Automated Accessibility Analysis of Dynamic Content Changes on Mobile Apps

Forough Mehralian
University of California, Irvine
fmehrali@uci.edu

Ziyao He
University of California, Irvine
ziyaoh5@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

***Abstract*—With mobile apps playing an increasingly vital role in our daily lives, the importance of ensuring their accessibility for users with disabilities is also growing. Despite this, app developers often overlook the accessibility challenges encountered by users of assistive technologies, such as screen readers. Screen reader users typically navigate content sequentially, focusing on one element at a time, unaware of changes occurring elsewhere in the app. While dynamic changes to content displayed on an app's user interface may be apparent to sighted users, they pose significant accessibility obstacles for screen reader users. Existing accessibility testing tools are unable to identify challenges faced by blind users resulting from dynamic content changes. In this work, we first conduct a formative user study on dynamic changes in Android apps and their accessibility barriers for screen reader users. We then present TIMESTUMP, an automated framework that leverages our findings in the formative study to detect accessibility issues regarding dynamic changes. Finally, we empirically evaluate TIMESTUMP on real-world apps to assess its effectiveness and efficiency in detecting such accessibility issues.**

***Index Terms*—Android, Accessibility, Screen Reader, Dynamic Content Changes**

I. INTRODUCTION

Dynamically changing visual content of screen (e.g., through animation) is a commonly used technique for enhancing the visual aesthetics of an app and to guide users' attention to specific parts of the app. However, these visually appealing techniques should not come at the cost of making apps inaccessible. In adherence to legal frameworks [1], [2], established guidelines [3]–[5], and ethical principles, the digital realm should be inclusive and accessible to all. This is especially crucial for the approximately 15% of the global population with some form of disability, including more than 300 million users that are blind or visually impaired [6].

Visually impaired users rely on assistive technologies like screen readers to interact with mobile apps. These tools enable users to navigate to a specific element on the screen and listen to the content in focus. However, the tunnel-like focus provided by screen readers may lead to unawareness of dynamic changes occurring elsewhere on the screen. A known example of such dynamic content is error notifications. When an app assesses user inputs and provides feedback, such as an error message through a notification, these changes may go unnoticed by screen reader users. Mobile platforms let developers designate these dynamically changing parts of a screen as "live regions", assisting screen readers to detect and announce such changes to users. Unfortunately, developers often neglect using proper accessibility attributes, posing significant accessibility challenges for the blind.

Earlier studies and guidelines addressing software accessibility have only scratched the surface of this critical issue. The related accessibility guidelines on this matter primarily center on designating live regions for screen reader announcements. Specifically, in scenarios involving error messages, Web Content Accessibility Guidelines (WCAG) success criterion 3.3.1 emphasizes the crucial need for users to be informed about errors and comprehend what went wrong and recommends techniques such as annotating error notifications as live regions [7]. However, the challenge extends beyond these scenarios. Dynamic changes have been neglected from prior studies and tools that rely on screen captures from an app to detect accessibility issues [8]–[10]. GUI crawlers and app explorers typically capture screenshots of an app under test after it is in stable conditions by waiting for certain amount of time [11], [12]. Unfortunately, these approaches fail to capture app states during the entire rendering process, overlooking changes that occur over time on the screen. Consequently, they are not capable of detecting accessibility issues caused by dynamic contents.

To bridge this gap, we initiated a formative study aimed at identifying various types of dynamic changes and assessing their impact on screen reader users. This study revealed characteristics of accessibility issues related to dynamic content changes that negatively impact blind users. Building on these insights, we developed TIMESTUMP, an automated framework designed to detect such issues in Android apps. TIMESTUMP comprises an automated crawler, randomly exploring diverse app states and capturing data before, during, and after each action. Subsequently, this data undergoes processing using the identified patterns from our initial study to pinpoint problematic dynamic changes for screen reader users. The identified issues are then reported and visualized for developers.

This paper makes the following contributions:

- The first study on accessibility issues arising from dy-

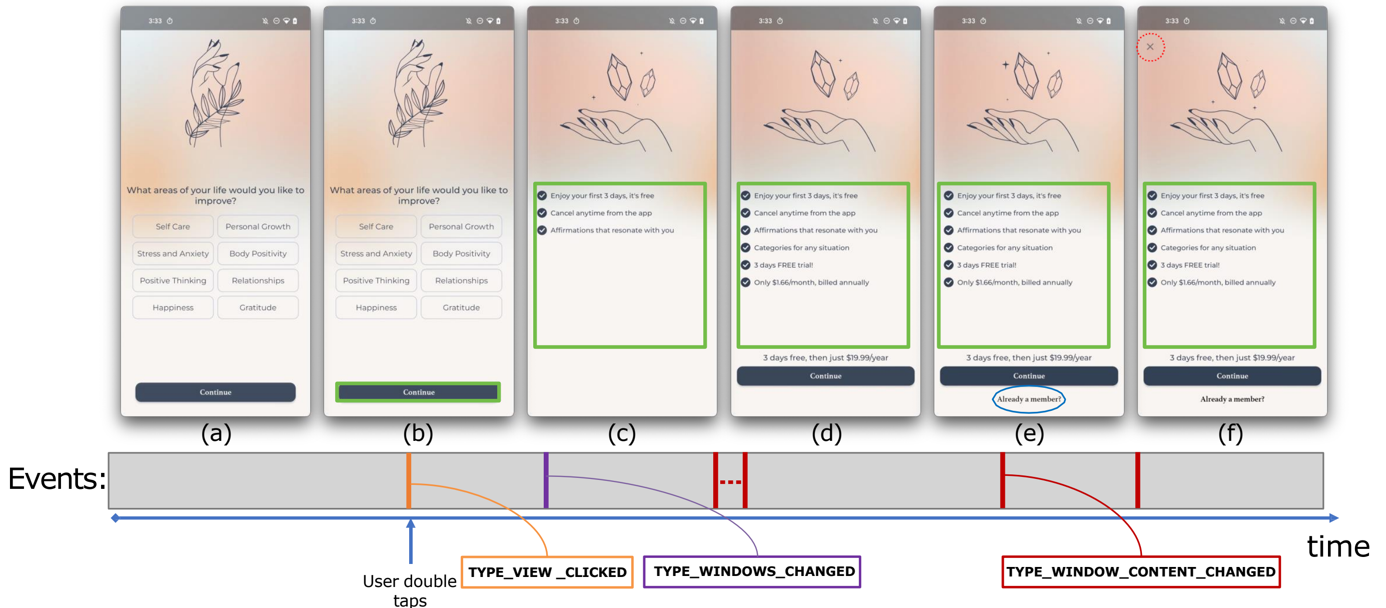

Fig. 1. Evolution of content loading on the screen across various states over time: (a) represents the initial screen state before the user initiates an action, (b) captures the moment when the user interacts with the app by clicking on a button, and (c) to (f) illustrate the gradual appearance of new screen content over time. Notably, in (f), the close button, indicated by a dashed red circle, appears above the accessibility focus. Since it is not tagged with `liveRegion` attribute, it is also not announced, and a screen reader user does not notice it.

namic content changes in Android apps.

- The introduction of the first automated crawler capable of capturing dynamic content changes, complemented by the creation of the initial dataset cataloging such behaviors.
- The development and public release of the first automated tool, named TIMESTUMP, designed for localizing and detecting accessibility issues related to dynamic content changes in Android apps [13].
- An empirical evaluation on real-world apps, corroborating the effectiveness of TIMESTUMP in detecting accessibility issues induced by dynamic screen changes.
- A user study involving blind participants to assess the impacts of dynamic screen change on app accessibility.

The remainder of this paper is organized as follows. Section II provides the background information. Section III describes our formative user study that motivated this work. Section IV presents TIMESTUMP, an automated approach for detection of problematic dynamic content changes. Section V details the evaluation of TIMESTUMP on real-world apps and in collaboration with blind participants. The paper concludes with a discussion of threats to validity, related research, and future work.

II. BACKGROUND

Mobile platforms offer the possibility of dynamic content changes, allowing developers to alter the screen content in real-time. Figure 1 displays an Android app called “I Am” [14] that provides daily affirmations for users and has more than 5 million downloads. When the screen reader focuses on the continue button as shown in Figure 1(b), the user can double-tap to perform the click gesture. Soon after clicking the button, the window changes, and some promotional content appears gradually, such as text, buttons, and other elements. For example, the “already a member” button, dotted in blue in Figure 1(e), and the close button, dashed red circle in Figure 1(f), appear after the bullet points are displayed.

This kind of screen rendering can pose severe challenges to screen reader users. Blind users utilize screen readers to interact with apps, and when encountering an unfamiliar app, they navigate through the on-screen elements sequentially to understand the app’s layout. The swipe right and left gestures allow the screen reader to move to the next or previous element, respectively, highlighting it with a green box as shown in Figure 1(b). When an element is focused, the screen reader vocalizes its textual description, enabling blind users to gauge its functionality in a manner analogous to how sighted users depend on the visual cues of an element. Should the textual description align with their expectations, blind users execute a double-tap, mirroring the single-tap action typical of sighted users. The following example illustrates the challenges blind users can face when dealing with dynamic changes. In Figure 1, as the user navigates to screen (c), the top element, which is the text view component, receives the accessibility focus. Screen reader users explore the screen by moving through the elements sequentially from top to bottom using a swipe-right gesture. However, the close button annotated in

dashed red is not recognizable as it appears on top of the screen and users are less likely to traverse backward, to the area they already visited. Such barriers can lead to unintentional interactions with ads or difficulties navigating away from them.

In Android, the guidelines suggest using an attribute called `liveRegion` to help screen readers recognize the appeared content. When an element is annotated as `liveRegion`, it is announced by the screen reader. Android system utilizes an event-based model to inform screen readers of changes in live regions. A GUI element emits an `AccessibilityEvent` when there are changes to its state, which is received by assistive technologies such as a screen reader. Figure 1 illustrates several different types of events that can be triggered during the loading of app content, with each color representing a distinct event. For example, `TYPE_VIEW_CLICKED` events occur after a view is clicked, `TYPE_WINDOWS_CHANGED` events happen when the app transitions to a different window, and `TYPE_WINDOW_CONTENT_CHANGED` events take place after the content inside a window changes.

Assistive technologies can also identify the element that is the source of events. In Android, GUI elements are represented by a tree of `AccessibilityNodeInfo` objects that mirror the XML hierarchy of elements and their attributes.

`AccessibilityNodeInfo` tree can be likened to the Document Object Model (DOM) tree in the case of web pages, offering a hierarchical representation of rendered elements on a web page. Prior studies on exploring various states of web apps for testing purposes have characterized dynamic content changes as modifications to the DOM that occur without reloading the page [15], [16]. These changes include updating or disappearing content [17], reordering elements [18], and inserting specific elements [19], as outlined in the WCAG guidelines. Similar to WCAG guidelines, Android suggests using accessibility attributes to notify screen reader users of such changes [20].

However, these guidelines only scratch the surface of the issues that may arise as a result of dynamic contents. For instance, when a temporary button, like an undo button, pops up on the bottom of the screen, users may struggle to locate it within the brief time frame of its visibility. This challenge intensifies when content disappears before users become aware of its existence. Merely relying on accessibility attributes does not fully resolve this issue.

III. Formative Study

We conducted a formative study to investigate the impact of various dynamic content changes on screen reader users.

A. Study Design

Prior studies and guidelines on Web defined dynamic content changes as modifications to the DOM that occur without reloading the page [15], [16]. Consequently, in Android apps, dynamic content changes encompasses any modifications to the hierarchical representation of elements, i.e., a tree of `AccessibilityNodeInfos`, in a rendered window. These modifications include adding, removing, or changing attributes of elements. To have a better understanding of different types of dynamic content changes in Android, two authors conducted an empirical analysis of 50 Android apps. These apps were randomly chosen from the Google Play Store, representing various app categories. Additionally, to ensure the significance and popularity of the apps studied, each app selected had a minimum of 1 million downloads. For each app, two authors manually explored the app screen using a combination of actions, such as clicking, scrolling, and typing, to observe if that would trigger dynamic content changes. If so, a recording from the screen is taken with a brief description of the dynamic content change. Following this, two authors engaged in an iterative open-coding process to categorize the types of dynamic content changes identified.

Through our empirical analysis, we identified 5 types of dynamic content changes.

Appearing Content. This content change type is characterized by an element that initially is not present on a loaded window, but appears a few moments later and remains on the screen. For example, in Figure 1(f), the close button, marked by a red circle, appears after a few seconds.

Disappearing Content. This content change type describes an element that disappears either after a set period or as a result of user interaction. For example, in Figure 2(i), the navigation bar at the bottom including element **a** as well as the more button on top, element **b**, disappear when users navigate through the list of items.

Short-Lived Content. This content change type relates to an element that initially is not present on the screen but appears and remains on the screen only for a brief duration. Due to its transient nature, we refer to this as short-lived content. For example, as illustrated in Figure 2(iii), when users save a restaurant, a notification message annotated as **e** appears to notify them that they have successfully saved the store and to offer an option to view all saved stores. This message disappears after a few seconds.

Moving Content. This content change type refers to an element that is initially visible on the screen but subsequently gets relocated to a different part of the screen. For instance, in Figure 2(ii), the app-related information, marked as **c**, is shifted to the bottom of the screen and goes out of screen bounds after user presses the install button. Users must locate that information at a different position within the sequence of elements on the screen, as perceived by screen readers.

Content Modification. This content change type pertains to an element that remains on the screen but its attributes change. For example, in Figure 2(ii), the `TextView` annotated as **d**, continuously refreshes its text to display the progress of the app installation process.

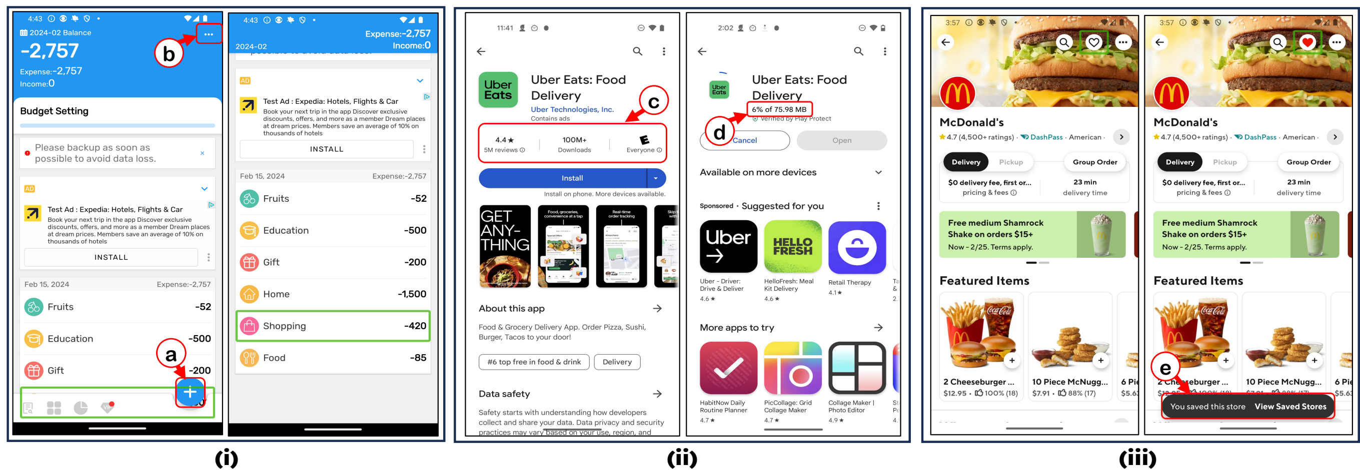

Fig. 2. Examples of dynamic content changes: (i) the add button (annotated as a) and the more button (annotated as b) disappear when users continue exploring the screen, (ii) the app information (marked as c) moves to the bottom of the screen after hitting the Install button; the text (annotated as d) constantly changes to indicate installation progress, (iii) the short-lived notification (annotated as e at the bottom) after saving a restaurant.

Having identified different types of dynamic content changes in Android, our objective was to understand their impacts on screen reader users. To this end, we selected 4 apps that collectively represented all identified types of dynamic content changes. We then designed 5 specific tasks that would involve interactions with these dynamic changes. Our objective was to understand whether the users can perform the tasks, whether they can perceive the dynamic changes in the apps, and to generally develop a better understanding of how the dynamic changes impact app accessibility. To recruit participants, we leveraged the Fable platform [21], which connects tech companies with disabled users for accessibility testing. Each user interview session was conducted over a one-hour period. During these sessions, we requested participants to share their phone screens and perform our designed tasks while vocalizing their thoughts and actions, aka think aloud [22]. This think-aloud method, combined with in situ questioning, enabled us to observe their understanding of the dynamic content changes and assess their ability to complete the tasks successfully. Our 3 blind participants included one female and two males, all of whom demonstrated a high proficiency in using the TalkBack.

B. Results

The user interviews focused on all the apps depicted in Figures 1 and 2. To generate a comprehensive list of accessibility issues related to dynamic content changes, two authors thoroughly examined each interview session. Initially, they independently identified areas where screen reader users faced confusion and attempted to ascertain the underlying reasons. Then, they engaged in discussions to reach a consensus. The following list outlines the dynamic content changes that proved challenging for screen reader users in our study.

Latent Appearing Content. When content appears without being annotated as a live region and is situated in an area previously explored by the user, it remains latent or unknown. For instance, the close button on an app's promotional page, as shown in Figure 1(f), emerges after a few seconds without alerting blind users. During the interview, blind users had already navigated past it, possibly interacting with elements located lower on the screen. This led to confusion when attempting to exit the promotional page, requiring users to employ various strategies such as using the TalkBack back gesture or re-navigating the screen.

Latent Disappearing Content. Content that vanishes before the user explores that region of the app stays undiscovered. In Figure 2(i), annotated buttons **b** and **a** disappear as users swipe through the list of items. The disappearance of the *add* button (element a) presented specific challenges for blind participants, as the button disappears in an unexplored area. As a result, they were unable to locate the element to add a new cost entry to the list. Conversely, the *more* button (element b) remains accessible, as users had already visited that element before its disappearance and when navigating backward, the more button becomes visible again.

Latent Short-Lived Content. When an element appears temporarily, it may be inaccessible to the user, especially if the element is actionable. The time it takes for the user to navigate to that element and perform the action might exceed the visibility period of short-lived content, leading to accessibility issues. In Figure 2(iii), we observe a brief notification annotated as **e** that emerges after users save a restaurant. Our user interviews revealed that participants were aware of this notification, understanding that they had successfully saved the restaurant and that the app offered an option to view all saved stores. However, this notification disappeared within a

few seconds. For this type of dynamic content, participants only partially grasped the situation. While they recognized the appearance of the notification, thanks to the live region annotation, they did not fully understood its transient nature and were unable to click on the *View Saved Stores* button. One interviewee expressed, “It was a flash or pop up, and it went away. I would expect to be able to navigate to that button, but when I move back and forth, it is gone.”

Latent Moving Content. When the location of a previously visited element changes, it can cause confusion for blind users. As illustrated in Figure 2(ii), the app-related information, including app rating and download number, relocates to the bottom of the screen after pressing the install button. In our study, blind users were assigned the task of installing an app and finding its download number. After installation, they navigated backward, relying on the previous announcement by TalkBack about the download number during their journey to the install button. However, to their surprise, upon navigating back, the information they sought was no longer present, resulting in confusion and a period of being stuck on the page. None of the participants completed the task, with one participant believed that he could eventually locate the download number by navigating further down, acknowledging it would take more time.

Latent Content Modification. Changes in attributes of elements that are noticeable by sighted users may go unnoticed by users relying on screen readers. During the formative study, the `TextView` (element **d**) in Figure 2(ii) continuously updated its text to reflect the progress of the app installation. The proper implementation of the `liveRegion` feature ensured that changes in the `TextView` content were effectively announced to blind participants, enabling them to accurately comprehend the status of the installation process. Conversely, this suggests that if the `liveRegion` feature is not correctly implemented, it becomes challenging for screen reader users to understand content modification, such as the installation status. In addition. changes in attributes such as size and color, which are not perceptible by screen readers, do not impact their perception of the app.

IV. Approach

Relying on the insights gained from formative interviews with screen reader users, we developed an automated framework called TIMESTUMP, designed to identify accessibility issues associated with dynamic content changes. Figure 3 provides an overview of TIMESTUMP, highlighting the three distinct phases of its operation. In the initial phase, we install an Android app on a Virtual Machine (VM) and utilize a GUI crawler to automatically explore the app, generating a diverse set of states in an app. The Snapshot Recorder tracks app states and records snapshots of distinct screens. In the second phase, we extract the list of actionable elements

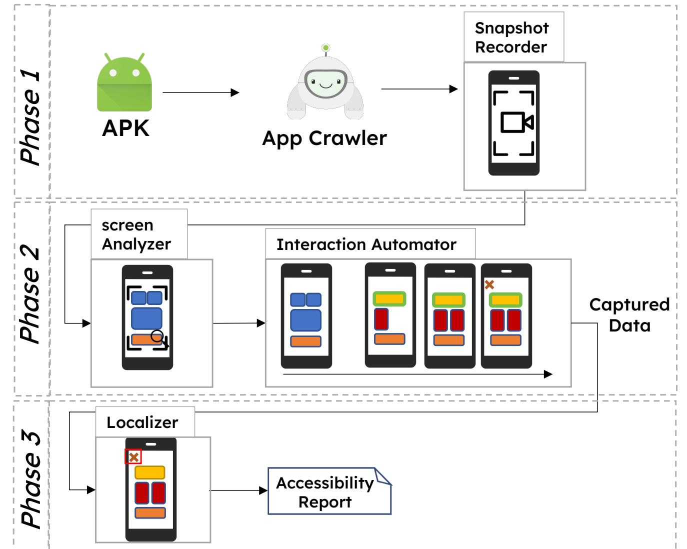

Fig. 3. TIMESTUMP’s approach overview.

in each recorded snapshot. Then, the Interaction Automator systematically executes each action, capturing information before, during, and after the action. This rich dataset is passed to the third phase, where the Localizer component assesses the gathered information, precisely flagging accessibility issues stemming from dynamic content changes.

We now describe the details of each phase.

A. Phase 1: Capturing Unique App Screens

The main goal of this phase is to navigate through an app and explore its different states for subsequent examination. Interacting with the app leads to various state changes, ranging from subtle modifications in attributes, such as selecting a checkbox on the screen, to more significant changes like transitioning to an entirely new screen, resulting in a completely different hierarchical structure of elements. In this phase, our focus is on identifying a diverse set of screens from each app that have undergone significant changes. Detailed assessment of minor changes is reserved for subsequent phases.

To facilitate the testing of GUI apps, a variety of tools, such as STOAT [23], Monkey [24], Spaienze [25], and APE [26], have been specifically designed to traverse the expansive domain of an app’s different states. TIMESTUMP seamlessly integrates with any existing app exploration tool, providing the flexibility to traverse various app states. Additionally, TIMESTUMP allows manual testers to explore the app with diverse scenarios in mind or leverage existing GUI test cases.

The Snapshot Recorder plays a pivotal role in tracking alterations. As the crawler interacts with the app, Snapshot Recorder keeps track of the screen structure and *Activities*, i.e., an Android component representing a single screen. It then captures VM snapshots from app states involving changes to activity names or the hash value of hierarchical structure of

elements. Similar to previous studies [11], the hash function excludes nodes that are not important for accessibility, i.e., those not notified by screen readers, as well as attributes such as `checked` or `enabled` that do not contribute to recognizing a different screen in an app. VM Snapshots enable us to load the app from the exact state and perform further analysis without concerns about the impacts of prior interactions.

B. Phase 2: Monitoring Apps in Action

During this phase, each potential interaction within a given app snapshot is automatically executed, all while monitoring the app for dynamic content changes.

To achieve this, we first load a VM snapshot. The Screen Analyzer employs an accessibility service to extract and dump the hierarchical structure of elements on the screen. The outcome is a tree structure wherein each node represents an element, accompanied by various attributes such as clickability. Parsing this node tree, the Screen Analyzer identifies all interactable elements and enumerates the types of actions they support, such as click, type, or swipe.

Every element is uniquely identified by its resource-id and a set of other attributes such as text, content description, and class name. The resource-id serves as a distinctive marker for locating each element. In instances where developers have not assigned a resource-id, the combination of other attributes can help in locating the element.

The Interaction Automator receives the comprehensive list of actions identified by the Screen Analyzer and executes them on the app while collecting certain data before, during, and after each action. The Interaction Automator consists of two main components: Controller and Accessibility Service.

The Controller functions as a server, sending commands to the client—the Accessibility Service, which operates in the background. The Accessibility Service is responsible for interacting with elements on the device. The client captures two frames of the app: *first frame* and *last frame*. The first frame corresponds to the initial state of the window, while the last frame corresponds to the app's state once all the content has finished rendering. The first frame primarily reflects the state of the app before any action takes place. However, if an action triggers a window change, this first frame then denotes the app's status immediately following the action. To accomplish this, TIMESTUMP tracks accessibility events that signal either the loading of a new window or shifts in accessibility focus. When a window change occurs, the first frame is recorded immediately after detecting the event that indicates a change in the window. For example, in Figure 1, the TYPE_WINDOWS_CHANGED event, highlighted in purple, signifies a window transition. Consequently, Figure 1(c) is identified as the first frame. In the absence of such events, the first frame defaults to the state of the app before executing the action.

For the last frame, TIMESTUMP listens for accessibility events indicating window content changes and, if none occur for more than 5 seconds, it captures that final state. For instance, Figure 1(f) is designated as the last frame, indicating that all changes on the app screen have been finalized. This practice is common in prior studies [11] and automation tools [27], ensuring app stability before data capture. The waiting time can be configured to be as long as necessary, or even adaptive to the specific app to optimize efficiency [12]. In instances of continual changes, such as animations or ads, a timeout period is implemented to bypass waiting.

The Controller stores these frames, as well as real-time logs of accessibility events generated by the Accessibility Service throughout the entire action execution period. Leveraging this extensive dataset empowers the Localizer to pinpoint accessibility issues arising from dynamic content changes.

C. Phase 3: Localizing Problematic Dynamic Changes

TIMESTUMP analyzes the collected data to identify various categories of latent content changes for screen reader users. This analysis is conducted across captured frames of the app, as well as the accessibility events captured during the execution of each action on the screen.

When a screen element undergoes a change, it triggers an event of type `Window_Content_Changed`. TIMESTUMP identifies sources of such events within the captured frames of the app, forming the initial set of candidate elements within a window that undergo a problematic change. The Localizer then compares elements in the final frame against those in the first frame to pinpoint the problematic changes.

As explained in Section IV-B, if the execution of an action results in a window transition, the first frame is the newly loaded window, i.e., the frame captured immediately after performing the action, similar to Figure 1(c). To detect window transitions, Localizer examines accessibility events of type `Windows_Changed` and `Window_State_Changed`, which are also used in the Android source code to detect the appearance of a new window [28]. Subsequently, we elaborate on the logic employed for detecting various types of problematic changes.

Due to space limits, we provide an intuitive explanation of how TIMESTUMP localizes each issue here, and provide the detailed algorithms on the companion website [13].

Latent Appearing Content: As explained in Section II and Section III, if certain content appears in previously explored areas (i.e., above the accessibility focus in default navigation order) and is not designated as a live region, it remains unknown to the screen reader user. The Localizer classifies elements in the final frame that trigger a content change event as latent appearing content if (1) they do not appear in the first frame, (2) are not designated as live regions, and (3) are positioned before the current accessibility focus.

Latent Disappearing Content: When an element disappears from the screen, a change event is triggered, similar to the case of appearing content. However, in this scenario, the event's source is the container of the vanishing element. For example, if a button within a linear layout disappears, the event source will be the linear layout, potentially covering the entire screen. Consequently, the Localizer evaluates all the children of an event publisher node in the first frame to verify their presence in the final frame. A child node is categorized as latent disappearing content if (1) it is not observed in the last frame, (2) it is not designated as a live region, and (3) it is positioned after the current accessibility focus.

Latent Short-lived Content: Elements that appear and disappear have a brief visibility period. Even if this content is announced, navigating to them and interacting with them using screen readers is challenging. The Localizer identifies these elements by searching for pairs of change events and localizing their sources denoted as <S1, S2> in two consecutive frames, checking whether S2 is the container of S1. For any such found pair, an element S1 is categorized as latent short-lived content if (1) S1 is not present in the first frame, (2) its container S2 is observed in the second frame, and (3) S1 is not designated as a live region or is actionable.

Latent Moving Content: The Localizer examines elements displaying a shift in their position on the screen across different frames while maintaining consistent identifiers such as resource-id, and content description. Elements are flagged as problematic if their changed position is above the accessibility focus or if they move beyond the screen boundaries.

Latent Content Modification: The change of attributes in an element refers to any modification in the properties that define an element's behavior, or metadata within a UI. Localizer uses a hash function to detect such modifications across different frames. The hash function encodes the element attributes that are important in exploring the app with screen readers. A discrepancy in the hash values of an element between any two frames signifies a modification in the element's content. When the `liveRegion` attribute is absent, the change remains unknown to screen reader users.

V. Evaluation

We evaluated TimeStump on real-world apps and with the help of several blind users to answer the following questions:

- **RQ1.** How accurate is TimeStump in detecting dynamic content changes and different categories of accessibility issues?
- **RQ2.** How do the issues reported by TimeStump impact the screen reader users?
- **RQ3.** What is the performance of TimeStump?

A. *RQ1. Accuracy of* TimeStump

1) Experimental Setup: For this experiment, we utilized STOAT [23] as the app exploration tool. We evaluated our approach on 30 real-world Android apps. Our test set consists of two groups of apps: (group1) 10 apps with manually verified dynamic content changes from different categories of Google play store, (group2) 20 randomly selected apps with known accessibility issues from a prior study [11]. For apps in group1, the authors installed top rated apps in different categories of Google play store and manually explored each app, looking for dynamic content. They captured a VM snapshot of each app at that state, with the accessibility focus set on the target element such that performing the action on the target element results in the dynamic change. For apps in group2, we set the crawler to automatically get VM snapshots from two random unique states of the app. The tool then explores the actionable elements in each state to get the real-time data.

The precision of the captured data directly impacts the ability of TimeStump to detect changes in dynamic content. Before evaluating TimeStump's effectiveness, we manually reviewed the captured data from a test set to confirm alignment with our definitions of the first and last frames, and encountered no issues. For this experiment, we chose 15 unique app states from 5 random apps from a prior study [12], encompassing a range of app transitions such as implicit loading, explicit loading, and transitions. Additional information about this study can be found on the companion website [13].

All experiments were conducted on a typical computer setup for development (MacBook Pro, Apple M1 Max, 32 GB memory). We used the most recent distributed Android OS (SDK34), and the latest versions of Android screen reader.

2) Results: To answer this question, the authors manually examined the issues reported by TimeStump and tagged them as False Positive (FP) if the reported issue is not correct, and True Positive (TP) if the reported issue correctly detects and categorizes problematic dynamic content changes. The authors used an emulator to load the captured snapshot and manually interacted with the app using TalkBack. This process allowed for the identification of legitimate dynamic elements in exploring the app with screen readers. We then report precision as the ratio of the number of TPs to the number of all detected issues. As shown in Table I, the overall precision over all the elements in 130 actions of apps is 0.94. To compute recall, we manually reviewed apps in both group 1 and group 2 to identify dynamic changes and establish the ground truth. In group 1, snapshots were captured during manual exploration of the app, revealing states with dynamic content changes. In contrast, for group 2 apps, we manually inspected automatically captured snapshots of random app states for dynamic content changes. Dynamic elements missed by TimeStump were identified and manually labeled as False Negatives (FN). The overall recall across 130 actions is 0.92, as depicted in Table I.

Figure 4 presents examples of problematic dynamic elements detected by TimeStump. In Figure 4(a), the four

elements highlighted by orange boxes emerge above the accessibility focus after tapping on the *plus* button. Figure 4(b) illustrates short-lived elements, including a button, indicated by blue boxes, appearing after adding a song to favorites. As all of those elements are actionable, TIMESTUMP reports them as problematic. In Figure 4(c), a `TextView`, annotated by the black box, is updated following the tap on the *CALCULATE* button. Since this element is not tagged as live region, it remained unannounced while exploring the app with TalkBack. Thus, TIMESTUMP reports it as problematic.

We analyzed the failures of TIMESTUMP and identified issues falling into two main themes, resulting in both false positives and false negatives.

The first pattern relates to inadequate identifiers assigned by developers to elements. For instance, in the Fuelio app, a `FrameLayout` serves as a container for its child elements, lacking essential identifiers such as `resource_id`, `text`, and `content_description`. As a result, its unique identification relies solely on its screen bounds. However, when an action triggers a layout modification, the element's screen bounds also change, leading TIMESTUMP to mistakenly interpret this as an appearing element, thus generating a false positive. Moreover, the absence of sufficient identifiers can result in false negatives. In the ESPN app, certain elements possess identifiers that are neither empty nor unique. TIMESTUMP primarily depends on these identifiers to match an element, but since they are not distinctive, TIMESTUMP fails to differentiate between elements on the screen, consequently failing to report associated issues.

Another category of failures occurs when the screen displays multiple windows, such as step-by-step guidelines overlaid on the main app. ADB allows us to capture the `AccessibilityNodeTree` of the foremost window only, thereby missing content in other windows. This limitation contributes to both false positives and false negatives.

B. RQ2. Qualitative Study

To assess the impact of the issues detected by TIMESTUMP on screen reader users, we conducted 15 user studies. We randomly selected three apps from RQ1, representing various types of dynamic content changes, corresponding to IDs *P8*, *G1*, and *G7*, as listed in Table I. Our qualitative study consisted of 10 self-guided tasks and 5 user interviews with blind testers, recruited through the Fable platform [21]. In the self-guided tasks, testers were given concise task descriptions to execute offline on apps while recording their screens and articulating their thoughts aloud. This approach, without a moderator present during sessions, helped mitigate interviewer bias. Additionally, user interviews were conducted to explore incidents in different states of one app, *G7*, and ask follow-up questions. Due to the limited size of the tester pool on the platform, some tasks involving different apps were assigned to the same tester. However, no tester evaluated the same app multiple times. In total, 8 distinct testers participated in this study: 6 males, 2 females, with 7 identifying as White and 1 as Asian. Participant ages ranged from 20 to 45. Below, we first outline the issues that users confirmed. We then discuss the observed shortcomings and insights gained.

1) User Confirmed Issues: Of the 30 issues identified, users directly confirmed 25, yielding a confirmation rate of 0.83.

Appearing elements in explored areas. Figure 4(a) depicts an instance from this category. Blind testers were tasked with locating the `Gas` entry button, which appears after tapping the `Plus` button. The target button, along with three other elements highlighted with orange boxes on Figure 4(a), emerged in areas previously explored by users, without any notification. Consequently, users felt as if nothing had changed after tapping the `Plus` button. One participant expressed, *"It's very confusing and disorienting when the screen changes without any audio feedback from the screen readers"* Another noted, *"Usually, new elements appear below [the current TalkBack focus], but in this case, they appeared above."* The appearance of elements in previously explored areas caused confusion for screen reader users, resulting in longer times to locate the desired element. Two out of five interviewees were unable to find the targeted button and complete the task, while others had to explore the screen multiple times to do so. A similar issue in self-guided tasks resulted in confusion for all the participants. For elements that appear dynamically, blind testers recommended setting the `liveRegion` attribute appropriately. They suggested moving the TalkBack focus to the first new element on the screen in cases of significant window changes. For minor window changes, introduce dynamic elements in unexplored screen areas.

Disappearing elements in unexplored areas. In one of the test apps, activating a switch at the top caused some form entries to disappear. Testers interacting with the switch were not informed of the changes and were confused as to why they could not find certain elements. Four out of five interviewees were unable to complete the task, concluding that the required element was not present on the screen. Conversely, one interviewee managed to find the desired element by turning off a switch, leveraging his prior experience with such controls. Among the interviewees who failed to perform the task, one person remarked, *"[I] Thought the Recurrence section was either not on the screen or not visible to TalkBack. I just couldn't find it."* The tester expressed a preference for receiving a notification indicating that *"new controls are available or shown"* once the checkbox is ticked. This would enable them to recognize that the layout of the app has changed on the same screen and understand how to revert the layout to its original configuration.

Short-lived buttons. As depicted in Figure 4(b), when users

TABLE I
THE ACCURACY OF TIMESTUMP ON SUBJECT APPS.

ID	App	Category	#Installs	# Issues	TP	FP	FN	Precision	Recall
P1	Autozone	Auto & Vehicles	>5M	5	5	1	0	0.83	1
P2	Duolingo	Education	>500M	19	18	1	1	0.94	0.94
P3	Forest	Productivity	>10M	7	5	0	2	1	0.71
P4	Gratitude	LifeStyle	>1M	8	8	0	0	1	1
P5	Motivation	Health & Fitness	>5M	3	3	0	0	1	1
P6	Starbucks	Food & Drink	>10M	4	4	1	0	0.8	1
P7	TicketMaster	Events	>10M	1	1	0	0	1	1
P8	Spotify	Music & Audio	>1B	5	4	0	1	1	0.8
P9	H&M	Lifestyle	>50M	1	1	0	0	1	1
P10	File Manager	Tools	>1B	20	11	0	9	1	0.55
G1	Booking.com	Travel & Local	>500M	39	36	0	3	1	0.92
G2	Easy Bills Reminder	Finance	>100K	2	2	0	0	1	1
G3	Burn	Education	NA	6	2	0	4	1	0.66
G4	Dictionary.com	Books & Reference	>10M	4	4	0	0	1	1
G5	ESPN	Sports	>50M	3	2	0	1	1	0.66
G6	Calorie Counter by FatSecret	Health & Fitness	>50M	58	58	5	0	0.92	1
G7	Fuelio	Auto & Vehicle	>1M	101	92	5	9	0.94	0.91
G8	Life360	Lifestyle	>100M	4	4	0	0	1	1
G9	Master Lock Vault Enterprise	Lifestyle	>100K	4	1	0	3	1	0.25
G10	Nike	Shopping	>50M	25	24	0	1	1	0.96
G11	Weee! Asian Grocery Delivery	Food & Drink	>1M	8	8	0	0	1	1
G12	Norton Secure VPN	Tools	>10M	2	2	0	0	1	1
G13	TripIt	Travel & Local	>5M	7	7	1	0	0.87	1
G14	ToonMe photo cartoon maker	Photography	>50M	2	2	0	0	1	1
G15	Vimeo	Entertainment	>10M	6	6	0	0	1	1
G16	Yelp	Food & Drink	>50M	1	1	0	0	1	1
G17	The Clock	Productivity	>10M	56	55	11	1	0.83	0.98
G18	King James Bible	Books& Reference	>50M	23	21	2	2	0.91	0.91
G19	Lyft	Maps & Navigation	>50M	11	11	0	0	1	1
G20	To-Do List - Schedule Planner	Productivity	>10M	50	47	2	3	0.95	0.94
Overall				485	445	29	40	0.94	0.92

add a song to their favorites, a notification pops up and let them revert the action by tapping on the `Change` button. Three of the users became aware that they could potentially use this element. Two participants missed the button because TalkBack simultaneously announced three short-lived elements, which overwhelmed them. Moreover, during self-guided tasks, none of the participants could interact with the `Change` button as it disappeared quickly. As a result, three participants could not accomplish the task for removing a song from the favorites. Two participants were able to remove the song through an alternative method, using the ticked button, annotated by a green box in Figure 4(b). Therefore, short-lived elements should not overwhelm blind users with excessive information. Additionally, it is recommended to avoid including clickable elements in a short-lived manner, as blind users navigating sequentially with a screen reader are likely to miss them before they vanish.

Unannounced Short-lived Elements. In the Spotify app, when users removed a song from their favorites, a short-lived notification, with the text *Removed from Liked Songs*, appeared signaling this change, providing immediate feedback to sighted users. However, TalkBack did not announce the change to the screen reader user, leading to confusion. Only two participants advanced to the step of removing a song from the favorites in our self-guided tasks. For those screen reader users, they were uncertain if the song had been successfully removed and felt compelled to navigate through the entire screen to verify that. When a song has not been added to favorites, the `Plus` button is labeled with the content description *Add Item.* Conversely, when the song is in the favorites, its description changes to *Item Added.* As a result, the blind users need to navigate the screen to check if the content description has reverted to *Add Item.* in order to confirm that their action was successful. Their experience suggested that the `liveRegion` attribute should be appropriately configured for short-lived notifications.

Unannounced Content Modification In Figure 4(c), tapping the `CALCULATE` button triggers an update of the result, indicated by the black box, appearing above the button. However, this change is not communicated to screen reader users, forcing them to navigate back to check the calculation result. Although all participants in our self-guided tasks managed to find the calculation result by navigating back and forth, they reported it as confusing and inconvenient. Additionally, if users press the `CALCULATE` button without providing any input for prior entries, they receive an error

message advising them to input values before proceeding. One participant noted, *"For the sake of consistency, having the calculation result announced just like the error message would not disappoint me."* Proper utilization of the `liveRegion` attribute would alleviate such issues.

2) Observed Shortcomings: The user study also shed light on TIMESTUMP's shortcomings and enhancement opportunities.

Reverting Changes. TIMESTUMP evaluates the changes resulting from each action. However, a series of actions may have counteractive impacts. For instance, For example, in the Spotify app, top views move up as users navigate toward the bottom. As soon as they attempt to navigate back to those top elements, the views are restored to their original position, and users do not perceive any problem. Our manual exploration of our test set reveals 8 elements with similar issues that may not be problematic for users. However, for users who rely on alternative interaction modes, such as *explore by touch* where TalkBack shifts its focus to the coordinates of the touch gesture, these cases can still be confusing.

Severity of Issues. The dynamic elements identified as problematic exhibit varying degrees of severity and impact on blind users, with TIMESTUMP unable to prioritize them by severity. Factors such as the frequency of the issue among different apps and users' familiarity with it contribute to its severity. For instance, during interactions with the Booking.com app, switching tabs changes the content of the window without altering the screen reader focus or providing any notification. While this issue caused confusion for participants, they relied on their intuition and manually adjusted the TalkBack focus to the top of the screen to access the new content. However, all participants expressed that it would be helpful if the focus were automatically moved to the newly appeared content. Another factor influencing the severity of the issues is the distance of the changed element from the accessibility focus. In Figure 4 (c), navigating one element back and forth could help users find the results, while if the appearance of the result is far from the current focus, it may become impossible for users to locate it.

Navigation Order. TIMESTUMP relies on the default navigation order of elements for TalkBack to determine if a dynamic change is problematic. However, users may have their own interaction preferences when using screen readers. In our study, some users rely on their prior knowledge and tap on specific parts of the app to find the requested element. One interviewee mentioned that it would be helpful if the change was announced, but he could still locate the dynamically updated `TextView`. Additionally, although customization of the navigation order of elements was not observed in the apps used in our experiments, it is important to note that developers can override the default TalkBack navigation order, e.g., allowing the topmost element to be designated as the last element focused by TalkBack on a screen. If so, the button in Figure 1(e), circled in red, would not trouble blind users as it would be in an unexplored area.

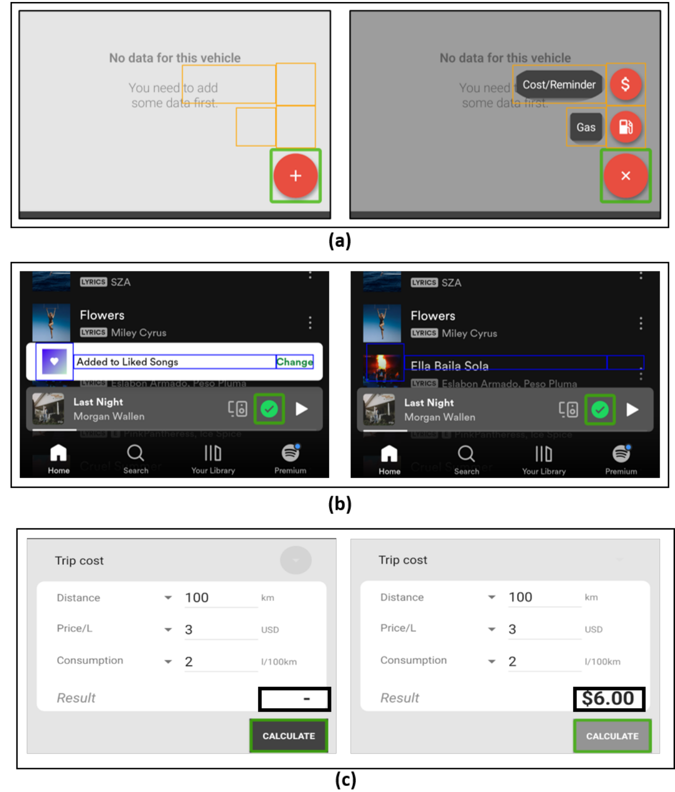

Fig. 4. Examples of detected issues by TIMESTUMP: (a) Appearing Content, (b) Short-Lived Content, and (c) Content Modification

C. RQ3. Performance

The performance evaluation of our tool, TIMESTUMP, is structured into three phases: app screen capturing (Phase 1), monitoring apps in action (Phase 2), and localizing problematic dynamic changes (Phase 3). In RQ1, Phase 1 involves automated capturing of two distinct app states from group2 apps, requiring an average of 167 seconds using STOAT [23]. Phase 2's analysis of each state for the number of actions is rapid; however, executing each action, capturing data, and transferring it to the server consumes about 66 seconds per action on average. The bulk of this time, approximately 20 seconds, is dedicated to dumping and transferring data, especially the captured video for each action. Developers can opt to disable video capturing in the tool, relying instead on screenshots, to significantly improve the tool's performance. In Phase 3, the post-analysis of the technique includes analysis of collected frames as well as the accessibility events, completing in approximately 7.5 seconds for each action.

VI. Threats to Validity

External Validity. In this study, we examined dynamic content changes following action execution or screen transition. However, ad-related pop-ups or random rating requests may appear without user actions. Investigating these requires analyzing the source code and library calls to find them. Future research could focus on identifying and understanding these instances.

Another concern is the completeness of our work, both in terms of the types of dynamic content changes and the challenges they pose. We carefully selected and manually explored a diverse range of apps to identify the types of dynamic change, ensuring these aligned with web testing definitions related to element structure and attribute modifications. Moreover, we utilized interviews to pinpoint scenarios where different types of dynamic content change might pose issues for screen reader users. Although our initial findings were extracted from three interviews, the subsequent user study in RQ2 reaffirms the validity of our conclusions.

Similarly, a concern related to RQ1 is the completeness of the identified accessibility issues. The ground truth was manually created due to the lack of preexisting datasets. To validate the manual construction of the ground truth, two authors independently reviewed the snapshots, including the `AccessibilityNodeInfo` tree and `AccessibilityEvents`, to identify problematic dynamic changes. Subsequently, they engaged in discussions to ensure agreement in their evaluations.

Internal Validity. TimeStump integrates various libraries and tools, including Stoat, ADB, AVD, and AccessibilityService, raising potential risks of defects. Additionally, there is a possibility of defects in our prototype's implementation. To counteract, we utilized the latest version of third-party tools, conducted Github code reviews, and tested on varied apps. We also assessed data capture accuracy on apps with different transitions, detailed on our website [13]. For rigorous testing, we used different sets of apps for our formative studies, accuracy evaluations, and tool assessments.

VII. Related Works

Automated accessibility testing includes various tools and studies for both web and mobile app accessibility analysis.

Web Pages: Web accessibility testing primarily relies on the WCAG guidelines [3]. These guidelines have led to the development of tools assessing web page accessibility compliance [29]–[34]. However, the guidelines overlook various accessibility challenges encountered by assistive technology users, especially in the context of dynamic changes. While a few criteria mandate developers to ensure dynamically displayed error/success messages are accessible to all, they fail to address other issues arising from dynamic content changes. Existing tools [29]–[34] cover only a fraction of the standards, thus inadequately detecting these issues on web pages.

To address limitations of guidelines, dynamic techniques have been proposed to assess apps while interacting with them. They resulted in studies that detect accessibility issues during interactions with web pages [35]–[37] or evaluate and infer correct accessibility attributes, like ARIA labels [38], for web content [39], [40]. In the evaluation of interaction issues, recent studies have attempted to utilize assistive technologies, similar to how an end user explores the app. They also account for changes introduced by JavaScript by evaluating multiple states that a single web page can take [41]–[47]. These studies focus on interaction failures which are only a subset of the challenges posed by dynamic changes. While exploring the app dynamically, they overlook real-time changes, like unannounced buttons appearing in previously explored areas. Furthermore, these studies miss changes like content modification that does not involve altering the DOM structure.

Mobile Apps: Similar to web accessibility testing, various automated tools are designed for mobile apps to assess specific app states and report their adherence to accessibility guidelines [48]–[57]. Recognizing the limitations of accessibility guidelines and the unique interaction modes of assistive technologies, recent studies have focused on identifying inaccessible content by utilizing assistive technologies to navigate various app states and comparing it with exploring the app without assistive technologies [11], [58]–[63]. However, no technique tackles the challenges of dynamic content changes.

VIII. Conclusion

The broad impacts of dynamic content changes on accessibility issues have not been thoroughly examined in prior research. We presented TimeStump, an automated framework identifying accessibility issues due to dynamic screen changes. TimeStump navigates through app states, collects data before, during and after each action, and applies a set of rules to detect dynamic screen changes that may lead to accessibility problems for the blind. An empirical study on real-world apps and a user study with blind participants prove its efficacy.

Future directions involve extending our work to ad-related pop-ups and unexpected rating requests that may appear without user actions, and expanding our implementation to other platforms, such as Web and iOS.

Our research artifacts are available publicly [13].

Acknowledgment

This work has been supported, in part, by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation. We thank Fable for their collaboration in making this research possible. We are grateful for the detailed feedback from the anonymous reviewers of this paper, which helped improve this work.

References

[1] U. D. of Justice, "Americans with disabilities act," https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX

[2] ——, "A guide to disability rights laws," https://www.ada.gov/cguide.htm, U.S. Department of Justice, 2020, Last Accessed: February 12, 2024.

[3] W3C, "Wcag 2 overview," https://www.w3.org/WAI/standards-guidelines/wcag/, W3C, 2023, last Accessed: December 5, 2023.

[4] Apple, "Accessibility on ios," https://developer.apple.com/accessibility/ios/, Apple, 2022, last Accessed: May 6, 2021.

[5] Android, "Build more accessible apps," https://developer.android.com/guide/topics/ui/accessibility, Google, 2022, last Accessed: May 6, 2022.

[6] WHO, "World report on disability," https://www.who.int/news-room/fact-sheets/detail/disability-and-health, WHO, 2023, Last Accessed: July 18, 2024.

[7] WAI, "Understanding sc 3.3.1: Error identification (level a)," https://www.w3.org/WAI/WCAG21/Understanding/error-identification.html#:~:text=Providing%20information%20about%20input%20errors, icons%20and%20other%20visual%20cues., W3C, 2024, last Accessed: March 5, 2024.

[8] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 569–579.

[9] A. Mathur, G. Acar, M. J. Friedman, E. Lucherini, J. Mayer, M. Chetty, and A. Narayanan, "Dark patterns at scale: Findings from a crawl of 11k shopping websites," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–32, 2019.

[10] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th annual ACM symposium on user interface software and technology*, 2017, pp. 845–854.

[11] N. Salehnamadi, F. Mehralian, and S. Malek, "Groundhog: An automated accessibility crawler for mobile apps," in *2022 37th IEEE/ACM International Conference on Automated Software Engineering*, IEEE. Rochester, Michigan, USA: ACM New York, NY, USA, 2022.

[12] S. Feng, M. Xie, and C. Chen, "Efficiency matters: Speeding up automated testing with gui rendering inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 906–918.

[13] F. Mehralian and Z. He, "Timestump companion website," https://github.com/seal-hub/Timestump, SEAL, 2024, Last Accessed: Aug 16, 2024.

[14] Monkey Taps LLC, "I am - Daily Affirmations," https://play.google.com/store/apps/details?id=com.hrd.iam&hl=en_US&gl=US, 2024, accessed: 2024-02-12.

[15] N. F. Malik, A. Nadeem, and M. A. Sindhu, "Achieving state space reduction in generated ajax web application state machine." *Intelligent Automation & Soft Computing*, vol. 33, no. 1, 2022.

[16] K. J. Koswara and Y. D. W. Asnar, "Improving vulnerability scanner performance in detecting ajax application vulnerabilities," in *2019 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2019, pp. 1–5.

[17] W3C, "Understanding success criterion 2.2.2 — understanding wcag 2.0," https://www.w3.org/TR/UNDERSTANDING-WCAG20/time-limits-pause.html, W3C, 2023, Last Accessed: December 7, 2023.

[18] ——, "Reordering page sections using the document object model," https://www.w3.org/WAI/WCAG22/Techniques/client-side-script/SCR27.html, W3C, 2023, Last Accessed: December 7, 2023.

[19] ——, "Inserting dynamic content into the document object model immediately following its trigger element," https://www.w3.org/WAI/WCAG22/Techniques/client-side-script/SCR26.html/, W3C, 2023, Last Accessed: December 7, 2023.

[20] WAI, "Using aria role=alert or live regions to identify errors," https://www.w3.org/WAI/WCAG21/Techniques/aria/ARIA19, W3C, 2024, last Accessed: March 5, 2024.

[21] F. T. Labs, "Fable — digital accessibility, powered by people with disabilities," https://makeitfable.com/, Fable Tech Labs, 2023, Last Accessed: December 7, 2023.

[22] M. Van Someren, Y. F. Barnard, and J. Sandberg, "The think aloud method: a practical approach to modelling cognitive," *London: AcademicPress*, vol. 11, pp. 29–41, 1994.

[23] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM New York, NY, USA, 2017, pp. 245–256.

[24] Google, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, Google, 2022, Last Accessed: May 6, 2022.

[25] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrücken, Germany: ACM New York, NY, USA, 2016, pp. 94–105.

[26] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE. Montreal, Canada: IEEE, 2019, pp. 269–280.

[27] Google, "Dumpcommand," https://android.googlesource.com/platform/frameworks/testing/+/jb-mr2-release/uiautomator/cmds/uiautomator/src/com/android/commands/uiautomator/DumpCommand.java#87, Google, 2024, Last Accessed: Feb 18, 2024.

[28] ——, "clickandwaitfornewwindow," https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/cmds/uiautomator/library/core-src/com/android/uiautomator/core/InteractionController.java#252, Google, 2024, Last ACCESSED: Feb 18, 2024.

[29] G. Broccia, M. Manca, F. Paternò, and F. Pulina, "Flexible automatic support for web accessibility validation," *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. EICS, pp. 1–24, 2020.

[30] G. Gay and C. Q. Li, "Achecker: open, interactive, customizable, web accessibility checking," in *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. Raleigh, USA: Association for Computing Machinery, 2010, pp. 1–2.

[31] A. M. Agency, "Access monitor plus," https://accessmonitor.acessibilidade.gov.pt/, Administrative Modernization Agency, 2021, Last Accessed: December 5, 2023.

[32] C. Benavidez, "Examinator," 2015. [Online]. Available: http://examinator.net/

[33] WebAIM, "Wave web accessibility evaluation tool," https://wave.webaim.org/, WebAIM, 2023, last Accessed: December 5, 2023.

[34] accessiBe, "accessscan - website accessibility checker - free & instant - accessibe," https://accessibe.com/accessscan, accessiBe, 2023, last Accessed: December 5, 2023.

[35] H. Takagi, C. Asakawa, K. Fukuda, and J. Maeda, "Accessibility designer: visualizing usability for the blind," *ACM SIGACCESS accessibility and computing*, no. 77-78, pp. 177–184, 2003.

[36] J. P. Bigham, J. T. Brudvik, and B. Zhang, "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*. Orlando, USA: Association for Computing Machinery, 2010, pp. 35–42.

[37] F. Durgam, J. Grigera, and A. Garrido, "Dynamic detection of accessibility smells," *Universal Access in the Information Society*, pp. 1–12, 2023.

[38] W3C, "Accessible rich internet applications (wai-aria) 1.2," World Wide Web Consortium (W3C), Tech. Rep., 2014. [Online]. Available: https://www.w3.org/TR/wai-aria/

[39] C. Duarte, A. Salvado, M. E. Akpinar, Y. Yeşilada, and L. Carriço, "Automatic role detection of visual elements of web pages for automatic accessibility evaluation," ser. W4A '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3192714.3196827

[40] M. Bajammal and A. Mesbah, "Semantic web accessibility testing via hierarchical visual analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1610–1621.

[41] W. M. Watanabe, R. P. Fortes, and A. L. Dias, "Acceptance tests for validating aria requirements in widgets," *Universal Access in the Information Society*, vol. 16, pp. 3–27, 2017.

[42] N. Fernandes, D. Costa, S. Neves, C. Duarte, and L. Carriço, "Evaluating the accessibility of rich internet applications," in *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, 2012, pp. 1–4.

[43] N. Fernandes, D. Costa, C. Duarte, and L. Carriço, "Evaluating the accessibility of web applications," *Procedia Computer Science*, vol. 14, pp. 28–35, 2012.

[44] L. Sensiate, H. Lidio Antonelli, W. Massami Watanabe, and R. Pontin de Mattos Fortes, "A mechanism for identifying dynamic components in rich internet applications," in *Proceedings of the 38th ACM International Conference on Design of Communication*, ser. SIGDOC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3380851.3416785

[45] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond, "Detecting and localizing keyboard accessibility failures in web applications," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 855–867. [Online]. Available: https://doi.org/10.1145/3468264.3468581

[46] P. T. Chiou, A. S. Alotaibi, and W. G. Halfond, "Bagel: An approach to automatically detect navigation-based web accessibility barriers for keyboard users," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–17.

[47] ——, "Detecting dialog-related keyboard navigation failures in web applications," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1368–1380.

[48] Android, "Improve your code with lint checks," https://developer.android.com/studio/write/lint?hl=en, Google, 2023, last Accessed: December 7, 2023.

[49] ——, "Accessibility scanner - apps on google play," https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor\&hl=en_US, Google, 2023, last Accessed: December 7, 2023.

[50] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. Bretton Woods, New Hampshire, USA: ACM New York, NY, USA, 2014, pp. 204–217.

[51] Android, "Espresso : Android developers," https://developer.android.com/training/testing/espresso, Google, 2023, last Accessed: December 7, 2023.

[52] Robolectric, "Android unit testing framework," https://github.com/robolectric/robolectric, Robolectric, 2023, Last Accessed: December 7, 2023.

[53] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, "Automated accessibility testing of mobile apps," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. Västerås, Sweden: ICST, 2018, pp. 116–126.

[54] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu, "Accessible or not an empirical investigation of android app accessibility," *IEEE Transactions on Software Engineering*, vol. 48, pp. 3954–3968, 2021.

[55] KIF, "Keep it functional - an ios functional testing framework," https://github.com/kif-framework/KIF, KIF, 2023, last Accessed: December 7, 2023.

[56] H. N. da Silva, S. R. Vergilio, and A. T. Endo, "Accessibility mutation testing of android applications," *Journal of Software Engineering Research and Development*, vol. 10, pp. 8–1, 2022.

[57] L. Li, R. Wang, X. Zhan, Y. Wang, C. Gao, S. Wang, and Y. Liu, "What you see is what you get? it is not the case! detecting misleading icons for mobile applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 538–550.

[58] M. Taeb, A. Swearngin, E. School, R. Cheng, Y. Jiang, and J. Nichols, "Axnav: Replaying accessibility tests from natural language," *arXiv preprint arXiv:2310.02424*, 2023.

[59] F. Mehralian, N. Salehnamadi, S. F. Huq, and S. Malek, "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. Rochester, Michigan, USA: ACM New York, NY, USA, 2022.

[60] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, "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*. Virtual, Okohama, Japan: ACM New York, NY, USA, 2021, pp. 1–11.

[61] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, "Automated detection of talkback interactive accessibility failures in android applications," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 232–243.

[62] A. Alshayban and S. Malek, "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*, 2022, pp. 984–995.

[63] N. Salehnamadi, Z. He, and S. Malek, "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*, 2023, pp. 1–20.