements that were *too short only*, Button, Radio Button, and Checkbox elements had longer

Fig. 16. A Checkbox element that, stand-alone, is *too small in both* dimensions and *missing a label* if without a `contentDescription`. If the clickable region includes the text label, then the element is *too short only* and *labeled*. Expanding the region to the full width and height of the row layout makes the element *big enough*. No modifications affect the visual appearance.

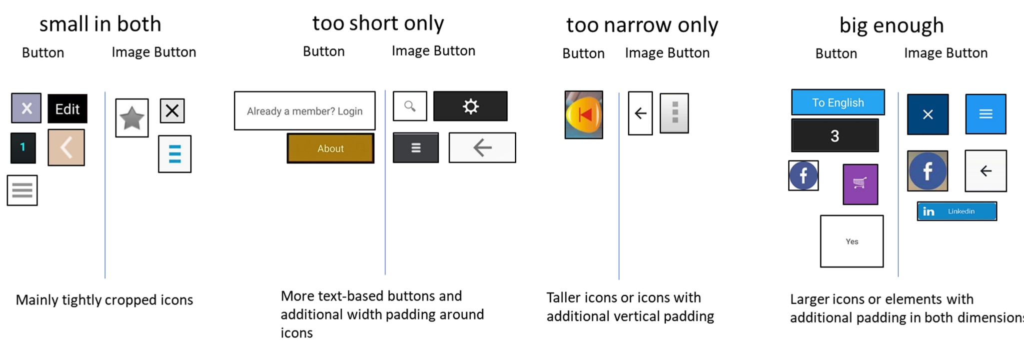

Fig. 17. Example Buttons and Image Buttons that had size-based accessibility barriers in different dimensions: *too small in both, too short only, too narrow only*, or *big enough*.

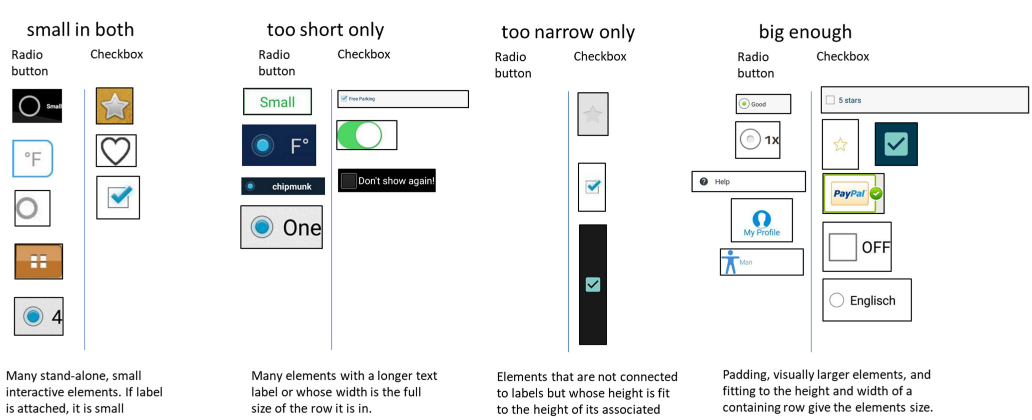

Fig. 18. Example Radio Buttons and Checkboxes that were *too small in both, too short only, too narrow only*, or *big* enough. What components are included in the clickable element affects the likelihood the element is *big enough*.

text-based content while Image Buttons had additional horizontal padding around icons. Styles that gave *too narrow only* elements sufficient height included using taller icons or adding vertical padding around icons. We did not note any text-based elements that were *too narrow only*. Vertical and horizontal padding was a key component of Button and Image Button classes that were *big enough*. Checkboxes and Radio Buttons often achieved this padding by matching the height or width of a container element. Figure 16 illustrates this technique and Figure 18 shows examples

Table 12. Prevalence of Size-based Errors in Google+ (*Yei* and *ztu* Classes) and Facebook Login Elements

Class	**# elements**	**# apps using class**	**% too small in either**	**% too small in both**	**% too short only**	**% too narrow only**
`yei`	318	37	100%	62.6%	37.4%	0%
`ztu`	369	56	98.9%	26.8%	72.1%	0%
combined	687	90	99.4%	43.2%	56.0%	0%
`com.facebook.login.widget.LoginButton`	306	132	82.7%	0.3%	81.7%	0.7%

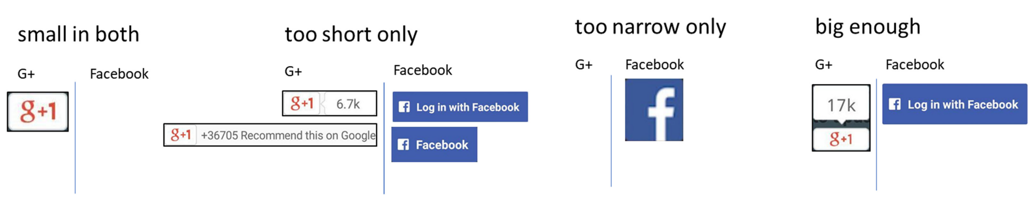

Fig. 19. Sample Google+ (*g*+1) and Facebook Login elements that had the size-based errors of being *too small in both, too short only, too narrow only*, or *big enough*, based on Android Accessibility Guidelines [9]. The comparison highlights how design choices made by third-party providers can impact many apps.

from our “such as the “Help” Radio Button and “5 stars” Checkboxes with clickable regions the height and width of their menu row”.

Facebook Login and Google+ elements also had high prevalence of the *size-based* accessibility barriers, as presented in Table 12. Google+ elements were *too small in both* when the element was only the g+1 logo became wider and were too short only when the logo also had a status indicator placed next to it (e.g., showing the number of likes). In one case, the status indicator was placed above the logo, which resulted in an element that was *big enough*. Figure 19 illustrates these differences.

9 DISCUSSION

We return to Ross et al.’s [33] epidemiology-inspired framework to discuss our findings. Accessibility barriers are posed as *diseases* in the *population* of Android apps. The interconnected population of apps is posed within a rich ecosystem of *factors*. Figure 20 from Ross et al.’s framework illustrates example factors. This ecosystem approach can guide the identification of potential factors that have widespread impact on app accessibility.

We focus our discussion first on the highly severe *few TalkBack-focusable elements* barrier and then on the highly prevalent *label-based* and *size-based* barriers. We address an epidemiology-inspired objective of *determining the extent of a disease in a population* by revisiting some key prevalence findings and their connection to Yan and Ramachandran’s large-scale app accessibility analyses [41]. Determining the extent of an *inaccessibility disease* (i.e., an accessibility barrier) is a powerful tool for educating stakeholders on the frequency of accessibility barriers, for planning how resources should be allocated to repair apps, and for tracking progress on app accessibility over time.

We then discuss our case studies through the *multi-factor environment* lens from the epidemiology-inspired framework. Our findings include *intrinsic factors* (i.e., closely related to a single app) such as visual design and code reuse as well as more *extrinsic factors* (i.e., that have a higher-level impact on many apps) such as app creation tools and app category. This ecosystem view situates our prevalence measures, accessibility measures from prior work, and our case

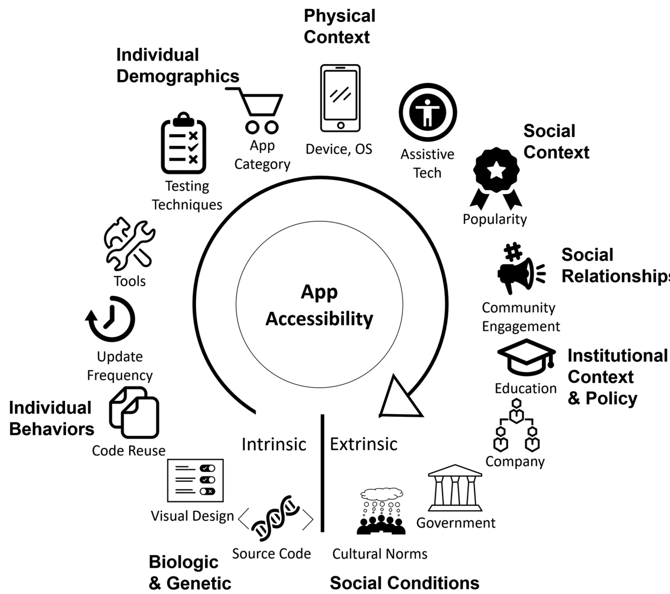

Fig. 20. An ecosystem of factors that can impact app accessibility at a population level. Intrinsic factors are closely associated with a single app; extrinsic factors have more widespread impact. Diagram re-published with permission from Ross et al. [33].

studies into a richer view of how app accessibility is influenced at scale. The richness and interconnectedness of the app ecosystem then guides our discussion of potential *treatment*s (i.e., approaches for improving app accessibility).

9.1 Few TalkBack-focusable Elements

Our prevalence metric does not take into account barrier *lethality*; how likely it is that a barrier will cause a person to abandon the app (i.e., a "death of usage" in epidemiology-framework terms). The *few TalkBack-focusable elements disease* is an example of a barrier that had low prevalence but has a high severity. Thus, even the 8% prevalence rate we observed is highly problematic.

Yan and Ramachandran [41] found a related focus-based accessibility barrier to be prominent in their analyses. Their focus-based error tested whether any element on a screen from a predefined set of classes was not focusable. This error differs from our *few TalkBack-focusable elements*, thus the prevalence values reported by Yan and Ramachandran are not directly comparable to our measures. However, Yan and Ramachandran's findings support the importance of focus-related barriers. They found that 53.4% of their measured violations were focus errors. Further analyses could explore *environmental factors* that impact Yan and Ramachandran's definition of focus errors, perhaps following the techniques we applied to the *few TalkBack-focusable elements* barrier.

We identified third-party hybrid app creation tools and game engines as *environmental risk factors* that were frequently used in apps with the *few TalkBack-focusable elements* barrier. These results suggest further work needs to be done to understand why developers are using third-party hybrid tools and game engines for app production. Moreover, we identified Education apps as a *high-risk population*; Education apps were disproportionally likely to have this barrier. Future work should also explore what aspects of Education app development makes those apps more prone to this barrier. We use the epidemiology-inspired multi-factor framework to consider possible factors that impact the choice of app creation tools.

One factor could be the tools themselves offer functionalities that more mainstream Android API widgets do not, such as game-like interaction mechanics, non-traditional layouts, and portability between devices. In such a case, two interventions that might reduce the prevalence of the *few TalkBack-focusable elements* inaccessibility disease include: (1) supporting third parties in creating tools that better integrate with Android's accessibility infrastructure or (2) enhancing Android tools and APIs to address underlying needs of these types of app creators while maintaining the accessibility of resulting apps.

Another possibility is that current tools that create more accessible apps *do* meet the needs of developers using game engines and hybrid app tools, but developers are nonetheless choosing hybrid and game engine tools that create less accessible apps. Factors impacting this decision could include a lack of awareness of the more traditional Android tools and the impact on app accessibility that tool choice has; game engines or hybrid tools could be used in popular tutorials; or a single company that internally chose a set of tools for all their developers could be responsible for a large majority of the apps we tested with this barrier. In these cases, possible treatments include (1) increasing the visibility of existing tools that meet design needs and accessibility needs for apps, (2) increasing awareness around accessibility and how some hybrid app and game engine tools do not support accessibility, (3) providing tutorials for using tools that create more accessible apps in hybrid app or game-engine styles, or (4) working with prominent companies that use hybrid development or game engines to shift their practices. Further work is needed to identify which factors play a large role in hybrid tool and game engine usage and thus guide which *treatments* are most promising.

9.2 Label-based Inaccessibility

In our analysis of label-based inaccessibility, we focus on four *determinants,* or types, of label-based accessibility barriers. Considering *missing label, duplicate label, uninformative label,* and *editable TextView with contentDescription* barriers as four manifestations of a label-based *inaccessibility disease* (i.e., an accessibility barrier that affects what label a screen reader will announce) emphasizes the connection between these barriers. For example, approaches for reducing *missing labels* may increase the number of *uninformative labels* if not implemented mindfully. Additionally, understanding when one type of label-based accessibility barrier occurs can help us understand and treat other label-based barriers.

Duplicate label, uninformative label, and *editable TextView with contentDescription* accessibility barriers had much lower prevalence relative to the *missing label* barrier. This trend suggests that if an element is labeled, that label is often an attempt at a meaningful label. Yan and Ramachandran [41] similarly found labeling problems a prominent barrier; 19.6% of their violations by rule and 77.8% of their potential violations were from their description error. Although differences in their analysis approach and reported statistics do not allow direct comparison to our results, both their and our analyses highlight and measure how widespread and frequent the *missing label* disease is. As noted in Section 6.2.3, more robust methods are needed to determine if the quality of the labels passes more than the most basic thresholds for which we tested. The lower prevalence of poor labeling suggests efforts should focus first on getting elements labeled at all, rather than trying to identify what labels are of low quality.

Out of all barriers, the most prevalent barrier was the *missing label* inaccessibility disease on image-based buttons (Section 9.2) and on `contentDescription`-dependent elements (Section 6.2.1). Yan and Ramachandran [41] similarly identified elements that had Description errors primarily came from the Button, Image View, and View classes.

Applying the app ecosystem lens, we looked for patterns of labeling across apps. These patterns can be indicative of more extrinsic factors that impact labeling practices at scale. Viewing these patterns in context of their population-level implications can guide the development of interventions. In our case study, we compared elements that were *missing labels* to elements of the same classes that were *labeled.* One pattern that emerged was the use of common icons (e.g., three-line hamburger menu icon, settings cog, search magnifying glass) in Button and Image Button elements that were both *missing labels* and *labeled.* This use of standard icons presents an opportunity to automatically label elements using labels from elements with similar icons, as explored in the web [25]. Another labeling pattern that occurred in Buttons, Image Buttons, Radio Buttons, and Checkboxes with the *missing label* inaccessibility disease was the use of image-based text. Using automated text recognition to construct a `contentDescription` for the element or converting it into a text-based element could remedy the missing label.

We use the epidemiology-inspired framework to ideate on how automated labeling, like the techniques discussed above, could be applied in practice. The epidemiology-inspired framework suggests considering beyond the effect individual app creators have on their own element to population-level factors. These considerations surface opportunities throughout an app's lifecycle to apply the idea of automated labeling. One approach could apply automated labeling as a *preventative treatment* that occurs before an app is released by integrating labeling into app developer creation or testing tools. Another approach could have a service that performs a *therapeutic treatment* by labeling elements in released apps (e.g., using techniques as developed by Zhang et al. [42]). Further work is needed to assess which approach or set of approaches is most effective in which conditions, but the multi-factor lens also highlights that a combination of approaches could be deployed to effect change in the population over time.

9.3 Size-based Inaccessibility

Size-based barriers were among the more prevalent barriers in our app population. Yan and Ramachandran [41] similarly identified element size as a prominent problem; element size errors accounted for 17% of warnings. We ideate on why and under what circumstances developers do not make their elements large enough to meet accessibility standards (i.e., what are the impactful *environmental factor*s).

Element sizing can directly affect an app's visual appearance and interaction experience. This is contrasted with more "invisible" accessibility-specific practices like labeling (i.e., these accessibility practices do not affect the visual appearance of an app). The more direct connection of element sizing to app design and development practices affects how various factors in the ecosystem must be acknowledged or changed to remedy undersized elements. For example, instead of needing app creators to add information to their apps, such as labels, it may be necessary to encourage them to make visible changes to their designs or restructure existing screen hierarchies.

Further work is needed to understand the most prominent forces that dictate element sizing in apps. If visual design *is* a main concern, then approaches that minimally disrupt visual design could be very impactful. Our analysis of Checkboxes and Radio Buttons revealed one promising technique of grouping those elements with visible labels or their container elements (e.g., a menu row). Figure 16 shows an example of these approaches. Grouping an icon-only element with its visible label has the added benefit of giving the element an automatic and likely meaningful label. Additional work is needed to explore how frequently these techniques can be applied in practice and what other barriers developers encounter in creating larger elements. After more detailed explorations are performed, large-scale analyses can be filtered using new knowledge-based filters and potentially provide inspiration for repair suggestions.

The epidemiology-inspired perspective drove us to think beyond the effect that individual app creators have on their own element size. We considered the population-wide impact of popular third-party plug-ins. The Google+ and Facebook Login plug-in elements had notable prevalence of size-based errors. The consistency in visual design of these elements across apps suggests that third party plug-in creators are an important, more *extrinsic environmental factor* that influences how developers size plug-in elements. Potential influence may come from documentation and example code from the third parties that give a default size. Although individual app developers can resize plug-in elements (e.g., padding the elements), developers may trust that documentation from well-known third-party companies give the best implementation strategy for that element. One approach to enhancing plug-in sizes would be for third parties to change documentation or default sizing for their elements. Another technique, should the third party be unable or unwilling to make those changes, would be to inform developers using the plug-ins about the likelihood a *size-based inaccessibility disease* (i.e., the barrier) would be *transmitted* to their app through their use of the plug-in.

10 CONCLUSION

As articulated through Ross et al.'s epidemiology-inspired framework, understanding the extent of accessibility barriers in the app population can help identify breakdowns in the development process and inform enhancement efforts. Driven by that framework, we performed large-scale analyses to characterize the state of app accessibility, testing 9,999 apps for seven accessibility barriers: *few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextViews with contentDescriptions*, *fully overlapping clickable elements,* and *undersized elements.*

Our analyses revealed *missing labels* and *undersized elements* were particularly prevalent across apps. Future analyses can use these results to: (1) track prevalence of inaccessibility diseases over time and (2) evaluate intervention efficacy. We also explored case studies of specific types of elements that met or failed to meet accessibility guidelines. Understanding patterns of meeting or failing accessibility standards can inform interventions that support app creators in addressing the accessibility of individual apps. Potential solutions can leverage existing practices identified through large-scale analyses (e.g., using element grouping to address size and labeling barriers without impacting the visual design). Patterns in large-scale accessibility data can also help surface more extrinsic factors, which exist outside of any individual app and have an impact across the app population. Example extrinsic factor improvements include game engines that better support accessible development. The epidemiology-inspired framework allowed us to situate our prevalence analyses as a product of the rich multi-factor environment apps exist within. The framework further motivated our case studies on particular factors of interest, such as the role of third parties in plug-in element sizes. Finally, the framework inspired potential avenues for future work. Our overall work characterizes the current state of app accessibility, presents insights into improving the app accessibility ecosystem, and demonstrates analysis techniques that can be applied in further assessments.

REFERENCES

[1] Adobe. Adobe AIR. Retrieved on 17 April, 2019 from https://get.adobe.com/air.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR'16)*. 468–471. DOI:http://doi.org/10.1145/2901739.2903508

[3] Android Open Source. Making Applications Accessible. Retrieved on 17 April, 2019 from http://developer.android.com/guide/topics/ui/accessibility/apps.html.

[4] Android Open Source Project. Accessibility Developer Checklist. Retrieved on 17 April, 2019 from http://developer.android.com/guide/topics/ui/accessibility/checklist.html#requirements.

[5] "Android Open Source Project." FloatingActionButtonBasic | Android Developers. Retrieved on 17 April, 2019 from https://developer.android.com/samples/FloatingActionButtonBasic/res/layout/fab_layout.html.

[6] "Android Open Source Project." ImageButton | Android Developers. Retrieved on 17 April, 2019 from https://developer.android.com/reference/android/widget/ImageButton.html.
[7] "Android Open Source Project." ImageView | Android Developers. Retrieved on 17 April, 2019 from https://developer.android.com/reference/android/widget/ImageView.html.
[8] Android Open Source Project. Improve Your Code with Lint. Retrieved on 17 April, 2019 from https://developer.android.com/studio/write/lint.html.
[9] "Android Open Source Project." Making Apps More Accessible | Android Developers. Retrieved on 17 April, 2019 from https://developer.android.com/guide/topics/ui/accessibility/apps.html.
[10] Apache Cordova. Retrieved on 3 April, 2019 from https://cordova.apache.org/.
[11] AppCompatImageView | Android Developers. Retrieved on 30 April, 2019 from https://developer.android.com/reference/android/support/v7/widget/AppCompatImageView.
[12] Apple Accessibility Scanner. Retrieved on 17 April, 2018 from https://developer.apple.com/library/content/documentation/Accsssibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html.
[13] Apple Inc. 2012. Accessibility Programming Guide for iOS. Retrieved on 14 July, 2018 from https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/iPhoneAccessibility/Introduction/Introduction.html.
[14] 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 Computer Systems (IHC'16)*. DOI: http://doi.org/10.1145/3033701.3033718
[15] Raphael Clegg-Vinell, Christopher Bailey, and Voula Gkatzidou. 2014. Investigating the Appropriateness and Relevance of Mobile Web Accessibility Guidelines. In *Proceedings of the International Cross-disciplinary Conference on Web Accessibility (W4A'14)*. 1–4. DOI: http://doi.org/10.1145/2596695.2596717
[16] Cocos2d-x - World's #1 Open-Source Game Development Platform. Retrieved on 30 April, 2019 from https://cocos2d-x.org.
[17] Michael Cooper, Peter Korn, Andi Snow-Weaver, Gregg Vanderheiden, Loïc Martínez Normand, and Mike Pluke. 2013. *Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT)*. Retrieved from http://www.w3.org/TR/wcag2ict/.
[18] "Data Driven Design Group." Rico: A Mobile App Dataset of Building Data-Driven Design Applications. Retrieved on 17 April, 2019 from http://interactionmining.org/rico.
[19] 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 ACM Symposium on User Interface Software and Technology (UIST'17)*. 845–854. DOI: http://doi.org/10.1145/3126594.3126651
[20] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction mining mobile apps. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'16)*. 767–776. DOI: http://doi.org/10.1145/2984511.2984581
[21] Trinh-Minh-Tri Do and Daniel Gatica-Perez. 2010. By their apps you shall understand them: Mining large-scale patterns of mobile phone usage. In *Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia (MUM'10)* 1–10. DOI: http://doi.org/10.1145/1899475.1899502
[22] Google. 2016. Accessibility Scanner. Retrieved from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor.
[23] Google. 2015. Accessibility test framework for android. Retrieved from https://github.com/google/Accessibility-Test-Framework-for-Android.
[24] Google. Android Accessibility Developer Guidelines. Retrieved on 28 August, 2015 from https://developer.android.com/guide/topics/ui/accessibility.
[25] Darren Guinness, Edward Cutrell, and Meredith Ringel Morris. 2018. Caption crawler: Enabling reusable alternative text descriptions using reverse image search. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI'18)*. 1–11. DOI: http://doi.org/10.1145/3173574.3174092
[26] Vicki L. Hanson and John T. Richards. 2013. Progress on website accessibility? *ACM Trans. Web* 7, 1 (2013), 1–30. DOI: http://doi.org/10.1145/2435215.2435217
[27] 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 *. DOI: http://doi.org/10.1145/2594368.2594390
[28] Shaun K. Kane, Jessie A. Shulman, Timothy J. Shockley, and Richard E. Ladner. 2007. A web accessibility report card for top international university web sites. In *Proceedings of the International Cross-disciplinary Conference on Web Accessibility (W4A'07)*. 148. DOI: http://doi.org/10.1145/1243441.1243472
[29] Lauren R. Milne, Cynthia L. Bennett, and Richard E. Ladner. 2014. The accessibility of mobile health sensors for blind users. In *Proceedings of the International Technology and Persons with Disabilities Conference (CSUN'14)*. 166–175. DOI: http://doi.org/10211.3/133384

[30] Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. 2014. A large-scale empirical study on software reuse in mobile apps. *IEEE Softw* 31, 2 (2014), 78–86. DOI:http://doi.org/10.1109/MS.2013.142

[31] Kyudong Park, Taedong Goh, Hyo-Jeong So, Hyo-Jeong Association for Computing Machinery., HCI Society of Korea, and Hanbit Media (Firm). 2014. Toward accessible mobile application design: Developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of the International Conference on Human-Computer Interaction (HCI Korea'14)*. 478. Retrieved from https://dl.acm.org/citation.cfm?id=2729491.

[32] John T. Richards, Kyle Montague, and Vicki L. Hanson. 2012. Web accessibility as a side effect. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS'12)*. 79. DOI:http://doi.org/10.1145/2384916.2384931

[33] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2017. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS'17)*. 2–11. DOI:http://doi.org/10.1145/3132525.3132547

[34] Anne Spencer Ross, Xiaoyi Zhang, Jacob O. Wobbrock, and James Fogarty. 2018. Examining image-based button labeling for accessibility in android apps through large-scale analysis. In *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS'18)*.

[35] 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 Comput. Sci.* 67, 348–357. DOI:http://doi.org/10.1016/J.PROCS.2015.09.279

[36] Android Developers. Shrink, Obfuscate, and Optimize Your App. Retrieved on 29 April, 2019 from https://developer.android.com/studio/build/shrink-code.html.

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

[38] Android Developers. Support Different Pixel Densities. Retrieved on 23 April, 2019 from https://developer.android.com/training/multiscreen/screendensities.html#TaskUseDP.

[39] The Crosswalk Project. Retrieved on 30 April, 2019 from crosswalk-project.org/.

[40] Unity. Retrieved on 30 April, 2019 from https://unity.com/.

[41] Shunguo Yan and P. G. Ramachandran. 2019. The current status of accessibility in mobile apps. *ACM Trans. Access. Comput.* 12, 1 (2019), 1–31. DOI:http://doi.org/10.1145/3300176

[42] 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 International Conference on Human Factors in Computing Systems (CHI'17)*. 6024–6037. DOI:https://doi.org/10.1145/3025453.3025846

Received May 2019; revised November 2019; accepted December 2019