# Groundhog: An Automated Accessibility Crawler for Mobile Apps

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

Forough Mehralian*
fmehrali@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

Accessibility is a critical software quality affecting more than 15% of the world's population with some form of disabilities. Modern mobile platforms, i.e., iOS and Android, provide guidelines and testing tools for developers to assess the accessibility of their apps. The main focus of the testing tools is on examining a particular screen's compliance with some predefined rules derived from accessibility guidelines. Unfortunately, these tools cannot detect accessibility issues that manifest themselves in interactions with apps using assistive services, e.g., screen readers. A few recent studies have proposed assistive-service driven testing; however, they require manually constructed inputs from developers to evaluate a specific screen or presume availability of UI test cases. In this work, we propose an automated accessibility crawler for mobile apps, Groundhog, that explores an app with the purpose of finding accessibility issues without any manual effort from developers. Groundhog assesses the functionality of UI elements in an app with and without assistive services and pinpoints accessibility issues with an intuitive video of how to replicate them. Our experiments show Groundhog is highly effective in detecting accessibility barriers that existing techniques cannot discover. Powered by Groundhog, we conducted an empirical study on a large set of real-world apps and found new classes of critical accessibility issues that should be the focus of future work in this area.

## CCS CONCEPTS

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

## KEYWORDS

Android, Accessibility, Software Testing, AssistiveTechnology

**ACM Reference Format:**
Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556905

*Both authors contributed equally to this research.


*ASE '22, October 10–14, 2022, Rochester, MI, USA*

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556905

## 1 INTRODUCTION

The ever-growing reliance of people on mobile apps to perform daily tasks necessitates app accessibility for all, notably for more than 15% of the population who have disabilities [51]. Developers are obliged by law and expected by ethical principles to build accessible apps for users regardless of their abilities. However, prior studies reveal that many popular apps are shipped with some form of accessibility issues, hindering disabled users ability to interact with them [4, 21, 42].

To assist developers in enhancing app accessibility, technology institutes such as World Wide Web Consortium [49] and companies such as Apple [18] and Google [10] have published accessibility guidelines and best practices. These guidelines are backed by accessibility analysis tools to automatically analyze app compliance with guidelines and detect accessibility issues [5, 8, 19, 20]. For instance, by analyzing User Interface (UI) elements, they can report whether the contrast between elements and their backgrounds are above a certain threshold or the area of a button is smaller than a specific area defined in the guidelines.

Unfortunately, guidelines cannot detect about %50 of the accessibility issues that users with disabilities may encounter while interacting with apps [40]. Static assessment of UI specifications cannot reveal many critical accessibility issues that only manifest themselves in interacting with an app using assistive services, such as a screen reader. For example, users with visual impairment rely on screen readers, i.e., TalkBack in Android, to navigate through UI elements and perform actions on them. TalkBack users click on a desired element by double tap gesture. When this gesture entails no change in the app state, the element is not actionable by TalkBack and can render the app inaccessible.

The great majority of prior automated accessibility testing techniques do not take assistive services into account in their analysis. Salehnamadi, et. al [44] incorporate assistive services in evaluating the feasibility of executing GUI test cases. Their work, however, assumes the availability of GUI tests for validating the functionalities of the app under test, which are then repurposed for accessibility analysis. Unfortunately, developers do not usually write GUI tests for their apps, making their approach applicable to only situations in which GUI tests are available. Studies show that more than 92% of Android app developers do not have any GUI test for their apps [33]. Even if GUI tests are available for proprietary apps, the test cases are rarely available to the public or app store operators that may want to assess the accessibility of apps for users. Furthermore, GUI tests may fail to achieve good coverage, making their approach ineffective at finding accessibility issues in uncovered parts of the app under test. The work by Alotaibi, et. al [3] also utilizes TalkBack to find inaccessible elements in navigating sequentially through all the elements on the screen without GUI tests. This work is limited

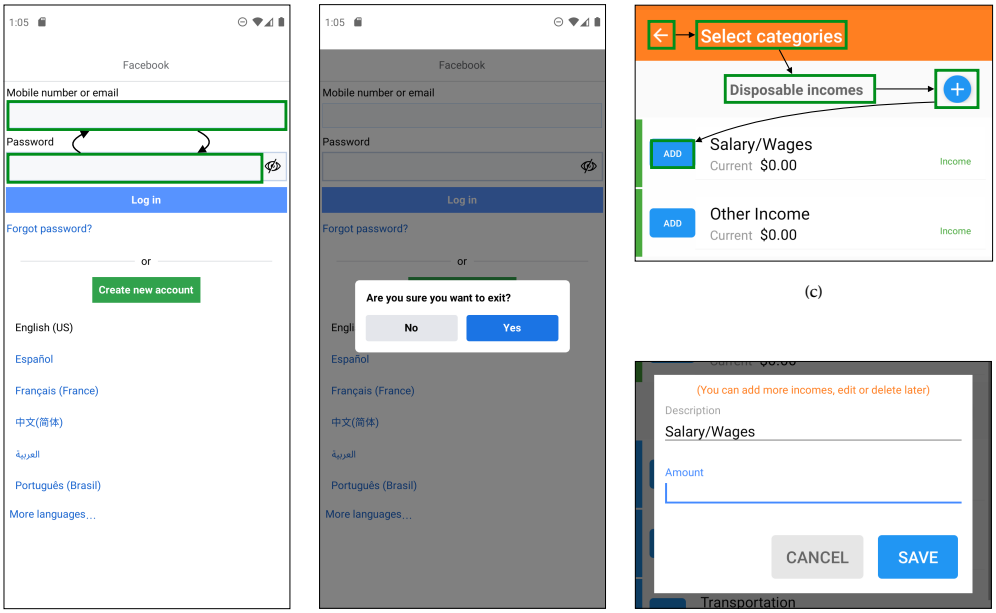


**Figure 1: (a) The login activity of Facebook app, (b) The exit dialog appears when users press back button on Facebook app, (c) a screen in BudgetPlanner app, the highlighted boxes and arrows depicts the directional navigation to the "ADD" button by TalkBack, (d) a dialog appears after tapping "ADD" button**

to analyzing a single screen that developers should provide manually. Moreover, it cannot detect other types of reachability issues that may occur while exploring the app with TalkBack by touch or with other assistive services. Their work also does not consider the different ways of performing actions with and without assistive services, potentially resulting in unactionable elements for assistive services

To address the limitations of existing tools, we have developed a fully automated approach, called Groundhog, for validating the accessibility of Android apps that replicates the manner in which disabled users actually interact with apps, i.e., using assistive services. Groundhog gets the app in a binary form, i.e., APK, and installs it on a Virtual Machine (VM). It utilizes an app crawler to explore a diverse set of screens to be assessed. For each screen, Groundhog extracts all the possible actions and executes the same action with different interaction models, including different assistive services, to validate if the app is accessible. Groundhog leverages the VM to repeatedly reevaluate the app from the same state, performing the same action using different assistive services to identify the accessibility issues that may affect users with various forms of disability.[1] In particular, Groundhog checks if UI elements can be located by users, i.e., **locatability**, and all actions can be performed, i.e., **actionability**, regardless of the way users interact with the device, e.g., touch-based interaction or assistive-service interaction. Instead of just reporting violations of accessibility guidelines as in prior work, Groundhog produces a summary of the accessibility issues containing a video that describes how a user with disability cannot perform an action in an app. This type of reporting can help developers to pinpoint the issue and increase their awareness of the challenges faced by users with disability.

Our empirical experiments show that Groundhog can detect 293 accessibility issues that could not be detected by existing accessibility testing tools.

This paper makes the following contributions:

- A novel, high-fidelity, and fully automated form of automated accessibility analysis that evaluates the accessibility of mobile apps from the perspective of users with various forms of disability.
- A publicly available implementation of the above-mentioned approach for Android, called Groundhog [45];
- An empirical evaluation on a large set of real-world Android apps, showing the effectiveness of Groundhog in detecting new accessibility issues in popular apps (even with more than 1 billions installs on Google Play), and
- A qualitative study of the different types of accessibility issues that can be detected by Groundhog, which can aid future researchers with developing automated means of fixing these specific kinds of issues.

The rest of this paper is organized as follows: Section 2 motivates this study with an example. Section 3 provides a background on accessibility testing and Android fundamentals. Section 4 explains the details of our approach, and Section 5 describes the optimizations over our technique. The evaluation of Groundhog on real-world

[1]The name of our tools is inspired by the popular Hollywood movie "Groundhog Day" from 1993, where the lead character is stuck in a time loop, forcing him to relive the same day, which is akin to our repeated reevaluation of an app from the same state.

apps is finally presented in Section 7. The paper concludes with a discussion of the related research and avenues of future work.

## 2 MOTIVATING EXAMPLE

In this section, we provide two examples to illustrate the types of accessibility issues that cannot be detected with conventional accessibility testing tools and prior studies.

Figure 1(a) shows the login screen of the Facebook app with more than 1 billion installs on Google Play [28]. This screen provides the ability for the user to log in, which obviously is crucial to be accessible, since it is the entry point of the app.

A user with a disability relies on an assistive service to interact with the app. For example, a blind user utilizes TalkBack [13], the standard screen reader in Android, to perceive the screen content by listening to what TalkBack announces for each element on the screen. A TalkBack user can navigate through the elements sequentially by swipe (*Directional Exploration*), or focus on a specific element by touch (*Touch Exploration*). Using either of these exploration strategies on the app screen illustrated in Figure 1(a), TalkBack can only detect the two text boxes, annotated in green in Figure 1(a), and is incapable of detecting the rest of the elements, including crucial buttons such as "Log in" or "Create new account". However, a regular user can see all the elements on the screen, provide login credentials, and tap on the buttons to log in and use the app without any problem. Interestingly, a TalkBack user cannot even exit the app using the back button as none of the elements on the exit dialog, in Figure 1(b), are accessible by TalkBack. This is an example of **locatability** issue, since a user with a disability cannot locate (reach) an element on the screen.

Existing accessibility testing approaches are not capable of detecting these issues. Google Accessibility Scanner [5] evaluates the top screen on a device, checks a few rules for the elements, and reports their violations as accessibility issues. In running Scanner on the screen in Figure 1(a) 4 issues are detected for text boxes, 2 of them are warning about their *"small touch target size"*, and 2 of them are noting the *"missing speakable text"* for them. Neither Scanner nor other rule-based accessibility testing tools [12, 15] are capable of detecting navigational issues in Android apps.

Assistive services also enable users to perform actions on elements. When there are no locatability issues, Assistive services such as TalkBack can focus on the desired element. In the case of TalkBack, double-tap gesture triggers the "Click" action on the focused element. Unfortunately, actions performed under different interaction models may have inconsistent behaviors. Figure 1(c) shows a screen in a popular budget tracker app, with more than 1 million installs, where users can add income or expenses to their budgets. To add an income to the budget, a user without a disability simply taps on the "ADD" button and a form appears to input the amount, as shown in Figure 1(d). For the same action, a TalkBack user, first locates the "ADD" button, either by touch exploration (tapping on the location of the button) or directional exploration (swiping right until the target element is focused, as shown by arrows in Figure 1(c)). Once the element is located, the user double taps to perform a click action through TalkBack. However, in this case, The income addition form in Figure 1(d) will not be shown, preventing TalkBack users from adding any income and rendering the app inaccessible for them as a result. This is an example of **actionability** issue, since the action is not supported consistently under different interaction models.

```
1  <node class="android.widget.FrameLayout" ...>
2      <node class="android.widget.LinearLayout" ...>
3          <node class="android.widget.FrameLayout" ...>
4              ...
5          </node>
6          <node class="android.widget.FrameLayout" ...>
7              ...
8              <node bounds="[44,560][182,648]" class="android.widget.Button"
9                    clickable="true" clickableSpan="false"content-desc=""
10                   enabled="true" focusable="true" focused="false" index="0"
11                   importantForAccessibility="true" long-clickable="false"
12                   package="com.colpit.diamondcoming.isavemoney" password="false"
13                   resource-id="com.colpit.diamondcoming.isavemoney:id/btn_add"
14                   scrollable="false" text="ADD" visible="true"/>
15             ...
16         </node>
17         ...
18     </node>
19 </node>
```

**Figure 2: A part of the excerpted XML representation of UI structure in the Budget Tracker app shown in Figure 1(c).**

The insight underlying our work is that the two types of accessibility issues discussed above cannot be revealed accurately unless the apps are examined in the manner disabled users interact with apps, i.e., using assistive services.

## 3 BACKGROUND

We provide a brief overview of User Interface (UI) components and accessibility support in Android to help the reader understand the material that is presented later.

### 3.1 Android UI

Android provides a variety of pre-built UI components such as structured layouts and widgets that allow developers to build the GUI of their app. This section provides background on UI components and GUI testing in Android.

The UI of an Android app is a single-root hierarchical tree where the leaf nodes are called *Views* or *Widgets* that users can see and interact with, e.g., buttons, text fields, and check boxes. The non-leaf nodes, on the other hand, are invisible to user. These non-leaf nodes are called *ViewGroups* or *Layouts* and used for arranging or positioning the widgets.

Both *Widgets* and *Layouts* have variety of attributes. For example, the *content-desc* attribute is used by accessibility services to provide description for widgets without textual representation or *clickable* attribute shows if the widget is clickable. The UI hierarchy of a screen in an Android device can be retrieved as an XML file. Figure 2 shows part of the UI structure in the Budget Tracker app. Lines 8-14 represents the first "ADD" button in Figure 1(c) where its attributes such as clickable or text are represented.

XPath [50] (XML Path Language) is an expression language that supports various queries in XML documents. In particular, XPath can be used to identify nodes accurately using the structural information. For example, the first "ADD" button in Figure 1(a) can be identified by its absolute path in XPath created by the class attribute as "/Framelayout/LinearLayout/FrameLayout[2]/Button" (the "android.widget" part is ommitted from classes). Since the class of an android widgets cannot be changed at runtime, the absolute

path in XPath, or in short *apath*, is a reliable identifier of widgets in Android.

## 3.2 Accessibility in Android

Android provides an accessibility API for alternative modes of interacting with a device. It also offers several assistive services, including TalkBack, which is the official screen reader in Android and built on top of the accessibility API. We briefly describe accessibility API in Android and how an assistive service like TalkBack can leverage this API.

The Android framework provides an abstract service, called *AccessibilityService*, to assist users with disabilities. The official assistive tools in Android, such as TalkBack, are also implementations of this abstract service [7]. *AccessibilityService* works as a wrapper around an Android device interacting with it (performing actions on and receiving feedback from it).

The feedback is delivered to accessibility services through *AccessibilityEvent* objects. Accessibility services should implement the method *onAccessibilityEvent* to receive feedbacks in form of *AccessibilityEvent* objects. *AccessibilityManager* is a system-level service that monitors any changes in device and manage accessibility services. When anything important happens on the device, *AccessibilityManager* creates an *AccessibilityEvent* object that describes the changes and passes it to *onAccessibilityEvent* method of accessibility services. The accessibility services can analyze the delivered event and may provide feedback to the user. For example, TalkBack announces the textual description of an element to the user when it is focused. An *AccessibilityEvent* object is associated with an *AccessibilityNodeInfo* object that contains the element's attributes. For instance, when a user clicks on "ADD" button ( Figure 1(c)), the system creates an *AccessibilityEvent* of type *TYPE_VIEW_CLICKED*, which is associated with the *AccessibilityNodeInfo* object corresponding to the element shown in lines 8-14 in Figure 2.

Moreover, an *AccessibilityService* can access all GUI elements on the screen in the form of an *AccessibilityNodeInfo* object. An `AccessibilityNodeInfo` object not only represents the attributes of a GUI element on the screen, but also provides the ability to perform actions on the corresponding element. For example, an *AccessibilityService* can perform a click action on an *AccessibilityNodeInfo* by sending *ACTION_CLICK* event to it.

# 4 APPROACH

Regardless of different interaction models, the ability to locate elements on the screen and perform actions consistently are fundamental needs in app accessibility. As a result, *locatability* and *actionability* form the basis of our approach. The goal of our approach is to automatically detect apps that fail to meet these accessibility requirements at runtime.

To that end, we propose Groundhog, an automated assistive-service driven testing tool. Figure 3 shows the overview of our approach. Groundhog utilizes an *App Crawler* to explore different states of the app. After each change in the app, *App Crawler* invokes the *Snapshot Manager* to capture a VM snapshot if the current state (screen) has not already been seen. *Snapshot Manager* provides the VM Snapshots to *Action Extractor*, where all the possible actions on the given state of the app are subsequently extracted. Groundhog

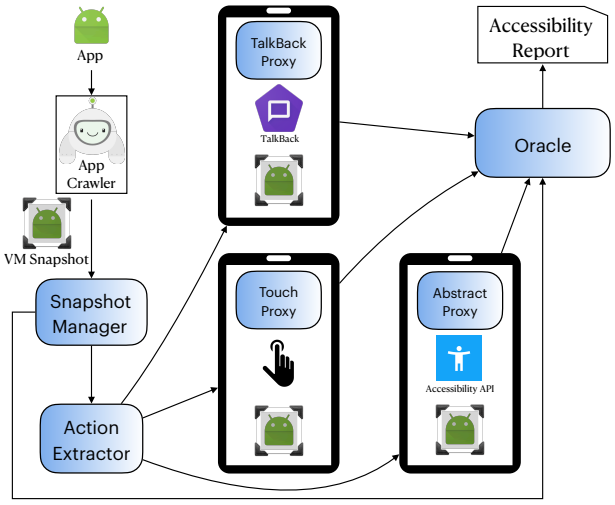


**Figure 3: An overview of Groundhog**

then tries to locate the elements and perform these actions on them using three different proxies: Touch Proxy, TalkBack Proxy, and Abstract Proxy. Finally, the new state of the app after performing the action is provided to the *Oracle* along with the initial app state. *Oracle* assesses if each proxy successfully performs the action and produces the final report.

In this section, we describe each component of Groundhog in detail.

## 4.1 Snapshot Manager

The goal of Snapshot Manager is to allow a diverse set of app states obtained through crawling to be later analyzed. Snapshot Manager is a connector between an app crawler and the rest of the system. Groundhog can be integrated with any of the existing app crawling techniques like Monkey [29], Stoat [47], Ape [30], Sapienz [34], etc. These crawlers employ various techniques in modeling the app to trigger transitions between app states. For example, Stoat models app behavior as a Finite State Machine (FSM) whose nodes are UI elements and attempts to maximize node coverage as well as code coverage. In Groundhog, even a human agent, e.g., developer or tester, can be involved to replace or enhance an automated app crawler to reach any desired state of the app.

For each app state, Snapshot Manager checks whether this state is a newly discovered state to take a snapshot for further analysis or not. To that end, Snapshot Manager calculates a hash value of the hierarchical representation of UI elements on the screen. Screen hash calculation in Snapshot Manager only incorporates elements and attributes that impact obtaining a diverse set of app screens. For example, elements that do not belong to the app under test, i.e., have a different package name or belong to an advertisement widget, are not included. Similarly, not all elements' attributes can distinguish different screens. For example, if the app crawler taps on an edit text box or writes a random string in it, its *focused* and *text* attributes change; however, they are not indicators of a new

screen. The practice of excluding some values in defining GUI states is also widely adopted in Mobile GUI testing studies [23, 24, 26].

Snapshot Manager provides VM snapshots of diverse app screens to the rest of Groundhog's components.

## 4.2 Action Extractor

The *Action Extractor* component takes a VM snapshot of an app state as input and extracts a list of available actions from it. To that end, *Action Extractor* loads the snapshot on a VM equipped with an Accessibility Service such as UIAutomator. This service runs in the background and enables capturing a hierarchical representation of UI elements, similar to Figure 2. *Action Extractor* performs further analysis on the dumped hierarchy of UI elements. It explores the tree of elements and searches for those that support action, e.g., have *clickable=true* in their attributes.

An action consists of two parts: the operation, e.g., click, and an identifier of the element on which the action is performed. The target element can be identified uniquely by its *apath*, i.e., the absolute path from the root to the target node in the UI hierarchy tree. For example, assuming the target element is the first "ADD" button in Figure 1(c), the corresponding *apath* is */Framelayout/LinearLayout/FrameLayout[2]/Button*, as shown in lines 8-14 in Figure 2. Also, the operation of this action can be determined from the "clickable" attribute (line 9 in Figure 2). Therefore, this action can be represented as the following JSON object that is passed to proxies to be executed in different interaction modes:

```
{
    operation: 'click',
    apath: '/Framelayout/LinearLayout/FrameLayout[2]/Button'
}
```

## 4.3 Proxies

Proxies represent various interaction models with a device. The goal of each Proxy is to execute a given action on a given app state and return the app state after performing the action along with the execution logs. To that end, each Proxy utilizes Android's *AccessibilityService* to support two main functionalities: (1) locating the element specified in action and (2) performing the intended action on the element. In this study, we consider three different models: Touch, TalkBack, and an Abstract assistive service with all the capabilities of accessibility API. The details of each Proxy are as follows.

*4.3.1 Touch Proxy.* This Proxy interacts with the system from the standpoint of users without disabilities. These users do not use any assistive service and see the elements that are depicted on the screen to locate them. Touch Proxy first determines the coordinates of the bounding box of the element on the screen to locate the element. It then sends a tap gesture event to the element to simulate the touch-based interaction model.

To locate an element, Touch Proxy searches for the corresponding node of the target element identified by its *apath* in the UI hierarchy. It starts from the root node of the screen and follows the address specified in the apath in a depth-first traversal order of the UI tree. If at a level, no branch matches the apath, it means the node is not *locatable*.

To perform an action on the located element, Proxy calculates a tap point considering the bounding box of the element. To that end, it calculates the coordinate of the center of the element using its bounding box coordinates. For example, the coordinate for the tap action on the "ADD" button (113, 604) can be calculated from line 8 in Figure 2. Once the coordinate of the target element is determined, Touch Proxy performs the click operation by sending a tap gesture event for that coordinate.

*4.3.2 TalkBack Proxy.* This Proxy utilizes TalkBack to interact with the device. TalkBack supports two UI exploration modes: Directional Exploration (by swiping) and Touch Exploration (touching different screen parts). Similarly, TalkBack Proxy leverage both exploration modes. When we enable TalkBack, it focuses on the first node on the screen. Swipe right (left) changes the focus on the next (previous) element on the screen. The Proxy first employs directional exploration to locate an element, i.e., iteratively draws swipe right gestures using the Accessibility API to navigate to the desired element. The Proxy terminates navigation if it focuses on the desired node or visits an element twice. The latter case indicates either there is a navigation loop or all existing elements have been visited once. When this process fails in locating the element, there is a *locatability* issue in using directional exploration. For example, a revolving list of elements can cause a navigation loop for a TalkBack user, preventing the user from reaching the elements residing afterward. To alleviate this problem, in practice, disabled users transition to explore by touch mode to focus on a random element outside of the loop and resume directional exploration forward or backward from there. This Proxy, similarly, tries to use touch exploration.

If the element is not found in Directional Exploration, TalkBack Proxy tries Touch Exploration mode by touching on the coordinates of the target element. If the element cannot be focused, TalkBack Proxy reports a violation of *Locatable* rule for the element. Once the element is located (focused), TalkBack Proxy uses Accessibility API to perform the intended operation., e.g., perform a double tap on the screen to click on the focused element. If the target element cannot be focused by Directional Exploration or Touch Exploration modes, a *locatability* failure is reported using TalkBack.

*4.3.3 Abstract Proxy.* As mentioned in Section 3, all assistive services in Android are built on top of the Accessibility API. To evaluate the app accessibility, given all the capabilities of Accessibility API, we introduce Abstract Proxy. Accessibility issues revealed for Absract Proxy exist for all other assistive services (e.g., SwitchAccess [11] for users with motor impairment) since they use Accessibility API to locate elements and perform actions on them.

For locating an element, Abstract Proxy locates the elements by their *apath* similar to what was explained for Touch Proxy. Then, it sends the event corresponding to the action, e.g., *ACTION_CLICK* to the located node using Accessibility API.

All the proxies return the next state of the app along with the execution logs to the Oracle component to be further analyzed. The execution logs contain all the triggered events, the action specification, node information, and failure reports.

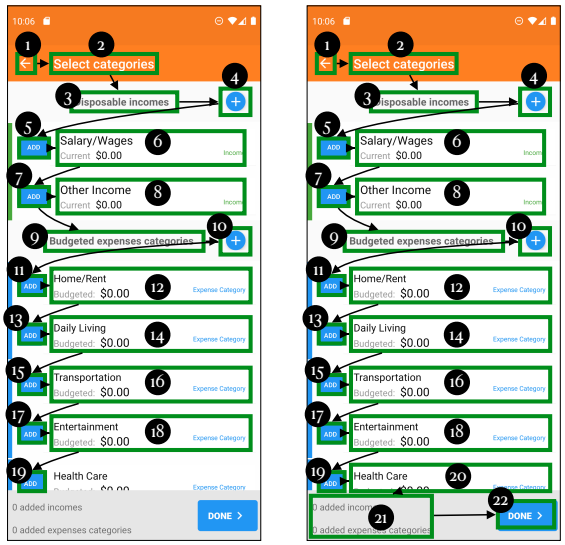


**Figure 4: Locating (a) the last "ADD" button, and (b) the "Done" button with TalkBack Proxy in directional navigation. 18 directional navigation interactions in (b) are redundant since they have been performed in (a) already.**

### 4.4 Oracle

The Oracle component is responsible for analyzing each app state and corresponding execution logs to determine if an accessibility issue exists in executing an action with a proxy.

For *locatability* issue, Oracle refers to failure reports of proxies to check if the Proxy was successfully locating the element. For *actionability* issue, Oracle first analyzes event logs to check if the events which are indicating a change in the content of the UI, i.e., *TYPE_WINDOW_CONTENT_CHANGED*, and executing an action, e.g., *TYPE_VIEW_CLICKED*, occurred. It also compares the app's previous state with the new state to ensure the event occurred. In comparing app states, Oracle compares their UI hierarchy similar to *Snapshot Manager* by comparing their hash values. However, Oracle does not exclude the same attributes as *Snapshot Manager* in calculating the hash value. For example, changes in the *text* attribute are not demonstrating a new screen for *Snapshot Manager* but can indicate an action execution. In the end, if the UI hierarchy before and after the action execution is the same, and there is no corresponding *AccessibilityEvent* of the executed action, the oracle reports an *actionability* issue for a given User Proxy.

Furthermore, the Oracle compares the *actionability* of each element across different proxies to check if there exists at least one Proxy that can successfully perform the action. This way, we are assured the element is associated with behavior (it is operative) and not just a decorative element.

## 5 OPTIMIZATION

In the previous section, we explained how, given a snapshot of an app, Groundhog extracts all possible actions for each of them, and locates and performs the available actions using different proxies. For example, Figure 4 depicts the process of locating two elements (a) the last "ADD" button, and (b) the "Done" button. Note that TalkBack traverses the UI hierarchy with each swipe starting from the top left element on the screen. As can be seen in Figure 4, the elements 1 to 19 appear both in (a) and (b). In other words, there is substantial redundancy between the steps required to locate these two elements.

We introduce an optimization technique using a memoization algorithm to minimize the number of interactions in the Directional Exploration strategy without losing the accuracy of detecting locatability issues in an app. The basic idea is to memorize the elements that TalkBack has located directionally in previous action executions and start the exploration from the closest located element to the target element. To locate the target element, TalkBack Proxy first sends a direct *AccessibilityEvent*, called *ACTION_FOCUS* to element $e$ which asks TalkBack to focus on it directly. The element $e$ is a visited element in the past action executions of TalkBack Proxy, closest to the target element in the UI hierarchy. This way, all directional navigation from the start to the element $e$ is bypassed, allowing the exploration to proceed much faster.

## 6 IMPLEMENTATION

Groundhog is designed as a Client-Server architecture model where the server is on the host machine and the client resides on an Android device. The server side, implemented in Python, orchestrates the whole analysis from running an app crawler, taking snapshots, executing actions with proxies, creating reports, and visualizing the results. The client, implemented in Java, is basically an accessibility service, i.e., proxies, that controls the device to execute actions.

Groundhog utilizes Android Debug Bridge (ADB) [9] to manage communications between the server and client. Groundhog also modifies Stoat app crawler [47] and employs it to explore different states of the app. As discussed in Section 4, any app crawler can be used in Groundhog. The rationale behind choosing Stoat is that it is completely open-source and conveniently works with the latest Android versions. It also has been widely used in previous studies. Lastly, Pillow [25] Python imaging library and Flask [39], python web framework, assist in visualizing the detected accessibility issues.

In our experiments, for actionability evaluation of GUI elements, we only focused on click actions that are most commonly associated with app behaviors. However, Groundhog can be similarly configured for any other type of action, e.g., long-click.

## 7 EVALUATION

We conduct several research experiments to evaluate Groundhog and answer the following research questions:

**RQ1.** How effective is Groundhog in detecting accessibility issues?
**RQ2.** How does Groundhog compare to Google Accessibility Scanner (the official accessibility testing tool in Android)?
**RQ3.** What are the characteristics of the detected accessibility issues? How do they impact app usage for users with disabilities?

**RQ4.** What is the performance of Groundhog? To what extent optimization improves its performance?

## 7.1 Experimental Setup

We evaluate Groundhog on three different sets of real-world apps. First, a set of 20 random apps with more than 10 million installs in Google Play Store [14] (labeled as **P**). Second, 20 randomly selected apps from AndroZoo [1], a collection of Android apps collected from several sources including Google Play (labeled as **A**). All of these 40 apps are published in Google Play in 2021 and 2022. We also included 17 apps from the 20 apps that were evaluated by Latte [44] (labeled as **L**).[2] Latte is a related prior tool, discussed in Section 1, to which we compare against. Out of the 17 apps from the Latte dataset included in our study, 11 have confirmed accessibility issues.

In total, our dataset consists of 57 apps that have been published in 21 different categories in Play Store. The complete list of datasets can be found on our companion website [45]. We ran Groundhog on each app until at least 10 states (screens) were captured (in total 570 different states).

To answer the research questions, we carefully examined the results to check if the reported issue is correct (true positive) or wrong (false positive). Therefore, we create a smaller set of results by selecting 5 UI states from 10 apps in each dataset (P, A, and L). In total, a set of 150 different UI states with 1,133 actions is created which can be seen in Table 1 (sorted based on installs).

All experiments were conducted on a typical computer setup for development (MacBook Pro, 2.8 GHz Core i7 CPU, 16 GB memory). We used the most recent distributed Android OS (SDK30), and the latest versions of assistive services, i.e., TalkBack 12.1 and SwitchAccess 12.1.

## 7.2 RQ1. Effectiveness of Groundhog

Table 1 summarizes the accessibility issues detected by Groundhog. The *Actions* column represents the total number of extracted actions from all different states of the app and the number of actions that Groundhog found to be operative, i.e., leading to a modification in the GUI state. As shown in the Table, on average, each snapshot has 7.5 actions to be evaluated by proxies. The columns entitled *TalkBack Unlocatable*, *TalkBack Unactionable.*, and *Abstract Unactionable* represent locatability and actionability issues by TalkBack Proxy, and actionability issues by Abstract Proxy, respectively. For each type of issue, we show the total number of detected issues and the number of issues manually verified by authors or True Positives (TP).

To verify if an issue is detected correctly by Groundhog, we load the corresponding snapshot on an emulator and interact with the app manually. For TalkBack locatability issues, we explored the app using TalkBack's two exploration modes, i.e., Directional and Touch Exploration strategy and check if the target element cannot be located in either way. Note that since Abstract Proxy directly interacts with the corresponding *AccessibilityNodeInfo* objects, it has no locatability issue.

For the actionability issues, first, we perform the action with touch (by tapping on the element) and observe the changes in the app state, e.g., by tapping on a checked box, its state changes, or by clicking a button, a new page may appear. Once we confirmed the target element is associated with an action by touch, we reload the snapshot to the same state two other times. The first time, we use TalkBack to click on the element (double tap), and the second time we send *ACTION_CLICK* to the target element using ADB and Groundhog. Then we monitored all changes to see if anything happened. We follow a conservative strategy and assume that any changes after clicking (even if it is not the same as the change after tapping the element) show the element is actionable.

With the number of verified issues (TPs), we evaluated the effectiveness of Groundhog in terms of Precision as the ratio of the number of TPs to the number of all detected issues. We also report Action Coverage and Recall of Groundhog as follows.

*7.2.1 Precision.* The number of locatability and actionability issues that are confirmed manually are shown in Table 1. In total, Groundhog could detect 293 true accessibility issues with a precision of 86%. Two-thirds of the apps in our test set have locatability issues. Note that, when an element is not locatable by TalkBack, it cannot be verified if it is actionable. Therefore, the number of TalkBack Proxy actionability issues is expected to be less than Abstract Proxy actionability issues. A9 and A11 are the only two exceptions in our test set. Our further investigations of these apps reveal that TalkBack dispatches touch events to the screen when performing *ACTION_CLICK* fails. TalkBack utilizes this workaround to overcome some accessibility issues in apps.

Our analysis of Groundhog's failures showed that 39 out of 48 false positives could be fixed by rerunning Groundhog on the app for the second time. The reason for these failures in the first attempt is the improper timing between performing an action and retrieving the results from the device, e.g., some of *AccessibilityEvent*s are not captured, which is a common challenge in dynamic analysis techniques due to concurrency issues.

In a few of the false positives, although the assistive services did not make any changes to the app's state, the changes by touch interaction do not contribute to any functionality of the app. For example, Figure 5 (a) shows the login page of MicrosoftTeams app (P7). Clicking on the email text box on the login page results in different behaviors based on the way it is performed. When a user with an assistive service clicks on the text box, nothing happens; however, if a user touches the text box, the decorative figure disappears, as shown in Figure 5 (b). Groundhog reports this as an actionability issue. However, since this change does not impact assistive-service users, we mark it as a false positive.

Some false positives happen because of changes in the app state during exploration. For example, Groundhog reports a button in a slider list of the To-Do-List app (A5) as locatabilty issue, as shown in Figure 5 (c). However, the reason behind this is that the element is the last item on the list and when TalkBack focuses on it, the sliding widget fetches new elements and moves the elements to the front. This changes the GUI hierarchy layout and Groundhog does not realize the current first element is the same as the last element on the list seen previously. Moreover, Groundhog detects a TalkBack actionability issue for the SchoolPlanner app (L8), as shown in Figure 5 (d), since performing a click on the focused element does not change the UI state (since the tab is already active). However,

[2]We had to exclude 3 outdated apps that do not work anymore.

**Table 1: The evaluation subject apps with the details of detected accessibility issues by Groundhog**

| Id | App | Category | #Installs | #Actions | | TalkBack Unlocatable | | Talkback Unactionable | | Abstract Unactionable | | | #All Issues | | Scanner |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Total | Operative | Total | TP | Total | TP | Total | TP | SA | Total | TP | |
| P1 | Instagram | Social | >1B | 31 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 |
| P2 | FacebookLite | Social | >1B | 20 | 18 | 14 | 14 | 0 | 0 | 7 | 6 | 6 | 21 | 20 | 33 |
| P4 | Zoom | Business | >500M | 26 | 25 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 13 |
| P7 | MicrosoftTeams | Business | >100M | 23 | 19 | 0 | 0 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 6 |
| P11 | MovetoiOS | Tools | >100M | 12 | 10 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 11 |
| P12 | Bible | Books | >50M | 44 | 39 | 6 | 6 | 0 | 0 | 0 | 0 | 0 | 6 | 6 | 20 |
| P13 | ToonMe | Photography | >50M | 48 | 41 | 18 | 17 | 1 | 0 | 0 | 0 | 0 | 19 | 17 | 43 |
| P19 | Venmo | Finance | >10M | 24 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| P21 | Lyft | Navigation | >10M | 21 | 18 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 |
| P22 | Expedia | Travel | >10M | 40 | 34 | 9 | 6 | 0 | 0 | 0 | 0 | 0 | 9 | 6 | 71 |
| A1 | YONO | Finance | >100M | 92 | 59 | 54 | 41 | 9 | 9 | 1 | 1 | 1 | 64 | 51 | 39 |
| A2 | NortonVPN | Tools | >10M | 21 | 16 | 9 | 8 | 1 | 0 | 0 | 0 | 0 | 10 | 8 | 8 |
| A3 | DigitalClock | Tools | >10M | 57 | 42 | 7 | 7 | 0 | 0 | 1 | 0 | 0 | 8 | 7 | 21 |
| A5 | To-Do-List | Productivity | >5M | 45 | 32 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 19 |
| A6 | Estapar | Vehicles | >1M | 41 | 31 | 23 | 21 | 2 | 0 | 0 | 0 | 0 | 25 | 21 | 11 |
| A9 | MyCentsys | House | >10K | 34 | 19 | 0 | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 9 | 14 |
| A10 | HManager | Productivity | >10K | 17 | 17 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 5 |
| A11 | Greysheet | Lifestyle | >10K | 44 | 24 | 1 | 0 | 0 | 0 | 19 | 18 | 18 | 20 | 18 | 10 |
| A13 | MGFlasher | Vehicles | <10K | 54 | 37 | 5 | 5 | 2 | 2 | 6 | 6 | 6 | 11 | 11 | 19 |
| A18 | AuditManager | Productivity | <10K | 15 | 10 | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 6 |
| L3 | Yelp | Food | >50M | 62 | 56 | 10 | 9 | 0 | 0 | 0 | 0 | 0 | 10 | 9 | 9 |
| L4 | GeekShopping | Shopping | >10M | 29 | 28 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 5 | 3 | 13 |
| L5 | Dictionary | Books | >10M | 42 | 38 | 3 | 1 | 0 | 0 | 2 | 1 | 0 | 5 | 2 | 16 |
| L6 | FatSecret | Health | >10M | 37 | 37 | 11 | 9 | 1 | 1 | 0 | 0 | 0 | 12 | 10 | 14 |
| L8 | SchoolPlanner | Education | >10M | 52 | 48 | 8 | 8 | 0 | 0 | 1 | 0 | 0 | 9 | 8 | 52 |
| L9 | Checkout51 | Shopping | >10M | 29 | 22 | 6 | 6 | 0 | 0 | 0 | 0 | 0 | 6 | 6 | 4 |
| L11 | TripIt | Tavel | >5M | 52 | 39 | 9 | 8 | 0 | 0 | 0 | 0 | 0 | 9 | 8 | 6 |
| L12 | ZipRecruiter | Business | >5M | 31 | 27 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 5 |
| L13 | Feedly | News | >5M | 63 | 34 | 34 | 34 | 14 | 12 | 24 | 23 | 23 | 58 | 57 | 1 |
| L15 | BudgetPlanner | Finance | >1M | 27 | 25 | 2 | 0 | 6 | 6 | 6 | 6 | 6 | 8 | 6 | 26 |
| Total | | | | 1133 | 879 | 244 | 209 | 43 | 34 | 83 | 75 | 74 | 341 | 293 | 512 |
| Precision | | | | | | 0.85 | | 0.79 | | 0.90 | | | 0.86 | | |

by touching on the tab, we are in fact touching on the overlay, resulting in the disappearance of the overlay element.

*7.2.2 Action Coverage.* To understand the effectiveness of Groundhog in extracting all possible actions from the screen, we manually examined all 150 UI states by touch interactions to extract the set of all elements that are associated with an action. In total, we found 1,149 actions, where Groundhog could extract 1,133 of them (98% action coverage). In cases that Groundhog missed an action, there was a custom-implemented UI widget without proper specifications for accessibility services. For example, two missing actions, back and search buttons from apps Greysheet and Feedly apps (A11 and L13), depicted in Figure 5(e), and (f), are layouts with attribute *clickable* set to *False*. Thus, Groundhog cannot identify them as actionable elements.

*7.2.3 Recall.* To calculate the recall of Groundhog in detecting real accessibility issues, we used the set of confirmed accessibility issues by Latte [44] as the ground truth. In total, Latte found 12 accessibility issues, where 10 of them could be detected by Groundhog (83% recall in detecting existing issues). One false negative happens for the Feedly app where Groundhog did not extract the search button, depicted in Figure 5 (e), as an action. The other false negative happens in the Dictionary app, where the accessibility issue can be revealed after performing three consecutive actions on the app. Since Groundhog analyzes an app only with one action, this issue could not be detected. We also found 87 new accessibility issues in the dataset of apps from Latte that were not detected by Latte.

In comparison with Latte, we can see Groundhog is able to detect a much larger number of accessibility issues. This is mainly because Latte assumes the availability of manually written GUI tests and does not achieve the same level of coverage as Groundhog that uses a crawling technique. At the same time, in a few cases, Groundhog is missing certain accessibility issues that are detected by Latte because manually written tests can exercise non-native UI elements that do not have a proper specification for accessibility services (i.e., attributes of *AccessibilityNodeInfo* object are not

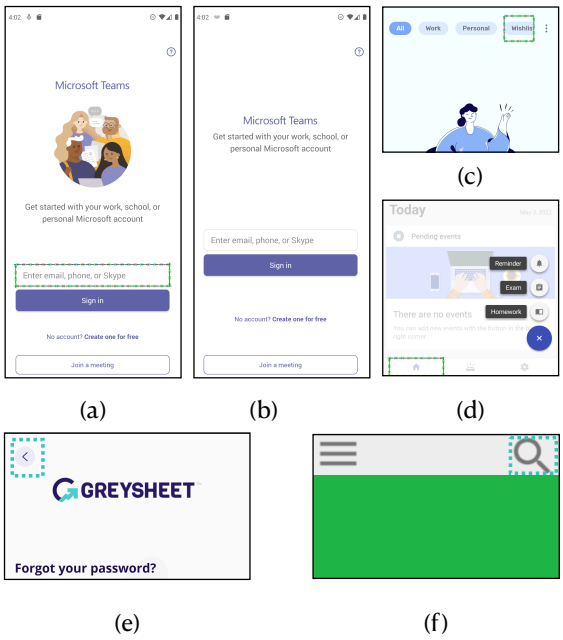


**Figure 5: (a-d) are examples of false positives, and (e-f) are examples of missing actions in Groundhog**

properly set), while Groundhog cannot properly analyze such elements.

## 7.3 RQ2. Comparison with Scanner

Google Accessibility Scanner [5], or Scanner for short, is the most widely used accessibility analyzer for Android. Scanner leverages Accessibility Testing Framework (ATF) [6] to evaluate screen accessibility. To compare Groundhog with Scanner, we analyzed all the examined app states in Table 1 with Scanner and checked what it reports. The last column of Table 1 displays the number of issues detected by Scanner. By comparing the accessibility issues reported by Groundhog against what Scanner reports, we found that there is no intersection between the type of issues each of them detects. Scanner evaluates a screen against predefined accessibility rules and reports issues such as low contrast, small touch target size, and missing speakable text for unlabeled icons. It cannot detect issues related to interactions with an app using assistive services. However, the accessibility issues reported by Scanner are also important to be addressed to have an accessible app. We believe that Groundhog complements Scanner and other ATF-based testing techniques [4, 27, 31] in evaluating app accessibility.

## 7.4 RQ3. Qualitative Study

We manually examined all the detected accessibility issues to understand how the issues affect users with a disability and what are their root causes. We found four different categories of issues as follows.

*7.4.1 Unlocatable elements with TalkBack.* Groundhog evaluates locatability of elements by TalkBack in using both directional and touch exploration strategies. In severe cases, neither of these strategies can locate an element. For example, Figure 6 (a) shows a screen in the Expedia app where none of its elements, even the back button, can be detected by TalkBack. We found that the root cause of this issue is having the *important-for-accessibility* attribute set to false, meaning that TalkBack should treat them as decorative elements and skip them in exploring the app. Developers should set this attribute properly. We found this issue in Facebook, Expedia, Checkout51, ToonMe, SchoolPlanner, and Yelp apps.

In some cases, the element can be located by directional exploration, but not by touch exploration. For example, Figure 6 (b) depicts the entry screen of YONO (a banking app), where the highlighted button can be located by directional exploration, yet, the element does not get accessibility focus when touched. This issue happens when there is an overlap among the active elements on a screen, similar to Figure 6 (b), where the highlighted button is placed under the top layout. Such elements confuse users about the screen's content and may also have security implications when a malicious functionality is hidden by malware authors in such elements. The security implications of this accessibility issue are further studied in [36]. This type of issue can be found in YONO, Feedly, Dictionary, Estapar, TripIt, NortonVPN, Facebook, DigitalClock, ToonMe, AuditManager, and SchoolPlanner apps.

The remaining cases of locatability issues occur in elements that TalkBack skips in directional exploration but can be focused on by touch. For example, Figure 6 (c) shows a part of the Bible app, when the user uses TalkBack in Directional exploration and reaches the end of the text, the highlighted bottom menu disappears. For a sighted user who sees all the changes on the screen, the disappearance of the menu can aid in reading the rest of the text more conveniently; however, it confuses blind users who may not even know the menu exists in the first place. The FatSecret, Geek, ToonMe, TripIt, Bible, MoveToiOS, and HManager apps have this type of issue.

*7.4.2 Actionability.* This issue manifests itself when an assistive service cannot be used to perform an action. Groundhog could find this type of issue in Facebook, Dictionary, Feedly, BudgetPlanner, MyCentsys, Greysheet, MGFlasher, AuditManager, FatSecret apps. For example, Figure 6 (d) shows a button in Feedly app that can only be clicked by touch.

Generally speaking, Abstract Proxy has more capabilities than TalkBack in performing actions as it uses Accessibility API to directly click on *AccessibilityNodeInfo* object. However, this was not the case in MyCentsys and Greysheet apps. Our further investigation and study on TalkBack source code [16] revealed that TalkBack utilizes a workaround to mitigate accessibility issues in apps. Talkback first uses Accessibility API to perform and check if the action is sent successfully; otherwise, it sends a touch event to the center of the focused element. Although this workaround may address inaccessibility in some situations, it may confuse users even more in some other situations. For example, Figure 6 (e) highlights a button under the Register button with the text "Help". However, when a TalkBack user double taps, the Register button is clicked instead.

A common theme of apps with actionability issues is that they are developed using hybrid frameworks or utilize WebViews [17]. Hybrid frameworks enable a developer to implement mobile apps in

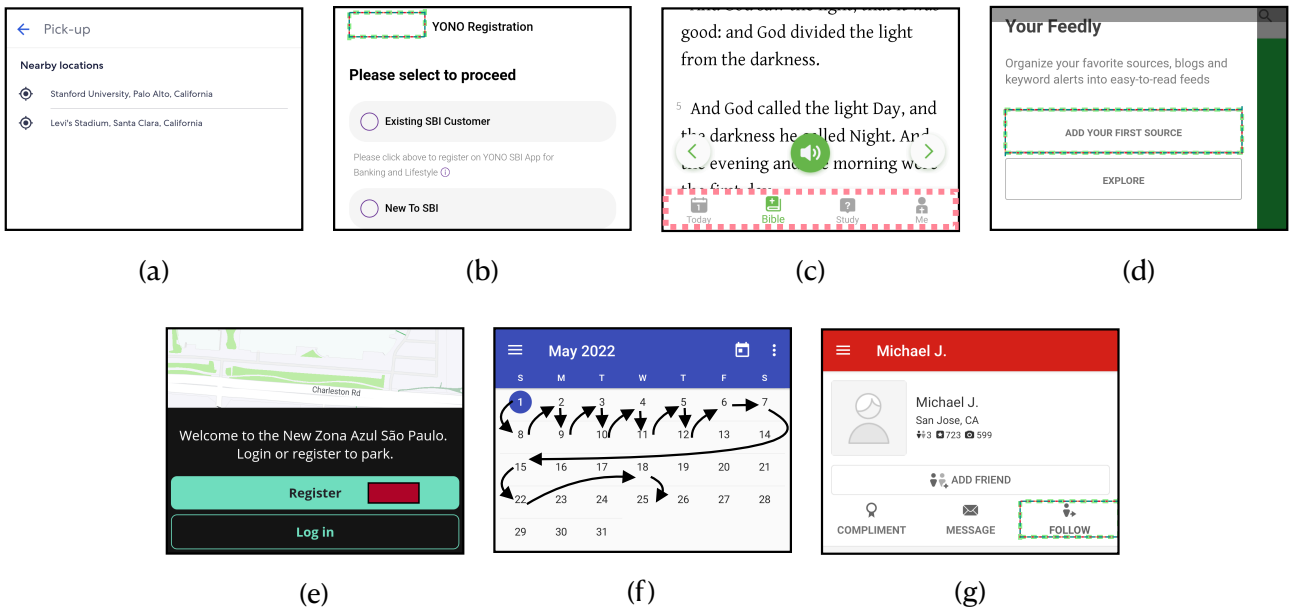


**Figure 6: Qualitative study of Groundhog's report on subject apps**

one codebase with one language, like C# in Xamarin[38]. Similarly, WebView renders web elements that are developed in HTML, CSS, and JavaScript code in mobile apps. One of the advantages of hybrid apps and Webviews is reusing the same code on different platforms, like iOS, Android, and even the Web. We could find YONO, ToonMe, Estapar, and Greysheet apps in Apple Store with similar accessibility issues detected by Groundhog, manifested by Voiceover (the iOS' official screen reader). We believe further studies are required to assess the accessibility issues resulting from hybrid frameworks.

*7.4.3 Counterintuitive Navigation.* One type of information produced by Groundhog as part of its reporting is short videos in GIF format showing how Talkback navigates directionally to reach an element. Checking these videos revealed a new type of accessibility issue where developers set an unexpected traversal order for elements. For example, Figure 6 (f) shows the visiting order of a calendar's elements in SchoolPlanner. As seen, there is no pattern in visiting the elements. In another example, Yelp's home page has counterintuitive navigation where the search button (which is at the top of the page) will be reached when all other elements have been visited.

*7.4.4 Inoperative Actions.* We examined the inoperative actions reported by Groundhog to see how they impact users with disabilities. Such clickable elements without any impact on the app content increase the number of interactions for TalkBack users to reach an element. For example, it takes 25 directional navigation to reach the farthest element in a state of DigitalClock; however, if the inoperative actions are removed by developers it can be reduced to 20 interactions, saving 20% of time spent by users with disabilities.

However, in some instances, there is a usability bug in inoperative actions which concerns regular users. For example, Figure 6 (g) shows a profile page of a user in Yelp where Groundhog detects the Follow button is not operative. Here, the other buttons in the same row (Compliment and Message) are associated with an action (the login page appears). It seems, there is a bug that makes the Follow button inoperative.

## 7.5 RQ4. Performance

We measured the time that Groundhog takes to create reports to understand how Groundhog can be integrated into the development lifecycle. For an app on average, Groundhog takes 3,541 seconds to explore an app, execute all actions using different proxies, and produce an accessibility report with visualized information. Since Groundhog does not require any manual input from developers, analyzing an app in less than an hour is completely practical, and can be done on a nightly basis.

The breakdown of the execution time is as follows. The app crawler (Stoat) takes 420 seconds on average to explore different states of the app. The action extraction part virtually takes no time (less than a second). The heavy part of Groundhog is executing each action via proxies. Groundhog executes each action in 21, 24, and 40 seconds for Abstract, Touch, and TalkBack Proxy, respectively. There are some common time-consuming parts for all proxies: reloading snapshot takes 4.1 seconds, reconnecting ADB takes between 2 to 12 seconds, and Groundhog waits for 5 seconds after each action is executed to ensure all changes in the app state are finalized. TalkBack Proxy takes more time to execute because the communication between Groundhog and TalkBack is a slow process since Groundhog actually performs touch gestures and waits for TalkBack to change its internal state.

Groundhog's performance can be improved significantly by parallelizing the snapshot analysis thanks to its Client-Server model.

Each VM snapshot is less than 1GB of data and can be easily transferred in less than 10 seconds.

Locating an element using TalkBack Proxy takes 9.71 seconds on average per action. Without our optimization technique, it would take 26 seconds on average. In other words, the optimization improves the performance of this aspect of Groundhog by more than 2.5 times per action, which reduces the app analysis time by 10 minutes on average.

## 8 THREATS TO VALIDITY

**External validity.** A key threat to validity is preserving the state of the app under test since three different proxies should perform the same action on the same element. We mitigate this threat by capturing a VM snapshot of the device used for all proxies. The virtualization technique may not preserve the state of apps that update their content dynamically or retrieve information from the server. For example, in a shopping app, if one proxy adds an item to an empty shopping cart that is synchronized with an external database, the same VM snapshot may be in a different state when it is loaded for another proxy. We have not observed this situation occurring in our experiments; however, to prevent reporting false positives/negatives in similar cases, we check the UI hierarchy of the apps after loading the VM snapshots. If they are not exactly similar, we report a flag indicating that the VM snapshot is different and the result may not be reliable. It would be interesting for future work to examine elegant solutions for handling dynamic and online content.

Another threat resides in the variety of actions supported by Groundhog. Our current implementation supports clicking action. Other touch gestures are not implemented. Although clicking is one of the most essential touch gestures for interacting with GUI elements, our claimed benefits of Groundhog can be more confidently generalized by providing and evaluating support for other types of actions. However, it is worth noting that most other complex touch gestures, like pinching in/out or double-tap, are not supported by assistive services in the first place. For example, pinching can be used for zooming in on an image, but it does not have an equivalent in TalkBack since blind users may not see visual images.

**Internal validity.** We implemented Groundhog using several libraries and tools, including *ADB*, *Android Virtual Device*, *Stoat* [30], and *AccessibilityService* in Android, which may introduce defects in the crawling and analysis steps of our implementation. Furthermore, our prototype may contain bugs in its implementation. We have tried to minimize this threat by upgrading all libraries to the latest available versions, writing automated unit tests, and conducting code reviews. In addition, we tested the prototype extensively on numerous popular Android apps.

## 9 RELATED WORK

Empirical studies on mobile accessibility [4, 21, 43, 48] have revealed the prevalence of various accessibility issues in mobile apps, preventing disabled users from utilizing their services. These findings have motivated the research community to develop techniques to automatically detect accessibility issues [4, 15, 22, 27], and to repair the detected issues [2, 21, 37, 52].

In general, automated accessibility testing techniques evaluate app compliance with accessibility guidelines [49] using static or dynamic analysis approaches [46]. Static analysis approaches such as Lint [15] identify accessibility violations in the source code upon compilation. Thus, they are not able to detect issues that can be detected at runtime. To mitigate their limitations, dynamic analysis techniques are proposed to analyze the runtime attributes of rendered UI components on the screen. Google accessibility Scanner [5] and other tools that are built on top of Accessibility Testing Framework [27, 31, 32] take a single app screen from the developers to run their tests and report issues such as small touch target size or duplicate name issues. The capabilities of these tools are limited to a small number of issues that were supported by accessibility guidelines that are found to only cover around 50% of the issues [40]. Thereby, they are not able to detect issues that manifest themselves in interactions with apps. This limitation, similarly, exists for enhanced dynamic techniques that evaluate the same accessibility rules but replace the developers' effort in exploring an app with a crawler [4, 27] or provide the ability to write app exploration scenarios in form of GUI tests [12, 41].

A related prior work is Latte [44], which was already discussed in Section 1 and empirically compared against in Section 7. Alotaibi, et al. [3] have proposed a method of detecting certain accessibility failures that may occur when using TalkBack. However, in contrast to Groundhog, their approach requires the developer to manually navigate through the app, i.e., the input to their tool is a screen of an app, rather than the app under test. Furthermore, their approach cannot detect unactionable elements. Such manual exploration is expensive, time-consuming, and may not result in good coverage.

Unlike prior testing techniques, Groundhog is a fully automated accessibility testing technique that only requires app in binary form and detects accessibility issues in interactions with the app using several interaction models. Groundhog can be generalized to any assistive service in the context of Android and with different exploration modes to evaluate all GUI elements at each state.

## 10 CONCLUSION

Prior accessibility testing tools can only point out a small portion of the problems that people with disabilities encounter while interacting with an app [35]. In this work, we proposed Groundhog, a fully automated assistive-service driven accessibility crawler to detect accessibility issues that only manifest themselves through interactions with the app. Groundhog explores apps and assesses the locatability and actionability of each element on the screen using different interaction modes provided by assistive services. Our future work involves evaluating the extent to which the ideas presented here can be applied to other computing domains (e.g., iOS, Web), and expanding Groundhog's support to additional assistive services and more complex gestures.

## ACKNOWLEDGMENTS

This work was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

# REFERENCES


[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, IEEE, Austin, TX, 468–471.

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

[3] 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.

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

[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. *Accessibility Testing Framework*. Google. Retrieved May 6, 2022 from https://github.com/google/Accessibility-Test-Framework-for-Android

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

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

[9] Android. 2022. *Android Debug Bridge*. Google. Retrieved May 6, 2020 from https://developer.android.com/studio/command-line/adb

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

[11] Android. 2022. *Control your Android device with Switch Access*. Google. Retrieved May 6, 2022 from https://support.google.com/accessibility/android/answer/6122836?hl=en

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

[13] 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

[14] Android. 2022. *Google Play*. Google. Retrieved May 6, 2022 from https://play.google.com/store/apps

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

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

[17] Android. 2022. *WebView - Android Documentation*. Google. Retrieved May 6, 2022 from https://developer.android.com/reference/android/webkit/WebView

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

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

[20] 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

[21] 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.

[22] Paul T Chiou, Ali S Alotaibi, and William GJ Halfond. 2021. Detecting and localizing keyboard accessibility failures in web applications. 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, 855–867.

[23] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.

[24] Wontae Choi, Koushik Sen, George Necul, and Wenyu Wang. 2018. DetReduce: minimizing Android GUI test suites for regression testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Gothenburg, Sweden, 445–455.

[25] Alex Clark and Contributors. 2022. *Pillow, Python Imaging Library*. Pillow. Retrieved May 6, 2022 from https://pillow.readthedocs.io/en/stable/

[26] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE, Seoul, South Korea, 1–12.

[27] 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.

[28] Google. 2022. *Facebook Lite - Apps on Google Play*. Meta. Retrieved May 6, 2022 from https://play.google.com/store/apps/details?id=com.facebook.lite&hl=en_US&gl=US

[29] Google. 2022. *UI/Application Exerciser Monkey*. Google. Retrieved May 6, 2022 from https://developer.android.com/studio/test/monkey

[30] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Montreal, Canada, 269–280.

[31] 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.

[32] IBM. 2022. *IBM Accessibility Requirements*. IBM. Retrieved May 6, 2020 from https://www.ibm.com/able/guidelines/ci162/accessibility_checklist.html

[33] 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.

[34] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM New York, NY, USA, Saarbrücken, Germany, 94–105.

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

[36] 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, Michigan, USA.

[37] 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.

[38] Microsoft. 2022. *An app platform for building Android and iOS apps with .NET and C#*. Microsoft. Retrieved May 6, 2022 from https://dotnet.microsoft.com/en-us/apps/xamarin

[39] Pallets. 2022. *Flask, The Python micro framework for building web applications*. Pallets. Retrieved May 6, 2022 from https://github.com/pallets/flask

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

[41] Robolectric. 2019. robolectric/robolectric. https://github.com/robolectric/robolectric

[42] 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.

[43] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2020. An epidemiology-inspired large-scale analysis of android app accessibility. *ACM Transactions on Accessible Computing* 13, 1 (2020), 1–36.

[44] 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.

[45] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. *Groundhog companion website*. University of California, Irvine. Retrieved September 1, 2022 from https://github.com/seal-hub/Groundhog

[46] 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.

[47] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM New York, NY, USA, Paderborn, Germany, 245–256.

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

[49] 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/

[50] W3. 2022. *XML Path Language*. W3. Retrieved May 6, 2022 from https://www.w3.org/TR/2017/REC-xpath-31-20170321/

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

[52] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM New York, NY, USA, Virtual, Okohama, Japan, 1–15.