Towards Automated Accessibility Report Generation for Mobile Apps

AMANDA SWEARNGIN, Apple, USA
JASON WU*, HCI Institute, Carnegie Mellon University, USA
XIAOYI ZHANG, Apple, USA
ESTEBAN GOMEZ, Apple, USA
JEN COUGHENOUR, Apple, USA
RACHEL STUKENBORG, Apple, USA
BHAVYA GARG, Apple, USA
GREG HUGHES, Apple, USA
ADRIANA HILLIARD, Apple, USA
JEFFREY P. BIGHAM, Apple, USA
JEFFREY NICHOLS, Apple, USA

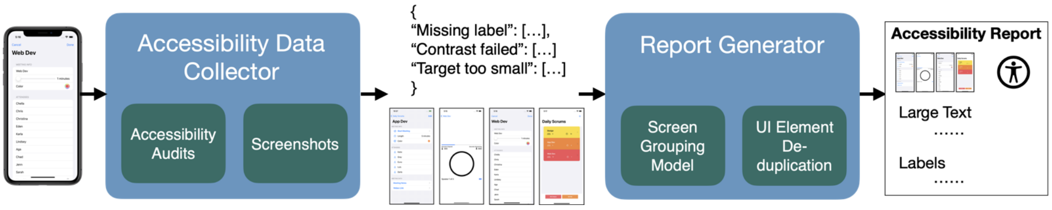


Fig. 1. Our approach to accessibility report generation collects screenshots and accessibility audits through an app crawler or a manual recording tool. From the collected data, a report generator uses a screen grouping model and UI element de-duplication heuristics to generate an interactive report.

Many apps have basic accessibility issues, like missing labels or low contrast. To supplement manual testing, automated tools can help developers and QA testers find basic accessibility issues, but they can be laborious to use or require writing dedicated tests. To motivate our work, we interviewed eight accessibility QA professionals at a large technology company. From these interviews, we synthesized three design goals for accessibility report generation systems. Motivated by these goals, we developed a system to generate whole app accessibility reports by combining varied data collection methods (e.g., app crawling, manual recording) with an existing accessibility scanner. Many such scanners are based on single-screen scanning, and a key problem in whole app accessibility reporting is to effectively de-duplicate and summarize issues collected

*Work done while Jason Wu was an intern at Apple

Authors' addresses: Amanda Swearngin, aswearngin@apple.com, Apple, USA; Jason Wu, jsonwu@cmu.edu, HCI Institute, Carnegie Mellon University, Pittsburgh, PA, USA; Xiaoyi Zhang, xiaoyiz@apple.com, Apple, USA; Esteban Gomez, estebangomez@apple.com, Apple, USA; Jen Coughenour, jjco@apple.com, Apple, USA; Rachel Stukenborg, rstukenborg@apple.com, Apple, USA; Bhavya Garg, bgarg@apple.com, Apple, USA; Greg Hughes, Apple, USA; Adriana Hilliard, adri@apple.com, Apple, USA; Jeffrey P. Bigham, adri@apple.com, Apple, USA; Jeffrey Nichols, jwnichols@apple.com, Apple, USA.

1073-0516/2024/1-ART1
https://doi.org/10.1145/3674967

across an app. To this end, we developed a screen grouping model with 96.9% accuracy (88.8% F1-score) and UI element matching heuristics with 97% accuracy (98.2% F1-score). We combine these technologies in a system to report and summarize unique issues across an app, and enable a unique pixel-based ignore feature to help engineers and testers better manage reported issues across their app's lifetime. We conducted a user study where 19 accessibility engineers and testers used multiple tools to create lists of prioritized issues in the context of an accessibility audit. Our system helped them create lists they were more satisfied with while addressing key limitations of current accessibility scanning tools.

CCS Concepts: • **Human-centered computing** → **Accessibility design and evaluation methods**; **Interactive systems and tools**.

Additional Key Words and Phrases: UI understanding, app crawling, accessibility

ACM Reference Format:
Amanda Swearngin, Jason Wu, Xiaoyi Zhang, Esteban Gomez, Jen Coughenour, Rachel Stukenborg, Bhavya Garg, Greg Hughes, Adriana Hilliard, Jeffrey P. Bigham, and Jeffrey Nichols. 2024. Towards Automated Accessibility Report Generation for Mobile Apps. *ACM Trans. Comput.-Hum. Interact.* 1, 1, Article 1 (January 2024), 37 pages. https://doi.org/10.1145/3674967

1 INTRODUCTION

Recent studies have found that a large number of both Android [56] and iOS apps [74] are still missing basic accessibility. This lack of accessibility can result from misleading or missing labels that provide descriptions of UI elements to accessibility services, or result from UI elements being completely missing from their apps' accessibility meta-data and thus unavailable for interaction [74]. Why are developers not making their apps more accessible? For some, they may be unaware of accessibility requirements, and others may choose to deprioritize accessibility in favor of other app features [4, 64].

Another cause may be a lack of efficient and effective accessibility testing tools. A variety of companies provide accessibility scanners, such as Accessibility Scanner for Android [30] and Accessibility Inspector on iOS [36], which must be manually activated on each screen of an app to dynamically test for a variety of accessibility issues. Unfortunately, it is laborious and time consuming for a developer to process and analyze a complete scan of their app. First, they must manually visit each screen in the app and collect a scan. Once they have completed scanning the whole app, developers must go through a lengthy process to examine the separate reports for each screen, identify true errors from false positives, and prioritize errors to fix. Huq et al. [34] note that the long list of reported errors from this process, which may contain many duplicates, often demotivates developers from addressing accessibility issues. We conducted a formative study, and found that some developers analyze reports scan by scan, instead of in aggregate, creating duplicate work when there are overlaps between scans. Most tools also have no memory from scan to scan, so each time they scan a screen, the developer must manually filter out false positives they have already identified as incorrect from previous scans.

Prior work has tried to address these challenges by providing accessibility app crawlers [18, 58, 59] that randomly or through record & replay approaches crawl an app to detect accessibility issues. There are two limitations to these approaches. First, they rely on accessible view hierarchies to drive the crawling itself which prior work has demonstrated to be often incomplete or unavailable for highly inaccessible apps [40, 74], which are the kinds of apps we most aim to support with our system. Zhang et. al. [74] found that 59% of app screens had at least one UI element that could not be matched to an element in the view hierarchy, and 94% of apps had at least one such screen. Second, none of these works has yet studied how users interact with and interpret information from these accessibility reports, and what features are important in an accessibility report generation tool.

In this paper, we introduce a pixel-based report generation system using an off-the shelf accessibility scanning tool which has been instrumented to detect inaccessible elements with no corresponding match in the underlying view hierarchy. This can enable our work to report on a more diverse set of apps with incomplete or missing view hierarchies. Our accessibility report generation system applies a multi-step workflow that uses pixel-based machine learning models and heuristics to generate a high level summary of results and filter false positive issues (see Figure 1).

To motivate our system, we interviewed eight accessibility engineers and QA testers about their pain points in using current accessibility scanning tools. Through these interviews, we identified the following user needs for an accessibility report generation system:

(1) Reduce the time required for developers and QA testers to manually scan individual screens with accessibility scanning tools.
(2) Provide developers and QA testers with an overall accessibility report of scan results.
(3) Enable developers and QA testers to reduce noise by ignoring false positives or previously addressed issues.

Finally, we conducted a user study of our system where participants interacted with reports generated by a manual scanning tool and an app crawler. Participants were more satisfied with their accessibility audit summaries created with the help our system vs a baseline tool, and it helped them quickly prioritize important issues. Our study provides insights into the features that should be supported by accessibility report generation systems.

It is important to recognize that while this paper focuses entirely on improving automated accessibility testing, this is just a part of the larger testing ecosystem that is needed to produce highly accessible apps. First, testing should be done throughout the development workflow starting as early as possible, for example using static analysis tools to evaluate source code for accessibility issues [25, 28]. Although many of our subjects and users work on products in later stages of development, it is worth noting that our tool can be used as soon as the app under development has a functional UI, and could in principle be used in continuous integration to find and surface accessibility errors regularly during development. Second, it is well known that automated testing cannot detect all issues [2, 34, 54], and manual testing should also be done to ensure the most accessible app experience. Both accessibility testing specialists [9] and people with disabilities [46] should be recruited for manual testing. Our user study suggests that one benefit of our tool may be that it reduces the time testers spend reading test reports and allows them to perform more complex manual testing, though this deserves further study. We recognize that automated accessibility testing systems, like the one we describe here, are only a part of the solution and should not be used as the only measure to ensure accessibility.

The contributions of this paper include:

- A formative study to identify key limitations and inefficiencies with accessibility scanning tools, inspiring 3 key design goals for a system to generate basic accessibility reports.
- An accessibility report generation system instantiating these design goals through pixel-based UI understanding models and heuristics. The system combines an accurate screen grouping model (96.9% accuracy, 88.8% F1) with UI element detection [74] and matching to build an application storyboard of unique screens, de-duplicate issues, enable an ignore feature, and filter false positives.
- A user study with 19 app developers and QA testers demonstrating that our report generation system can generate clean and accurate reports which can help them quickly prioritize and find common issues across an app. The study also reveals design insights for accessibility reporting interfaces and features needed to make them more effective in future systems.

Development Stage	Approach
Development (Static)	**Source Code Analysis**: Android Lint [28], AATK [25]
Test (Dynamic)	**UI Testing Frameworks**: Espresso [27], Roboelectric [55], Earl Gray [24], XCTest [37] **Accessibility Testing Frameworks**: Latte [57]
Runtime (Dynamic)	**App Crawling**: MATE [18], Alshayban et al. [4], Groundhog [59], Xbot [14] **Scanners**: Accessibility Inspector [36], Accessibility Scanner [30], Evinced [20], Lighthouse [31] **Record & Replay**: A11yPuppetry [58] **Scanning, App Crawling, Summarized Report:** Our Work

Table 1. Tools and frameworks for automated accessibility testing span across development, test, and run-time stages. Our work uses a dynamic approach that relies on app crawling and a scanner to generate a basic accessibility report.

2 RELATED WORK

Previous research systems have explored how to automatically collect, report, and repair accessibility issues. We primarily review tools for automated detection and reporting in mobile app contexts; however, as our work also provides insights into the challenges of current accessibility scanning tools and outcomes on how developers interact with accessibility reports for mobile apps, we review related work in the web context on automated accessibility reporting which may share similar issues and insights. We also review work in methods for UI element and screen identification as our work improves upon existing models and enables use cases beyond accessibility report generation.

2.1 Challenges of Accessibility Testing Tools

While automated accessibility testing tools are valuable to QA professionals and developers in testing their apps' accessibility support, they do have a number of challenges that have been studied by prior work. While there are multiple types of accessibility testing tools such as accessibility testing frameworks (e.g, Latte [57]), in our work, we focus on accessibility auditing tools which scan and report accessibility issues in developed software. Our work relates to three key challenges with these tools. First, these tools often produce false positives [1, 34] which we also validated in our formative study. Our work aims to alleviate one category of false positive issues commonly appearing in the results of accessibility scanners – i.e., issues reported with no corresponding visible UI element – by applying a pixel-based UI understanding model [74]. While this only addresses one common type of false positive, there is potential for accessibility testing tools to adopt ML technologies to both detect new categories of issues and filter false positives.

The second challenge our work examines is automatic report generation. Some work has found varied levels of coverage in accessibility testing tools covering different success criteria [52, 65]. Therefore, testers may need to apply multiple automated accessibility auditing tools to their mobile or web interfaces to get more complete coverage [1, 7, 34]. After testers run these tools, they will likely also need to aggregate, summarize, and compare the results of each tool [1, 34] which can be difficult and labor intensive. Our work targets these challenges in two ways. First, we provide a method for automatic summarization and generation of a report from the results of raw accessibility audit results to alleviate the need for developers and QA testers to do this manually. Second, our report generation method is agnostic to the underlying accessibility auditing tool. Our system uses Accessibility Inspector [36], but it could also replace this or augment its results with the results of other accessibility auditing tools if they provide an API or provide the results in an easily consumable output format (e.g., JSON).

Another challenge of many accessibility auditing tools is that they require testers to traverse and scan screens to be audited. However, this manual effort in scanning is a burden to testers and is not scalable or easily repeatable. Our work applies an app crawler to automatically navigate app interfaces to collect screens to be audited. Other work also uses app crawlers for accessibility testing, but focuses less on summarization and reporting [18, 59]. Much of the work in large scale web accessibility testing also uses crawling [35] to collect accessibility audits, and even enables reporting metrics that would not be possible without an automated solution [47]. Our work contributes an approach to collect the underlying data that would be necessary to report such metrics for mobile apps in the future.

2.2 Automated Tools and Frameworks for Mobile Accessibility Testing

Several tools exist to check accessibility properties of apps [60], and they can generally be categorized as development-, test-, or run-time (Table 1). Development-time tools, like AndroidLint [28], use static analysis techniques to examine code and declarative user interface descriptions for potential issues. These tools do not have access to user interface elements that may be created programmatically or data that is downloaded at run-time.

Test-time tools [24, 27, 37, 55, 57] are integrated into functional or UI testing processes, and collect data when tests are run. These tools collect data from the running UI to find issues development-time tools would miss, but may be limited by the completeness and coverage of the tests. Unfortunately, past work has shown that mobile apps are often tested in an ad-hoc manner [16] or not at all [39]. Latte [57] eases the process of accessibility test creation by working with test cases written for functional UI correctness, which are easier to author than UI integration tests. Latte tests for accessibility by replaying test cases through available accessibility services, such as SwitchAccess or TalkBack [29] on Android. While Latte can detect more accessibility issues than prior work, it remains only as effective as the coverage of the test cases across the entire app.

Run-time tools [30, 36, 58] examine the running user interface through dynamic analysis, which emulates the way an end user would see the app, and can find issues development-time tools would miss [18]. Accessibility scanners, like Accessibility Inspector for iOS [36] and Accessibility Scanner for Android [30], require developers to visit and scan each screen of an app, and they often do not provide summarization across screens.

To alleviate the burden of manual navigation on the developer, other run-time tools use app crawling to find accessibility issues. An app crawler is a program that runs to autonomously interact with, explore, and collect data from apps. MATE [18] uses an app crawler for accessibility testing through dynamic, random exploration of Android apps and detects accessibility issues on encountered screen states. When an app crawler uses a random exploration strategy, it typically uses a service (e.g., UIAutomator) to identify interactive elements on a UI screen and randomly selects one of them which it will interact with through simulation of user actions. The crawler repeats this process until it has reached a specified number of screens, UI elements, or when a time budget is reached. To our knowledge, all past systems work on the Android platform and rely on a view hierarchy and accessibility metadata to understand the app contents. While some common Android components are accessible by default (e.g., View), this is often not the case for custom widgets, and such UI elements may not be visible to app crawlers relying on accessibility services.

Both Xbot [14] and Alshayban et al. [4] crawl apps to collect accessibility issues but either rely on app instrumentation or static analysis of source code to extract intents, which they note will only work for a limited number of Android apps [14]. All three [4, 14, 18] appear to rely on Android

Input	Issue Type
View Hierarchy, Screenshot	**Icon Labels**: LabelDroid [11], COALA [49] **Visual or Display Issues:** Owl Eyes [44], Iris [76] **Target Size:** Alotaibi et al. [3]
Human annotated labels, Screenshot	**UI Element Detection:** Screen Recognition [74], Ours **Icon Labels:** Chen et al.[13], Screen Recognition [74]

Table 2. Prior work introduced methods detect accessibility issues using machine learning. The inputs to these models include view hierarchies, screenshots, and human labels app interfaces to detect a variety of issues types. Our work detects and surfaces missing accessibility elements (i.e., UI elements not exposed to accessibility hierarchies) using UI element detection models.

Activity and heuristics to determine screen states, and primarily focus on generating issue counts rather than producing a summarized report that developers can interact with.

More recently, the accessibility app crawler Groundhog [59] crawls apps through accessibility services to detect additional classes of issues (i.e., locatability, actionability), producing a report of issues, and a video to visualize navigational failures (i.e., TalkBack [29]). However, the paper does not provide details on the interface for the output report or study how users interact with it. A11yPuppetry [58] uses a record and replay approach with a similar infrastructure to replay tests through accessibility services and reports a few classes of issues. In contrast, our system uses an existing accessibility scanning tool within an app crawler, thus reports on more and different classes of issues than A11yPuppetry. By leveraging a pixel-based app crawler, similar to Wu et al.'s [72], our system can navigate to areas of the UI that would be inaccessible through accessibility services, which Groundhog relies on. Prior studies [74] have demonstrated a large amount of apps still have many UI elements and screens that are not exposed to the accessibility hierarchy. In contrast to prior works [58, 59], we also present methods to report and summarize detected issues and evaluate this summarization through a user study with app developers and QA testers.

In summary, prior approaches have detected accessibility errors through different ways of exploring an app: manual capture, integration with existing UI tests, or automated app crawlers. We instead focus on assembling the results of accessibility error reports, agnostic to the collection method, into a single app report with an overall summary. We also present new technologies to summarize unique issues, filter false positives, and enable developers to ignore issues in future reports, which can help reduce noise when adopting such systems for regression monitoring over time.

2.3 Machine Learning-Aided Accessibility Issue Detection

Beyond the tools and frameworks for accessibility testing, prior work has explored detection and repairing *specific* accessibility issues (e.g., target size, unlabeled elements) using machine learning. These models may be used within the prior tools and frameworks which currently rely on static or dynamic analysis [30, 36] or heuristics. Machine learning models can operate on additional inputs in addition to screenshots (Table 2).

View hierarchy based methods [3, 44, 49] train models on screenshots and semantic information extracted from view hierarchies to generate icon labels for accessibility services [11, 49], repair size-based accessibility issues [3], and detect visual display issues [44]. These approaches demonstrate that machine learning can generically detect and even repair accessibility problems. In our work, we apply a pixel-only based approach. This follows previous work on UI element detection [74] which detects and repairs UI element labels for a screen reader. Pixel-only approaches, also used to generate

Scope	Method
Across Apps	**CNN-based Object Detection:** Gallery DC (mining design semantics) [10] **Autoencoder-based:** Liu et al.(mining design semantics) [43], Rico (similar screen search) [17] **CNN-based embedding:** VINS (visual search) [8], Swire (sketch search) [33], Screen Parsing (similar screen search) [73]
Within Apps	**Hashing:** Puppetdroid [26] **Heuristics:** Zhang et al.(accessibility annotation) [75], Bility (accessibility annotation) [66], Fragdroid (app crawling) [12], Humanoid (app crawling) [42], Droidbot (app crawling) [41], Jiang et al.(app crawling) [38] **Modeling:** Feiz et al. [21], **Ours**

Table 3. Our work uses machine learning based UI Understanding to generate accessibility reports organized by screen using a screen similarity transformer model. Prior work has used machine learning, hashing, and heuristics to search for similar screens across apps or within apps.

icon labels [13, 74], may benefit from relying on the visible interface screen over view hierarchies which can be unreliable and difficult to interpret [40]. We use machine learning approaches in our system to detect a particular class of issues and to summarize and de-duplicate issues in the generation of accessibility reports. Our system primarily focuses on reporting issues, and it could be used in combination with prior work to detect issues and alert developers of accessibility issues earlier.

2.4 UI & Screen Identification

Our work presents a new model for screen grouping and heuristics to identify UI elements across different instances of a UI screen within a mobile app. This work contributes to a body of work in computational screen and UI understanding that has been applied to diverse use cases. Some examples include accessibility [75], app crawling [42], and design search [10] (Table 3). We specifically focus on work within UI Understanding for mobile applications. Mobile platforms can be more challenging in some ways compared to other platforms (i.e., web) as the underlying source code and UI hierarchy are often not available or are incomplete.

Across apps, some work builds models trained on UI screen datasets to support design search for *similar screens* [8, 10, 17, 43, 73]. Swire applies a similar concept to search screens based on UI sketches [33]. While some ML aspects of these works may aid in our screen grouping problem, they apply screen similarity detection across similar screen types in different apps, while we aim to group screen types within an app. Additionally, some of this work [8, 17, 73] does not appear to incorporate visual information into the similarity problem, which we believe can provide important cues for screen similarity.

Other work aims to identify “same screen” instances within an app. In this paper, the terminology “same screen” refers to two screens within an app used for the same purpose, to accomplish the same task, or to view the same type or category of information. This is adapted from the work of Feiz et al. [21] which introduces a “screen similarity” model to identify same screen instances within an app. This was primarily motivated by app crawling use cases (e.g., Humanoid [42]) as app crawlers need to understand if the screen they are currently viewing is the same as one previously encountered to focus efforts on covering unexplored areas of the app. Identifying “same screen” instances within an app is challenging as different instances of the same screen may have different data, scrolling, have keyboards open, or dialogs opening and closing.

In this work, we developed an improved screen similarity model from Feiz et al. [21] which we use both within an app crawler and to generated summarized accessibility reports which group detected issues by screen. For same screen detection within an app, prior work has proposed heuristics-, modeling-, and hashing-based approaches. Earlier works applied a perceptual hash [26] to detect same screens within an app, but later work showed that hashing techniques have high precision but very low recall [21]. In accessibility report generation, this performance could result in much noisier and less usable reports. Zhang et. al. [75] present screen and UI element equivalency heuristics based on identifiers and structures in Android view hierarchies. App crawling, a key use case for same screen detection, relies on similar heuristics based on view hierarchy structures [12, 38, 41, 42] which prevents them from being generalizable to other platforms. Feiz et al. [21] presents a machine learning approach for same screen detection within an app. We build on this work, but we modified the definition of "same screen" problem, collected cleaner annotations, and updated the model architecture to produce embeddings for faster computation across large sets of screenshots. We use this screen grouping model in our app crawler which explores apps to collect screenshots and accessibility scans, and in report generation where we use the model to group accessibility scan results by screen and de-duplicate them.

3 BACKGROUND & USER INTERVIEWS

This project began as a collaboration with our research team, the accessibility engineering team from a large technology organization and a product manager in charge of app accessibility for a different team in the same organization. From these stakeholders, we initially learned about the challenges of collecting and assembling accessibility reports for a full app. To understand more about these challenges from a larger group, we interviewed eight accessibility-focused developers, testers, and managers from five diverse product and research-focused teams at the same organization (4M, 4F). All participants had been working in accessibility focused roles for at least one year; however, the median years of experience working in accessibility engineering or accessibility QA roles was five. Through a small set of structured questions, we discussed their experiences testing app accessibility in 30 minute exploratory interviews. We provide a copy of the questions asked for this study in the appendix (Appendix **??**). The interviews were semi-structured, so we started with these questions and asked further follow-up questions if warranted.

First, we asked participants to describe which parts of testing for app accessibility they found challenging. Through our preliminary investigation of prior work, and interviews with accessibility stakeholders, our hypothesis was that accessibility scanning tools could provide value to developers and QA testers, but were having limited adoption during development and testing due to many inefficiencies and limitations. Thus we asked participants to detail what they liked and disliked about accessibility scanning tools. The primary tool our participants had used was Accessibility Inspector [36], however, participants also mentioned using the Evinced scanner [20], Lighthouse [31], and Android tools (e.g., [30]). While the participants sometimes wrote accessibility tests and used automated scanners, they reported primarily manually testing their apps. From this formative interview, we conducted a qualitative analysis of the interview transcripts [32] and the following themes emerged from that analysis which center around the challenges the participants encountered when using accessibility scanning tools and the reasons some participants noted for not adopting them in their workflows as much as they would like.

Current tools provide no results overview: Our participants mentioned that since scanning tools provide results per screen, they can't easily see an overview of results – *"I can't really get an overview of an app's accessibility just from that tool"* or *"view the issues of a particular type across the* *app"*. Some participants said it can be hard to give feedback on app accessibility to teams that may not understand accessibility well or know what to test. Such teams might benefit from feedback on issue patterns across the app (e.g., missing Large Text support), which for some participants to compile may require *"toggling on that feature and navigating through every single screen of the app myself to get an idea of whether this app does or does not support those [accessibility features]"*

Current tools are too noisy: Participants mentioned that results of current tools *"can be quite noisy at times and so we end up with a lot of false positives"*. For someone less familiar with accessibility features, *"they have a really hard time understanding what's signal and what's noise from the report"*. Participants also mentioned many issues detected by these tools are lower priority to fix – *"false positives are confusing. There's definitely a difference between elements that can't be visited in any way, and are totally inaccessible, compared to some of the smaller nit picks that get presented"*. This feedback echoes prior work on mobile [34] and web [1] accessibility testing tools which often provide false positives, requiring testers to incorporate the results from multiple tools.

Manual scanning introduces inefficiencies: Participants also recounted the manual effort and time to use accessibility scanning tools *"it will take me a couple hours just to get through a couple screens, like a few screens usually"*. The amount of effort involved often leads them to scan their apps infrequently. When multiple developers or teams contribute to the same app, manually scanning after each change is infeasible. Accessibility regressions can be created and persist for quite some time. It can also be infeasible to run these scans across the multitude of combinations of devices and accessibility settings they would like to test.

3.1 Design Goals

From these formative interviews, we formulated the following design goals for a system to generate accessibility reports for accessiblity scanners.

- *D1*: Reduce the time required for developers and QA testers to manually scan individual screens with accessibility scanning tools.
- *D2*: Provide developers and QA testers with an overall accessibility report of scan results.
- *D3*: Enable developers and QA testers to reduce noise by ignoring false positives or previously addressed issues.

For *D1*, we adopt an app crawler introduced in prior work [72] that we modified to scan each screen using the Accessibility Inspector [36]. For *D2*, we provide an output HTML report in a live webpage (Figure 5). We also provide a manual tool for accessibility scanning which generates a multi-screen report. For *D3*, we developed features to ignore false positives & previously addressed issues which are available in the web report.

4 ACCESSIBILITY REPORT GENERATION

Figure 2 illustrates our report generation approach. First, a data collector, such as manual capturing tool, an app crawler, or a test case-based recorder, captures screenshots and accessibility data (a). Next, a *report generator* generates a summarized report by first building a storyboard of same screen instances detected by a screen grouping model (Figure 2.b). The report generator then uses UI element de-duplication heuristics to de-duplicate issues (c) and hide previously ignored issues (d) that users have marked in prior runs. Lastly, the report generator uses a UI element detection model [74] to filter false positives (e) and produce a report with an overall summary and results grouped by screen.

We implemented our prototype as a Flask-based web server that controls data collection via a proprietary device cloud, generates reports, and hosts reports for later viewing by users. We currently only support generating accessibility reports for iOS-based apps.

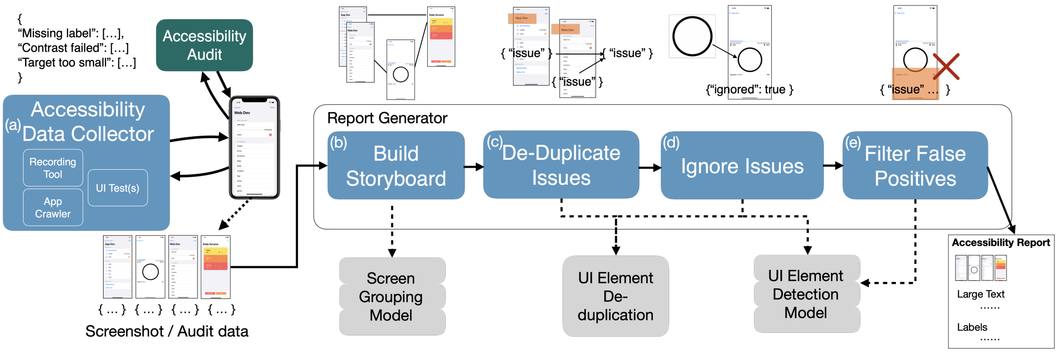


Fig. 2. Our system for accessibility report generation. A data collector (a) captures screenshots and accessibility audits, followed by a 4-stage process that produces a summarized accessibility report. The stages consist of 2) "Build Storyboard" (b) which uses a pixel-based screen grouping model to build a storyboard of app screens, (c) & (d) "De-Duplicate Issues" and "Ignore Issues" which use UI element matching heuristics based on a UI element detection model to de-duplicate and filter out issues that QA users have ignored, and (e) "Filter False Positives" which removes any issues with no corresponding UI element on the screen as detected by the UI element detection model.

4.1 Accessibility Data Collection

The first step of report generation is capturing accessibility data to report (Figure 2.a). This is aimed towards design goals *D1* and *D2*. Our prototype offers two options for data collection: a manual recording tool and a random app crawler using the architecture adapted from Wu et al. [72].

The random app crawler runs on a remote cloud device or a locally attached iOS device. The random app crawler detects clickable UI elements on each screen using UI element detection [74], and interacts with them to explore the app. It uses our screen grouping model to find new screens to attempt to maximize coverage. On each screen, it captures an accessibility scan and screenshot and may capture a screen in various states. This is a quicker option to generate a report, but provides no guarantees of obtaining complete coverage over all screens in the app.

Users interact with the manual recording tool via desktop MacOS interface. The interface connects to a locally attached device or simulator and provides a button "Run Audit" to capture an accessibility scan and screenshot. While this does not directly meet design goal *D1*, the user cannot examine reports as they are generated, and may be less likely to get distracted by the results until they have finished capturing. Once finished, the system generates a summary report for all screens captured by the user (*D2*). Using this data collection method requires manual effort, but gives the user control over what screens are captured.

Both of these methods collect accessibility scans and screenshots. We currently use Xcode's Accessibility Inspector [36] feature via a command line tool on each device, which produces a JSON report listing all detected issues with their associated bounding box on the screen. The Accessibility Inspector supports 29 accessibility checks, categorized by *Element Description*, *Contrast*, *Hit Region*, *Element Detection*, *Clipped Text*, *Traits*, and *Large Text*[1]. In the future, it should be possible to add other accessibility scanning tools to our capture process, provided they can run on a live device and produce JSON output.

[1]How well does Accessibility Inspector cover accessibility criteria for mobile apps? According to the WCAG guidelines, all information presented digitally should be perceivable, operable, understandable, and robust [68]. W3C provides a mapping of guidelines within these principles for mobile apps [67]. Accessibility Inspector largely includes checks under the *perceivable* principle, such as resizable text, element descriptions, and sufficient contrast. Under *operable*, it includes

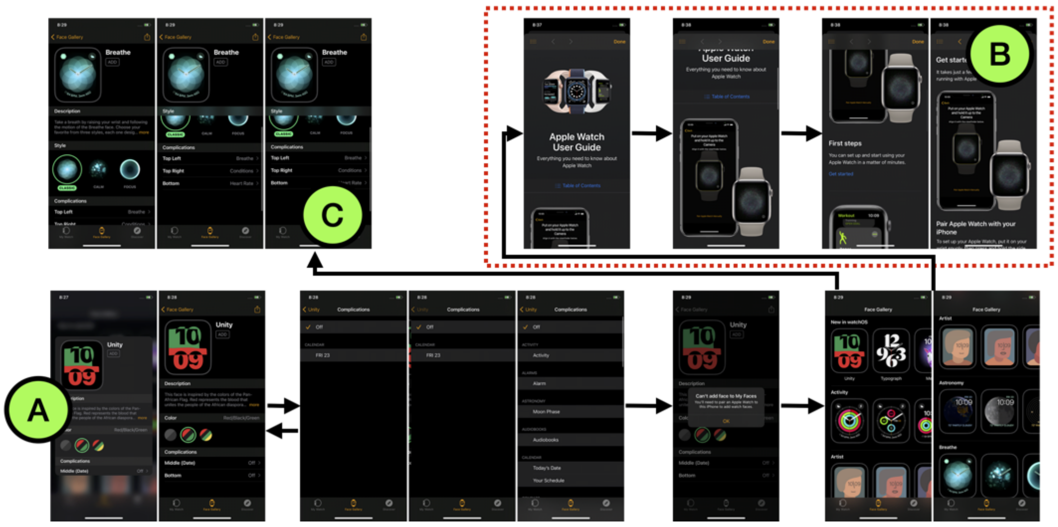


Fig. 3. An example of an app storyboard generated by our screen grouping model. Screens grouped together (e.g., A, C) were predicted as "same screen" by our screen grouping model. The arrows between groups mark transitions between each screen(s) where interacting with a UI element on the origin screen transitions the app to the destination screen. The red box (B) marks a missed opportunity for grouping in where the screen has been scrolled but all four screens should still be considered to be the "same screen" since they are the "Apple Watch User Guide" screen in various states of scrolling.

4.2 Build Storyboard

Data collectors may often capture multiple instances of "same screens" that appear slightly differently, perhaps because they are scrolled or contain dynamic content. If we reported the accessibility scanner's results for each screen instance individually, the overall report would be noisy and contain duplicates. To mitigate this, the report generator aims to group together the results from "same screen" instances into a single section of the report. To do this, the report generator uses a screen grouping model that operates on app screenshots to build an app storyboard. The app storyboard clusters the results from different instances of the same screen together (Figure 2.b). In this work, we use the term "storyboard" from UI builders (e.g., Xcode Storyboards) which describes a visualization of relationships between views in an app, rather than term "storyboard" to convey a user story in UX design.

Figure 3 shows example storyboards generated by our model from a set of app screenshots collected by an app crawler. We present the storyboards in a graph visualization of the screens grouped together by our screen grouping model to help in visualizing the models full understanding of which screens in the app should be grouped together. Clusters of screens placed together (Figure 3.C) indicate the model considered those screens to be the same, whereas screens that are not grouped together (e.g., A and C) are not the same. The arrows between the groups mark transitions where interacting with a UI element on the origin screen would take the user to the destination screen. The goal of the "Build Storyboard" step of our report generator is to generate a representation it uses to group and organize the results in the accessibility report, which will

checks for minimum touch target sizes and accessibility actions. Finally, it includes some checks applying to *understandable* such as grouping and element detection. Please see the documentation for more information on the checks it covers – https://developer.apple.com/documentation/accessibility/performing-accessibility-audits-for-your-app.

ultimately present the results for each group of screens in a separate tab to make them easier for developers to digest (Figure 5). The report generator also uses the screen groups to create a summarized overview of issues across the app. Screen grouping enables the report generator to summarize the results without repeating the results for a single issue when it captures the same screen with different variations (e.g., scrolled down, with extra UI elements). We describe this de-duplication process in Section 4.3.

4.2.1 Screen Grouping Model. Screen grouping is a key technology used across multiple stages of our system (Figure 2.b-d) to de-duplicate screens and issues. For screen grouping, we built upon the Feiz model [21] in prior work. The input to this model is two screens, s_1 and s_2, and the output is a binary prediction of *"same screen"* or *"different screen"*.

The original Feiz model was trained on a dataset constructed from 77k screenshots across 1,110 iOS applications [21]. The paper further details the data collection process used for capturing these screenshots. Despite various experiments with model parameters, we were unable to improve the original model's performance beyond 75.8% F1-score when trained and evaluated on its original dataset. Note that this F1 score is lower than previously reported [21], we believe due to different choices in training, validation and test set splits. We discovered the annotation process in the original work had a low agreement rate. To address this, many apps were dropped from the final dataset. We subsequently focused our efforts on improving the annotation process, both to increase annotation agreement and to make more data available for training and evaluation.

We examined 1000 failure cases from the original similarity transformer model trained on the Feiz dataset [21]. Model prediction errors occurred primarily when same screen instances were scrolled or had structural differences (e.g., keyboard open / closed, search with and without data) and were predicted as different, or for different screens that were structurally and visually similar. The annotated labels contained disagreements on some cases when screens had different tabs or page controls active, or displayed different data.

4.2.2 Screen Grouping Data Collection. To improve the quality of this dataset for our report generation use case, we collected an entirely new labeled dataset of 750,000 grouped screens from 6700 free apps using a similar data collection process to Feiz et al. [21]. To collect the 750k app screens, ten annotators (i.e., crowd workers) manually explored apps through a remote device in a web interface with the instruction to find as many unique screens as possible within a 10 minute limit. While the annotators crawled the apps, the system captured screens every second provided the screen changed. A separate team in our company hired and paid the annotators with legal and ethical approval, similar to the review of an Institutional Review Board (IRB), and paid them a competitive hourly wage based on their location.

A different set of 15 annotators grouped the screenshots of each app into same screen groups using a card sorting style interface, similar to that of Feiz et. al. [21]. We generated initial groupings using our trained screen similarity model with 75.8% F1 -score to cluster the screens. Thus, annotators only needed to fix the model's mistakes rather than starting from scratch. Annotators also discarded invalid screens (e.g., home screen, loading screens, landscape orientation). The screenshots from each app were only grouped by single annotator; accordingly, there were no disagreements in the groupings of the screens within each app.

Our annotation guidelines defined same screen as two screens used for the same purpose, to accomplish the same task, or to view the same type or category of information. For the annotation task instructions, we defined a list of possible variations a screen can undergo to be considered the same screen including:

- Same screen with different data

- Partially scrolled down
- Sections expanded or collapsed
- Keyboards open or closed
- Non-modal application dialog open or closed
- Same modal menus or dialogs on top of different content.

We compiled this list iteratively where multiple researchers, who are also domain experts in UI understanding, met multiple times to examine "same" and "different" screen examples in a large dataset and resolved any disagreements.

As nearly all accessibility scanners operate on the topmost, non-occluded layer (e.g., contrast checks), we modified Feiz et al.'s definitions [21] by grouping screens based on the topmost dialog or layer of interactive content (i.e., two screens with the same dialog open over different backgrounds are considered the same screen).

We trained the annotators on these guidelines through an extensive slideshow with positive and negative screenshot examples of each category, along with explanations. The labeled data contains 70,882 groups across 6,332 apps, with a mean of 10.8 groups per app (Med: 10, Std: 7.3) and 3.3 screenshots per group (Med: 2, Std: 6.7). As the screens from a single app were only grouped by a single annotator, we cannot report an IRR metric for these annotations. However, our company employs a separate group of expert QA annotators who reviewed 10% of the annotations at random, until they certified that the annotations achieved a 98% accuracy threshold using the annotation guidelines as the source of truth. We compute the 98% accuracy value as the number of screens that the QA annotators moved to a new group out of the total number of screens in the set.

4.2.3 Model Modifications. To understand the impact of these annotations, we trained two additional screen grouping models to compare with the original Feiz model [21]. The first is the original similarity transformer Feiz model trained on the new data we collected, and we created a modified version of the Feiz model that produces an embedding for each screen. The Feiz model is transformer model which applies a cross-encoder to predict similarity for a pair of screens by minimizing the binary cross-entropy loss on the predicted similarity label.

One challenge with this pairwise model is that it can be computationally costly ($O(n^2)$) as each screen in a set is added as it requires a new model call to compare a newly added screen to each existing screen in the set. Our ultimate goal is for our reports to be generated on demand, ideally within a few minutes. This model architecture would severely limit the ability of our report generation system to be run efficiently and interactively. To mitigate this, we created a modified Bi-Encoder embedding model that uses a transformer to generate an embedding for each screen by encoding and pooling the output of pre-trained object detection model [76] on the screenshot. The model takes in pairs of screens as input, and learns to minimize the distance between the embeddings of pairs of same screens while maximizing the distance for pairs of different screens. We also apply the masked prediction objective to this embedding model from Feiz et al. [21].

This modified architecture enables our system to compute screen embeddings for each screen a priori and calculate the distance between embeddings to determine similarity versus conducting a pairwise inference with all known screens. If the difference between the pair of screen embeddings is less than or equal to a *margin* value, the model's prediction is "same screen". Conversely, if the difference between the pair of screen embeddings is greater than *margin*, the model's prediction is "different screen". The *margin* is a value that we tuned specifically for the Bi-Encoder model to achieve the highest accuracy and best balance between precision and recall on the test samples.

We compare multiple versions of screen grouping models in our evaluation (Section 5.1).

	Train	Val	Test	#Apps	#Screens
Feiz Model - Baseline					
SSim Transformer (FD)	70%	15%	15%	1,110	77k
SSim Transformer (1k)	70%	15%	15%	1,110	77k
New Dataset & Model - Our Work					
SSim Transformer (6k)	94%	3%	3%	6,332	226k
SSim Bi-Encoder (Embedding) (6k)	94%	3%	3%	6,332	226k

Table 4. Details of the model baselines and new models we trained for screen grouping, along with their split percentages (i.e., training dataset percentage – Train, validation dataset percentage – Val, and test dataset percentage – Test) and dataset sizes by number of apps and screens.

4.2.4 Dataset Splits & Model Training. We split the data into training, validation, and test sets by app to ensure that screens from the same app appear in only one set. We trained multiple versions of the models on various subsets of the data. The original Feiz model on the original dataset [21] (FD) uses split percentages of 70% training, 15% validation, and 15% test, and the dataset contains over 77k screens across 1,110 apps. To help us understand whether our new annotations improved the model's performance, we trained another version of the original model on a 1k (containing the screens from the original 1,110 apps) subset of our newly annotated data using the exact app splits and percentages as the original model. Table 4 summarizes the dataset sizes and splits for these two models. The only difference between these two models is that the input data was entirely re-annotated.

To evaluate the impact of our significantly larger dataset, we trained two additional models on our full 6k dataset including the original Feiz model and the modified Bi-Encoder embedding model. This dataset contains all app's data from our screen grouping data collection of over 226k screens across 6,332 apps. To enable the models to be more comparable to the baselines, we added all apps and screens not in the original 1k subset to the training data split, and kept the validation and test sets the same size. As a result, the training splits for the models we trained on the 6k dataset were %94 training, %3 validation, and 3% test. Table 4 summarizes these splits and dataset sizes.

The training input to each model are pairs of "same" and "different" screens generated from the annotated groups of each app. We include all possible pairs of screens within a group as "same" screen samples and all possible pairs of screens not in the same group as "different" screen samples. The full 6k dataset is significantly unbalanced, containing 8.2 million pairs of "different" screens and 3.3 million pairs of "same" screens. Section 5.1 details our evaluation of these models.

4.2.5 Storyboard Generation. To build a storyboard of app screens using this model (Figure 2.b), the report generator processes each screen collected from the app consecutively. When it processes the first screen, the storyboard generator computes the screen's embedding score using the screen grouping embedding model. It then creates a new screen group for it and updates the mean embedding score for the group. When the storyboard generator adds a new screen to the set, it needs to determine if this new screen belongs in any existing group or if it belongs in its own new group. To do this, it computes the screen's embedding using the screen grouping model, and then computes the difference between the embedding and the mean embedding for each group of screens already in the storyboard. It then assigns the screen to the group with the smallest distance between the embedding scores, if the difference is less than the *margin* value of *0.34*. If the difference from the screens embedding to the mean embedding from any group is greater than the *margin* value, the storyboard generator creates a new group for it.

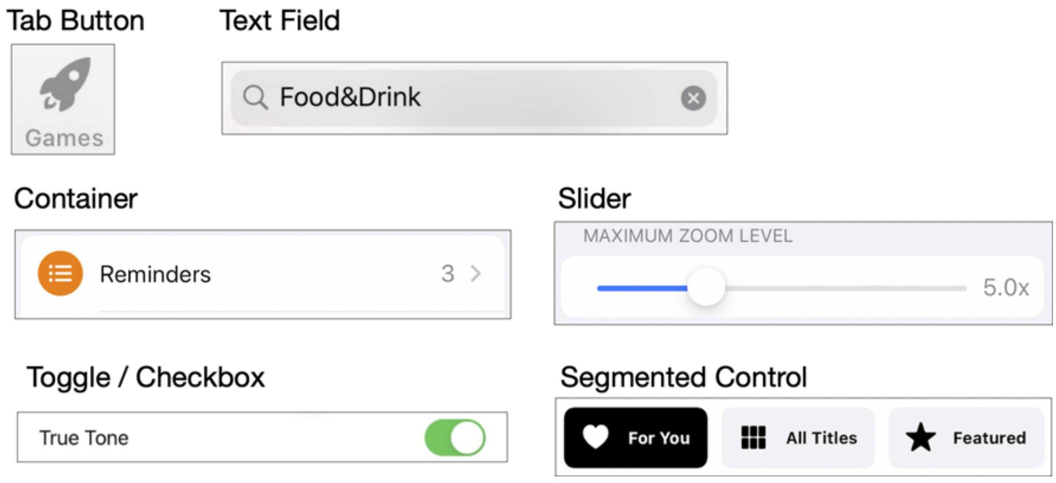


Fig. 4. UI element groupings used by UI element de-duplication heuristics to find matching UI elements in a new screen. The examples include several UI types including Tab Button, Text Field, Container, Toggle / Checkbox, Slider, and Segmented Control.

The storyboard generator also adds transitions between screens (i.e., arrows in Figure 3) based on the order the screens were captured in the crawl or recording tool. Our report generation system does not directly use these transition edges in creating its output report; however, they can be potentially applied to other use cases that would like to apply similar screen grouping to improved app understanding.

4.3 De-duplicate Issues

A key requirement of report generation motivated from our interviews is to avoid noise (design goal *D3*). Formative study participants noted that accessibility scanners can often produce noise (see Section 3), so our system should reduce noise when summarizing results across multiple screens. After the *Build Storyboard* step, our report generator (Figure 2.c-e) applies heuristics to identify multiple instances of the same element across screens and de-duplicate them, and summarizes results on each screen and across an app. The system applies these same heuristics to ignore issues, which users can mark in the report interface (Figure 5), and re-identifies them on future reports of the same app. The goal of these heuristics is to *find the same UI element across two different instances of the same screen within an app.*

4.3.1 Pre-processing of template screen and UI element. The input to the heuristics is a pair of same screen instances formulated as a *template* screen T_s and UI element T_{ui}, and a new screen N_s. The goal is to find the best matching UI element N_{ui} within the new screen N_s for a template UI element T_{ui}.

To de-duplicate UI elements, the heuristics first pre-process the template screen T_s, template UI element T_{ui}, and new screen N_s by detecting UI elements in T_s, N_s and grouping them using modified grouping heuristics introduced by [74] (see Figure 4 for examples of the UI element groupings constructed by these heuristics):

Tab Button: A tab button group often contains a icon and text, and sometimes contains only a icon.

Toggle or Checkbox: A toggle (or checkbox) group contains that element and its text description, which is often the closest text on the same row.

Segmented Control: A segmented control group contains the border and the text of a segmented control.

Text Field: A text field group contains the border of the text field and UI detections inside it.

Slider: A slider group contains the slider and text on the same row and the closest text above.

Container: A container group contains the border of the container and UI element detections inside it.

4.3.2 Finding the best matching UI element. Our de-duplication procedure then applies heuristics to find the best match for a template UI element T_{ui} in a new screen N_s by comparing each UI detection in N_s with T_{ui} to get their similarity scores and selecting the one with the highest similarity score. The heuristics for computing similarity scores are as follows based on the UI element type of T_{ui}:

Text: Make all text lowercase and keep only alphanumeric characters and spaces. Apply fuzzy matching to compute similarity score [6].

Icon and Picture: Search the area around the Icon or Picture detection using image template matching [51] (template = the cropped pixels of T_{ui}) in multiple scales[2]. The similarity score is the max value of template matching among all scales.

Tab Button: If the template Tab Button contains only an Icon, find the Icon using template matching. When the template Tab Button contains both Icon and Text, run the Text matching method.

Toggle, Checkbox, Segmented Control, Slider and Text Field: Run the Text matching above on its grouped Text.

Page Control and Dialog: Normally, there is at most one Page Control or Dialog on a screen. When there are multiple, compute the distance (normalized with screen width) between T_{ui} and N_{ui}. The similarity score is $1 - distance$.

Container: If a Container only contains Icons, use the template matching procedure. Otherwise, run the Text matching above on each contained Text (in reading order).

If no UI elements on N_s pass a threshold[3], then there is no match. Otherwise, we pick the UI element with the highest similarity on N_s. When a Text or Icon is in a grouping, our method first determines if the grouping is a match, and then prioritizes candidates in the same grouping.

4.4 Building the Final Report

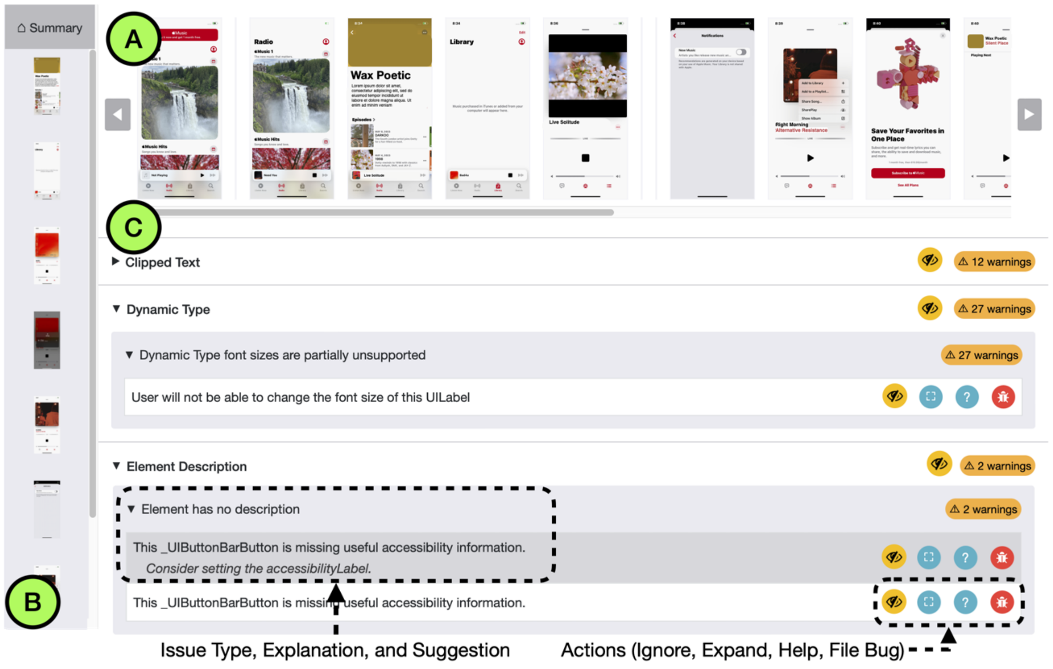
Finally, the report generator builds the final report by detecting and hiding previously ignored issues and filtering false positives (Figure 2.d-e). These are key stages towards design goal *D3* of reducing noise in the report.

First, the report generator retrieves the ignored issues, elements, and screens from a database. Report users can save these using the interface pictured in Figure 5 by clicking the Ignore button (represented by an eye icon). For each ignored issue, the report generator finds the matching screen it is located on in the report using the screen grouping model. On the matching screen, the report generator uses the UI element de-duplication heuristics (subsection 4.3.2) to find a matching UI element. If any of the same issue types are already reported for this UI element, the report generator marks them as "ignored" and moves them to a collapsed section.

Second, the report generator filters false positives using a basic heuristic (Figure 2.e). Any issue reported in the scanner's results with no visible UI element overlapping it is assumed to be a false positive and the report generator hides it in the output report.

[2]S = $\frac{T_s width}{N_s width}$; Scales = [0.91*S, 0.94*S, 0.97*S, 1.0*S, 1.03*S, 1.06*S, 1.09*S]

[3]Text: 90%, Icon: 80%, Picture: 50%, as determined empirically on our screen grouping dataset detailed in subsection 4.2


Summary
A
B
C
Wax Poetic
Clipped Text
12 warnings
Dynamic Type
27 warnings
Dynamic Type font sizes are partially unsupported
27 warnings
Element Description
2 warnings
Element has no description
2 warnings
Issue Type, Explanation, and Suggestion
Actions (Ignore, Expand, Help, File Bug)

User will not be able to change the font size of this UILabel

This _UIButtonBarButton is missing useful accessibility information.
Consider setting the accessibilityLabel.

This _UIButtonBarButton is missing useful accessibility information.

Fig. 5. The prototype interactive HTML report generated by our report generation system. The interface has three main areas: a) A carousel displaying all screens captured in the report, b) A menu to toggle between the "Summary" view which is currently selected, and screen-by-screen views represented by thumbnail screenshots, and c) a table of summarized results of issues found, grouped by category. The report provides actions users can take including ignoring, viewing suggestions, filing bugs, and expanding the screenshot.

The report generator processes the underlying screen and audit data per screen group detected in the storyboard generation step, and assigns a screen group identifier to each screen within a group. It then produces a self-contained JSON file with a summary of unique issues detected for each screen group of the app and an overall summary across screens categorized by accessibility issue category (e.g., Element Description, Dynamic Type) and subcategory (e.g., Element has no description, Dynamic Type partially unsupported). The JSON is then transformed into an HTML report, shown in Figure 5 which groups the results by each screen group into tabs along the side. It also provides a "Summary" tab that contains a table of all results across the app and a row of all screens across the top. Each screen tab, represented by thumbnail screenshots of screen in the group, shows a similar view as the "Summary" tab within only the results for that screen group and the screen(s) within it. Since our system generates the report in a self-contained JSON file, the report could be further processed in a continuous integration pipeline in the future.

4.4.1 Report Interface. The interface displays a summary table to explore all issues found across the app (Figure 5.C) which it categorizes by issue type (e.g., Element Description) with an overall count of each category. A person examining the report can click categories or issue headers to display all screens impacted by the issue (in Figure 5.A). The report interface highlights impacted UI elements on screens currently being inspected. To view the results for a specific screen, the report user can click through each screen tab along the side (Figure 5.B), which the report visualizes

	P	R	F1	Acc.	T(s)
Feiz Dataset - Baseline					
SSim Transformer (FD test set)	76.9%	74.8%	75.8%	92%	-
SSim Transformer (1k test set)	77.1%	87.1%	81.8%	94.5%	-
New Dataset - Our Work					
SSim Transformer (1k)	82.2%	94.1%	87.7%	96.3%	-
SSim Transformer (6k)	89.5%	88.2%	**88.8%**	**96.9%**	42.2s
SSim Bi-Encoder (Embedding) (6k)	91.1%	81.2%	**85.9%**	**96.7%**	5s

Table 5. Performance results for the screen similarity (SSim) transformer and Bi-Encoder embedding based models (distance threshold 0.34) demonstrating an improvement from our work of 13% in F1 score from the baseline.

with small thumbnail images. The reports on each screen tab are presented similarly to those on the summary tab.

The interface provides a few options to triage and reduce noise in future reports (design goal *D3*), including bug filing, or ignoring specific issues, issue type, category, or screens in a future report. The interface hides the ignores (in a collapsed section) for this app using the screen grouping model and UI element de-duplication heuristics as previously described. The ignores can be removed at anytime through a separate section. Beside each issue row, the report UI includes a question mark button which provides additional info and fix suggestions for the issue. The interface also displays a storyboard of the screens of the app captured during data collection on a separate tab in the report (not pictured but similar to Figure 3).

5 TECHNICAL EVALUATION

To evaluate the accuracy of our report generator, we evaluated the technical accuracy of two of its subcomponents – our screen grouping model and our UI element de-duplication heuristics. For the screen grouping model, we report the overall results on the test dataset, shown in Table 5, and for UI element de-duplication, we collected and evaluated the heuristics on a large evaluation dataset.

5.1 Screen Grouping Model

Overall, we can see that the transformer trained on 6k is the best performing model (Table 5), although the transformer trained on 1k performs surprisingly similarly. The transformers trained with new annotations also perform noticeably better than those trained on the old annotations.

We also evaluated the transformer trained on FD (Feiz dataset [21]) with the test set from 1k. Interestingly, this model performs better on the 1k test set than the FD test set, which may indicate it was able to learn some concepts through annotation noise in FD that were more applicable in 1k.

The Bi-Encoder embedding model, also trained on the full 6k dataset, performs slightly worse but still very close in accuracy to the transformer model. For the Bi-Encoder model, we use a distance threshold of 0.34 for "same screen" (experimentally determined in our model evaluation). This model achieves within 3% F1-score of the similarity transformer and is able to generate a storyboard 8.4 times faster than the similarity transformer.

Depending on user preference, our system can adopt different models. We currently use the 6k similarity transformer in our systems as our users noted that accuracy was more important to them in our formative studies. However, in cases where the system requires quick performance for generating reports for larger apps, systems can adopt the Bi-Encoder embedding model for improved system efficiency.

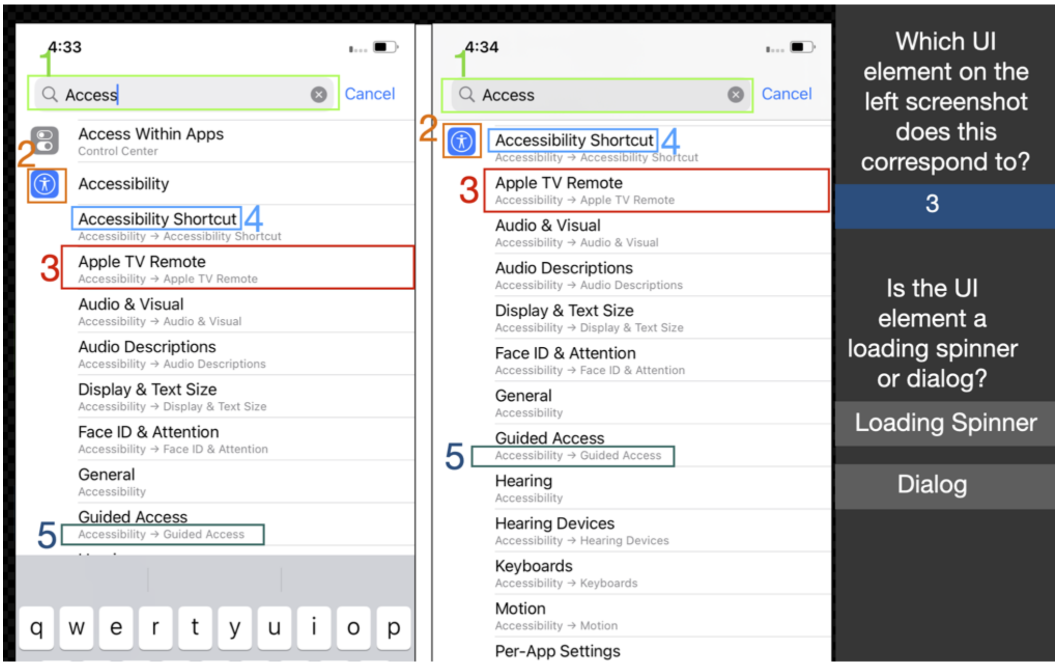

4:33
Access
Cancel
Access Within Apps
Control Center
Accessibility
Accessibility Shortcut
Accessibility → Accessibility Shortcut
Apple TV Remote
Accessibility → Apple TV Remote
Audio & Visual
Accessibility → Audio & Visual
Audio Descriptions
Accessibility → Audio Descriptions
Display & Text Size
Accessibility → Display & Text Size
Face ID & Attention
Accessibility → Face ID & Attention
General
Accessibility
Guided Access
Accessibility → Guided Access
q w e r t y u i o p
4:34
Access
Cancel
Accessibility Shortcut
Accessibility → Accessibility Shortcut
Apple TV Remote
Accessibility → Apple TV Remote
Audio & Visual
Accessibility → Audio & Visual
Audio Descriptions
Accessibility → Audio Descriptions
Display & Text Size
Accessibility → Display & Text Size
Face ID & Attention
Accessibility → Face ID & Attention
General
Accessibility
Guided Access
Accessibility → Guided Access
Hearing
Accessibility
Hearing Devices
Accessibility → Hearing Devices
Keyboards
Accessibility → Keyboards
Motion
Accessibility → Motion
Per-App Settings
1
2
3
4
5
Which UI element on the left screenshot does this correspond to?
3
Is the UI element a loading spinner or dialog?
Loading Spinner
Dialog

Fig. 6. The interface crowd workers used to label the evaluation set for UI element de-duplication. Each task displayed a left and right screenshot of same screen instances. Crowd workers drew boxes and labeled corresponding UI elements on the right screenshot, using the left as a baseline.

5.2 UI Element De-duplication Heuristics

To evaluate the accuracy of our UI de-duplication heuristics, we collected a dataset of 138k UI element correspondence labels across 25k same screen pairs from our annotated screen grouping dataset. We employed the same set of 15 annotators to label the data for this task as for our screen grouping model data collection (Section 4.2).

5.2.1 Data Collection & Annotation. The annotators used the labeling interface pictured in Figure 6. Each annotation task contained two screens. The left screen (T_s) showed 6-8 highlighted template UI elements with corresponding numbers of UI elements to be matched in the right screen (N_s). To train the annotators, we provided them with several examples of UI element correspondences and a set of definitions. Within each pair, our annotators consider two UI elements to be a match if they a) serve the same purpose (i.e., have the same functionality or convey the same information), b) are actionable and would lead to the same next screen in the app, and c) have the same grouping (e.g., an icon inside a container should be matched with the same icon and not the container). Each task, consisting of a pair of screens and UI elements to be matched, was annotated by a single annotator. We did not replicate these annotation tasks with multiple annotators. However, our data was also reviewed by the expert QA team (mentioned in Section 4.2), and they randomly reviewed the accuracy of batches of 10% of the data until a 98% accuracy threshold was reached. If the initial accuracy of the 10% did not not achieve the 98% threshold, all batches of data were reviewed and corrected by the 15 annotators and the QA expert annotators.

In the annotated dataset, half of the screen pairs (53.6%) are very similar (MSE [63] < 30). 4.4% of pairs are the same screen with some content scrolled, while the remaining pairs have other content

	Precision	Recall	F1	Time
Template Matching Only	87.5%	96.3%	91.7%	2.57s
Exact Text Matching Added	89.4%	95.4%	92.3%	1.11s
Fuzzy Text Matching Added	88.9%	96.8%	92.7%	1.12s
All Heuristics	**97.7%**	**98.7%**	**98.2%**	**0.35s**

Table 6. Performance results for UI element matching[4], reported by the subset of matching heuristics applied.

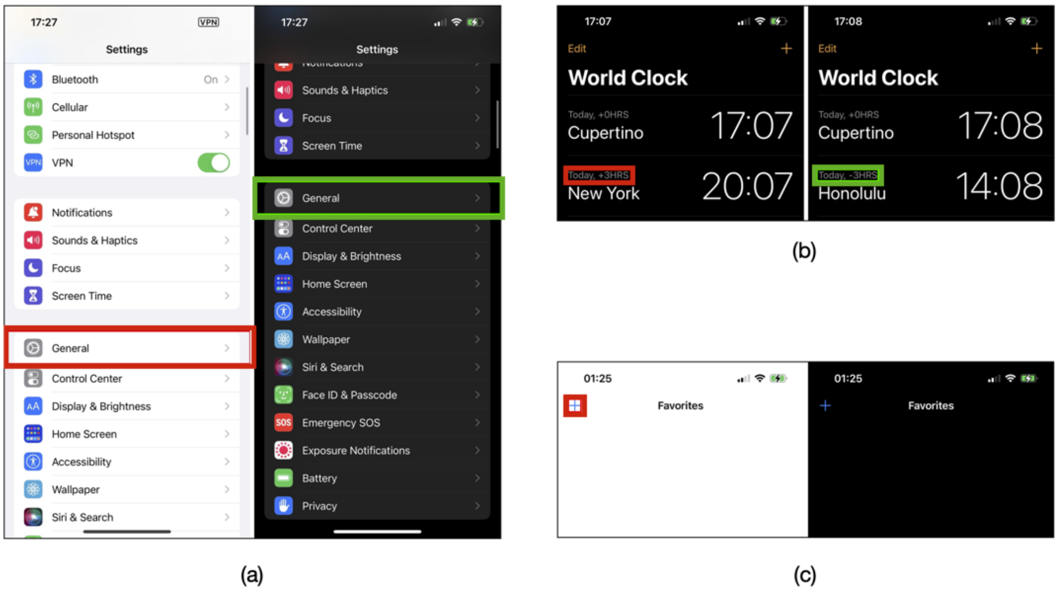


Fig. 7. Examples of True Positive (a), False Positive (b), and False Negative (c) of UI element de-duplication heuristic results. The red box indicates target UI on original screen, while the green box indicates the matched UI on the new screen. In (b), the text above New York and Honolulu was matched incorrectly. In (c), the heuristics (for icons, using template matching) did not find a match.

changes (e.g., added UI elements, removed UI elements, text content changes). In the end, 17,913 (13.0%) template UI elements (T_{ui}) do not have any matching UI element in the new screen (N_s).

We evaluated these heuristics on the 138k UI element matching annotations. To improve our heuristics, we examined and updated them on a small dataset of 991 screen pairs (5,420 template UIs), and then evaluated the heuristics on our full dataset. We report precision, recall, and F1-score metrics using the following definitions:

True Positive: The match found by our heuristics and the match found by the annotators are the same UI element.

True Negative: Our heuristics did not find a match and the annotators did not find a match.

False Positive: Our heuristics found a match but the annotators did not, or the match found by our heuristics and the match found by the annotators are NOT the same.

False Negative: Our heuristics did not find a match but the annotators found a match.

[4]The average matching time for each template UI element was measured on a Macbook Pro with 2.4 GHz 8-Core Intel i9 / 32G memory

Table 6 presents the results of this evaluation with a few variations. First, using template matching alone achieves 91.7% F1-score, in part because many non-Icon UI elements are not correctly matched. Next, we improve on the template matching method by adding exact text matching, which improves the F1-score by 0.6% and doubles the speed. Adding fuzzy text matching to the previous methods further improves the F1-score by 0.4%. Fuzzy text matching tolerates small mistakes introduced by OCR imperfections, but may also create error when there are Text elements with small differences (see appendix). None of the baselines approach our full set of heuristics, which achieved a F1-score of 98.2%.

We also examined the performance of our heuristics on three common cases of same screen pairs. For screens with very few differences our method achieves very high accuracy (99.2% Precision, 99.5% Recall, 99.4% F1-score). It also works well on scrolled screen pairs (91.7% Precision, 96.8% Recall, 94.2% F1-Score), and screen pairs with other content changes (95.1% Precision, 97.0% Recall, 96.1% F1-Score). We examined the failure cases and share common patterns in the appendix. Figure 7 shows an example of a true positive, false positive and a false negative match produced by our UI element de-duplication heuristics.

6 USER STUDY

To evaluate our report generation system, we conducted a user study to better understand how the system can impact QA testers and developers' ability to gain awareness and prioritize issues to fix. We evaluated the following research questions:

(1) How does the mode of accessibility scanning impact users' interpretation and summarization of accessibility issues?
(2) How do users perceive the quality of automatically generated reports collected using an app crawler?
(3) How might accessibility reporting tools support prioritization and quick discovery?
(4) How can accessibility reports fit into participants' workflows?

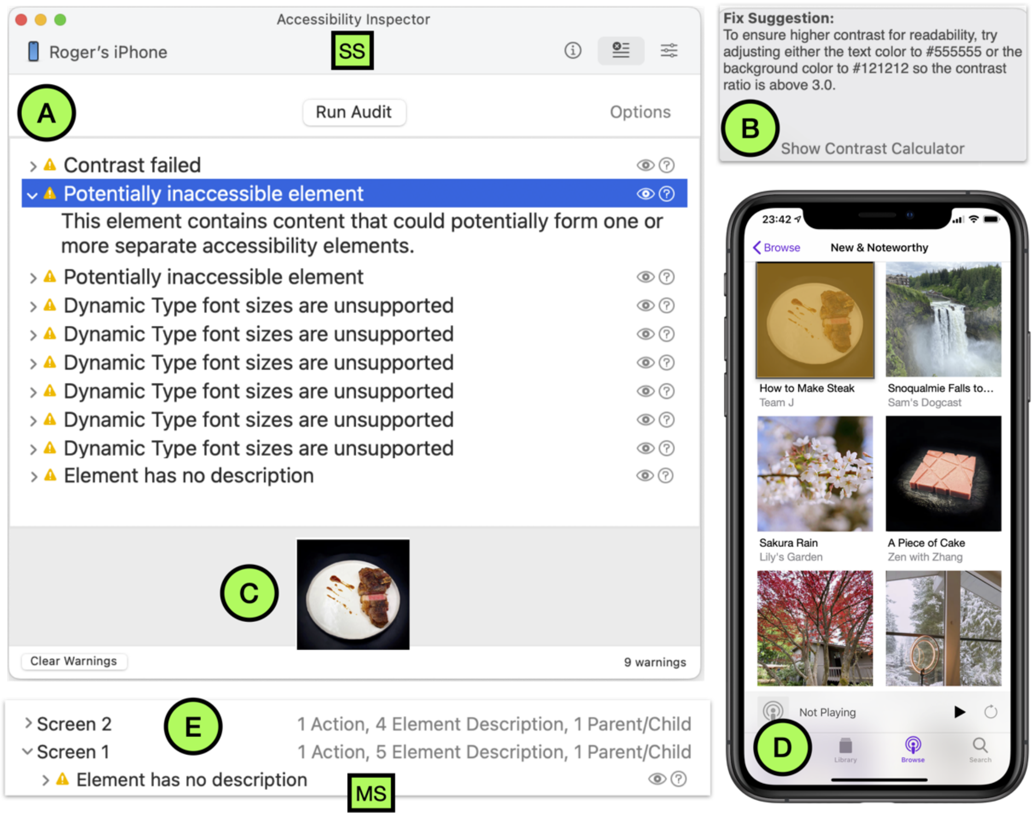
6.1 Participants

We recruited 19 (5F, 14M) participants across a large technology company to take part in the study, across varied roles including software engineer (9), QA or Automation engineer (7), accessibility evangelists (1), and managers (2). Participants mean self-rated expertise in iOS app development was 3.31 (Med: 4, Std: 1.6) and in accessibility testing was 3.8 (Med: 4, Std: 1.01). Participants self-reported their expertise in these two categories from a scale of 1 to 5, including: 1 - No experience, 2 - Beginner, 3 - Advanced Beginner, 4 - Intermediate, 5 - Expert. 18 participants were sighted, some used varying degrees of magnifications features, and 1 participant used a screen reader. None of these 19 participants participated in our formative study summarized in Section 3. We selected an entirely new group of participants for this study as the study occurred more than one year after the formative study.

6.2 Procedure

We first asked the participants to describe their prior experience in using accessibility testing and reporting tools. Then, participants completed 3 accessibility auditing tasks in a counterbalanced order. For each task, we instructed the participants to conduct accessibility audits of 3 different apps using 3 different sets of tools. The three sets of tools included:

- *Single screen accessibility inspector (SS)* - This tool, shown in Figure 8.A, supports single screen scanning with a button "Run Audit" which when clicked returns a list of possible issues for a


Accessibility Inspector

Roger's iPhone

SS

A

Run Audit

Options

Contrast failed

Potentially inaccessible element

This element contains content that could potentially form one or more separate accessibility elements.

Potentially inaccessible element

Dynamic Type font sizes are unsupported

Dynamic Type font sizes are unsupported

Dynamic Type font sizes are unsupported

Dynamic Type font sizes are unsupported

Dynamic Type font sizes are unsupported

Dynamic Type font sizes are unsupported

Element has no description

C

Clear Warnings

9 warnings

Fix Suggestion:
To ensure higher contrast for readability, try adjusting either the text color to #555555 or the background color to #121212 so the contrast ratio is above 3.0.

B

Show Contrast Calculator

Screen 2 — 1 Action, 4 Element Description, 1 Parent/Child

Screen 1 — 1 Action, 5 Element Description, 1 Parent/Child

Element has no description

E

MS

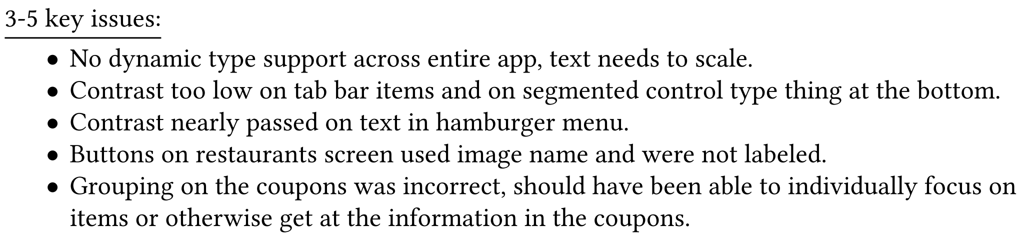
D

Fig. 8. Xcode's Accessibility Inspector [36] in single screen mode (SS) with the scanner's results for an iPhone. The Inspector has four main areas: (A) a table of detected issues for the current screen that a user requests by clicking the "Run Audit" button, (B) Fix suggestions that appear when a user clicks on the "?" icon on a selected row, (C) A preview window with a screenshot of the impacted UI element, and (D) a device (either on a simulator or live), or an application window. Our study evaluates two modes: SS, which provides features A-D, and MS, which provides features A-E. (E) adds a history of scanned screens and headers to summarize result counts by category.

screen across 29 possible issue types. Clicking "Run Audit" clears the results from the prior scan.

- *Multi screen accessibility inspector (MS)* - Using SS, users cannot compare results across screens directly in the tool as results are cleared when scanning a new screen. With MS, we added features targeted to help users summarize and prioritize issues across multiple screens, while still requiring manual navigation and scanning. MS contains a header on the results for each screen with the number issues found in each category for that screen (e.g., 3 Element Description, 2 Contrast), and adds a history to track results from multiple screens (Figure 8.E).
- *App crawler with generated report (AC)* - This tool provides a pre-generated accessibility report, generated by our system's app crawler, with results hosted in a web page for participants to examine (see Figure 5 for an example report).

The output of the *SS* and *MS* tools also includes explanations of each issue type in a submenu, along with fix suggestions for that issue category (Figure 8). The *AC* tool includes the same



3-5 key issues:

- No dynamic type support across entire app, text needs to scale.
- Contrast too low on tab bar items and on segmented control type thing at the bottom.
- Contrast nearly passed on text in hamburger menu.
- Buttons on restaurants screen used image name and were not labeled.
- Grouping on the coupons was incorrect, should have been able to individually focus on items or otherwise get at the information in the coupons.

Fig. 9. An example of a list of key issues compiled by a participant in our user study for an accessibility auditing task, created with the assistance of the provided scanning and reporting tools - SS, MS, and AC. In this case, the participant used AC to audit app B.

descriptions and suggestions in the interface as shown in Figure 8. Before participants completed each of the three tasks, we showed them a brief (~2 min) tutorial video demonstrating the key features of each tool and how to use it.

6.2.1 Tasks. For each accessibility auditing task, participants used each of the three tools to conduct accessibility audits where the output was a list of 3-5 key issues across the app to be sent to a development team to fix, along with the output of any scanning or reporting tools. This type of task is common with accessibility auditing with automated tools where the auditer, often a QA tester, groups and prioritizes issues before handing them over to the developer [34]. We instructed participants to add context to the list to help the developers reproduce or interpret it, which could also include screenshots. Figure 9 shows an example list created by one of our participants.

All participants had some experience with accessibility testing, and should be familiar with the accessibility issue types detected by our tools. However, in they case they did not understand an issue reported by the tool, they could read the provided descriptions and suggestion information in each interface or ask the researcher conducting the study clarifying questions. We did not give them specific guidance for issues to focus on, as we wanted to see how they used our tools to prioritize the issues given their own expertise.

6.2.2 Task Workflow. Participants completed 3 such app auditing tasks where we assigned one tool (as listed in 6.2) and one app to audit for each task. We counterbalanced the order participants used each tool and the apps being audited using a Latin square ordering across participants.

As we found during the formative study, participants typically also manually validate issues found by accessibility scanning tools. Therefore, we also allowed participants to manually validate issues reported by tools using a locally attached iPhone. Participants could enable VoiceOver to manually validate VoiceOver related issues, for example, or they could toggle on accessibility settings such as Dynamic Type. During the study, we did see that participants did often turn on accessibility features to manually validate the issues reported by our tools.

In practice, it is challenging to recruit professionals in roles such as those of our participants for studies lasting longer than one hour, so we kept each session to 45 minutes in length. We gave participants six minutes to complete each auditing task. The reason for making this task six minutes was twofold. First, we wanted to see how our tools could support conducting quick app accessibility audits. Second, this was also the maximum time we could allow for each task and keep the total session length under 45 minutes while leaving time for the video tutorials and study context, transitioning between tasks, follow up interviews after each task, and a follow up interview and survey at the end of the study. We piloted the study several times to optimize the timing of each component.

After the participants completed the 3 tasks, we interviewed them and they completed a follow-up survey to compare and contrast their experiences using each available tool for the task.

6.2.3 Materials. For the auditing tasks, the participants audited three publicly available apps (which we refer to as app A, B, and C) from the top 100 apps for each of three categories in the App Store initially at random; and then to have apps with a roughly equivalent complexity and number of accessibility issues. The selected apps are from the categories of: Food & Drink (A), Shopping (B), and Sports (C). We ran our app crawler on each app to produce a report (like that seen in Figure 5). The number of screens found in each app was 39, 63, and 58 for apps A, B, and C, respectively. The accessibility report (AC) surfaced 444, 596, and 522 accessibility warnings, respectively.

6.2.4 Design & Analysis. We recorded audio for each session and took notes on how many screens each participant scanned for the SS and MS tasks. We saved each participant's accessibility audit list of 3-5 key issues in a document. Two researchers conducted a qualitative thematic analysis [32] of the interview transcripts using an inductive method. Two authors also rated the quality of listed issues with a rubric to measure the specificity, scope, and importantness of the issues reported by participants in their lists. We detail this rubric along with the results of RQ1. We statistically analyzed the results of the ratings using an aligned rank transform analysis [71]. Some of the survey questions included Likert scale responses, which we also statistically analyzed using an aligned rank transform [71].

6.2.5 Ethical Review and Consent. A user studies review board internal to our company reviewed and approved our study. This board reviews the ethics of all human subjects studies and data collection procedures conducted within our company. All participants in our study signed an approved consent form prior to the session and were able to freely end the study at any time they chose.

6.2.6 Apparatus. The participants completed the tasks on a MacBook pro running an M1 Max chip and MacOS Sonoma. They used a locally attached iPhone, running iOS 17.0, to run the accessibility auditing tools using SS and MS modes of the Accessibility Inspector, shown in Figure 8. We pre-installed the apps for each auditing task on the device. Since the Accessibility Inspector is hosted as an Xcode developer tool, Xcode was also running on the MacBook. The participants also had a Notes file open to create their issue lists, and Safari was open to a tab showing the AC report. The studies took place in person in a conference room. A researcher, also an author on the paper, was in the room to facilitate the sessions and interview the participants. The researcher also took notes and recorded audio. For some sessions, a second researcher was present in person or remotely via Webex to observe and take notes.

6.3 Results

In this section, we summarize the results per each research question we examined in the study. For any numerical results including comparisons of all three scanning tools, we include the results for 18 of the 19 participants. We report the screen reader user's results separately as they only completed two tasks during the allotted time for the session. However, they provided valuable feedback that may make our system more accessible in the future to screen-reader users. We also included some of their feedback in our qualitative analysis reported for RQ3 and RQ4.

6.3.1 RQ1: How does the mode of accessibility scanning impact users' interpretation and summarization of accessibility issues? First, we report which tool the participants preferred the most out of the three auditing tools they used. Overall, 13 participants preferred AC (app crawler) the most, while 2 preferred MS (Multi-screen accessibility inspector) and 1 preferred SS (Single-screen accessibility inspector). Two participants did not prefer any specific tool. One of those two participants mentioned they still preferred manual testing over all tools as they thought it would be faster to move through the issues by hand. The other participant mentioned seeing value in all three tools for different tasks.

Participants gave varied reasons for preferring AC the most such as "giving a more holistic overview", "saving manual effort", "removing the dependency on the Xcode and the device", being "more sharable", and "reducing friction and context switching".

Which tool helped participants create a better accessibility audit list? To better understand how each tool impacted the participants' key takeaways from their audits, which they compiled in the form of accessibility audit lists, we collected the users' satisfaction ratings with their lists. Participants were overall more satisfied (using a 5-point Likert scale for satisfaction) with their lists created using AC (Mean: 4.27, Med: 4, Std: 0.75, n=18), compared to MS (Mean: 3.61, Med: 3, Std: 0.91, n=18) and SS (Mean: 3.5, Med: 3, Std: 0.86, n=18). We ran a non-parametric analysis of variance on the Likert scale data using the Aligned Rank Transform [71] with *Tool*, and *App* as factors and *Satisfaction* as the output response. The ART analysis indicated no statistically significant effect on *Satisfaction* of *App* (F(2, 28) = 0.52 = n.s.), but there was a significant effect of *Tool* (F(2, 28) = 5.09, p = 0.024). Posthoc pairwise comparisons ran using the ART-C procedure [19], and corrected with Holm's sequential Bonferroni procedure, indicated that AC versus MS was significant (t(28) = 3.17, p = 0.007) and AC versus SS was also significant (t(28) = 3.49, p = 0.005). No other pairwise comparisons were significant.

The accessibility audit lists (Figure 9 shows an example) contained issues related to dynamic type, missing labels, poor contrast, and small target size which are all issues reported by the underlying scanner. We also manually validated a subset of these issues ourselves on the apps and found the vast majority of them were real issues that should be fixed. However, participants also found and listed some issues through manual testing on the provided device. As satisfaction can be subjective, we also evaluated the contents of the lists using a rubric we developed. Two authors rated the issues in each list, without awareness of which app and condition the list was created with. The authors' rubric consisted of three categories - *Specific*, *High Level*, and *Important*. For *Specific*, we rated each issue on how easily we could deduce the UI elements or screens impacted by the issue. For high level, we rated each issue on whether it applied to a single element (1), a single screen (2), or multiple screens across the app (3). For *Important*, we rated the severity of the issue based on whether it would block any major usage of the app for key accessibility features (e.g., Voice Over, large text) on a scale of 1 to 3. Our goal through this rubric was to evaluate whether any particular tool helped participants create lists covering a wider scope of important issues across the apps.

Each rater rated all 163 issues listed by the participants along these dimensions. After rating, if any rating differed by at least two for a category, the raters discussed the grading and resolved disagreements if possible. Ultimately, the raters achieved an IRR (Percent Agreement) of 0.80 and IRR for ratings differing by 1 or less was 0.99.

Overall, the mean ratings per listed issue across all three modes were very similar for *Specific* – AC was 2.38 (Med: 2.5, Std: 0.61) while MS was 2.37 (Med: 2.5, Std: 0.65) and SS was 2.32 (Med: 2.5, Std 0.63). However, there was a larger difference for *High-Level* – The ratings for AC (Mean: 2.12, Med: 2, Std: 0.79) were 9% higher than MS (Mean: 1.95, Med: 2, Std: 0.72) and 16% higher than SS (Mean: 1.83, Med: 1.75, Std: 0.78). Finally for *Important*, the ratings for AC (Mean: 2.23, Med: 2.5, Std: 0.7) were 13.3% more than MS (Mean: 1.97, Med: 2, Std: 0.76) and SS (Mean: 1.93, Med: 2, 0.74). While there were differences between the means between tools, we found these differences not significant using the Aligned Rank Transform [71] for non-parametric data with *Tool*, *Experience*, and *App* as factors.

6.3.2 RQ2: How do users perceive the quality of automatically generated reports collected using an app crawler (AC)?. Overall, participants rated the AC report as "clean" (Likert scale, 5: Very clean, 1: Very messy) with a median rating of 4 (Mean: 3.85) and rated the screen grouping quality (Likert scale, 5: Very accurate, 1: Not accurate) as accurate (Med: 4, Mean: 3.65). When rating screen grouping quality, we asked participants to open the AC report and showed them a few examples of grouped screens across the report, as most participants did not notice this feature right away. Participants were also not highly familiar with the structure of each app and as such their responses to this question may not be as grounded as they might if they were the original developers of the apps.

6.3.3 RQ3: How might accessibility reporting tools support prioritization and quick discovery? We additionally examined whether any particular tool or features of any tool helped in prioritizing and summarizing issues by asking the following questions:

- Which tool helped you discover the issues *most quickly*?
- Which tool helped you most to find the *most common* issues?
- Which tool helped you most to prioritize the *most important* issues?

Which tool helped you discover the issues most quickly? The app crawler was rated as the most helpful tool in discovering the issues most quickly by 14 participants. Participants were also able to scan an order of magnitude more screens across the app using the crawler (A: 39, B: 63, and C: 58) vs the other modes. Using SS, participants scanned 3.9 screens on average (Med: 3, Std: 2.78) and with MS they scanned 4.5 screens on average (Med: 4, Std: 2.91). Participants (n=12) noted the manual modes could be a bottleneck, time consuming, and tedious, especially with limited time allocated during the study.

P2: *"(SS) took too much time and i wouldn't have been able to even get all the views up to look through them in time ... It was a bottleneck to have to keep loading and running the tool on each view."*

Another aspect of effort saved on the AC mode participants noted (n=5) was more hypothetical future use cases in that the report would ultimately "reduce friction" by removing dependencies on setting up a device for testing.

P1: *"I don't have to worry about being on the latest version or any incompatibilities ... I can just go to the website with all the data, interact with it, file my bugs and then go from there."*

Participants (n=6) also felt they were able to get more coverage and more information across the app to make decisions with the app crawler as compared to the other modes.

P13: *"But, yeah, I guess for even for, for contrast issues and stuff, like, the last tool is really great, because I felt confident that we got like, a really good look at the entire app."*

Which tool helped you most in finding the most common issues? The majority of participants reported that AC best helped them to find the most common issues (14 participants) across the app. Participants (n=8) noted that AC assisted them in finding the most common issues by providing a summary and counts for each category across the app. Participants noted that the counts and summary helped them to spot more prevalent issues across the app, like missing support for Dynamic Type, a pervasive issue among the apps tested in our study.

P12: *"You can click here if you want to know the count, but I'm gonna give you the full list and and after while you're going to be able to just to answer this and be like yeah, we got a big problem with the Dynamic Type. I just thought that was such a great delivery."*

Counts of issues and a "history" of scan results were also helpful to participants when using MS. Two participants mentioned this helped them find the common issues most by looking for issues appearing in multiple scans.

P9: *"I guess in the multiple, like, for example, if I'm seeing this area is too small and I'm seeing it multiple times. I kind of like, okay, this is maybe like a main issue on this."*

Which tool helped you most to prioritize the most important issues? When asking participants which tool helped them find *more important* issues, 7 participants chose AC while the remaining chose MS (6), SS (1) and Multiple Tools (2) and None of the above (2). Participants that chose either MS, SS or Multiple Tools mentioned that because the AC randomly explored the app, they had no control over whether important user tasks were covered in the app. In the other modes, they could quickly review key user scenarios by controlling which screens they scanned.

P14: *"If we can, if we can attach a process to this, and then I'm just going through, let's say, or an ordering flow, and then generates the report for this ordering flow ... let's just say, hypothetically, my team is in charge of building the order flow. I'm not going to care about accessibility of the other screens"*

This suggests that participants may benefit from having a mode that supports both *automated exploration* and *control* over which user scenarios and tasks are explored by the app crawler.

Prioritization and Discovery: Strengths and Potential Improvements. One key theme in our study was that a high level report across the app gave participants new capabilities and benefits compared to manual scanning modes. Participants (n=14) mentioned that the overview, summary, and total counts of the AC report helped them strategize and prioritize issues to fix. Total counts helped P19 – "I'm just going through and kind of looking at I'm trying to look at where are the warnings are the highest?" – to organize how they looked through the report. By clicking on categories in the summary tab, participants could view the total counts of issues and visualize all impacted screens at once. This helped them discover higher level patterns of issues.

While participants in general felt that the AC report helped them prioritize and report the issues across the apps more easily than the other modes, they gave several creative and insightful suggestions about how to make the reports more useful and more interpretable for future developers.

Several participants (n=6) in the study had questions about particular issues and what they mean, suggesting that the fix suggestions (such as those shown in Figure 8.B) provided in the AC report and the SS and MS tools could be clearer, provide more detail, or "link to additional resources" (P12).

Several participants also found the report initially to be "overwhelming" (n=5) and noted that it might discourage developers to see a big number of issues, such as P2 – "oh, now we have a, I got a huge list of, like, you know, 500 accessibility issues. Which ones do we start with?". These participants also gave suggestions for how the system could better prioritize issues to make them less overwhelming. Some participants recommended assigning severity or priority ratings to issues, or reporting statistics that emphasize the magnitude of impact on a population.

P14: *"Maybe we can actually empower people to get some, some stuff done. I mean, saying there's a 450 dynamic type issues versus saying oh, there's X number of 100,000 people that won't be able to use your your tool. This many of your bugs affect people with low vision, this many affect users of screen readers."*

Other participants recommended prioritizing issues based on the frequency of usage or importance of the UI elements impacted by the issue. This also relates to P13's suggestion which highlights the value of integrating screen and UI element identification into the report which can enable understanding the scope of issues across the app and report themes and higher level insights.

> P13: *"Like, if it went through every screen, it was like, okay, like, 90% of the elements didn't support dynamic type then probably like limited dynamic type for support altogether. Yeah. So, it'd be nice to just, you know, pull up these into one area for dynamic type."*

While our current system does not provide severity ratings or any of these suggested prioritization modes, the technology our system introduces (e.g., screen similarity and grouping, UI element matching) can potentially enable reporting these higher level themes automatically, and we plan to integrate some of these ideas into future versions of our system.

6.3.4 RQ4: How can accessibility reports fit into participants' workflows? One common theme coming up across the interview questions was participants hypothesizing about workflows where MS and AC modes might be useful. The AC report was seen as particularly useful for QA and reporting use cases (n=9), such as tracking the accessibility of the app over time or computing stats and trends.

> P16: *"Like, is there a way to do a checklist? So once you make fixes, can you show, like, how it's been fixed over time or changing over time, or something like that?"*

Related to this, participants (n=4) wanted to integrate AC into their continuous integration workflows, which would enable them to run the reports on a regular basis, across multiple devices and settings (e.g., dark and light mode), or for testing accessibility across multiple languages.

Supporting triaging and marking issues as ignored over time was also noted as an important feature in long term use, as participants mentioned they might be likely to file bugs or find issues in the report that they would mark as "won't fix" or "minor issues". These issues should then be filtered out of future reports automatically.

Aside from it's usefulness in QA and continuous integration, several participants (n=7, mostly software engineers) desired for the tool to be in Xcode where they typically develop their apps. As opposed to dynamic accessibility scanning which runs on a built app like our system, some participants desired for report generation while building the app or on specific screens when possible, similar to the functionality of existing accessibility linters [28] that statically analyze code for accessibility issues.

While it is possible to detect some accessibility issues through static analysis, some classes of issues need to be detected on a running app (e.g., contrast issues, layout or dynamic type resizing), and so systems should support how to best combine the benefits gained from dynamic audits at a higher level with the direct benefits of live code editing.

6.4 Screen Reader User Feedback

One participant in our study is a screen reader user who is legally blind. This participants role in the company is accessibility evangelist, and they self-rated their expertise in accessibility testing as Expert (5). The participant first encountered difficulties in setting up their external keyboard they rely on for screen navigation which cost a few minutes of time for the study session. It also took them longer to navigate the interfaces using VoiceOver on the Mac. Thus they were only able to complete two tasks during the 45-minute session, so we report the results separately. Their feedback reveals some benefits our system gave this participant over other scanning tools and areas of future improvement. First, a key benefit this participant noted was that for very inaccessible apps, our system would let them scan more screens compared to manual scanning tools that require use of the VoiceOver screen reader to navigate the app to each screen for scanning. Unexposed elements in very inaccessible apps might cause them to miss scanning key areas of the apps they would be unable to navigate.

However, as our system does not yet provide automatic alt-text to describe each screen at a high level, the participant struggled to establish context to navigate the report to determine which issues belonged to which screens in the app, which would be necessary to include when filing a bug report. In the future, we could integrate more recent UI understanding technologies that add screen descriptions or structure to generate better alt text [70, 73].

7 EXPERIENCE REPORTS

In our user study, participants audited apps they did not own themselves. Thus, we also gathered experience reports from five app developers (5M, between ages of 22 and 45) within our organization to understand how they might leverage our manual scanning mode (MS in our user study) and app crawler generated reports (AC). We generated AC reports to help them evaluate their own apps' accessibility and gave them our MS tool to use on their own apps (see the multi-screen version inspector in Figure 8.E). The apps they worked on were internal apps used within our organization for bug filing, device and build management, and sample apps from documentation. Each developer we interviewed did have QA in place for accessibility testing, although the majority of them (four) lacked automation tests and primarily relied on manual testing.

Issues Found Each developer looked through the AC report and noted any issues they found for which they might file a bug. The mean number of screens in these reports among the five apps was 30.4. Since the developers were already highly aware of accessibility, most of their apps did not have pervasive accessibility issues. However, each located screens in their app lacking Dynamic Type support, and at least one UI element with an incorrect or missing accessibility label. The developers typically manually verified these issues themselves outside of our tool, which may be due to lack of trust in prior tools which provided false positives. However, a common thread of feedback on both MS and AC tools was that the reports contained issues (e.g., low contrast, target size) flagged on system controls or system provided dialogs or screens they had no control over. A challenge for future versions of our system is to filter these out or report them separately.

Workflows Three developers noted some issues in the report they had decided to not fix, or had alternative solutions for after discussing with accessibility QA. They would ignore these issues if this tool was integrated into their workflows. All developers expressed excitement about the AC reports, requested access to it, and envisioned using it in their workflows. Developers thought that MS was better than prior versions of this tool (single screen mode), but would prefer to use the AC report if available for CI or through a command line tool.

Coverage Three developers noted that the AC report contained all the key screens they were able to think of while two found that it missed capturing some key screens and areas of their apps. Using the MS tool, those developers scanned those screens themselves but did not find any additional issues they would file in those two cases. In future versions of our system, we intend to improve our app crawlers to obtain more complete coverage.

8 DISCUSSION

Overall, our system received positive feedback from both our internal stakeholders and participants in our studies, indicating that accessibility reports generated by our system helped them summarize and prioritize issues. In this section, we summarize the key results and implications, discuss limitations, and point to ideas for future work that can study the use of report generation tools within the broader context of accessibility testing workflows.

Addressing the Limitations of Current Accessibility Scanning Tools First, one key contribution of our work is a better understanding of the limitations and inefficiencies in accessibility scanning tools. We re-framed these as design goals we instantiated in our second contribution – our system that uses state-of-the-art UI understanding models and heuristics to provide automation and summarization of issues to address these limitations.

Lastly, we evaluated our system with a large group of 19 app developers and QA testers and a smaller group of developers who provided experience reports. The findings were twofold - first, the participants were able to more quickly find and prioritize accessibility issues using our system versus a baseline single-screen auditing tool. This indicates that our report generation system and features, including our multi-screen (MS) and app crawler (AC) modes, can alleviate the effort required for single screen scanning tools while providing a better overview and summarization. Second, the findings reveal new insights about how to make such reports more effective in future systems.

Prior works reporting accessibility issues using app crawlers [18, 58, 59] have not studied users' interactions with their output in depth, so our study results contribute to a gap in the literature on how developers and QA testers can consume such reports. Currently, there are still large gaps in accessibility across app ecosystems [4, 56, 64, 74]. A key goal of our work is to reduce the scale of these issues over time by providing better tools for developers that can highlight these issues and motivate developers to fix them.

Reducing Effort and Enabling Finding Patterns Participants in our user study used our app crawler, i.e. AC, to help them discover and find the most common accessibility issues quickly, enabling them to audit an order of magnitude screens in the same amount of time as the modes we compared this to (i.e., MS and SS which require manually navigating through the app). Participants found the manual modes to be a bottleneck and time consuming for quick auditing efforts. Aside from saving developers time and effort, our system, including both MS and AC modes, can enable developers to find patterns in the accessibility audits when similar issues appear across multiple screens. The MS mode also adds a history of audits and counts which can enable users to find patterns in the data while also enabling them to to control which screens they audit as opposed to AC which reports screens through random exploration through the app. Future work should explore how to best support combining the benefits of random exploration while providing some level of control over which screens and user scenarios are included in the audit reports. A recent body of work exploits large language models [50] to interpret and automate tasks in UIs without any pre-training or fine-tuning [22, 62, 69]. This work could potentially be combined with our system to generate more targeted accessibility reports.

Supporting Prioritization through an Overview Participants were also significantly more satisfied with the audit lists they created using AC than the lists they created using the other modes, indicating that AC can help them quickly create satisfactory accessibility audit summaries they can share with others. Participants also rated the generated reports from AC to be clean and accurate. While the analysis of the contents of the lists did not reveal any statistically significant differences, the summary lists created using AC covered a wider scope of more important issues across the apps. The participants also noted that AC could help them find common issues in apps quickly to better prioritize what to fix. Prior work notes the benefits of accessibility auditing tools both to help developers in understanding how to improve their app's accessibility and as a resource for learning about accessibility [34]. However, users of these tools can become overwhelmed and demotivated when looking at the results which are not typically grouped or summarized effectively. Our results imply that systems like ours can better help developers and QA testers quickly narrow in on key accessibility issues to fix, which can potentially make these results more motivating and less overwhelming to fix.

Another major outcome of our study was in revealing participants insights and creative suggestions on how reports generated by our system can better support prioritization and discovery, such as providing contextualized fix suggestions, severity ratings, impact on users with specific disabilities, or reporting the scope of issues across the app. Some participants did find the reports "overwhelming" for the apps they audited, which did have a large number of accessibility issues. It will be important for future systems to incorporate these suggestions to make the results less overwhelming.

Integration into Workflows Participants in both our user study and in our experience reports (Section 7) desired to incorporate our system's reports into their workflows in various ways to enable new capabilities of tracking accessibility over time, computing trends, or scaling up testing across various settings (e.g., display, language). These are capabilities that most of them do not currently have in their accessibility testing toolkit, which our system can add. These findings suggest that app developers and QA testers, who are all involved in accessibility testing, can benefit from improved accessibility report generation tools to report and summarize basic issues. In turn, they may be more motivated to use these tools at all and can free up time in testing to focus on more complex accessibility testing that can be difficult to automate. Another way we could motivate participants to use these tools is by bringing them closer to their development environments, as indicated by our user study. Future work could explore linking these reports back to provided code suggestions for fixing the issues.

Supporting Highly Inaccessible Apps Another limitation of these prior systems is that they rely on accessible view hierarchies to drive their automation, which for inaccessible apps are often unavailable or incomplete [40, 74]. A major motivation for our work is to build a system that can report accessibility issues found in all kinds of apps, most importantly highly inaccessible apps. Hence, our system operates on the visible UI by using a machine learning UI detection model to explore the app which can enable it to explore and capture data from a larger scope of an inaccessible app. Future work should compare or even combine these two approaches to more effectively cover both accessible and inaccessible apps.

8.1 Study Limitations

In our formative study, we primarily focused on eliciting the design goals based on the use of automated tools such as accessibility scanners. These tools are only one part of the whole workflow of accessibility testing, and future studies should examine other aspects of how these tools fit into larger testing workflows. Additionally, accessibility scanners, such as Accessibility Inspector [36] used by our system, do not cover every accessibility issue.

Our system can incorporate more automated checks [3, 44, 49] to expand coverage. Our system is also agnostic to the underlying accessibility scanner; thus it could also be easily extended to incorporate the results from multiple scanners, if available, to provide more coverage of key issues. Recent work on web platforms has provided frameworks for interoperability of web accessibility requirements, enabling accessibility reporting tools to incorporate multiple sets of accessibility guidelines [7]. Future work in a mobile context should study if a similar framework can be adopted.

However, even for the portion of the app that our system crawls, it still does not generate a complete report of accessibility issues and should not be interpreted as such. As automated tools are unlikely to ever find all accessibility issues [1, 2, 48, 52, 65], it will always be imperative to conduct other types of accessibility testing, including manual testing by specialists [1, 9] and people with disabilities [46]. However, as our study participants and internal stakeholders note, this tool can be a valuable complement to their current testing and might motivate them to use these tools more regularly. We also recognize that these tools should make it transparent to developers the types of accessibility issues that they can and cannot detect [45], which is a subject for future work.

Our studies primarily evaluate our report generation technologies, and the overall idea and features of the system over the design of the report itself and how it conveys information. However, our study revealed insights around how future systems might surface accessibility issues in reports. Some of these insights can directly leverage our introduced models to report the scope of issues across an app, for example. We may also be able to take inspiration from work on large scale reporting for web interfaces on how to more effectively convey the reported accessibility information to relevant audiences [53, 61].

Our user study sessions comparing AC to MS and SS scanning modes were short and thus may not reflect how participants would use them for thorough accessibility audits or comparisons over time. We have deployed this system to a small pilot group in our organization and intend to study its impact over time to better address this limitation.

Related, we motivated and evaluated our system against app auditing in the iOS platform. The participants we interviewed for our study were also from a single company; thus their experiences may be skewed towards the iOS platform and the related accessibility scanning tools. However, some literature already points to issues more generally with accessibility scanning tools, which extends beyond the iOS platform [34].

Our participants were also relatively familiar with app accessibility. Future work should evaluate and explore how to best present and prioritize summarized issues to make them easily understandable to developers not as familiar with accessibility as our study participants. This work could in turn enable future studies on whether providing automated reports can increase awareness and understanding of accessibility among app developers [34].

In our user studies, participants audited apps they did not own themselves potentially limiting our evaluation findings to QA testing contexts where testers often audit apps for which they are not responsible. However, we also hope our system can ultimately increase developers' awareness of the accessibility of their own apps. To begin to explore this, we collected experience reports from developers in our organization which indicated our tools could help them find real, high impact accessibility issues with their apps. They also revealed future improvements to our system such as better filtering of issues from system controls and improving coverage of key screens in our AC reports. They also indicated an interest in using our tools on an ongoing basis with their own apps. However, the question of whether such tools can increase their awareness of accessibility of their apps should be explored further in future studies.

8.2 Technical Limitations

While our screen grouping model achieves high accuracy, our data still contains many annotation errors which we would like to resolve to improve the model further. We also plan to collect additional crawl data for screen variations the model has difficulty predicting (e.g., scrolled screens, keyboard open / closed). So far, we have trained and evaluated this model only on iOS screens. Future work can examine whether this model can generalize across platforms on datasets like Rico [17]. While improving these models, we will continue to evaluate our system with users to study the impact of accuracy improvements. For future work, we will apply these models in other contexts beyond accessibility report generation, such as UI testing, design, and record & replay systems.

Additionally, we plan to further explore methods of accessibility report data collection. Our prototype supports both manual capture and random crawling. While we used the random crawler to produce reports for our study, we have not evaluated crawler coverage. We plan to further develop and leverage improved app crawling in future work, based on feedback from our user studies indicating that users would like more control over reporting specific application flows that are relevant to them.

8.3 Future Work

Future work should examine the use of automated reports along with other evaluation techniques (e.g., manual testing) and whether deploying such reports increases time for manual testing beyond basic features. An interesting question to study might be whether accessibility reports can drive down the number of basic accessibility issues over time, in the vein of studies from Fok et al. [23] and Ross et al. [56]. Other work could also study whether automated accessibility reports can enable developers and QA testers to more easily monitor apps for accessibility regressions over time, a potential indicated by our user study. While large scale evaluation and tracking of issues over time has not been widely studied for mobile platforms, systems like ours can also take inspiration from work on large scale accessibility evaluation on the web [35, 53]. This area of work has developed automated metrics for tracking issues [47], and presents design requirements for interfaces for large-scale accessibility evaluation and reporting [53]. The technologies we introduce in this work can provide a foundation for building accessibility evaluation platforms to track and report on accessibility issues over time for mobile apps.

The two key technical contributions of this work, screen grouping and UI element matching, achieved high accuracy. We plan to continue improving these components of our system in future work while also addressing the improvements noted by our stakeholders and study participants. We also plan to explore additional ways machine learning can be used to detect and report issues (e.g., grouping, navigation order) to make the report more informative. However, while surfacing more issues can be helpful, caution must be taken to prevent developers and QA testers from relying only on the results especially in cases where the system makes false negative predictions or fails to report an issue [15]. Ours and similar systems can provide model confidence scores and explanations to increase transparency and explainability in accordance with the guidelines from Amershi et al. [5].

9 CONCLUSION

In this paper, we presented a system to generate accessibility reports for mobile apps from a variety of input sources. The models and algorithms in our system achieved high accuracy on large datasets. Our screen grouping model and UI element matching methods may also have implications in a number of UI testing and interactive applications beyond report generation.

User studies of our system demonstrate this approach is promising and can provide value as a tool in the accessibility testing process for mobile apps. Future studies should explore how systems like ours fit into the larger workflow of accessibility testing which includes manual assessments and evaluations by end users with disabilities. Going forward, we hope systems like ours lead to improvements in the number of accessibility issues being found in large scale analyses [56, 74]. Going forward, we will continue improving the accuracy of our methods and usability of our reports to enable developers and QA testers to quickly comprehend key areas where their app's accessibility should be improved.

REFERENCES

[1] Julio Abascal, Myriam Arrue, and Xabier Valencia. 2019. Tools for web accessibility evaluation. *Web accessibility: a foundation for research* (2019), 479–503.

[2] Nancy Alajarmeh. 2022. The extent of mobile accessibility coverage in WCAG 2.1: sufficiency of success criteria and appropriateness of relevant conformance levels pertaining to accessibility problems encountered by users who are visually impaired. *Universal Access in the Information Society* 21, 2 (2022), 507–532.

[3] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2021. Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 730–742.

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

[5] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300233

[6] Max Bachmann. 2024. RapidFuzz: Rapid fuzzy string matching in Python using various string metrics. https://github.com/maxbachmann/RapidFuzz

[7] Giovanna Broccia, Marco Manca, Fabio Paternò, and Francesca Pulina. 2020. Flexible automatic support for web accessibility validation. *Proceedings of the ACM on Human-Computer Interaction* 4, EICS (2020), 1–24.

[8] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. VINS: Visual Search for Mobile User Interface Design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[9] Rocio Calvo, Faezeh Seyedarabi, and Andreas Savva. 2016. Beyond web content accessibility guidelines: expert accessibility reviews. In *Proceedings of the 7th international conference on software development and technologies for enhancing accessibility and fighting info-exclusion*. 77–84.

[10] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.

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

[12] Jia Chen, Ge Han, Shanqing Guo, and Wenrui Diao. 2018. Fragdroid: Automated user interface interaction with activity and fragment analysis in android applications. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 398–409.

[13] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Towards Complete Icon Labeling in Mobile Applications. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*.

[14] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2021. Accessible or Not An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3108162

[15] Valerie Chen, Q Vera Liao, Jennifer Wortman Vaughan, and Gagan Bansal. 2023. Understanding the role of human intuition on reliance in human-AI decision-making with explanations. *arXiv preprint arXiv:2301.07255* (2023).

[16] Luis Cruz, Rui Abreu, and David Lo. 2019. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering* 24, 4 (2019), 2438–2468.

[17] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*.

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

[19] Lisa A Elkin, Matthew Kay, James J Higgins, and Jacob O Wobbrock. 2021. An aligned rank transform procedure for multifactor contrast tests. In *The 34th annual ACM symposium on user interface software and technology*. 754–768.

[20] Evinced. 2024. Evinced, inc. https://www.evinced.com/

[21] Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. 2022. Understanding Screen Relationships from Screenshots of Smartphone Applications. In *27th International Conference on Intelligent User Interfaces*.

[22] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *46th International Conference on Software Engineering (ICSE)*.

[23] Raymond Fok, Mingyuan Zhong, Anne Spencer Ross, James Fogarty, and Jacob O Wobbrock. 2022. A Large-Scale Longitudinal Analysis of Missing Label Accessibility Failures in Android Apps. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–16.

[24] Google (Open Source Framework). 2022. Earl Grey: iOS UI Automation Test Framework. https://github.com/google/EarlGrey

[25] Anderson Canale Garcia, Silvana Maria Affonso de Lara, Lianna Mara Castro Duarte, Renata Pontin de Mattos Fortes, and Kamila Rios Da Hora Rodrigues. 2023. Early accessibility testing–an automated kit for Android developers. In *Proceedings of the 29th Brazilian Symposium on Multimedia and the Web*. 11–15.

[26] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. 2014. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv preprint arXiv:1402.4826* (2014).

[27] Google. 2021. Espresso. https://developer.android.com/training/testing/espresso

[28] Google. 2022. Android Lint. https://support.google.com/accessibility/android/answer/6376570

[29] Google. 2023. Get started on Android with Talkback. https://support.google.com/accessibility/android/answer/6283677?hl=ens

[30] Google. 2024. Get started with Accessibility Scanner - Android accessibility help. https://support.google.com/accessibility/android/answer/6376570

[31] Google. 2024. Lighthouse | Tools for Web Developers. https://developers.google.com/web/tools/lighthouse

[32] Greg Guest, Kathleen M MacQueen, and Emily E Namey. 2011. *Applied thematic analysis.* sage publications.

[33] Forrest Huang, John F Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.* 1–10.

[34] Syed Fatiul Huq, Abdulaziz Alshayban, Ziyao He, and Sam Malek. 2023. # A11yDev: Understanding Contemporary Software Accessibility Practices from Twitter Conversations. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–18.

[35] Nicola Iannuzzi, Marco Manca, Fabio Paternò, and Carmen Santoro. 2023. Large scale automatic web accessibility validation. In *Proceedings of the 2023 ACM Conference on Information Technology for Social Good.* 307–314.

[36] Apple Inc. 2022. Accessibility Programming Guide for OS X: Testing for Accessibility on OS X. https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html

[37] Apple Inc. 2023. XCTest. https://developer.apple.com/documentation/xctest/user_interface_tests

[38] Zexun Jiang, Ruifeng Kuang, Jiaying Gong, Hao Yin, Yongqiang Lyu, and Xu Zhang. 2018. What makes a great mobile app? A quantitative study using a new mobile crawler. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE).* IEEE, 222–227.

[39] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 1–10.

[40] Gang Li, Gilles Baechler, Manuel Tragut, and Yang Li. 2022. Learning to denoise raw mobile UI layouts for improving datasets at scale. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems.* 1–13.

[41] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 23–26.

[42] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1070–1073.

[43] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18).* ACM, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[44] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 398–409.

[45] Marco Manca, Vanessa Palumbo, Fabio Paternò, and Carmen Santoro. 2023. The transparency of automatic web accessibility evaluation tools: design criteria, state of the art, and user perception. *ACM Transactions on Accessible Computing* 16, 1 (2023), 1–36.

[46] Jennifer Mankoff, Holly Fait, and Tu Tran. 2005. Is your web page accessible? A comparative study of methods for assessing web page accessibility for the blind. In *Proceedings of the SIGCHI conference on Human factors in computing systems.* 41–50.

[47] Beatriz Martins and Carlos Duarte. 2022. Large-scale study of web accessibility metrics. *Universal Access in the Information Society* (2022), 1–24.

[48] Delvani Antônio Mateus, Carlos Alberto Silva, Marcelo Medeiros Eler, and André Pimenta Freire. 2020. Accessibility of mobile applications: evaluation by users with visual impairment and by automated tools. In *Proceedings of the 19th Brazilian Symposium on Human Factors in Computing Systems.* 1–10.

[49] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 107–118.

[50] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[51] OpenCV. 2018. Template matching. https://docs.opencv.org/4.0.0/de/da9/tutorial_template_matching.html

[52] Marian Padure and Costin Pribeanu. 2020. Comparing six free accessibility evaluation tools. *Informatica Economica* 24, 1 (2020), 15–25.

[53] Fabio Paternò, Francesca Pulina, Carmen Santoro, Henrike Gappa, and Yehya Mohamad. 2020. Requirements for large scale web accessibility evaluation. In *Computers Helping People with Special Needs: 17th International Conference, ICCHP 2020, Lecco, Italy, September 9–11, 2020, Proceedings, Part I 17*. Springer, 275–283.

[54] Christopher Power, André Freire, Helen Petrie, and David Swallow. 2012. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 433–442.

[55] Roboelectric (Open Source Project). 2021. Roboelectric. http://robolectric.org/

[56] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2018. Examining image-based button labeling for accessibility in Android apps through large-scale analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. 119–130.

[57] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–11.

[58] Navid Salehnamadi, Ziyao He, and Sam Malek. 2023. Assistive-Technology Aided Manual Accessibility Testing in Mobile Apps, Powered by Record-and-Replay. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–20.

[59] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[60] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. 286–293.

[61] David Mark Swallow, Helen Petrie, and Christopher Douglas Power. 2016. Understanding and supporting web developers: design and evaluation of a web accessibility information resource (WebAIR). In *Universal Design 2016: Learning from the past, designing for the future (Proceedings of the 3rd International Conference on Universal Design, UD2016)*. IOS Press, 482–491.

[62] Maryam Taeb, Amanda Swearngin, Eldon Schoop, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2024. AXNav: Replaying Accessibility Tests from Natural Language. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (Conditionally Accepted, will revise upon final acceptance)*.

[63] Todd Veldhuizen. 1998. https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/VELDHUIZEN/node18.html

[64] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. 2019. Can everyone use my app? an empirical study on accessibility in android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–52.

[65] Markel Vigo, Justin Brown, and Vivienne Conway. 2013. Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests. In *Proceedings of the 10th international cross-disciplinary conference on web accessibility*. 1–10.

[66] Aaron Richard Vontell. 2019. *Bility: automated accessibility testing for mobile applications*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[67] W3C. 2015. Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile. https://www.w3.org/TR/mobile-accessibility-mapping/

[68] W3C. 2023. Introduction to Understanding WCAG 2.0. https://www.w3.org/TR/UNDERSTANDING-WCAG20/intro.html

[69] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI Using Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 432, 17 pages. https://doi.org/10.1145/3544548.3580895

[70] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2words: Automatic mobile UI summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 498–510.

[71] Jacob O Wobbrock, Leah Findlater, Darren Gergle, and James J Higgins. 2011. The aligned rank transform for nonparametric factorial analyses using only anova procedures. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 143–146.

[72] Jason Wu, Rebecca Krosnick, Eldon Schoop, Amanda Swearngin, Jeffrey P. Bigham, and Jeffrey Nichols. 2024. Never-ending Learning of User Interfaces. In *The ACM Symposium on User Interface Software and Technology (UIST)*. ACM.

[73] Jason Wu, Xiaoyi Zhang, Jeffrey Nichols, and Jeffrey P. Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *Proceedings of the 2021 ACM Symposium on User Interface Software & Technology*

(UIST). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3472749.3474763

[74] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[75] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 609–621.

[76] Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 2023. Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1255–1267.