Assistive-Technology Aided Manual Accessibility Testing in Mobile Apps, Powered by Record-and-Replay

Navid Salehnamadi
nsalehna@uci.edu
School of Information and Computer Sciences
University of California, Irvine, USA

Ziyao He
ziyaoh5@uci.edu
School of Information and Computer Sciences
University of California, Irvine, USA

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

ABSTRACT

Billions of people use smartphones on a daily basis, including 15% of the world's population with disabilities. Mobile platforms encourage developers to manually assess their apps' accessibility in the way disabled users interact with phones, i.e., through Assistive Technologies (AT) like screen readers. However, most developers only test their apps with touch gestures and do not have enough knowledge to use AT properly. Moreover, automated accessibility testing tools typically do not consider AT. This paper introduces a record-and-replay technique that records the developers' touch interactions, replays the same actions with an AT, and generates a visualized report of various ways of interacting with the app using ATs. Empirical evaluation of this technique on real-world apps revealed that while user study is the most reliable way of assessing accessibility, our technique can aid developers in detecting complex accessibility issues at different stages of development.

CCS CONCEPTS

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

KEYWORDS

Android, Accessibility, Software Testing, TalkBack, AssistiveTechnology

ACM Reference Format:
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 (CHI '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3544548.3580679

1 INTRODUCTION

Mobile devices are the most popular computing devices [62], and mobile applications are an integral part of people's daily lives. Modern mobile devices are equipped with touchscreens, providing rich experiences for users; however, they also force developers to test and validate the functionality of their apps either manually or using automated tools. In the testing process, developers may neglect to evaluate their software for approximately 15% of the world's population with disabilities [66], many of whom cannot use conventional interaction methods, such as touch gestures. According to law enforcement and social expectations, developers should design apps accessible to all users, regardless of their abilities. Still, prior studies have revealed that many popular apps ship with accessibility issues, preventing disabled users from using them effectively [2, 23, 56].

App developers are aided by accessibility guidelines published by companies such as Apple [16] and Google [8], as well as technology institutes such as the World Wide Web Consortium [65]. In order to understand how people with disabilities use mobile apps, developers are encouraged to conduct user studies with users (preferably with disabilities) using assistive services, such as screen readers. Despite the fact that software practitioners acknowledge the importance of human evaluation in accessibility testing, they admit that end-user feedback is difficult to obtain [19]. Furthermore, for small development teams with limited resources, finding users with various types of disabilities and conducting such evaluations can be prohibitively challenging and expensive.

Using accessibility analysis tools, app compliance with guidelines and accessibility issues can be detected automatically [5, 7, 17, 18]. An app's User Interface (UI) can be analyzed, for example, to determine if the contrast between elements and backgrounds is above a certain threshold or if the button area exceeds a specified area defined in the guidelines. Solely analyzing the UI specification of an app may not reveal many accessibility problems that are only present when assistive services are used, such as screen readers. Blind users, for instance, can use a screen reader like Android's TalkBack to navigate UI elements and perform actions. If TalkBack is unable to focus on an element, the element becomes completely inaccessible.

Generally, automated accessibility testing does not consider assistive services, except for a few recent research tools [1, 58, 60]. Latte [58] assumes the availability of GUI test cases for validating an app's functionality. The test cases are then repurposed to execute with assistive services, e.g., TalkBack, for accessibility analysis. Since developers rarely write GUI tests for their apps, Latte is limited to situations where GUI tests are available. According to a recent study, over 92% of Android app developers do not have GUI tests [46]. Other works try to mitigate this issue by analyzing a single app screen while ignoring the app's functionalities. ATARI [1] assesses the focusability of screen elements by navigating sequentially using TalkBack. However, ATARI does not consider any actions, e.g., clicking, and depends on the developers/testers to provide the screen. Moreover, both Latte and ATARI consider one

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04.
https://doi.org/10.1145/3544548.3580679

type of navigation in TalkBack: linear navigation. Groundhog [60] addresses this limitation by using an app crawler to visit multiple screens and assessing whether the elements are clickable by TalkBack. Although Groundhog considers performing actions, its analysis is limited to one action on one screen and cannot detect accessibility issues occurring in a sequence of interactions. Moreover, it is limited to one action, i.e., click, and does not support other actions like swipe or type. Finally, Groundhog cannot visit and analyze various parts of an app due to the limitation in random input generation, e.g., it cannot pass the login screen.

The key insights that guide our research are (1) mobile developers and testers still prefer manual testing in-app development [38, 41, 47], (2) assistive services need to be incorporated for evaluating apps' accessibility, and (3) there is a lack of expertise and knowledge among many mobile developers and testers on how to properly evaluate the accessibility of their apps with guidelines, automated tools, and assistive services. A survey found that 48% of Android developers cite lack of awareness as the main reason for accessibility issues in apps [2]. Another survey found that 45% of accessibility practitioners are experiencing problems related to accessibility development and design, such as inadequate resources and experts [19].

Informed by the above-mentioned insights, we have developed a new form of automated accessibility analysis, called A11yPuppetry, that aids developers in gaining insights on accessibility issues of their apps. Developers and testers can evaluate their apps manually by using touch gestures, while A11yPuppetry records these interactions. After that, A11yPuppetry interacts with the app on another device using an assistive service to perform the equivalent actions on behalf of the testers, regardless of their knowledge and expertise in accessibility and assistive services. A11yPuppetry is inspired by Record-and-Replay (RaR) techniques, such as [31, 33, 57], where a program records the user actions on an app and replays the same actions on the same app in another device. However, to the best of our knowledge, all existing RaR techniques replay the recorded actions exactly as they are performed. For example, if the user touches specific coordinates of the screen, the replayer program also sends a touch event for the same coordinates. A11yPuppetry is different from these techniques since the replaying part is completely done by an alternative way of interaction, e.g., a screen reader. More importantly, A11yPuppetry generates a fully visualized report for developers after replaying the recorded use case with assistive services, which are augmented by accessibility issues.

This paper makes the following contributions:

- A novel, high-fidelity, and semi-automated form of accessibility analysis that can be used by almost any mobile developer or tester to evaluate the accessibility of mobile apps with assistive services;
- A publicly available implementation of the above-mentioned approach for Android called A11yPuppetry [59];
- Conducting user studies with users with disabilities and creating a benchmark of real apps with accessibility issues confirmed by disabled users; and
- An extensive empirical evaluation demonstrating the effectiveness of A11yPuppetry in identifying issues that the existing automated techniques cannot detect.

The rest of this paper is organized as follows: Section 2 motivates this study with an example and explains the challenges that we are facing. Section 3 examines the related literature, next the Section 4 provides an overview of our approach and the following sections explain the details of our approach. The evaluation of A11yPuppetry on real-world apps is finally presented in Section 9. The paper concludes with a discussion of the avenues for future work.

2 MOTIVATING EXAMPLE

This section illustrates how users with visual impairments use screen readers to interact with apps. Further, we demonstrate a couple of accessibility issues that cannot be detected by conventional accessibility testing tools. Finally, we elaborate on the challenges of automatically recording touch gestures and replaying them with a screen reader.

Figure 1(a) shows the home page of the Dictionary.com app with more than ten million users in the Android Play store [25]. Assume a tester wants to validate the correctness of a use case which consists of 3 parts: Selecting the "word of the day" and listening to its pronunciation, marking the word as a favorite, and reviewing or removing favorite words.

A user without a disability who can see all elements on the screen and perform any touch gestures can perform this use case fairly easily. First, she taps on the word of the day, box 10 in Figure 1(a), then the app goes to Figure 1(b). Next, she taps on the speaker button to listen to the pronunciation, pink-dashed box in Figure 1(b). Then to mark the word as a favorite, she taps the star button, yellow-solid box in Figure 1(b), and she can get back to the home page, Figure 1(a), by pressing the back button. Next, to see the list of favorite words, she taps on box 2. The app will go to the state depicted in Figure 1(c). To remove a word, the user needs to tap on the edit button, yellow-solid box in Figure 1(c), then the navbar changes to depict the number of selected words, and the delete button, Figure 1(d). Finally, the user selects the checkbox next to the word, and taps on the delete button, the yellow-solid box in Figure 1(d).

To perform the same use case, users with visual impairments, particularly blind users, have a completely different experience. They rely on screen readers, e.g., TalkBack for Android [10], to interact with the app. Users can perceive the screen's content by navigating through elements and listening to the textual description of the focused element by TalkBack. A common accessibility issue among mobile apps is the lack of content description for visual icons [2, 23]. For example, if the star button in Figure 1(b) does not have a content description, a blind user cannot guess the functionality of this button. For the sake of this example, assume this app does not have such issues and all elements have proper textual description, e.g., box 2 in Figure 1(a) has a content description as "Favorites List".

There are several ways of navigating the elements of an app with TalkBack. Using *Linear Navigation*, the user can navigate to the next and previous element of the currently focused element by swiping right and left on the screen. For example, to reach the "word of the day" in Figure 1(a), which is *diphthongize*, the user can start from box 1 (top left icon) and navigate to the next

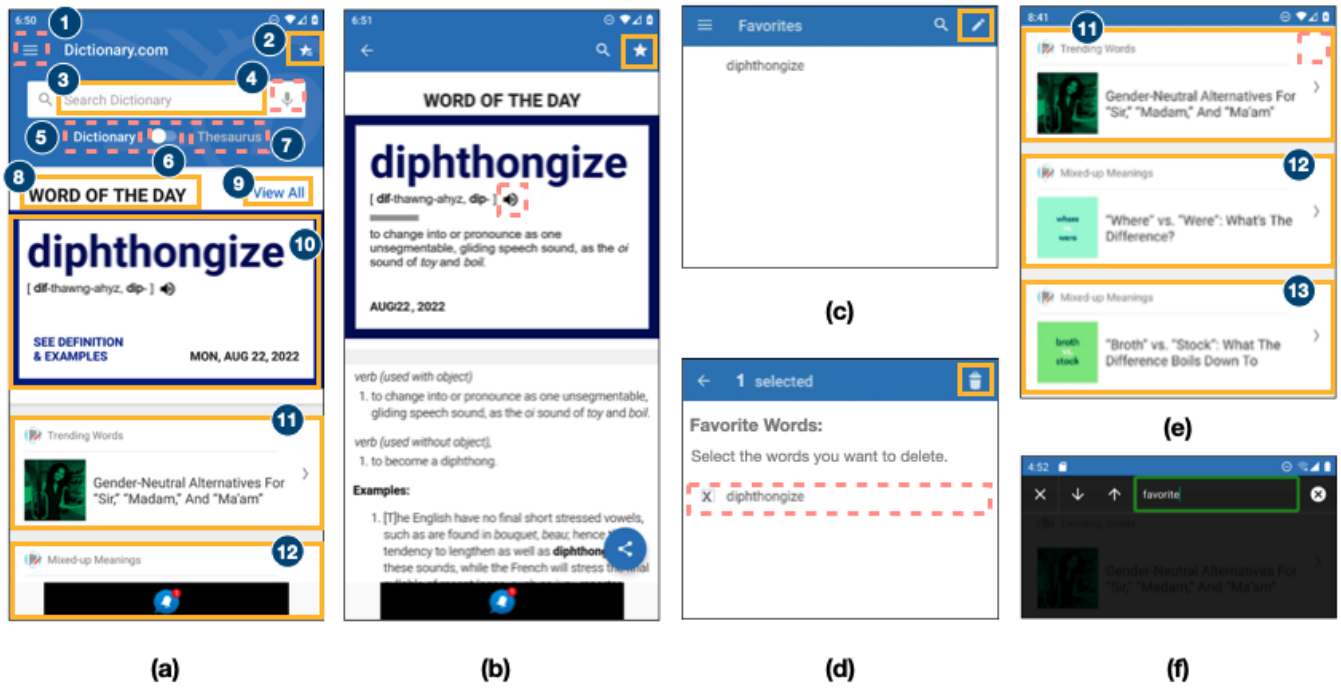


Figure 1: (a) The main page of Dictionary app; (b) The page after tapping on the word of the day; (c) The page showing all of the words favored by a user; (d) The page after user taps on the edit button in Figure 1(c); (e) Upper menu disappears when user scrolls down the page; (f) The Search Navigation provided by TalkBack.

elements until it reaches box 10. Note that TalkBack may group elements for a more fluent announcement, like here, where a couple of textual elements are grouped into box 10. Secondly, the user can utilize *Jump Navigation* to focus on elements with specific types, e.g., buttons or edit-text boxes. For example, by jumping in button elements, the user can focus on boxes 1, 4, 5, 6, and 7, pink-dashed boxes in Figure 1(a). The third way is *Touch Navigation* where the user touches different parts of the screen, and TalkBack focuses on the elements behind the user's finger. For example, if the user touches the top right of the screen in Figure 1(b), it focuses on box 2, and TalkBack announces "Favorites List". Another way is finding the element through a search. TalkBack user can enter the name of the element she is looking for, either by text entry or voice command, and TalkBack focuses on the element with the same text. For example, by searching "View All" TalkBack focuses on box 9 in Figure 1(a).

Besides these navigating ways for focusing on an element, there are alternative ways to perform touch gestures. For example, the user can replicate the scroll action by swiping on the screen with two fingers. Also, the user can execute some predefined actions by performing special gestures. For example, swiping up then left is equivalent to going to the device's home screen, or swiping left then right is equivalent to scrolling backward.

To click on an element, the user should perform a double-tap gesture on the screen when the target element is focused. TalkBack perceives this gesture and sends a click accessibility event, *ACTION_CLICK*, to the focused button, which is the equivalent of tapping on the button by touch. After getting to the word of the day page, to listen to the pronunciation, the user needs to locate the speaker button, pink-dashed box in Figure 1(b). However, the element cannot be focused on by TalkBack as developers only set focus to its ancestor, the **RelativeLayout**, and telling TalkBack to skip all the descendants, including the speaker button; therefore, this functionality is inaccessible for TalkBack users. While the unlocatablity of this element by TalkBack is a critical accessibility issue, Google's Accessibility Scanner, the most widely used accessibility analyzer for Android, cannot detect it since the scanner does not consider assistive services like TalkBack into account.

Assuming the mentioned accessibility issue does not exist, the blind user continues the rest of the use case by selecting the favorite button, the yellow-solid box with the star icon in Figure 1(b), and then returns to the home page. After returning to the home page, the user needs to find Favorites List or box 2 in Figure 1(a). However, since the user was previously on this page, box 10 is focused. By navigating to the next elements, boxes 11 and 12, TalkBack automatically scrolls forward to fetch the items below; however, the app makes the upper menu disappear as shown in Figure 1(e). A sighted user can notice this major change in the screen since she can observe all parts of the screen; however, a blind user may not notice it. Consequently, the blind user cannot locate the favorites list button, initially located at the top right of the display. Even if the user searches for the word "Favorite", Figure 1(f), there is no result since the favorites list button does not exist on the screen anymore. This is another example of accessibility issues that cannot

be detected without considering exactly how blind users interact with apps, i.e., through a screen reader such as TalkBack.

While it is straightforward for most app developers and testers without disabilities to perform the aforementioned use case with touch gestures, none of the accessibility issues above could be detected unless the same use case is performed using a screen reader. Our objective is to record touch gestures from an arbitrary app tester, execute them using a screen reader automatically, and generate a report with detected accessibility issues. Now, we explain the possible challenges to realizing this idea.

- **Action Mapping.** Although users with visual impairments also use touch gestures with screen readers like TalkBack, the way actions are performed are completely different. For example, as we mentioned in the example, clicking an element without a screen reader is a simple touch on the element's coordinates; however, a screen reader user needs to first locate the element and then perform a double tap gesture to initiate a click. There is no trivial mapping between the touch gestures and the screen reader's actions.
- **Action Approximation/Alternatives.** In the case of having a mapping between touch gestures and screen readers, the actions are not completely equivalent. For example, sighted users can scroll different parts of the screen with different velocities; however, TalkBack users can only perform four limited scrolling, left, right, forward, and backward, where their start/end points and velocity are constant, regardless of what TalkBack user wants. On the other hand, TalkBack users can scroll through lists by navigating through items via swiping left or right. Either way, although there are equal actions with and without screen readers, their effects are different, making it complicated to ensure the apps are in the same state.
- **Element Identification.** Besides the fact that actions are done differently with and without screen readers, the way elements are accessed is also different. As mentioned earlier, TalkBack may group multiple elements into one for a better user experience for visually impaired users. Moreover, if action is associated with an element of a group, TalkBack assigns the action to the whole group. For example, in Figure 1(d), a sighted user may tap on the checkbox to select the word; however, TalkBack focuses on the group of the checkbox and the word (pink-dashed box), not the checkbox itself. Therefore, to select the checkbox, a TalkBack user needs to focus on the group of elements and then perform a double-tap gesture.
- **Lack of Accessibility Knowledge.** In traditional record-and-replay techniques, a tester can easily identify the bugs and issues since the replaying is supposed to be identical to the recording, and all interactions are familiar to the tester. However, it is not trivial for a sighted user to understand accessibility issues in a screen reader's replays if she is not an experienced assistive-service user. That is why it is important to not only detect accessibility issues, but to also provide an explanation for developers as to how the detected issues hinder the visually impaired users.

3 RELATED WORK

3.1 Accessibility Testing

Accessibility testing aims to identify the accessibility issues that hinder disabled people from interacting with apps or software. Based on the previous study, accessibility testing can be categorized into two types: static accessibility testing and dynamic accessibility testing [61].

In general, static accessibility testing tries to find accessibility issues by investigating the source code. Android Lint [11] is a static analyzer embedded into Android Studio and can detect accessibility issues such as the lack of content descriptions. Prior researches also focus on finding certain accessibility issues, such as [23, 51] used deep learning technique to detect unlabeled icons and generate the corresponding labels for them. However, static approaches cannot detect problems that only manifest themselves at runtime.

Dynamic accessibility testing can detect runtime accessibility issues by analyzing the rendered UI components' attributes and the corresponding interactions with UI components. Accessibility Scanner [5], PUMA [35], MATE[26], Xbot [24], and KIF [40] rely on a single app screen to perform the testing, and they can report issues such as the inappropriate size of touch elements and the low text contrast. Other works [2, 26] utilize a crawler so that developers do not need to explore the app manually. However, all the aforementioned tools fail to detect accessibility issues that manifest themselves in interactions with apps. In addition, few prior works consider assistive services for accessibility testing except the following ones. Alotaibi, et al. [1] utilize TalkBack to identify accessibility issues, such as unfocusable elements. However, their tool, called ATARI, is limited to a single app screen and fails to detect accessibility issues related to actions such as clicking and typing. Latte [58] also employs TalkBack and executes GUI tests via assistive services to identify the related accessibility issues. As a result, Latte can detect accessibility issues related to actions. However, Latte assumes the availability of GUI tests, and prior work indicates that over 92% of Android app developers do not have GUI tests [46]. Furthermore, Latte and ATARI only implemented one type of navigation in TalkBack (linear navigation) while ignoring other navigation methods such as search and jump. Groundhog [60] is an accessibility app crawler, and therefore not limited to a single app screen. In addition, Groundhog assess whether the elements are clickable by TalkBack. Nevertheless, Groundhog only supports the click action and cannot detect accessibility issues in a sequence of interactions. OverSight [50] is an automatic tool that can detect overly accessible elements in Android. Overly accessible elements refer to the elements that provide additional information and functions to the user of AT, compared to what is available through the conventional interaction mode. OverSight dumps all the nodes belonging to the current window and lists potential overly accessible elements using predefined rules. Potential OA elements are then verified through AT on the actual device, and a report is generated to provide additional clues for developers. AccessiText [3] can automatically detect text accessibility issues that occur when using Text Scaling Assistive Service (TSAS) in Android. AccessiText executes the same GUI test on an app with both the default size text and the scaled text, and captures screenshots and metadata for

further analysis. It then analyzes the execution results and reports text accessibility issues to users.

Several accessibility testing tools have been developed for Web applications. MAUVE++ [22], AChecker[30], Accessibility Designer[63], and ABD[20] are dynamic accessibility testing tools for web applications. MAUVE++ and Achecker mainly rely on accessibility guidelines for analyzing the accessibility of websites. More specifically, MAUVE++ incorporates WCAG 2.1 as its guideline together with additional success criteria for mobile websites. Achecker enables users to select one of nine international accessibility standards during the check. It categorizes and prioritizes the accessibility issues into known problems, likely problems, and potential problems. Accessibility Designer enables users to find the mismatch between the voice generated by a screen reader and the original texts. In addition, it investigates the time-oriented aspects by calculating how long it takes to navigate from the top of the page to the rest of the page using a screen reader. ABD is a plug-in to the WebAnywhere screen reader that lets users record the accessibility issues they encounter as human-understandable macros. Then users can share the problems with developers in the form of a URL that encapsulates the assistive technology and the recorded interactions. Compared to A11yPuppetry, ABD mainly focuses on three accessibility issues (reading orders, alt orders, and alt text), while A11yPuppetry can detect additional accessibility issues, such as unlocatable elements and ineffective actions.

3.2 Record-and-Replay

There are lots of prior works related to record-and-replay in Android. RERAN [31], appetizer [15], Mosaic [34], and Orangutan[42] rely on the Linux Kernel for recording and replaying events. For example, RERAN requires a rooted device and employs the ADB commands `getevent` and `sendevent` to record and replay events. For tools that rely on Linux Kernel, the captured events are low-level and hard to translate to high-level gestures that are understandable by assistive services.

VALERA [36] has a high accuracy for recording and replying and can capture various events, such as network inputs. However, VALERA relies on a customized OS as it requires a modified Android system image, which imposes threats to its application.

Mobiplay [53], Espresso[9], Barista [27], Robotium[55], Culebra[52], Ranorex[54], SARA [33], RANDR [57], Sugilite [45] rely on the application layer to capture inputs. Mobiplay utilizes client-server architecture. The client app and target app run on an Android device and a remote server, respectively. Mobiplay identifies the targeted node based on the screen coordinates during the replay stage. Nevertheless, Mobiplay is not publicly available to researchers. Espresso can record motion events via an attached debugger but requires the recorded app's source code. Barista is a cross-platform record-and-replay tool. However, Barista fails to record and replay on non-open-source apps as it highly relies on the Espresso framework. Robotium can only capture widgets that are rendered by the app's main process, but normally the apps will run several processes [33]. Culebra provides a desktop GUI for user recordings, and the widget that interacts with users is identified via the view hierarchy. The drawback of Culebra is it causes a large overhead while identifying the view hierarchy of the interacted widget. Ranorex can record interactions via instrumentation, but the instrumentation fails when it encounters apps that have a large size. SARA can record and replay several input sources via dynamic instrumentation and the interaction can be recorded in the form of coordinate and the widget. Specifically, SARA records the interaction coordinate at first and identify the corresponding widget information via the self-replay technique. Then, SARA employs an adaptive replay method to replay captured interactions on different devices. The drawbacks of SARA are the lack of a graphical user interface and the high reliance on the third-party dynamic instrumentation tool called Frida. Frida cannot instrument classes that implement the Android Interface `android.text.Editable`, which cause SARA to lose essential interactions during the recording. RANDR utilizes both static and dynamic instrumentation so that it is able to record and replay multiple input sources, including external non-deterministic sources such as random numbers. While RANDR can record and replay abundant input sources, it does not require administrative device privileges or the access to the app source code. However, RANDR is not publicly available to researchers. In addition, as RANDR and SARA both utilize instrumentation to capture the events and interactions, the non-standard widgets such as `android.webkit.WebView` are ignored. Current popular android apps implement WebView to display web contents as a part of an activity layout, so failing to identify WebView makes the recorder lose essential interactions during the recording. Sugilite is a publicly available android application for record and replay that utilizes an overlay to intercept interactions, such as click and typing. Using an overlay enables Sugilite to capture various events and widgets, even the non-standard widgets such as WebView. For each interaction being recorded, users need to confirm whether the identified interaction is correct. After confirmation, Sugilite performs that interaction on behalf of users. However, it fails to recognize a node that is not clickable and gets stuck at the current window if the clicked node has accessibility issues. Overall, Sugilite is the most promising recorder for our project as it is publicly available to us and can capture various widgets, even WebView. The known drawbacks of Sugilite can be mitigated via re-implementations.

None of the tools mentioned above use assistive service to replay the recorded interactions.

Besides the Android platform, plenty of record and replay tools have been implemented for web applications. Actionshot[44], Coscripter[43], WebVCR[14], Smart Bookmarks[37], and WebRR[48] rely on the application layer for recording and replaying events. Actionshot and Coscripter can record users' browsing activity through interactions, such as the button click. They then generate the human-readable text scripts for the replay stage. However, they both have problems recording interactions for pages that employ complex HTML and JavaScript. WebVCR requires users to specify the start point and end point of each recording and utilize the DOM signature as part of the identifier. The recording is reflected as browsing steps in a smart bookmark that can be replayed later. As WebVCR relies on DOM API, interactions related to HTTP authentication cannot be precisely recorded. Smart Bookmarks trigger recording when Javascript events occur and capture the corresponding text label and XPath of

the interacted components as identifiers. During the replay, it runs each recorded interaction in sequence and utilizes Chickenfoot's algorithm[21] to match the desired element. WebRR refers to the self-replay technique SARA uses, and WebRR generates several identifiers to improve the robustness of recorded interactions. WebRR uses generated identifiers to locate the desired elements during the replay. Nevertheless, WebRR fails to capture interactions that happen through non-standard widgets and has problems recording interactions inside a dynamic iframe. WaRR[4] requires a customized browser and an interaction driver. Using WebKit, the WaRR recorder is able to capture interactions on multiple platforms, including desktop and mobile. The recorded interactions are saved in the form of WaRR Commands. During the replay, the browser interaction driver converts commands into ones that are understandable by a browser.

4 APPROACH OVERVIEW

A11yPuppetry consists of four main phases, (1) Record, (2) Action Translate, (3) Replay, and (4) Report. In this section, we provide an overview of the approach and in the next four sections, we explain the details of each phase.

Figure 2 depicts an overview of A11yPuppetry. The process starts with the Record phase when the user interacts with a device enabled with the Recorder service. The Recorder service listens to UI changes events and adds a transparent GUI widget overlay on top of the screen to record the user's touch gestures. After receiving a touch gesture on the overlay, the Recorder replicates the gesture on the underlying app, and sends the recorded information to the server as an *Action Execution Report*. The server will store the recorded information in the database.

In the second phase, Action Translation, the *Action Translator* component receives the *Action Execution Report* from the Recorder (containing UI hierarchy, screenshot, and the performed gesture) and translates it to its equivalent *TalkBack Action*. For example, touching on the coordinates of the favorite button in Figure 1(b) will be translated to focusing on the favorite button and performing a double-tap gesture.

In the Replay phase, the TalkBack Action is sent to several replayer devices that perform the action. Each replayer device has a running TB Replayer service that receives TalkBack Action from the server, creates and maintains a *TalkBack Element Navigation Graph* (**TENG**) of the app, and performs the received actions with a navigation mode. We will define and explain TENG and navigation modes in Sections 6 and 7 in detail; however, for now, assume TENG is a model of the app UI designed for TalkBack, and a navigation mode is a way of locating elements, e.g., Linear or Jump Navigation. Once an action is performed, a TalkBack Execution Report is stored in the database. The TalkBack Execution Report consists of actions that are executed with TalkBack, screenshots, and UI hierarchy files of the different states of the app before, during, and after execution.

In the final phase (Report), the *A11y Analyzer* component reads the stored information in the database, i.e., Action and TalkBack Execution Reports, and produces an *Aggregated Report* of the recording, replaying, and the detected accessibility issues. The user can access this report using a web application.

5 RECORDER

In this section, we first study the various touch gestures and explain how we model them. Next, we explain how we record the touch gestures of a user when she interacts with a mobile app.

5.1 Touch Gestures

To have a complete and sound understanding of the different ways of interaction, we used the official documentation of user interactions and touch gestures in Android [13, 49]. By analyzing the various touch gestures, we came up with two attributes for a touch gesture: (1) the number of involved fingers, and (2) motion. For example, a single tap is considered a touch gesture with one finger without any motion, or pinching-in is a touch gesture with two fingers with movement. We categorized the common touch gestures into several categories. However, due to space limits, we explain only two of these categories here since they are widely used in applications, and they have counterpart actions in screen readers.

- **PointGesture.** This is the most common way of interacting with a touch-based mobile device. To perform this type of gesture, the user uses one finger at a specific point on the screen without moving her finger to other parts of the screen. This type of touch gesture is identified as $\mathbb{PG}(t, p)$ where t is the type of the gesture, e.g., single-tap or long-press, and p is the coordinates of a point on the screen.
- **LineGesture.** In this type of touch gesture, the user puts her finger on the screen and draws a line. The movement's velocity and starting point may lead to different behaviors. For example, if the user draws the line from the edge of the display, it is considered an edge swipe that is usually associated with system actions, e.g., going to the home screen or navigating back. This type of touch gesture is identified as $\mathbb{LG}(l, v)$ where l is a straight line on the screen and v is the velocity of the gesture, i.e., fast, regular, and slow.

5.2 Implementation

In a nutshell, the recorder component uses two different ways to record the user's actions, (1) through a transparent overlay placed on top of the app and (2) by listening to system events related to the changes on the screen. We implement the recorder on top of Sugilite [45], a programming by demonstration tool for Android apps. We briefly explain some background for understanding the concept, then describe how the recorder is implemented.

With Android's Accessibility API, developers can create apps that interact with the device and receive feedback from it. These APIs can be accessed via the implementation of *AccessibilityService* [6], an abstract Android service that acts as a wrapper to interact with the device. *AccessibilityService* can augment the current screen by creating new GUI objects. For example, TalkBack is an implementation of *AccessibilityService*, which focuses on and highlights an element (by annotating a green rectangle GUI widget around the element), describes the focused element (by reading the textual description of the focused element), and interacts with the app on behalf of the user (by clicking on the focused element when the user double taps). By receiving *AccessibilityEvents*, an *AccessibilityService* can be notified when there is a significant change on the screen.

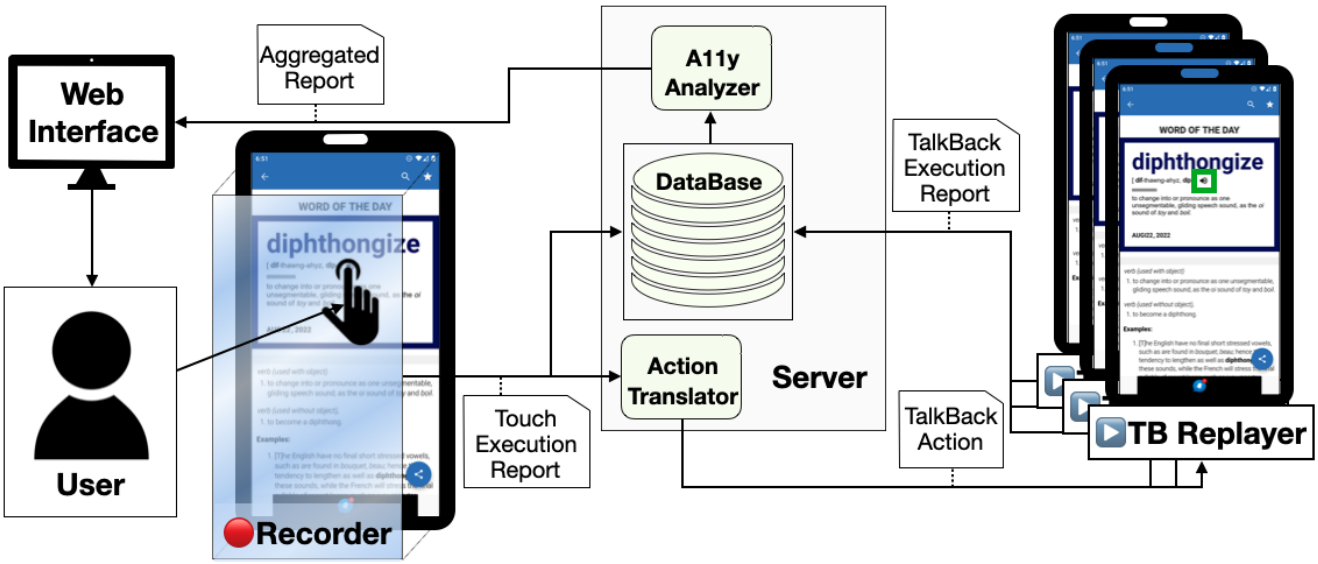


Figure 2: An overview of A11yPuppetry.

These events are generated by *AccessibilityManager*, which monitors the display and communicates with *AccessibilityServices*. An *AccessibilityEvent* is usually associated with an *AccessibilityNodeInfo* object which describes the attributes of the changed element, e.g., its text, content description, or class.

Although Accessibility API is mainly designed to assist users with disabilities, it can be used for other purposes. We implemented the recorder as an *AccessibilityService* to understand the user's action. When the recorder is enabled, it creates an overlay of the screen's size, which is an *android.view.object* and attaches it to the foreground window. This overlay acts as an echo component; it performs any received touch gestures on the app with Accessibility API. A touch gesture event, e.g., PointGesture $\mathbb{PG}$, is captured by *onTouchListener* which is enabled for the overlay. Once the touch gesture is received, a copy of the touch gesture, the UI hierarchy of the current screen, and a screenshot image are combined and packed as *Action Execution Report*.

Although the overlay object can record touch gestures, a few other actions such as adjusting volume with physical buttons or typing with a keyboard cannot be captured. To that end, the recorder listens to all *AccessibilityEvent*s and records the events that represent actions performed by the user. For example, when the user types on an EditText with a keyboard, the recorder will receive *AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED* containing the typed text. Similar to touch gestures, these events, along with the UI hierarchy and screenshot of the app, are packed and sent as *Action Execution Report*s.

Once the *Action Execution Report* is created, either by the overlay screen or *AccessibilityEvent*, the recorder sends it with WebSocket to the Server. Note that the recorder is an app inside an Android device or emulator, and all the storing, analysis and broadcasting is done on the external remote server.

6 ACTION TRANSLATOR

In the second phase of A11yPuppetry, the *Action Translator* component (Figure 2) translates actions recorded from the user using touch gestures to their counterparts in TalkBack. As seen in Section 2, the main challenge here is that there is no one-on-one mapping from touch gestures to actions that can be performed by TalkBack. To address this challenge, we first studied TalkBack and categorized its actions, which in turn allowed us to propose a mapping from touch gestures, i.e., Point and Line Gestures, to these categories.

6.1 TalkBack Actions

We studied and examined the Android documentation to understand TalkBack and its actions. Then, two authors followed official tutorials on TalkBack on Android devices and interacted with at least 5 popular Android apps. Finally, we interviewed a blind user who used TalkBack and asked him to perform a few use cases in an app to clearly understand different ways of interacting with TalkBack.

TalkBack, when it is enabled, creates a virtual layer between the app and the user to enable users to perceive the UI without performing unintended actions. TalkBack draws an overlay on the screen, receives touch gestures, and translates these gestures into different actions. We categorized the different ways of interaction into the following three categories:

- **ElementBased.** This type of interaction is mostly used to perceive the content of an element or perform a click or long-press on the focused element. TalkBack focuses on an element and announces its textual description. Given that the element is e and the type of the action is t, an ElementBased action can be defined as $\mathbb{EB}(t, e)$, meaning that the element e should be focused by TalkBack and action t, e.g., click, should be performed on the

focused element. There are various ways to focus on an element that previously were mentioned in Section 2.

- **LinearNavigation.** User can change the focus to the next and previous element of the currently focused element. The actions associated with linear navigation are swiping right and left. The order of the next and previous elements is determined based on their position in the UI hierarchy. Note that, TalkBack may also perform scroll action while navigating to the next or previous element if they are (partly) out of the screen, e.g., Figure 1(e).
- **JumpNavigation.** Users can jump through elements of certain types for faster navigation by swiping up and down. For example, users can go to the next heading, paragraph, control, or link instead of navigating element by element. Moreover, users can adjust the granularity of announcements to understand the content easier. For example, users can move to other lines, words, or even characters instead of focusing on elements.
- **SearchNavigation.** Users can search for a specific element on the screen with text or voice interface enabled by a three-finger long-press. It is similar to finding a specific word on a page in a text viewer/editor.
- **TouchNavigation.** Users touch a spot on the screen, and TalkBack focuses on the element on the same coordinates. This navigation method is usually used when the user has an estimation of the coordinates of the element she is looking for, e.g., top or bottom menu, or when the element could not be detected by the other navigation methods and the user has to conduct an exhaustive search to find all elements on the screen.

- **TouchGestureReplication.** Besides the click and long-press actions that can be done by ElementBased actions, users can replicate several other touch gestures, in particular, *LineGesture*s by bypassing the TalkBack overlay. A user can replicate scrolling, dragging, or edge swiping by swiping with two fingers when TalkBack is enabled. A TouchGestureReplication can be defined as $\mathbb{TGR}(lg)$ where lg is a LineGesture.
- **PredefinedActions.** Various actions that TalkBack can perform are not dependent on the app that the user is interacting with. For example, global actions, e.g., Home, Recent Apps, or Back, are not dependent on an app and can be performed with special gestures in TalkBack, e.g., swiping up then left will go to the home screen of the device. A PredefinedAction is $\mathbb{PA}(t)$ where t determines the action(s) to be performed, e.g., scroll forward or volume up.

6.2 Mapping

We can map the touch gestures, defined in Section 5, to these categories.

6.2.1 PointGesture. PointGestures, like single-tap or long-press, can be mapped to ElementBased actions in Talkback since a PointGesture is usually associated with a GUI element. In some cases, the PointGesture is not associated with a single element and the exact coordinate of the touched surface is important. For example, a painting app may have a large canvas where the user can paint and draw shapes by touch gestures. Although the underlying element of all these gestures is the canvas, the exact coordinate of the gesture is important to draw the lines precisely. We exclude these cases in this work since they require a fine visual perception of the screen to pinpoint the desired coordinates.

However, to precisely find the equivalent of a PointGesture, we also need to find the element associated with the touch gesture. To find the associated element, we list all elements in the UI hierarchy (recall that the Action Execution Report has the UI hierarchy of the app before the execution). Then filter the elements that enclose the touched point and sort them based on their z-index. An element with a greater z-index is always in front of an element with a lower z-index [28]. Then we iterate the list to find an element that has a matching attribute to the action that is performing. For example, if the PointGesture is single-tap or long-press, then the element should have a *clickable* or *long-clickable* attributes respectively. If no such element can be found, we choose the first element in the list.

6.2.2 LineGesture. LineGestures can be mapped to either TouchGestureReplication or PredefinedActions. For example, a swipe-up touch gesture can be performed in TalkBack either by swiping up with two fingers or performing the predefined action, swipe right then left.

Once the input action is translated into a TalkBack action, it will be sent to replayer devices, in particular, to their TB Replayer components.

7 REPLAYER

The third phase of A11yPuppetry replays the received *TalkBack Action* with TalkBack. Before the user starts interacting with the app, the recorder and replayer devices are in the same state, i.e., the app under test is installed and opened. In the replayer device, TalkBack and *TB Replayer* services are enabled. TB Replayer is an *AccessibilityService* similar to the Recorder service, which is responsible to communicate with TalkBack to perform the received action. For each navigation mode, i.e., Linear, Jump, Search, and Touch, there is one replayer device receiving the inputs from the server.

Recall that a TalkBack Action can be ElementBased ($\mathbb{EB}$), TouchGestureReplication ($\mathbb{TGR}$), or PredefinedAction ($\mathbb{PA}$). To perform $\mathbb{TGR}(lg)$, *TB Replayer* makes a copy of the LineGesture lg, called lg' and moves its coordinate 2cm toward the top or right of the display, then combine the two LineGestures (lg and lg') and perform them when TalkBack is enabled. Performing a $PA(t)$ is easier since it is predefined and not dependent on the app. TB Replayer has a database of PredefinedActions and can perform the actions accordingly, e.g., perform swipe right then left when t is "Scroll Forward".

However, performing an ElementBased action is relatively challenging since it requires finding and focusing on the element first. Moreover, there are various ways of navigating to locate an element, i.e., Linear, Jump, Search, and Touch. To that end, we introduce TENG (**T**alkBack **E**lement **N**avigation **G**raph) to model the different ways of navigating an app with TalkBack. After that, we define different strategies to guide TB Replayer on traversing the TENG of the app.

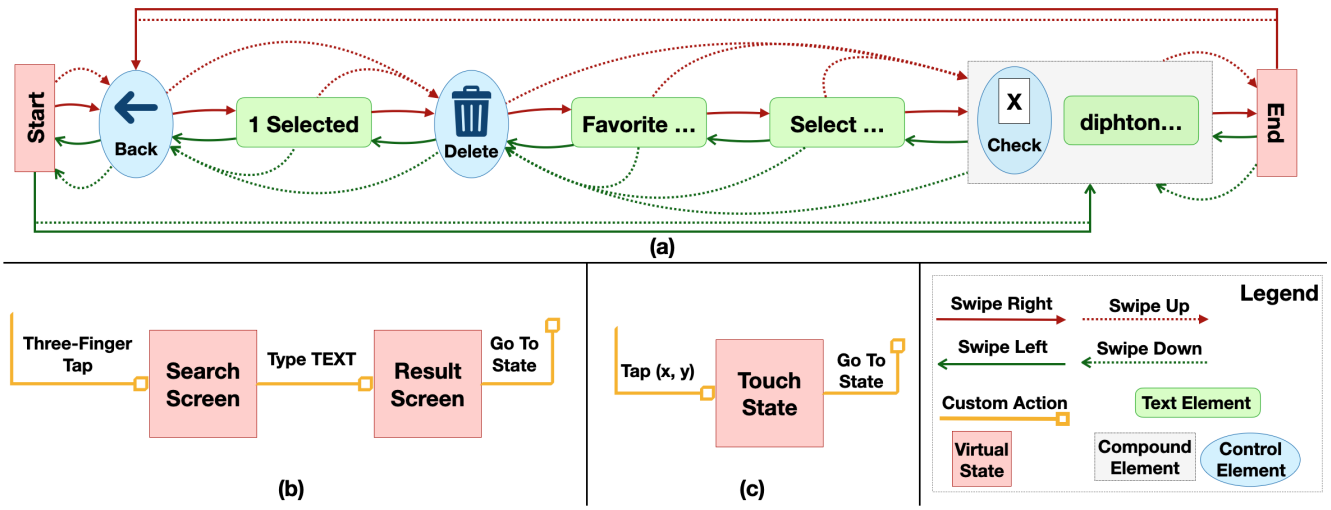


Figure 3: (a) TENG representing Linear Navigation of Figure 1(d); (b) TENG representing Search Navigation; (c) TENG representing Touch Navigation.

7.1 TENG

Simply put, TENG is a graph modeling the different states of TalkBack when enabled. TENG is defined over the UI hierarchy of an app screen, where the nodes include GUI elements that can be focused by TalkBack and the edges represent actions that can be done by the user (or TB Replayer) to change the focus from one node to another. For example, Figure 3(a) represents a part of the TENG of the app screen in Figure 1(d). For now, please ignore the Start and End red boxes, we will define and explain them shortly. The blue ovals represent control elements, e.g., buttons or checkboxes, and green-round boxes represent the textual elements. Also, the gray boxes are a View element containing a set of elements that are grouped by TalkBack to announce. Recall that in Section 2, we discussed TalkBack grouped elements that are related and associated the group with an action for a better user experience. In runtime, when Talkback is in any of these nodes (states), i.e., focused on their corresponding element, we call it an *active node*. The solid arrows in Figure 3(a) represent Linear Navigation between elements, e.g., red arrows are associated with swiping right or moving to the next element. The dotted arrows represent Jump Navigation which changes the active node to the next control element. For example, if the Delete node is active, by swiping right TalkBack focuses on the text element that starts with "Favorite" and by swiping down, TalkBack jumps on the previous control element which is "Back".

Besides the UI elements, TENG has some other nodes which we call *Virtual States*. These states do not correspond to an element on the screen; however, they represent some internal states of TalkBack. For example, the virtual states *Start* and *End* in Figure 3(a), represent the states where TalkBack reaches the first or last element on the screen and notifies the user there is no element left to visit. Note that, the user can still change the focus to other elements by Linear or Jump Navigation, even if TalkBack is in a virtual state, e.g., swiping left from Start changes the focus to the compound element in the end.

Recall that TalkBack supports two other navigation modes, i.e., Search and Touch. We model these navigations in TENG using virtual states. Figure 3(b) shows the part of TENG related to the search navigation. The entry edge is a representative edge that comes from all nodes in TENG and is associated with three-finger tap. We did not draw all edges to not make the figure complicated and messy. Once the Search Screen is activated, the user can type the text she is looking for, then the result appears in a list (Result Screen). Once the user selects a search entry, TalkBack focuses on the selected element. Finally, the Touch Navigation is modeled and depicted in Figure 3(c). Whenever the user taps somewhere on the screen, TalkBack finds the underlying element and focuses on it. Similar to Search Navigation in Figure 3(b), the entry edge of the Touch State comes from all nodes of TENG.

Given a target element, we can use TENG to plan a sequence of interaction with the device to focus on the element. For example, similar to the last step of our motivating example in Section 1, assume we want to click on the checkbox and at the beginning TalkBack is focused on the Back button. Therefore, the TENG's active node is the Back button in Figure 3(a), and the goal is focusing on the TENG's node containing the goal element (which is the compound element denoted by the grey box), and then performing double-tap. There are various ways to reach the target node, for instance, by performing two swipe up actions, TalkBack first jumps to the Delete button and then to the target node.

However, traversing with TalkBack is not as easy as it sounds. There are three reasons that TENG may be modified during the interaction with TalkBack. First, the app may dynamically update the visible elements on the screen. For example, a slide show constantly changes the visible content after showing it for a specific amount of time. Secondly, TalkBack may change the app state by performing extra gestures for navigation. For instance, recall that TalkBack scrolled the page once it reaches the last element visible on the screen in the motivating example, Figure 1(e). Lastly, the

app may change the focused element at runtime. For example, if developers do not want users to access certain elements, regardless of the rational behind this decision, they can focus on another element as soon as that element is focused by TalkBack. Therefore, we cannot rely solely on the TENG created UI hierarchy before navigation.

To that end, once TB Replayer performs an action associated with an edge, e.g., swiping right to focus to the next element, the service listens to any changes in the UI to determine if the UI hierarchy is changed. If anything changes, the TB Replayer recreates the TENG and continues the navigation. Otherwise, the service verifies if the current active node in TENG is focused by TalkBack. If it was not, then we mark the performed edge as *ineffective* and replan the locating path again.

7.2 Implementation

TB Replayer is an implementation of *AccessibilityService*. It builds the UI hierarchy by analyzing all visible *AccessibilityNodeInfo* on the screen. Then using the utility library provided by TalkBack [12], TB Replayer creates TENG from the UI hierarchy. Basically, this library has some helper methods to determine elements that can be focused by TalkBack and the linear order among them. The virtual states in TENG are created and maintained by the TB Replayer service.

Each TB Replayer in a device is responsible for one navigation mode, e.g., Linear or Jump. To locate an element, the TB Replayer only uses the edges in TENG that belong to its navigation mode. For example, to navigate to the checkbox element from the back element in Figure 3(a), the TB Replayer for Jump Navigation only uses the dotted arrows or the Search Navigation only uses the edges in Figure 3(b). Once the element is located, the TB Replayer performs the desired action, e.g., double tap for click or double tap and press for long-press.

TB Replayer compiles a set of information and sends it to the server, including the UI hierarchy, screenshot, TENG, and performed actions in all stages.

8 REPORT

In the final phase of A11yPuppetry (Report), the A11y Analyzer component in Figure 2 analyzes all information stored in the database and generated from the Recorder and TB Replayers, compiles and aggregates them, and shows the final report to the user via a web interface. Since the target users of A11yPuppetry are developers and testers with limited knowledge on accessibility, we implemented the following features to illustrate the accessibility barriers in their apps.

- **Annotated Video.** Once the record and replay for an app is completed, A11y Analyzer creates recorder videos using the captured screenshots, then animates the touch gesture on the image, as indicated by Figure 4(a). It also generates the replayer video and annotates the focused elements by TalkBack during the navigation, as indicated by Figure 4(b).
- **Action Detail.** In addition to the annotated video, we provide action detail of each step, which displays the essential information of each interaction, as indicated by Figure 4(c). The action detail includes the class name, the UI Hierarchy of the clicked element, and the text that belongs to the clicked element. We also highlight the clicked element using the red box.
- **Execution Result.** We display the execution result of each action to developers. Remember that TalkBack supports four different ways to focus on an element, i.e., Linear, Jump, Search, and Touch. Each navigation method corresponds to a TB Replayer. If the execution of a TB Replayer fails, we highlight the replayer using the red color, as indicated by Figure 4(d)—here indicating Replayer Search failed to execute the action. On the right of the execution result, we summarize the accessibility issue and provide the potential cause of the found issue.
- **Blindfold Mode.** The replayer video cannot represent the issues that blind users may face, especially the ones related to the semantics of the app. For example, when visual icons have content descriptions that are irrelevant to their corresponding buttons' functionality, blind users may become confused and not understand the app. We provide a blindfold mode in our report which lists the textual description of the items that have been navigated with TalkBack, as indicated by Figure 4(e). For example, the Blindfold Mode report of Linear Navigation for Figure 3(a) would be "(1) Back button, double tap to activate, (2) 1 Selected, (3) Delete button, double tap to activate, (4) Favorite ..., (5) Select, (6) diphtongize, not checked, checkbox, double tap to toggle."
- **State Comparison.** A11y Analyzer also compares the state of the apps in the recorder and replayer devices to see if there is any difference between them. Ideally, if all actions are performed correctly in all replayers, there should be no difference between the states. The comparison is done by checking the UI hierarchy of the apps before performing any action. In case of a difference between states, the web interface shows a warning sign near the state to show the issue.

8.1 Automated Issue Detection

A11y Analyzer detects and reports some of the accessibility issues automatically so that developers can pinpoint the accessibility problems more conveniently. In particular, three categories of issues can be detected automatically: Unfocusable Elements, Ineffective Actions, and Missing Speakable Texts. The last category can be detected easily by checking the existence of the content description attribute for the target element. However, the first and second categories are challenging to find.

Unfocusable Elements. As mentioned before, a TB Replayer executes an action by first locating the element (by focusing on it) and then performing the corresponding touch gesture, e.g. double-tap. If a TB Replayer cannot locate an element, A11y Analyzer reports that as an accessibility issue. There are various reasons an element cannot be located. For example, if the element does not exist on the screen or there is a navigational loop preventing TalkBack from focusing on the element. In the report section, we describe the reason why the element could not be focused.

Ineffective Action. Sometimes TB Replayer can locate the element and perform the action; however, the action is not effective, i.e., the intended functionality is not triggered. A11y Analyzer can detect such an issue by comparing the app's state before and after the execution by TB Replayer. Also, it considers the recorder's state for

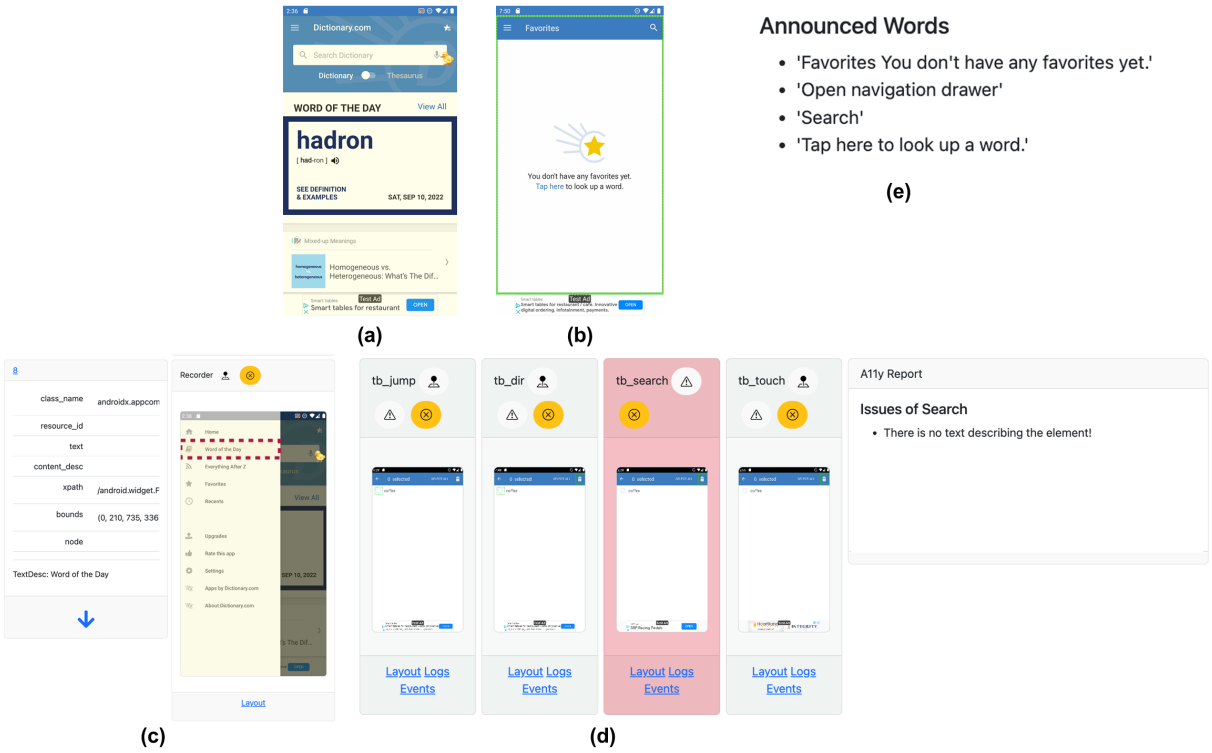


Figure 4: (a) The annotated recorder video; (b) The annotated replayer video; (c) The action detail; (d) The execution result; (e) The blindfold mode.

the same action as the reference. An action is reported as ineffective, if the before and after UI hierarchy of the app is identical and the corresponding action in the recorder state introduced changes.

In the next section, we provide examples of such automatically detected issues.

9 USER STUDIES

This section explains our experiments and user studies to evaluate the effectiveness and limitations of A11yPuppetry.

We selected five Android apps with possible accessibility issues reported in the literature [60] or online social media [39]. For each app, we designed a task (consisting of 21 to 33 actions) according to the functionalities of the app. Also, we included the parts of the app that were reported inaccessible in the task. The first four columns of Table 1 show some information about the subject apps and the number of actions involved in the designed tasks.

We use A11yPuppetry on each task of these five apps. We used an Android emulator with Android 11 and TalkBack (version 12.1) for both recording and replaying devices. Our prototype of A11yPuppetry enables us to perform the experiments synchronously (recorder and replayers are running simultaneously) or asynchronously (the recording can be done before the replaying). For the experiments, we use the asynchronous mode to not introduce any problem caused by network or other concurrency issues; however, in practice, the synchronous mode is more promising since the results can be obtained much faster.

To compare A11yPuppetry with existing work, we used Latte and Accessibility Scanner. Since Latte requires GUI test cases for the analysis, we transformed recorded use cases to GUI test cases. Scanner is not a use-case driven tool and scans the whole screen; therefore, we ran Scanner on the screens of the app after each interaction. Moreover, since in this experiments we are focused on blind users who uses TalkBack, we filter out issues that are not related to blind users, like small touch target size or low text contrast.

Besides experiments with these tools, we conducted two user studies with users with visual impairment who have experience working with TalkBack in Android. To connect to such users, we used the third-party service Fable.[1] Fable is a company that connects tech companies to users with disabilities for user research and accessibility testing. Fable compensates all user testers and is committed to fair pay for the testers.[2]

We used two services of Fable: Compatibility Test and User Interview. In the compatibility test, we provided the designed tasks and apps to Fable, then Fable distributed each task to three users with visual impairment. For all the users who participated in the compatibility tests, the assistive technology used was TalkBack (Screen Reader in Android). The users performed the tasks, and for each step of the task, they reported any issues they faced. Once we gathered all of the detected issues from A11yPuppetry and

[1]https://www.makeitfable.com
[2]https://makeitfable.com/article/why-fair-pay-for-testers-matters/

Table 1: The evaluation subject apps with the detected accessibility issues.

App	Category	#Installs	#Actions	#User Issues	#Scanner Issues	#Latte Issues	#A11yPuppetry Issues				
							Linear	Touch	Jump	Search	Total
ESPN	Sports	>50M	24	11	18	6	6	2	13	6	17
DoorDash	Food	>10M	23	8	22	10	9	1	13	9	15
Expedia	Travel	>10M	33	8	89	4	2	3	19	7	22
Dictionary	Books	>10M	21	8	113	6	4	2	13	5	15
iSaveMoney	Finance	>1M	21	5	35	2	10	9	10	2	11

Table 2: The percentage of the intersection of user-confirmed issues detected by Scanner, Latte, and A11yPuppetry to the total number of user-confirmed issues.

App	% Intersection with User-Confirmed Issues				
	Scanner	Latte	A11yPuppetry		
			Detected	Evidence	Total
ESPN	10%	18%	18%	45%	**63%**
DoorDash	25%	25%	50%	37%	**87%**
Expedia	12%	25%	50%	12%	**62%**
Dictionary	25%	50%	50%	37%	**87%**
iSaveMoney	40%	40%	40%	20%	**60%**

compatibility tests in Fable, we conducted a preliminary analysis and produced a comprehensive list of accessibility issues for each step. Then for each app, we sent requests for user interviews with Fable, where Fable scheduled a one-hour online interview with a blind user who uses TalkBack. During the interview, the user shared his/her Android phone screen. We asked the users to perform the designed tasks and explain their thoughts and understanding of the app's pages. When they faced an accessibility issue that prevented them from continuing the task, we intervened and guided them to skip to the next step. Once the users finished the tasks, we started a conversation and asked them some specific questions about the tasks or general questions about their experience in working with screen readers and apps. In summary, each app is assessed four times: three users in compatibility tests and one user in an online interview.

The source code of A11yPuppetry, a demo of the web interface, designed tasks, apps, and user responses can be found in our companion website [59].The designed tasks can also be found in the **appendix A**.

We would like to understand how A11yPuppetry can help detect accessibility issues confirmed by users with visual impairment. As discussed before, all five tasks from five subject apps are assessed by users with disabilities, Accessibility Scanner, Latte [58], and A11yPuppetry. For A11yPuppetry, we used four navigation modes (Linear, Touch, Jump, and Search). For user feedback, if at least one user expresses an issue with a certain action, we assume the action has an accessibility issue. The number of reported issues for each app can be found in Table 1. The last column (Total) represents the number of actions that at least one of the navigation modes in A11yPuppetry reported an issue. As can be seen, the issues detected by Latte and A11yPuppetry are proportional to the number of actions; however, Scanner reported many issues that can be difficult for testers to examine and verify.

Table 2 summarizes the effectiveness of Scanner, Latte, and A11yPuppetry in detecting issues confirmed by actual users. For each tool, we calculate the number of user-confirmed problems that the tool could automatically detect. The key insight for designing A11yPuppetry was that a human tester interacts with it and interprets the results to locate accessibility issues that could require human knowledge to detect. Therefore, for A11yPuppetry, we also calculate the number of user-confirmed issues for which evidence of the same issues exists in the report of A11yPuppetry. Table 2 shows the results obtained for each tool in comparison to the user-confirmed issues. As can be seen, even the automatically detected results of A11yPuppetry outperforms the existing tools. On average, A11yPuppetry could detect more than 70% of issues confirmed by users.

Results from Table 2 indicates that A11yPuppetry outperforms two existing accessibility checkers Latte and Accessibility Scanner. We further summarize what issues existing accessibility checkers can and cannot detect.

Accessibility Scanner is a dynamic accessibility testing tool on Google Play Store that provides accessibility suggestions based on scanned screens [5]. Developers can either scan a single screen or a series of snapshots through recording and Accessibility Scanner provides the results of the scan to them. According to its official documentation, Accessibility Scanner reports four types of accessibility issue.

- **Content Labeling.** Issues related to the content labels, such as missing labels, unclear and uninformative link text, and duplicate descriptions.
- **Implementation.** Issues inside View hierarchies that might hinder people with motor disabilities from interacting with a layout, such as duplicate clickable views that share the exact screen location, unsupported item types for Android Accessibility Service, traversal orders, and text scaling.
- **Touch Target Size.** Identifies the small touch elements. The threshold of the element size can be adjusted in Accessibility Scanner settings.
- **Low Contrast.** Identifies elements with a low contrast ratio between text and background or between background and foreground. Similar to touch target size, the threshold of the contrast ratio can be adjusted in Accessibility Scanner settings.

As Accessibility Scanner does not incorporate any assistive service during the evaluation apps, it cannot detect issues related to **unfocusable element** and **ineffective actions** nor provides evidence for **difficulties in reading** that A11yPuppetry supports. A detailed description of these issues is provided later in this section.

Latte [58] relies on the availability of the GUI test cases for detecting accessibility issues in Android and only supports the linear navigation of TalkBack for locating an element. Therefore, Latte can only detect unfocusable elements and ineffective actions related to linear navigation. Latte can neither provide any evidence for developers about the **uninformative textual description**, nor **difficulties in reading**.

To have a better understanding of the detected issues, we manually analyzed all reported issues and categorized them into five categories: (1) **Automated Detection** the ones that both users and A11yPuppetry reported, (2) **Evidence Provided** the ones that users reported and A11yPuppetry provide some evidence of the existence of such issue in its report which can guide the tester to detect the issue, (3) **Unsettled Issues** that A11yPuppetry reported, but users did not find significant, (4) **Flaky Issues** that A11yPuppetry mistakenly reported as issues, and (5) **Undetected Issues** are the one that users reported but A11yPuppetry did not provide any evidence of such issue. In the following, we explain the subcategories of each of these categories and provide illustrative examples.

9.1 Automated Detection

Missing Speakable Text. This issue (a visual element without the content description) is among the most common types of accessibility issues in mobile apps [23]. Due to the nature of this issue, existing accessibility testing techniques, like Accessibility Scanner, can detect this issue by only analyzing the layout of the app without considering assistive services. A11yPuppetry detects such issues using the Search navigation, i.e., if an element is not associated with a textual description, it cannot be searched with TalkBack.

Unfocusable Element. Here, an element associated with a functionality or certain data cannot be focused by TalkBack; as a result, TalkBack users cannot access them or even realize such an element exists. In Section 2, we gave an example of such an issue (the speaker button in Figure 1(b)). Note that this issue cannot be detected by Accessibility Scanner since it requires assessing whether the element is focusable by TalkBack in runtime.

Sometimes the unfocusable element belongs to a minor feature that the user may not need. For example, the collapse button in the iSaveMoney app that hides the details of expenses (red-dashed box in Figure 5(a)). However, sometimes this issue becomes critical. For example, on one of the search pages of Expedia, none of the elements on the screen, including Navigate Up Button, are focusable, making the user confused. A user mentioned: "After typing New York and pressing the search button, I am unable to move around the screen at all. None of the gestures that I use to navigate or read the screen work."

Ineffective Action. Sometimes elements are focused on by TalkBack, but the intended action cannot be performed. For example, in the iSaveMoney app, many buttons, including all yellow-solid boxes in Figure 5(a), can be focused by TalkBack. However, after performing a click action by double tapping, nothing happens. It seems the underlying reason behind this issue is the customized implementation of the button, which is sensitive to touch gesture and not click action. The issue is also found in Doordash when the user wants to change the delivery option to pick-up.

9.2 Evidence Provided

The following issues are reported by users and not by A11yPuppetry. However, the aggregated report of A11yPuppetry, including the annotated video and blindfold mode, provides evidence of these issues. The report can help accessibility testers find these types of issues faster without the need to interact with an app multiple times.

Uninformative Textual Description. The main purpose of content description for elements is to help users with visual impairment understand the app better; as a result, merely having a content description does not improve accessibility. A11yPuppetry is not capable of analyzing the semantics of content descriptions; however, its blindfold mode lists the texts that are announced while exploring the app. A developer/tester can determine whether the textual descriptions are informative or not by reading the blindfold mode report. The example of blindfold mode can be found in figure 4(e). Here are some examples of this type of issue confirmed by users.

- The textual element has some random or irrelevant data. For example, the notification icon in ESPN, highlighted button in Figure 5(b), has a content description "Í", which is not informative
- The elements associated with a functionality, e.g., button, checklist, or tab, should express their functionality. While TalkBack takes care of standard elements like *android.widget.button*, it does not announce the functionality of non-standard elements, e.g., a button which is a *android.widget.TextView*. Doordash app has many of these issues, e.g., "Save" without button or "Pickup/Delivery" without announcing toggle.
- The textual description should describe the purpose of the element completely. For example, on the renting page of Expedia, there is a compound element described as "Pick-Up"; however, it is unclear if it is related to location or date. A sighted user can easily recognize it by looking at the pinpoint icon inside this element which hints this element is related to the location of picking up.
- Sometimes, the textual descriptions provide complete information; however, they can be incorrect. For example, the traveler's element, highlighted in Figure 5(c), clearly shows there are 3 travelers selected, but its textual description is "Number of travelers. Button. Opens dialog. 1 traveler", which is incorrect.

Difficulties in Reading. Besides the textual description of elements, the way the texts are announced by TalkBack is important for understanding an app. We found a few accessibility issues reported by the users that make it difficult for them to perceive the text. This kind of issue can be detected by testers by manually analyzing the **annotated videos** and **blindfold mode**. The examples of annotated replayer video and blindfold mode can be found in figures 4(b) and 4(e), respectively. For example, in Dictionary, paragraphs of texts cannot be read as a whole; the user has to read a long text word by word. Or in the Doordash app, yellow-solid boxes in Figure 6(a), each category on the main page is announced two times, one time the visible text, e.g., "Grocery" or "Chicken", another time the image which does not have a textual description, announced

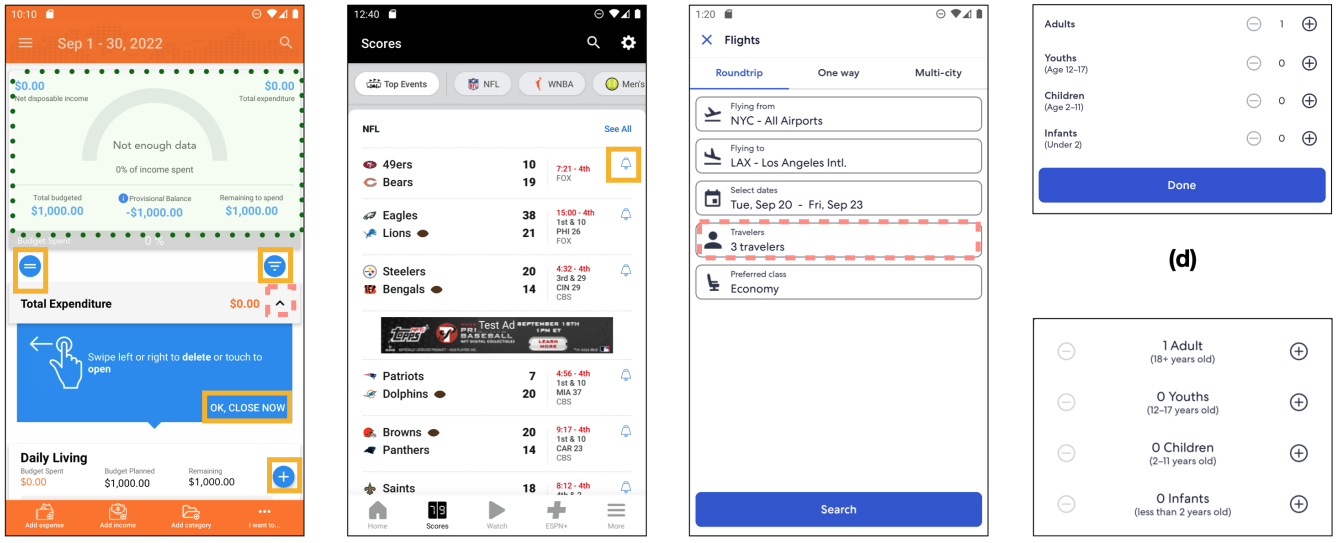


Figure 5: (a) The toggle button in iSaveMoney is not focusable and buttons indicated by yellow-solid boxes have ineffective action; (b) The content description of the notification icon in ESPN has unsupported characters; (c) The textual description of travelers numbers are different in Expedia; (d) (e) different fragments showing to different users.

as "unlabeled". In another example, all textual content of the summary block in the iSaveMoney app, green-dotted highlighted box in Figure 5(a), is announced altogether in an unintuitive order, and the user had to change the reading mode to understand each word. Although these issues do not make the app incomprehensible, they create barriers to blind users. We asked one of the interviewees how they felt about this kind of inaccessibility, and he said he could deal with them "but we, blind people or deaf people, deserved the same amount of dignity as others."

9.3 Unsettled Issues

A11yPuppetry detected some issues that the users in our user study did not find to be significant. Mainly these issues belong to Jump and Search navigation modes. In the Jump navigation mode, TB Replayer tries to locate the element using jump navigation (going to the next control or heading element); however, sometimes, it is not possible to reach to element since it does not have proper attributes, e.g., it is not a button. TB Replayer with Search navigation mode tries to locate the elements by searching their textual description; however, when there are multiple elements with the same description, this mode cannot locate the element correctly. Although users mentioned it would be nice if the attributes were set properly so they could use different navigation modes; they did not find these issues important since they usually do not use Jump and Search navigation modes. We further examined why users do not use these modes that often in Section 10.

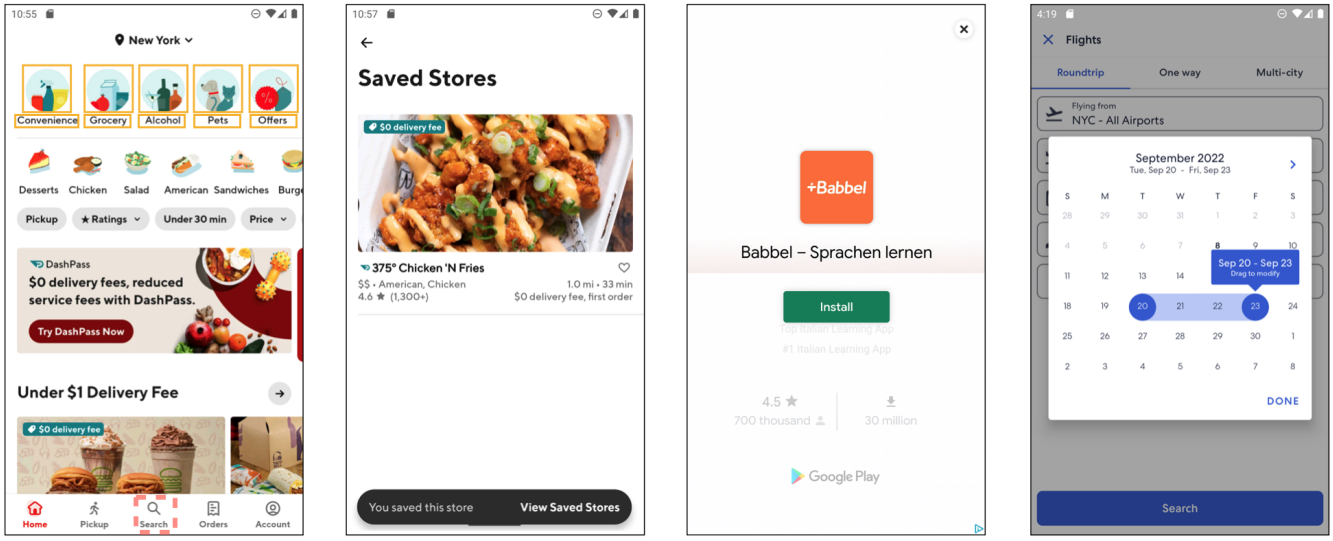
9.4 Flaky Issues

Sometimes A11yPuppetry reports issues that are not correct, which is caused by technical problems with the experiments. The main characteristic of this category is that by rerunning A11yPuppetry, the issue may not be reported again. There are three main technical problems. First, TalkBack sometimes freezes and does not respond properly and on time, making A11yPuppetry think the app has accessibility issues that do not let TalkBack continue the exploration. Secondly, the recorder may record incorrect an element; for example, on the signup page of the ESPN app, instead of recording a button, it records a transparent view covering the button, which does not interface with the touch interaction. Lastly, the apps can be changed and be in different states on TB Replayer devices. Mainly this issue is caused by A/B testing, where developers dynamically show different pages to different users to measure some metrics about their product. For example, Figures 5(d) and (e) are two different fragments of changing the number of travelers in the Expedia app. If the recorder records the action in Figure 5(d), the same element cannot be found in Figure 5(e) since the structure is totally different.

9.5 Undetected Issues

As expected, A11yPuppetry cannot detect all forms of accessibility problems, and the best way to evaluate the accessibility of apps is by conducting user studies with disabled users. We categorized the limitation of A11yPuppetry in the following categories.

Improper Change Announcement. As users interact with mobile apps, the layout constantly changes. A sighted user can monitor all of these changes to understand the latest state of the app, while



(a) (b) (c) (d)

Figure 6: (a) After pressing the search tab in DoorDash, a new search page appears without any announcement; (b) List of saved stores in DoorDash; (c) The interstitial ad in Dictionary app and the close tab is not focusable by TalkBack; (d)The accessible calendar in Expedia.

it is much more difficult for users with visual impairment to realize something is changed in the app. During our interview, users reported a couple of these kinds of issues. For example, when the user presses the search tab in the Doordash app, the red-dashed box in Figure 6(a), a completely new search page appears without any announcement for TalkBack users. One participant mentioned "My preference is that whenever something like that happens, [TalkBack] moves the focus up to where the new content begins because someone as a screen reader won't necessarily [realize the app is changed]."

Excessive Announcement. On the other hand, it can be problematic and annoying when TalkBack announces content more than a user's need. For example, in the Expedia app, when a user types a name in the search edit box, TalkBack interrupts the user by announcing "Suggestions are being loaded below". Although it is informative for users to know the search results are loaded on the fly, it is annoying to interrupt constantly.

Temporary Visible Elements. Sometimes apps introduce new elements for a short period to notify the user something has changed and let the user undo or do something relevant to this change. For example. in the Doordash app, when the user saves a restaurant as her favorite, a pop-up box appears, Figure 6(b), notifying the user the store is saved and disappears momentarily. A blind user is informed of this change, but does not have enough time to focus on the appearing dialogue box.

10 DISCUSSION

The previous section demonstrates the effectiveness of A11yPuppetry in providing insights and detecting accessibility issues. This section discusses other findings from the user studies that might be insightful for future research work.

TalkBack Interaction Preferences. We further examined how users with visual impairments interact with apps using TalkBack. We asked the interviewees to explain the different ways they use TalkBack. If they did not mention any of the navigation ways that we found in TalkBack documentation, we asked them if they are aware of them.

Generally, the primary way of navigation mode for all participants is Linear navigation. A user mentioned "I'm more into the flick, element to element, to explore an app and understand its layout." This mode is especially used when the user interacts with an app or page that is unfamiliar.

The next favorite way of navigating is through Touch mode; however, it is usually used in certain scenarios. For example, when a user knows about the possible location of elements, the user is likely to use the Touch navigation mode. One participant mentioned "The back buttons are always at the top left, usually so... I'm going to put my finger at the top left to find that back button.". Also, when a user cannot find the element or is stuck in a loop, the user is more likely to use touch to find the target element.

Some interviewees said they might use Jump navigation for headings in the apps that they are familiar with. One participant said "If I don't know [the app] well enough ... I'm going to flick through the whole thing to figure out the layout. If I know it well enough, then I probably would switch to the heading option and

then search by heading". However, almost none of the participants are willing to use the Search navigation mode. One user mentioned "I know [search] is there. But I prefer to just hunt for [the elements]. It gives me a more experience with the app."

We also realized users do not want to use other actions like scrolling, since scrolling confuses them in understanding the new state of the app. A user said "[I use scrolling] if I know an app really well. But sometimes I find that when I do the scrolling thing, it'll get me into something else... sometimes it'll get me where I really don't want to be. So I have a tendency not to want to do it."

Context. A common accessibility issue in mobile apps is missing speakable text [2, 23]. Although missing speakable text degrades the user experience and ability to locate elements, sometimes users can infer the functionality of an unlabeled button given its context. For example, the user can view the list of saved stores in Doordash and remove any of them, as depicted in Figure 6(b). The element for removing a store is an icon with the shape of a heart without a content description. However, our interviewee did not have a problem with locating this button. He mentioned "That is a good layout, an accessible checkbox next to [the restaurant], which is checked unchecked. I have seen these checkboxes on the home screen. I don't like them on the home screen because the user doesn't know what that checkbox actually does. The common sense here would tell you I'm in the saved stores' section. So if I uncheck a box, it's going to remove that." Anyway, this observation should not encourage developers not to care about missing content descriptions; on the contrary, it emphasizes the importance of context for users with visual impairment to understand the app better.

Advertisement. In our experiments with A11yPuppetry, we did not observe any ads. However, if an interstitial ad appears during the replay process, A11yPuppetry may fail to continue as the appearance of ads is random and irregular. For example, for the Dictionary app, the interstitial ad, such as Figure 6(c), might appear when the user searches for a word. Disabled users have difficulty noticing the occurrence of the ads until they get stuck in the ads window for a few minutes. Even if they are aware of the ads, closing them and returning to the previously interrupted use case is challenging. One of the interviewees tried to locate the app with Linear and Touch navigation modes, but the ad's close button was not focusable by TalkBack. As a result, the user had to restart the app (close and open again) to continue the task.

All the interviewees are cautious about the in-app advertisements. As one stated, "I tend not to open [the in-app advertisements] because half of the time, these advertisements cause problems." In addition, most interviewees expressed a willingness to pay for the ad-free version if the price is not too high, so they do not have to deal with ads while navigating apps. A user mentioned: "If the app gives me the option to do without ads with a small price, I pay the small price just so I don't have to deal with the ads. Most of the time [the ads] don't work with the screen readers." Nevertheless, previous research indicates that some apps still contain ads even if users pay ad-free fees [32].

To the best of our knowledge, only one previous research investigated the impact of ads on disabled users. The research found that most ads are represented in GIFs, and more than half of the sampled ads have no ALT tag [64]. Therefore, screen readers cannot read the contents of the ads to blind users. Other researchers investigated the impact of ads on the whole user group, not just disabled users. The negative influences of ads include privacy threats, significant battery consumption, slowing down the app, and disabling an app's normal function [29, 32]. We believe that this negative impact is further magnified for disabled users.

There are some design implications for in-app advertisements. Generally, ads that take the entire screen are named interstitial ads, while ads that are represented as horizontal strips are named banner ads. The ads should be announced correctly via Assistive Services so disabled users can know the occurrence of the ads. In addition, developers are encouraged to design banner ads, since the banner ads usually do not disable an app's functionality. By contrast, interstitial ads significantly attract users' attention and even require users to close the ad manually [32].

Guided Navigation. The interviewees enjoyed interacting with an app when the app guided them through the process. In particular, Expedia did a great job in reserving flights: it consists of several steps like asking about the origin and destination airports, and dates. Once each step is done, the focus is changed to the next question and also announces the changes. Users are also able to get out of this selection and get back to the search page to change or view other information. One of the interviewees was especially happy about the calendar, Figure 6(d), and mentioned "That was one of the coolest mobile calendars I've ever used because it walked me through where I was. I selected the start date, and it told me that, and then it said, pick your end date, and then it summarized with states, like September 19th Start date or September 20th in the trip."

Alternative Suggestion. As we discussed before, there are several complex touch gestures that do not have a corresponding equivalent in TalkBack, e.g., dragging or pinching. Developers are recommended to provide alternative interactions for complex gestures. For example, the calendar widget in Expedia, Figure 6(d), is designed to allow sighted users to modify their travel dates by dragging the start date to end date. For TalkBack users, the app is designed to announce "Select dates again to modify" which is an alternative way of modifying the dates.

Common Sense. During the interviews, we noticed participants sometimes locate certain elements much faster than other elements. In particular, for elements like "Search" or "Back", instead of using Linear navigation, they explored certain parts of the app by Touch navigation to locate the element. We asked how they locate these elements and they generally responded to do so with the help of common sense. For example, the back button or open navigation drawer is usually located on the top left of the element, or menus are located in the footer. Common sense is not limited to similar elements on the screen. In the interview for the Doordash app, the interviewee found the button that shows the address of a restaurant pretty fast, even though the button was unlabeled. When we asked how he found such an element, he responded "A normal company would put the address on top, you know. So I'm using it. That's common sense." Therefore, it is important for developers to not change the spatial aspects of UI elements without considering users' habits.

11 CONCLUDING REMARKS

In this work, we introduced A11yPuppetry, a semi-automated record-and-replay technique for detecting accessibility issues in Android apps using TalkBack, the official screen reader in Android. A11yPuppetry records the user touch gestures in a device, translates the gestures into their equivalent action in TalkBack, and performs them on four different devices with four navigation modes in TalkBack. Finally, A11yPuppetry analyzes reports of the recorder and replayers and generates aggregated and visualized reports for developers. We evaluated A11yPuppetry by conducting user studies with users with visual impairments. We showed A11yPuppetry detects various types of accessibility issues that cannot be detected by existing tools. Our experiments also suggest the informative reports produced by A11yPuppetry can potentially aid developers with understanding and resolving the accessibility barriers in their apps.

In our future work, we would like to conduct a developer study to determine to what extent our tool can provide clues to help developers understand and resolve accessibility issues. We are also interested in exploring the application of A11yPuppetry as a pedagogical tool, e.g., using A11yPuppetry to help software engineering students learn about the impact of their implementation choices on users that interact with their software through an assistive service. Furthermore, we are interested in extending our implementation to support other assistive services and possibly on different platforms.[3] Our ultimate goal is to introduce this record-and-replay technique in the industry to evaluate its effectiveness in a large-scale setting.

ACKNOWLEDGMENTS

This work was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation and Sigma Xi Grants in Aid of Research. We would like to thank Yasaman Razeghi and Forough Mehralian for their valuable discussions and feedback on this work. We also acknowledge and appreciate the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2022. Automated Detection of TalkBack Interactive Accessibility Failures in Android Applications. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, IEEE, Virtual, 232–243.

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

[3] Abdulaziz Alshayban and Sam Malek. 2022. AccessiText: Automated Detection of Text Accessibility Issues in Android Apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 984–995. https://doi.org/10.1145/3540250.3549118

[4] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, Hong Kong, China, 403–410.

[5] Android. 2022. *Accessibility Scanner - Apps on Google Play*. Google. Retrieved May 6, 2022 from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US

[6] Android. 2022. *AccessibilityService in Android*. Google. Retrieved May 6, 2022 from https://developer.android.com/guide/topics/ui/accessibility/service

[7] Android. 2022. *Android accessibility overview*. Google. Retrieved May 6, 2022 from https://support.google.com/accessibility/android/answer/6006564

[8] Android. 2022. *Build more accessible apps*. Google. Retrieved May 6, 2022 from https://developer.android.com/guide/topics/ui/accessibility

[9] Android. 2022. *Espresso : Android Developers*. Google. Retrieved May 6, 2022 from https://developer.android.com/training/testing/espresso

[10] Android. 2022. *Get started on android with talkback - android accessibility help*. Google. Retrieved May 6, 2022 from https://support.google.com/accessibility/android/answer/6283677?hl=en

[11] Android. 2022. *Improve your code with lint checks*. Google. Retrieved May 6, 2020 from https://developer.android.com/studio/write/lint?hl=en

[12] Android. 2022. *TalkBack source code by Google*. Google. Retrieved May 6, 2022 from https://github.com/google/talkback

[13] Android. 2022. *Use Touch Gestures*. Google Inc. Retrieved August 29, 2022 from https://developer.android.com/develop/ui/views/touch-and-input/gestures

[14] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. 2000. Automating Web navigation with the WebVCR. *Computer Networks* 33, 1-6 (2000), 503–517.

[15] appetizerio. 2022. *Replaykit*. appetizerio. Retrieved September 2, 2022 from https://github.com/appetizerio/replaykit

[16] Apple. 2022. *Accessibility on iOS*. Apple. Retrieved May 6, 2021 from https://developer.apple.com/accessibility/ios/

[17] Apple. 2022. *Apple Accessibility*. Apple. Retrieved May 6, 2020 from https://www.apple.com/accessibility/iphone/

[18] Apple. 2022. *Debug Accessibility in iOS Simulator with the Accessibility Inspector*. Apple. Retrieved May 6, 2022 from https://developer.apple.com/library/archive/technotes/TestingAccessibilityOfiOSApps/TestAccessibilityiniOSSimulatorwithAccessibilityInspector/TestAccessibilityiniOSSimulatorwithAccessibilityInspector.html#//apple_ref/doc/uid/TP40012619-CH4-SW1

[19] Tingting Bi, Xin Xia, David Lo, John Grundy, Thomas Zimmermann, and Denae Ford. 2022. Accessibility in software practice: A practitioner's perspective. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–26.

[20] Jeffrey P Bigham, Jeremy T Brudvik, and Bernie Zhang. 2010. Accessibility by demonstration: enabling end users to guide developers to web accessibility solutions. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*. Association for Computing Machinery, Orlando, USA, 35–42.

[21] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C Miller. 2005. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. Association for Computing Machinery, Seattle, USA, 163–172.

[22] 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.

[23] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Guoqiang Li. 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, Virtual, 322–334.

[24] 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* 48 (2021), 3954–3968.

[25] Dictionary.Com. 2022. *Dictionary.com English Word Meanings & Definitions*. Dictionary.Com. Retrieved August 29, 2022 from https://play.google.com/store/apps/details?id=com.dictionary

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

[27] Mattia Fazzini, Eduardo Noronha De A Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A technique for recording, encoding, and running platform independent android tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, IEEE, Tokyo, Japan, 149–160.

[28] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2014. Tivos: Trusted visual i/o paths for android. *University of Michigan CSE Technical Report CSE-TR-586-14* (2014), 12 pages.

[29] Cuiyun Gao, Jichuan Zeng, Federica Sarro, David Lo, Irwin King, and Michael R Lyu. 2021. Do users care about ad's performance costs? Exploring the effects of the performance costs of in-app ads on user experience. *Information and Software Technology* 132 (2021), 106471.

[30] Greg Gay and Cindy Qi Li. 2010. AChecker: open, interactive, customizable, web accessibility checking. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. Association for Computing Machinery, Raleigh, USA, 1–2.

[3] We chose TalkBack for this study because its source code is publicly available.

[31] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, IEEE, San Francisco, CA, USA, 72–81.

[32] Jiaping Gui, Meiyappan Nagappan, and William GJ Halfond. 2017. What aspects of mobile ads do users care about? an empirical study of mobile in-app ad reviews. *arXiv preprint arXiv:1702.07681* (2017), 10 pages.

[33] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: self-replay augmented record and replay for Android in industrial cases. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*. Association for Computing Machinery, Beijing, China, 90–100.

[34] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, IEEE, Philadelphia, PA, USA, 215–224.

[35] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM New York, NY, USA, Bretton Woods, New Hampshire, USA, 204–217.

[36] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Association for Computing Machinery, Auckland , New Zealand, 349–366.

[37] Darris Hupp and Robert C Miller. 2007. Smart bookmarks: automatic retroactive macro recording on the web. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. Association for Computing Machinery, Newport, USA, 81–90.

[38] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, IEEE, Baltimore, MD, USA, 15–24.

[39] Kiran Kaja. 2022. *DoorDash Issue Tweet*. Retrieved September 15, 2022 from https://twitter.com/kirankaja12/status/1551710324016836608

[40] KIF. 2022. *Keep It Functional - An iOS Functional Testing Framework*. Retrieved November 28, 2022 from https://github.com/kif-framework/KIF

[41] 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, IEEE, Graz, Austria, 1–10.

[42] Will Lachance. 2022. *Orangutan*. wlach. Retrieved September 2, 2022 from https://github.com/wlach/orangutan

[43] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, Florence, Italy, 1719–1728.

[44] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's what I did: Sharing and reusing web activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, Atlanta, USA, 723–732.

[45] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. Association for Computing Machinery, Bremen , Germany, 6038–6049.

[46] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2020. Test automation in open-source android apps: A large-scale empirical study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM New York, NY, USA, Virtual, Australia, 1078–1089.

[47] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE, Shanghai, China, 613–622.

[48] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Virtual Event, USA, 1498–1508.

[49] Google Material Design. 2022. *Gestures*. Google Inc. Retrieved August 29, 2022 from https://material.io/design/interaction/gestures.html#principles

[50] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2022. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, ACM New York, NY, USA, Rochester, Michigan, USA, 13 pages.

[51] 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*. ACM New York, NY, USA, Virtual, Athens, Greece, 107–118.

[52] Diego Torres Milano. 2022. *Culebra*. Diego Torres Milano. Retrieved September 2, 2022 from https://github.com/dtmilano/AndroidViewClient/wiki/culebra

[53] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. Association for Computing Machinery, Texas, Austin, 571–582.

[54] Ranorex. 2022. *ranorex*. Idera, Inc. Retrieved September 2, 2022 from https://www.ranorex.com/mobile-automation-testing/android-test-automation/

[55] RobotiumTech. 2022. *robotiumrecorder*. RobotiumTech. Retrieved September 2, 2022 from https://github.com/RobotiumTech/robotium

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

[57] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE, San Diego, CA, USA, 128–138.

[58] 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*. ACM New York, NY, USA, Virtual, Okohama, Japan, 1–11.

[59] Navid Salehnamadi, Ziyao He, and Sam Malek. 2022. *A11yPuppetry companion website*. Retrieved Jan 31, 2023 from https://github.com/seal-hub/A11yPuppetry

[60] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. GroundHog: An Automated Accessibility Crawler for Mobile Apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, ACM New York, NY, USA, Rochester, Michigan, USA, 13 pages.

[61] 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*. DSAI, Thessaloniki, Greece, 286–293.

[62] StatCounter. 2022. *Desktop vs Mobile vs Tablet vs Console Market Share Worldwide*. StatCounter. Retrieved September 15, 2022 from https://gs.statcounter.com/platform-market-share

[63] Hironobu Takagi, Chieko Asakawa, Kentarou Fukuda, and Junji Maeda. 2003. Accessibility designer: visualizing usability for the blind. *ACM SIGACCESS accessibility and computing* 77-78 (2003), 177–184.

[64] David Thompson and Birgit Wassmuth. 2001. Accessibility of online advertising: a content analysis of alternative text for banner ad images in online newspapers. *Disability Studies Quarterly* 21, 2 (2001), 26 pages.

[65] W3. 2022. *Web Content Accessibility Guidelines (WCAG) Overview*. World Wide Web Consortium. Retrieved May 6, 2022 from https://www.w3.org/WAI/standards-guidelines/wcag/

[66] WHO. 2011. *World report on disability*. World Health Organization. Retrieved May 6, 2022 from https://www.who.int/disabilities/world_report/2011/report/en/

A USER STUDY TASKS

A.1 Dictionary

(1) Type the word "Coffee" in the search bar, the app should provide a list of entries, please select the first entry (which should be "Coffee").
(2) Listen to the pronunciation of the word by selecting the speaker button. Then read the IPA (International Phonetic Alphabet) of the word. You may need to select the "Show IPA" link to reveal the IPA of the word.
(3) On the same page, read the definition of the word "Coffee". It should start with "a beverage consisting of".
(4) Mark the word coffee as a favorite word by selecting the star button.
(5) Select the back or navigate up button in the app (not Android's general back button) to go to the main page. Then open the menu by selecting the navigation drawer button.
(6) In the menu, select "Word of the Day". Then on the new page, select the second word in the list.

(7) On this page, listen to the pronunciation of the word by selecting the speaker button. Then read the first example of this word which is located under "Examples:" section.
(8) Select the back or navigate up button in the app (not Android's general back button) to go to the main page. Then open the menu by selecting the navigation drawer button. In the menu, select "Favorites".
(9) On the "Favorites" page you should see the word "Coffee" which was marked as a favorite in step 4. Remove this word by selecting the edit button, then select the word "Coffee", and finally select the "Delete" button.
(10) After deleting "Coffee", the favorite page should be empty with a text in the middle saying "You don't have any favorites yet. Tap here to look up a word". Please select the "Tap here" link, and search for the word "Tea".

A.2 DoorDash

(1) Select the "Continue as guest" button, type "New York" in the address bar, and select the first entry (which should be "New York").
(2) In the address settings page, do not change anything and select "Save". Sometimes, a pop-up window will appear to inform you, "New! Send a gift to your loved ones". In that case, select "Go Back".
(3) Select the "Search" button, and type "Chicken" in the search bar. Please do not hit enter or search button once you're done typing.
(4) Select the second entry of the search result. In the restaurant page, save the restaurant by selecting the save button (with a heart icon). A window appears with the title "You've saved your first store"., In this window, select "View Saved Stores".
(5) Remove the saved restaurant by selecting the save button (with a heart icon). After selecting, the button should be toggled.
(6) Go back to the search screen by selecting the back or "Navigate up" button two times. Then Select the "Home" button.
(7) Go to the grocery category page by selecting the "Grocery" button. Then select the first store.
(8) Change the delivery option to pickup by selecting "Pickup" button (if you do not see the pickup button, try selecting another store). Once you select this button, the address of the store should be available below it under the title "This is a Pickup order"
(9) Now select the info button (with an exclamation icon) under the name of the store. It should take you to a new screen with information of the store such as address and phone number.
(10) Navigate back two times by selecting the "Navigate up" or back button, and finally select the "Orders" button. In the new screen, there should be a text "No recent orders".

A.3 ESPN

(1) Select the "Sign Up" button, and then select the "Change" link. It may or may not ask an email, you can provide a random email just to proceed.
(2) On the new window with the title "Where do you live?", choose "United States" from the drop down menu. Then select the "Done" button.
(3) Then press the back button (or reopen the app) to be on the first screen. Now select the "Sign Up Later" button. It may ask you to choose a region, select any region and select next.
(4) On the new screen, select one favorite league, e.g., "NBA" and then select the "Next" button.
(5) On the new screen, select one team, e.g., "Lakers", and then select the "Finish" button.
(6) A new screen may appear with the title "Stream your favorite teams and sports. Get ESPN+ now!". In that case, perform the back button. Now you should be on the main screen of the ESPN app.
(7) Select the "Scores" button on the bottom menu, and in the "Scores" screen, select the button next to the "Top Events" button, e.g., "NFL". In the showing results, either select the "HIGHLIGHTS" button or the notification button (with a bell icon) for one of the shown matches. Regardless of the button you selected, select the "Navigate up" or back button.
(8) Select "ESPN+" on the bottom menu, select the "Settings" button, and then select the "Edition" button.
(9) On the list of editions, select "Global", a dialogue appears to ask if you want to switch, select "Continue".
(10) You should be in the main screen, select the "Search" button, and type "NFL" in the search bar. An entry "National Footbal League" should be shown under "LEAGUES" section, select that button.

A.4 Expedia

(1) After opening the app for the first time, it shows an introduction page. Select the "Next" button until you reach the last screen. Then select "LET'S GO" button.
(2) Close the sign-in page by selecting the "Close" button. On the main screen, select the "Flights" button.
(3) On the the "Flights" page, select the "Flying from" button, type "New York" and select the first entry. Then it asks you for the destination or "Flying to". Type "Los Angeles" and select the first entry.
(4) Once you enter the airports, the app shows a calendar window to select the departure date. Select August 23rd and August 26th buttons, and then press the "Done" button.
(5) Now you should be on the Flights page. Change the traveler's number to 3 by selecting the Travelers button, then increase the number of adults by selecting the plus button two times. Then select the "Done" button.
(6) Now you should be on the flight's page. Select the "Search" button. Once the search results are provided, Then go to the main screen by selecting the "Navigate up" (or back) button and then the "Close" button, this should close the flights page.
(7) Select the "Cars" button. On the "Cars" page, select the "Pick-up" button and type "New York". Then select the first entry (which should be "New York").
(8) On the Cars page, select the "Search" button. Then go to the main screen by selecting the "Navigate up" (or back) button and then the "Close" button.
(9) Select the "Account" button. Under the "Settings" section, select "Choose a theme" button. Then select the "Dark" button and press "Done".

(10) Select the "Trips" button and then select "Sign in or create free account".

A.5 iSaveMoney

(1) Skip the tutorial by selecting Next.
(2) Once you get to the actual app page of iSaveMoney, select "Create your first budget" button.
(3) In the "New Budget" page, do not change the start and end dates, and just select "Next" button.
(4) In the "Select Categories" page, select the "ADD" button for "Daily Living" category.
(5) When the dialouge appears, type 1000 for the "Estimated Budget" field, then select the "Save" button, and finally, select the "Done" button.
(6) Now, you should be on the current budget page, where the title is the current month, for example, Jul 1 - 31, 2022. Select the "Add Expense" at the bottom. If you are a screen reader user, it may be the second "Add Expense" to select.
(7) In the "Add Expense" page, fill the form by picking a category (Daily Living), Writing something on the Description, e.g., "SomeExpense", and entering 500 in the "Amounts" textbox. Finally, Select the "Save Button".
(8) Now, you should be in the budget page. Try to collapse the "Total Expendture" section, by selecting the arrow inside this section.