Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps

Forough Mehralian*
fmehrali@uci.edu
School of Information and Computer Sciences
University of California, Irvine, USA

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

Syed Fatiul Huq
fsyedhuq@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

Mobile apps, an essential technology in today's world, should provide equal access to all, including 15% of the world population with disabilities. Assistive Technologies (AT), with the help of Accessibility APIs, provide alternative ways of interaction with apps for disabled users who cannot see or touch the screen. Prior studies have shown that mobile apps are prone to the *under-access* problem, i.e., a condition in which functionalities in an app are not accessible to disabled users, even with the use of ATs. We study the dual of this problem, called the *over-access* problem, and defined as a condition in which an AT can be used to gain access to functionalities in an app that are inaccessible otherwise. Over-access has severe security and privacy implications, allowing one to bypass protected functionalities using ATs, e.g., using VoiceOver to read notes on a locked phone. Over-access also degrades the accessibility of apps by presenting to disabled users information that is actually not intended to be available on a screen, thereby confusing and hindering their ability to effectively navigate. In this work, we first empirically study overly accessible elements in Android apps and define a set of conditions that can result in over-access problem. We then present OverSight, an automated framework that leverages these conditions to detect overly accessible elements and verifies their accessibility dynamically using an AT. Our empirical evaluation of OverSight on real-world apps demonstrates OverSight's effectiveness in detecting previously unknown security threats, workflow violations, and accessibility issues.

KEYWORDS

Android, Accessibility, Security, and Software Testing

*Both authors contributed equally to this research.

1 INTRODUCTION

Principles of universal design [16] dictate that technologies and services, including mobile apps, must be accessible to everyone regardless of their abilities. These principles are often overlooked in development practices, where developers build and test their apps based on the assumption that by default, a user views the app content on the screen and interacts with it by touch. Such assumptions exclude about 15% of the world's population with some form of disability, especially users with visual and fine-motor impairments. To facilitate disabled users' interaction with apps, mobile platforms support Assistive Technologies (AT) such as screen readers or special physical keyboards, which utilize the information exposed by Accessibility APIs to provide an alternative interaction model.

Prior studies have shown that many apps are shipped with functionalities that are not accessible using ATs [3, 42]. We call this the *under-access* problem. In this paper, we look at the dual of this issue, which we call the *over-access* problem. That is, some apps are shipped with functionalities that in certain states can be accessed using ATs but not otherwise.

An element is *Overly Accessible* (OA) when it provides more information and functionality to AT users than regular users. In security-sensitive apps, OA elements can jeopardize the security of password-protected apps such as banking, investment, health, etc. Case in point, for several iOS versions, users have reported scenarios of using VoiceOver, the standard screen reader in iPhones, to bypass iOS passcode and gain access to contacts, photos, notes, etc [12, 30, 31]. Moreover, OA elements can be used to provide unauthorized access to premium functionalities in apps with in-app purchases, endangering around 60% of companies on app stores that derive revenue from such functionalities in their apps [35]. As an example, the Mediation Moments app [11] has premium articles that are available to subscribed users; however, we found that an AT user can read these articles without purchasing the subscription. Lastly, bypassing the designed workflow can result in invalid inputs to be provided to an app, breaking its logic and leading to unexpected crashes. For example, in using the Airbnb app to book a place, the "decrement" button is disabled for touch when there is only one traveler, preventing zero and negative inputs.

We found that an AT user can still click this button and submit a request for a room for a negative number of people.

Interestingly, over-access also degrades the accessibility of apps. Blind users utilize screen readers to navigate through the elements on a screen sequentially. Even if the OA elements are not security-sensitive, presenting information that the developer did not intend to be available on the screen can confuse the screen-reader users. OA elements also increase the number of required interactions to reach the desired element, resulting in a less optimal user experience.

Despite the severe impacts of OA elements, they have received practically no attention in prior accessibility analysis of apps or security-related studies. Neither Google Accessibility Scanner [6], nor Apple Accessibility Inspector [10] check any rules for over accessibility. They only check a set of accessibility rules (e.g., proper text size and color) on displayed UI elements. Most other accessibility testing studies [7, 40] extend the accessibility rules of standard scanners and cannot detect OA elements consequently. A recent accessibility testing study proposed Latte [42], an accessibility testing framework to examine the accessibility of UI elements by executing a specific use case using AT. Nevertheless, OA elements are not a concern of Latte as it focuses on finding inaccessible elements.

Prior security-related studies [25, 36] have investigated the feasibility of constructing malicious software (e.g., malware) to launch a security attack by exploiting accessibility APIs. No prior study has investigated the vulnerabilities caused by OA elements in benign apps that can be exploited by any user, and using the standard ATs.

To fill this gap, we conducted an empirical study on 100 different UIs from 20 randomly selected apps to understand OA elements and their specifications. We then developed a tool, OverSight, to automatically detect them on a given state of the app.

OverSight first leverages the findings of our empirical study and devises a static checker to analyze currently displayed UI elements and localize *OA smells*, i.e., elements with one of the OA characteristics that may lead to revealing information or functionality that is unavailable for sighted users and available for AT users. Then, OverSight validates the accessibility of these elements dynamically using a custom AT with all the capabilities of Accessibility API and Talkback, which is the standard screen reader on Android devices. Finally, OverSight reports accessibility issues resulting from OA elements. Our empirical evaluation on 30 apps reveals that OverSight can precisely detect more than 83% of OA elements.

This paper makes the following contributions:

- First study that introduces the problems caused by apps that are overly accessible.
- An empirical study of OA elements and their characteristics.
- The first automated tool, called OverSight, for localizing and detecting OA elements in Android apps, which has been made publicly available [37].
- An empirical evaluation on real-world apps, corroborating the effectiveness of OverSight in detecting OA elements.

The remainder of this paper is organized as follows. Section 2 motivates this study with an example and provides background information. Section 3 introduces OA elements according to our empirical study. Section 4 explains OverSight, an automated approach to detect OA elements. In Section 5, the evaluation of OverSight on
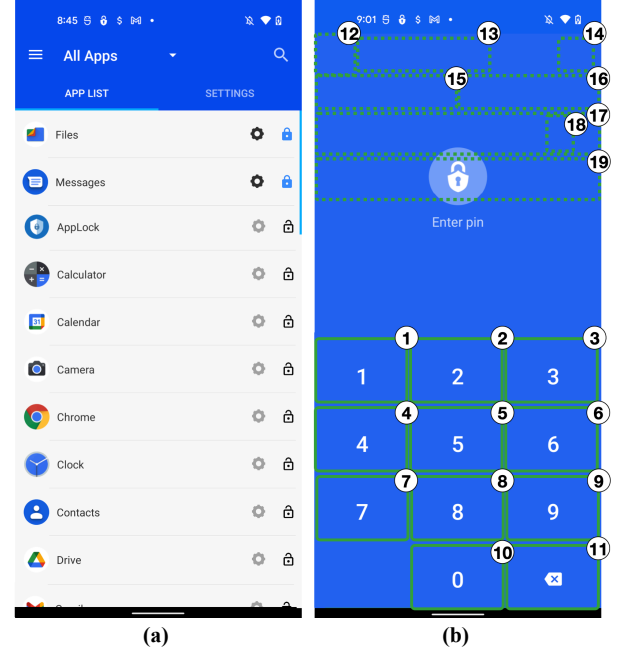

Figure 1: Built-in lock for a security-sensitive app.

real-world apps is presented. The paper concludes with a discussion of the related research and avenues of future work.

2 MOTIVATING EXAMPLE & BACKGROUND

Figure 1 shows screenshots of AppLock [28], a popular app locker with more than 5,000,000 installations and rating of 4.2. As shown in Figure 1 (a), the app lists all the installed apps on a phone on its first page, enabling users to add a lock to any desired app. App lockers protect themselves and other requested apps by preventing access to their content without providing a secret pattern or passcode. When a user opens the AppLock or any locked apps, e.g., Files or Messages as shown in Figure 1(a), she first sees the lock screen, depicted in Figure 1(b), and should first unlock it with a preset pin. Many other types of apps (e.g., investment, health monitoring, diary, etc.) employ a similar protection strategy for their contents.

A user without disability can see the pin pad and the text asking to "Enter pin" on the screen. She would try to unlock the app by entering the pin through touching the numbers on the screen. However, a user with disability has to rely on ATs to interact with apps. Mobile platforms such as Android have integrated ATs such as TalkBack [20]—the standard Android screen reader—and SwitchAccess [5]—a special keyboard with two keys, *Next* and *Select*—to enable app exploration for disabled users. Both of these ATs focus on each element on the screen and navigate through them sequentially, from top left to bottom right. The *Select* switch in SwitchAccess or the *double tap* gesture in TalkBack perform the *Click* action that is similar to touching the element without ATs. To represent each element to blind users, TalkBack also announces a textual description of the focused element on the screen. For visual elements like icons, these textual descriptions, which are called *Content Description* in Android, should be provided by developers in the

UI specification, a hierarchical structure of elements represented in an XML file.

Unfortunately, developers oftentimes only test their apps' functionality under conventional ways of interaction, leading to many inaccessible functionalities in apps. A developer who is aware of the disabled users' limitations may utilize accessibility testing tools, such as Google Accessibility Scanner [6], to evaluate the accessibility of their app. For example, for the lock page of AppLock, Accessibilty Scanner reports an issue for the text contrast of "Enter pin". Accessibility Scanner may also report "missing speakable text" if there is a clickable image without a content description, or "small touch target size" if the clickable area is too small for an element. Google Accessibility Scanner, as well as all other prior accessibility testing tools (e.g., [6, 8, 34, 42]), are aimed at finding *under-access*, i.e., features that should be available to the user but cannot be accessed using ATs. None of these tools report issues related to *over-access*, i.e., features that should not be available to the user but can be accessed using ATs.

In practice, a blind user may need to understand the screen content by exploring and navigating through all the elements on the screen. Figure 1(b) shows which elements can be focused by TalkBack. The numbers indicate the order in which elements are focused. After passing pin pad elements, TalkBack detects some elements that are not visible to sighted users. We call these elements Overly Accessible (OA) as they are not visible to sighted users or clickable by touch. Announcing these elements not only misleads the blind user about the content of the page, but in many cases also requires an exorbitant number of interactions to pass a long list of OA elements until the user reaches the visible functionality that the developer intended to be available. Such OA elements remain undetected in the prior accessibility testing tools.

These OA elements, as specified in Figure 1(b), can also pose security concerns. By listening to what TalkBack announces, we can understand that the OA elements correspond to the first page of AppLock as shown in Figure 1(a). This page contains the list of device apps and the mechanism to enable or disable their locks. For instance, element 17 in Figure 1(b) is the lock toggle for the Files app. This means that, using TalkBack, a user can access the locked apps and disable their protections, without even entering the pin code. In essence, she can bypass the lock screen protection. Prior research has demonstrated how Accessibility APIs can be used by malware authors to launch a security attack [25, 36] and how to prevent such attacks [38, 39]. No prior work, however, has aimed to develop a method of assisting developers with detecting vulnerabilities caused by OA elements in benign apps that can be readily exploited by any user, and using the standard ATs.

To fill this gap, we took a deeper look at how UI elements are represented to ATs. In modern platforms such as Android, Accessibility Service runs in the background and provides the required information about a window's content to ATs. From the perspective of Accessibility Service in Android, a window's content is presented as a tree of `AccessibilityNodeInfos` (nodes) [23]. Android 12 documentation lists 65 different types of information that are provided by nodes. Table 1 illustrates a sample set of this information. We hypothesize that nodes with peculiar specifications can lead to OA elements. For example, in Figure 1(b), by comparing the Bounds and DrawingOrder of elements, the second and third method in Table 1,

Table 1: Sample types of information exposed from nodes to ATs.

	Attribute	Description
1	**ActionList**	The actions that can be performed on the node.
2	**Bounds**	The coordinates of the bounding box of the node.
3	**DrawingOrder**	The drawing order of the view of this node.
4	**Text**	The text of this node.
5	**Enabled**	Whether this node is enabled.
6	**VisibleToUser**	Whether this node is visible to the user.
7	**Clickable**	Whether this node is clickable.
8	**ContentDesc**	The content description of this node.
9	**ChildCount**	The number of children.
10	**PackageName**	The package this node comes from.

we found that the layout that expands the whole window is drawn on top of some of the elements. While the elements underneath are covered for a sighted user, an AT can still navigate through them and announce them to an AT user. Our objective in this study is to study specifications of OA elements and propose an automated tool to detect such OA elements that can have severe security, privacy, and accessibility impacts on apps.

3 OVERLY ACCESSIBLE ELEMENTS

An element is OA if it is exposing more information/functionality to ATs than what is available through the conventional interaction mode. To understand to what extent node specifications can reveal OA elements, we perform an empirical study on manually detected OA elements on some real world apps. In this section, we explain the data collection and results of this study.

3.1 Data Collection

Our goal is to collect all the available information from nodes to ATs. To that end, we first developed an accessibility service, called OVERSIGHT Service (OSS), which is capable of capturing different types of information exposed from nodes. OSS runs in the background on an Android device and receives commands from Android Debug Bridge (ADB) [9], a command line tool that ships with Android devices. Using this service, we conducted an empirical study on 100 different screens of 20 real world apps. Our app list consists of 5 apps with built-in lock from Google play and 15 randomly selected apps from 38,106 apps that were published in 2021 in AndroZoo [1]. We installed each app on a Google Pixel 4 device, along with OSS. Then, one author interacted with each app to find 5 different states and explored each state with TalkBack and without it. We aimed at finding elements that are not visible to sighted users but TalkBack announces them or performs an action on them. We utilized OSS to dump OA nodes screenshot and specification in the hierarchy of nodes.

We then performed open coding of these elements iteratively. Two authors of the paper coded the elements, noting any condition that was not discovered before. To facilitate efficient coding, we developed a web application to visualize unannotated elements with search and batch tagging capabilities. In this way, authors can search and tag elements in batches using queries specified by different types of information from nodes, for example, Text ≠ ∅ ∨ ContentDesc ≠ ∅ filters elements without any information. After the initial coding, the authors discussed disagreements to reach a consensus.
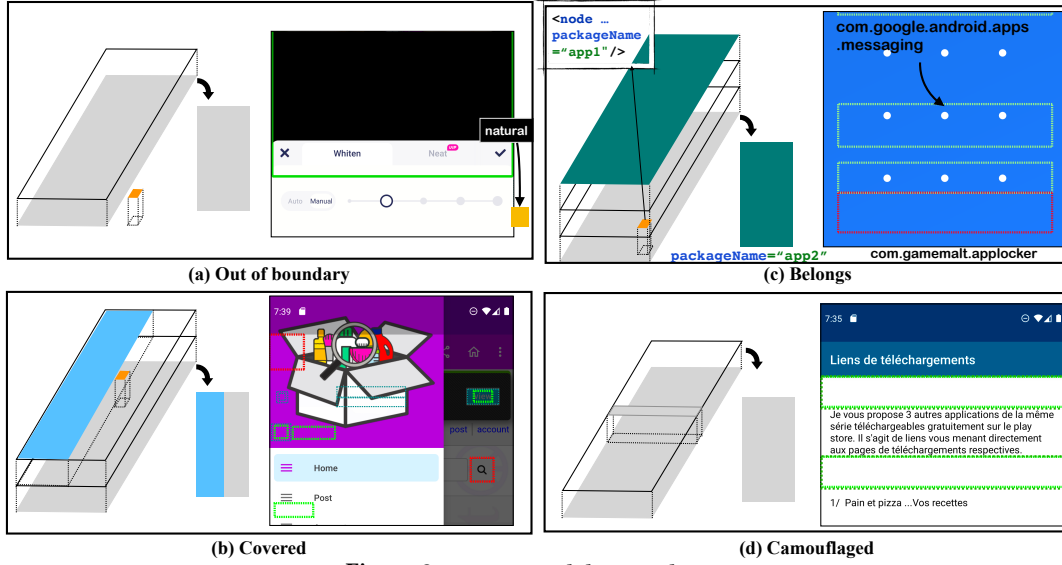
(a) Out of boundary

(c) Belongs

(b) Covered

(d) Camouflaged

Figure 2: Over Accessibility Conditions.

3.2 Results

We categorized the conditions of OA elements that were yielded during the coding procedure into two main classes:

- **Overly Perceivable:** elements that reveal content to an AT that is not available through regular interaction mode.
- **Overly Actionable:** elements that provide action to an AT that is not available through regular interaction mode.

These classes are inline with two accessibility principles from Web Content Accessibility Guidelines (WCAG) [46]: (1) Content should be equally perceivable by different users [47], and (2) UI elements should be equally operable by different users [48]. These principles can be violated due to bias in the level of access granted to any type of user, e.g., screen reader users vs. sighted users. While providing more access through conventional interaction modes, i.e., under-access problem, has been studied extensively and supported by a series of guidelines, not many works have investigated its counterpart, i.e., over-access problem. Our study is based on these principles and we organize detected OA elements' conditions under them. These conditions can be considered as accessibility guidelines to be later expanded or tailored to different platforms. Below, we list the conditions of OA elements we found in Android apps.

3.2.1 Overly Perceivable. A node with a textual data or content description is Overly Perceivable if it cannot be read or viewed by a sighted user, but can be accessed through programmatic means. We found the following conditions for such elements that are *hidden* to sighted users:

P1. Out of boundary: Nodes that are outside of the screen boundary, either with negative coordinates or with coordinates exceeding the device size. On the left, Figure 2(a) illustrates a schematic of the screen in layers corresponding to the drawing order of comprising elements. The orange element is OA as it is out of screen boundary and is not visible on the rendered screen. Figure 2(a) also shows an example in our empirical study on the right.

P2. Covered: Nodes that are covered by other nodes in the rendered UI. Dashed boxes in Figure 1 are examples of covered nodes.

Figure 2(b) also schematically shows how the orange OA element is covered by a blue sliding pane.

P3. Zero area: Nodes whose bounding box has zero area. These nodes will not be depicted on the screen but can be focused by an AT that will announce their content.

P4. Invalid bounds: Nodes whose captured bounds contradict the bounding box definition in Android documentation. The bounds attribute is supposed to be presented as the coordinates of the top-left and bottom-right points of the box. For example, if the coordinates of the ending point are smaller than the start point, the node has invalid bounds.

P5. Android invisible: Nodes that are not out of screen boundary and have positive area but they are specified as invisible to user.

P6. Belongs: Nodes that belong to a package name that is different from the app under test. Left side of Figure 2(c) illustrates that the green screen from app2 is placed on top of the elements of app1. In the rendered screen, the elements from app1 are not visible to sighted user but may be announced by ATs. The right side of Figure 2(c) shows a locker in our study, in which the elements of the Messages app are detected on the lock screen.

3.2.2 Overly Actionable. The `ActionList` attribute of nodes specifies the list of actions available to ATs. When a node support click action for ATs, the following conditions are barriers in performing that action through conventional interaction modes.

A1. Hidden: Nodes that are hidden to sighted users, i.e., with any of P1 to P6 conditions stated above.

A2. Disabled: Nodes that are disabled under certain conditions in the app and cannot be triggered by touch. Figure 3 provides an example for this condition, where the teeth correction function is disabled for unsubscribed user but using TalkBack, the user can activate it.

A3. Camouflaged: Empty nodes that are used as placeholders and are not detectable by sighted users, e.g., empty text boxes. Figure 2(d) provides the schematic placement of these nodes on the screen on the left and a real example on the right.
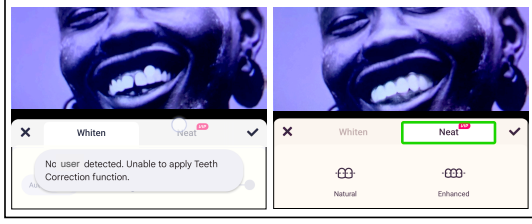
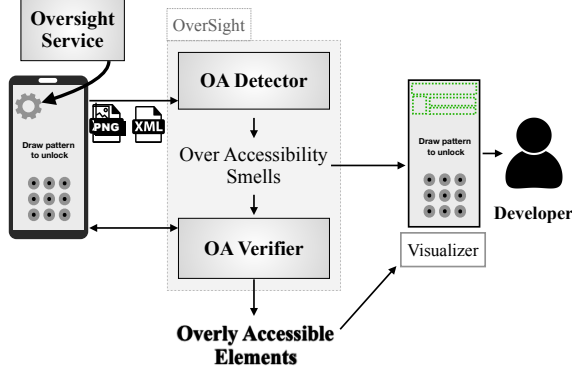**Figure 3: Neat button is not working when touched by enabled users but is available to TalkBack users.**

Figure 4: Overview of OverSight framework.

4 APPROACH

In this section, we introduce OverSight, an automated tool that gets the information from a specific state of the app and returns a list of OA elements confirmed by an AT. Figure 4 illustrates the overview of our approach. OverSight engine consists of two main components: OA Detector (Section 4.1) and OA Verifier (Section 4.2).

OA Detector gets a window's content specification in XML along with its screenshot through OverSight Service (OSS). As described in Section 3, OSS runs in the background, dumps hierarchical representation of nodes in an XML file, and enables communication with the device through broadcast messages. OA Detector analyzes nodes on the window and returns *Over Accessibility Smells*, i.e., nodes that meet one of the conditions derived from our empirical study (Section 3). Confirming over accessibility issues in these nodes is the responsibility of OA Verifier . Our approach only relies on available information to AccessibilityServices; therefore, it is applicable to any app regardless of its technology or even if it is obfuscated. OA Verifier communicates with the device and explores the window with an AT to validate the reachability and actionability of over accessibility smells. OverSight also visualizes over accessibility smells as well as OA elements on the screenshot along with their specification for developers. In the following sections, we describe the details of each component.

4.1 OA Detector

Our empirical study organizes a set of conditions under the basis of over-perceivability and over-actionability. OA Detector implements these conditions to automatically check the nodes against them.

Here, we describe the details of P1, P3-P6, and A1-A2 as implemented, and also elaborate on the algorithms used to calculate covered nodes (P2) and camouflaged nodes (A3). Implementation details of all conditions are available with our open-source tool

available at [37]. First, we define the conditions for perceivable and actionable nodes, and then we formally define all over-perceivable and over-actionable nodes. Recall from the description of these conditions in the previous section that all P1 to P6 nodes must be perceivable, and all A1 to A3 nodes must be also actionable.

Perceivable: A node is perceivable if it has a textual information. Based on Table 1, the attributes *text* and *ContentDesc* may contain such information.

$$\forall n \in Node; n.Text \neq \varnothing \vee n.ContentDesc \neq \varnothing$$
$$\Rightarrow perceivable(n) = True \quad (1)$$

Actionable: A node is actionable if it has an attribute that is associated with an action, e.g., *Clickable*, *LongClickable*, or the existence of these actions in *ActionList*.

$$\forall n \in Node; n.Clickable \vee n.LongClickable \vee$$
$$\{CLICK, LONGCLICK\} \cap n.ActionList \neq \varnothing \quad (2)$$
$$\Rightarrow actionable(n) = True$$

P1. Out of boundary: To detect these nodes, we compare the boundary of the node with the size of the window, i.e. $Window.width$, $Window.height$. The bounds of an element is shown as the coordinates of the top left and bottom right of its bounding box.

$$\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1];$$
$$x_0 < 0 \vee x_1 > Window.width \vee y_0 < 0 \vee y_1 > Window.height$$
$$\Rightarrow out\_of\_bound(n) = True$$
$$\quad (3)$$

P2. Covered: To find out covered elements, we investigate how Android draws elements on a window. Android draws a window starting from the root node and recursively draws the child elements according to their drawingOrder. To determine what nodes are covered, we simulate Android's drawing in reverse order using a depth-first search algorithm. We start visiting nodes from the last drawn node to the first drawn node and keep track of covered areas. A node is "covered" if any of the covered areas obscure its bounding box.

Algorithm 1 explains our approach in details. For a given node, n, and a set of bounds that may cover it, B_C, DetectCovered first checks if n is covered to set all the descendants up to the leaf node as covered. (Line 2-4) If n is not covered, we will assess if its children are covered. To that end, we first sort the children in descending order based on their drawingOrder in line 5. The first element in the *ordered* list is the last child drawn by Android on the window among the other children. Then, in line 6, we iterated through the children and check if they are covered by any bounds in B_C. If that is the case, in the recursion call, the algorithm set all the descendants covered. Otherwise, in the recursion call, children of node m will be assessed. In line 10, we add the bounds of node m to the set of covering bounds since the other children in the for loop of line 6 may be covered by m.

P3. Zero area: The bounding box of any node forms a rectangle which can have a zero area.

$$\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1];$$
$$x_0 = x_1 \vee y_0 = y_1 \Rightarrow zero\_area(n) = True \quad (4)$$

P4. Invalid bounds: We use Equation 5 to find the nodes whose bounding box – bottom-left and top-right coordinates – is not a

---

Algorithm 1: Overlap Analysis Algorithm

Input: $n \in Node$ (The visiting node),
  $B_C : \{b_1, \cdots, b_k\}$ (The set of covering bounds)
1 **Function** DetectCovered(n, B_C):
2   **if** $n.covered$ **then**
3         $\forall d \in n.descendants : d.covered \leftarrow True$
4   **return**
5     $ordered \leftarrow$ Sort $n.children$ based on decreasing order of $drawingOrder$
6   **foreach** $m \in ordered$ **do**
7   **if** $m.bounds$ is covered by B_C **then**
8             $m.covered \leftarrow True$
9   DetectCovered(m, B_C)
10         $B_C \leftarrow B_C \cup m.bounds$

---

rectangle.

$$\forall n \in Node, n.bounds \equiv [x_0, y_0, x_1, y_1]; x_0 > x_1 \vee y_0 > y_1 \\ \Rightarrow invalid\_bounds(n) = True \tag{5}$$

P5. Android invisible: To detect nodes with this condition, we look for nodes without any of the above-mentioned conditions that are marked as invisible to user in node attributes. (Recall row 6 in Table 1)

$$\forall n \in Node; \neg n.Visible \wedge \neg out\_of\_bound(n) \wedge \\ \neg n.covered \wedge \neg zero\_area(n) \wedge \neg invalid\_bounds(n) \\ \Rightarrow android\_invisible(n) = True \tag{6}$$

P6. Belongs: We compare the package name of nodes with the package name of the UI under test (UIUT) to find if nodes belong to its corresponding app.

$$\forall n \in Node; n.pkgName \neq UIUT.pkgName \\ \Rightarrow belongs(n) = True \tag{7}$$

A1. Hidden: An actionable node that has any conditions in Equations 3 to 7 is considered hidden, since a sighted user cannot perform any touch gesture on it.

$$\forall n \in Node; out\_of\_bounds(n) \vee n.covered \vee zero\_area(n) \vee \\ android\_invisible(n) \vee invalid\_bounds(n) \vee belongs(n) \\ \Rightarrow hidden(n) = True \tag{8}$$

A2. Disabled: The *enabled* attribute of a disabled actionable node should be *False* to be considered as over-actionable (Recall row 5 in Table 1).

$$\forall n \in Node; \neg n.enabled \Rightarrow disabled(n) = True \tag{9}$$

A3. Camouflaged: Detecting camouflaged nodes (A3) is challenging since there is no attribute in nodes indicating their color. This condition occurs when developers want to utilize some empty views as a placeholder. To detect these elements, we filter out nodes that have any child. Then, we evaluate the image associated to the remaining nodes. To get the image, we crop the screenshot based on the coordinates of the bounding box of the node. Then, we check if all the pixels of the image have the same color. With the advent of advanced computer vision and machine learning algorithms, analyzing app screenshots has been recently studied in prior works [13, 15, 50]. Such techniques can infer not only UI nodes, but also their structure from screenshots. While these advanced

UI analysis techniques can be adopted here, we opt for the simple aforementioned technique that can effectively detect empty boxes without the need for complex models.

OA Detector evaluates compliance of each node with the defined conditions to find nodes that has *Over Accessibility Smells*, i.e., they have symptoms that can lead to revealing information or functionality to AT users that is not available to sighted users. To verify their accessibility with an AT, we propose OA Verifier as below.

4.2 OA Verifier

The behavior of different ATs in focusing on the elements and performing an action on them cannot be predicted statically. To confirm if an AT can reach the detected over accessibility smells, we utilize OA Verifier . The goal of this component is to evaluate the reachability and actionability of nodes identified by OA Detector on a real device with an AT. To interact with the device, we expand the capabilities of OverSight Service (OSS) that was previously only responsible for capturing information from nodes. OSS receives commands from OA Verifier, perform the required gestures on the device, broadcast commands and return the results.

To achieve its objective, OA Verifier uses two subcomponents: 1) Reachability Analyzer, and 2) Actionability Analyzer. The first component verifies if an AT can focus on the node, while the second one checks if the AT can perform the action on it. In this work, we describe our approach for TalkBack as the standard screen reader in Android, and a custom AT, called Super AT (SAT), as it has all the information and functionalities provided by Accessibility Service. As briefly mentioned in Section 2, Accessibility Service in Android runs in the background and provides the required information to ATs. Each AT specifies a list of *flags* [21] to request for the corresponding information and capabilities from the Accessibility Service. For example, flagRetrieveWindowContent is required to be able to get the events indicating that something on the window has changed. In this work, we give all the capabilities to SAT, making it a representative of all ATs that are using a subset of its capabilities. In other words, SAT-verified nodes show what can *potentially* be accessible to different ATs, while OA nodes verified by TalkBack show that any user, who utilizes the standard platform screen reader, can get access to their content. The input for both of these subcomponents is an emulator snapshot, captured from a specific state of the UI under test, and a list of nodes to be verified, i.e., over accessibility smells.

4.2.1 Reachability Analyzer. If an AT can focus on a node, we call the node reachable by that AT. In Android, Accessibility API can perform actions on given nodes by calling the performAction method. OA Verifier identifies nodes by their XPath, i.e., their absolute path from the root node, and performs the focus action, ACTION_ACCESSIBILITY_FOCUS, on them. If this focused node, returned by AccessibilityService, is the desired node, OA Verifier determines the node reachable by SAT.

To assess the reachability of a node with TalkBack, we utilize the "Explore by swiping" strategy instead of the touch exploration as OA elements are not viewable on the screen to be enabled by tapping/touching. Since OA elements most likely appear after the ones that are visible to sighted users, OA Verifier first explores the screen backward by drawing "swipe left" gesture. Whenever

TalkBack focuses on a node, OA Verifier calls the node TalkBack Reachable. TalkBack continues screen exploration until either it reaches all the nodes in the given list, or sees a repetitive node.

Some UI components such as scrollable widgets may render some elements on the app unreachable. In practice, to break such infinite loops, a screen-reader user can touch on an element outside of the loop and resume exploring the app. To work around these loops, OA Verifier performs both forward and backward navigation from the top of screen when it does not meet its stopping criteria. Eventually, nodes that TalkBack cannot focus on by either backward or forward app exploration are determined to be unreachable by TalkBack.

4.2.2 Actionability Analyzer. An element is considered actionable, if it 1) is reachable and 2) performs the action successfully. Thus, Actionability Analyzer first evaluates reachability of over actionability smells using the same strategy as Reachability Analyzer.

Once Actionability Analyzer determines reachable nodes, it attempts to performing the action on them. This means it requires to first focus on the element and trigger the action using TalkBack or SAT. Since reachability of these nodes have already confirmed, we directly put the accessibility focus on the node under test using Accessibility API. Then, we utilize the specific AT to perform action. For TalkBack, OA Verifier performs a double-tap gesture to click the focused node. For SAT, OA Verifier calls performAction(ACTION_-CLICK) on any given node. To verify if the action was performed successfully, OA Verifier listens to the AccessibilityEvents and denotes the node clickable by either TalkBack or SAT if VIEW_CLICK event or WINDOW_CONTENT_CHANGED was logged.

5 EVALUATION

In this section, we evaluate OverSight on real-world apps to answer the following research questions:

RQ1. How accurate is OverSight in detecting OA elements?
RQ2. How prevalent are over-access problems in security-concerned apps?
RQ3. What are the potential impacts of OA elements on different apps and communities?
RQ4. What is the performance of OverSight?

5.1 Experimental Setup

5.1.1 Datasets. We evaluated our approach on 60 app screens from 30 real-world Android apps. Our test set consists of three groups of apps: (*group1*) 10 app lockers similar to the motivation example from Google Play, (*group2*) 10 apps with known accessibility issues in a prior study [42], and (*group3*) 10 randomly selected apps from different categories of Google Play. For each app, we captured two different states of the app. For apps in group1, the first state is the lock screen of the app itself, and the second state is the lock screen that protects a third-party app, e.g., Messages, when it is locked. For apps in group2, we selected two different screens of the app with the confirmed accessibility issue. Lastly, for apps in group3, we randomly explored the apps and captured two different screens. For the second question, we mainly focus on app lockers, security-critical apps that are responsible for protecting user apps. We picked 5 highest-rated, 5 lowest-rated, and 5 randomly selected app lockers from Google Play and followed the same strategy as group1 to capture two different states from each app. We did not

incorporate the low-rated app lockers in RQ1 to keep the quality of apps in that study consistent.

5.1.2 Implementation details. We ran our experiments on an Android emulator based on Android 11.0 and with TalkBack version 12.1 on a typical development machine, using a MacBook Pro with 2.4 GHz core i7 CPU and 16 GB memory. OverSight Service is implemented in Kotlin and communicates with OA Detector and OA Verifier components, implemented in Python, using ADB [9].

5.2 RQ1. Accuracy of OverSight

To answer this question, we ran OverSight on each snapshot in our test set and carefully examined the reports. We separately evaluate OverSight's two main components, OA Detector and OA Verifier. **OA Detector:** To evaluate OA Detector, we carefully checked the reported OA smells in each category and tagged them as True Positive (TP) if it was correctly detected with one of the OA conditions and False Positive (FP) otherwise. We then calculate OA Detector's precision as the ratio of the number of nodes that were correctly detected by OA Detector to the number of all detected OA smells.

Table 2 summarizes the results of this experiment. Each row in this Table corresponds to one state of an app. The number of nodes in each state varies as shown in the second column (N). In our test set, it can be as few as 6 nodes and as many as 656. *Smell* column indicates the number of nodes with Overly Perceivable (P) or Overly Actionable (A) conditions on each screen. We display the precision per app state under the *DP* (Detector Precision) column, and the average precision is in the last row.

As shown in the Table 2, on average, OA Detector has a precision of 84.23% in detecting OA smells. For 56 different states in 28 number of apps, the precision is 100%. We analyzed the elements recognized as False Positive, i.e., with FP tag, to better understand OA Detector failures. Figure 5 shows some examples where OA Detector erroneously evaluates a node as OA. In Figure 5(a), the map and the text on it is annotated as OA. Further inspection of this layout showed us that the map is behind a transparent layout and made our algorithm classify the underlying nodes as "covered" (recall P.2 in Section 4.1). In Android, transparent layouts pass the touch gesture to the underlying elements so that they are not recognizable through conventional interaction modes. Since layout colors are not included in node information, OA Detector cannot distinguish transparent layouts from color-filled ones. Moreover, having a stack of transparent nodes, if not maintained properly, can cause troubles for AT users. For example, if all the stacked nodes are focusable, AT will focus on each of them separately, confusing AT users about what is shown on the screen and resulting in a less optimal navigation experience. Partially covered nodes are another failure of OA Detector as shown in Figure 5(b). There is a "Sort and Filter" button covering the elements underneath. However, as the underlying texts are partially recognizable to sighted users they are tagged as FPs. OA Detector does not exclude partially covered elements in the "covered" category since a developer may have intentionally blocked access to part of a node content.

To evaluate if OA Detector fails to detect any issues, i.e., False Negatives, we ran OA Detector on a set of apps with known issues. OA Detector's False Negatives are the OA elements that OA Detector fails to report. Since no prior dataset exists, we take our

apps from the empirical study that were manually investigated for OA elements (recall Section 3). We investigated the manually confirmed OA issues that do not appear in the list of OA smells. Figure 3 shows the only case that the OA Detector failed to detect. The reason for this failure is that instead of disabling the button, the app intercepts the click event at runtime when it is touched to show an error message. This means the button performs the click action successfully with and without AT. However, its inconsistent behavior cannot be detected by OA Detector. We also noticed in some cases the issue was captured not in the first attempt but after the second attempt. This issue is due to the challenges of interacting with the device using OSS and analyzing the results at a proper time. To mitigate such validity threats, we ran our experiments 3 times on each app.

Further investigation of conditions of detected OA elements revealed that the "covered" condition (recall P.2 in Section 4.1) is the most frequent symptom of OA elements. 18 apps out of 30 had at least one "covered" OA element. According to Android documentation, Android attempts to evaluate whether a node is visible to user [19] (recall row 6 of Table 1) to be announced by TalkBack. However, our review of Android's source code [22] indicates the platform only compares the bounds of a child node with its parents to evaluate if they are visible to user (i.e., the corresponding `VisibleToUser` flag is set to true). However, such a comparison does not exist for nodes that are siblings or children of siblings. We believe Android platform should reassess its strategy of detecting visible nodes to minimize such issues.

OA Verifier: To evaluate the OA Verifier component, we investigate the nodes specified as reachable and actionable with TalkBack and SAT. To check the reported nodes by OA Verifier, we load the corresponding snapshots of the app states on the emulator and utilize an AT, e.g., TalkBack, to explore the app and assess Reachability (R) and Actionability (A) of OA smells. In terms of reachability, if the AT can focuses on a node, we consider it reachable. For actionability, the node is actionable if it is reachable and is clickable, i.e., the click gesture, such as double tap in TalkBack, broadcasts a click event. When an element is clicked successfully in Android, an `AccessibilityEvent`, called `VIEW_CLICKED`, is created and sent to `AccessibilityServices`. To determine if the action was performed, OverSight service captures the events and shows if an event of type `VIEW_CLICKED` or `WINDOW_CONTENT_-CHANGED` is logged. Since OA Verifier follows the same strategy in detecting clicked nodes, the accuracy of OA Verifier equals to its accuracy in detecting reachable nodes. Thereby, we label the output of OA Verifier as true if it matches with our manual investigation and false otherwise. Using these tags, we calculate precision and recall of OA Verifier as follows: Precision is the ratio of number of nodes that correctly verified to be reachable to the number of reachable nodes detected by OA Verifier, while recall is the ratio of number of nodes that correctly verified to be reachable to the number of OA smells that are manually verified to be reachable.

Table 2 shows the average precision and recall of OA Verifier using TalkBack and SAT in the last two columns, VP (Verifier Precision) and VR (Verifier Recall). As shown in the last row, the average precision and recall on all apps is 100% and 83.27% respectively. While OA Verifier is 100% precise in its reports, the recall shows that it has missed some issues. Figure 5(c) shows an example of a set
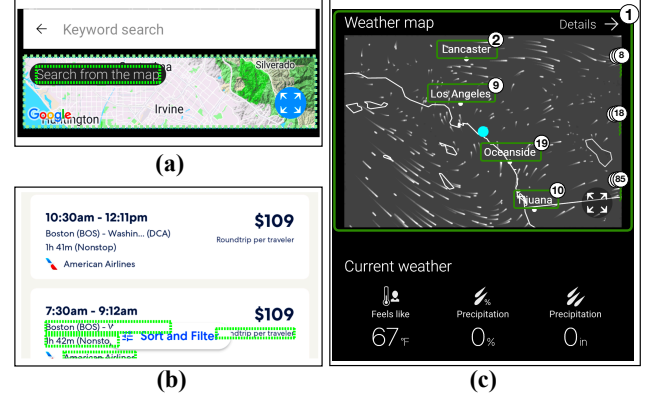

Figure 5: OverSight Failures: (a) and (b) are false positives of OA Detector, where dashed green boxes are erroneously detected as covered; (c) is a false negative of OA Verifier, where TalkBack is stuck in the world map.

of nodes that were erroneously detected to be unreachable by OA Verifier. On this state of the "Weawow" app, there is a map of all the cities that a user can get the weather information for. When TalkBack reaches this widget, it navigates through all the nodes on the map, as depicted by number annotations on the map, and gets stuck there in an infinite loop. Thus, all the nodes on the second half of the screen were mistakenly reported by OA Verifier as unreachable or not over accessible (False Negative). OverSight attempted to address such issues for scrollable widgets by navigating both forward and backward on the screen. However, backward navigation on this app does not help since the app content loads dynamically in forward navigation, while scrolling to the bottom. OA Verifier also has a similar issue in web apps such as Dictionary. In this app, every time the app is scrolled forward, it fetches a totally new UI specification which although looks visually similar, uses different XPaths for nodes, making the logged information inaccurate.

5.3 RQ2. OA Elements in Security-Sensitive Apps

OA elements in security-sensitive apps, such as app lockers, put the privacy and security of both users and apps at stake by divulging private content or granting access to functionality that they are supposed to protect. To understand the prevalence of such critical issue in these apps, we utilize OverSight to test 15 real-world app lockers. All app lockers require two permissions from the users to work, 1) "Usage Access", to track what other apps the users are using, 2) "Display Over other Apps", to place their lock screen on top of the other apps. We grant the required permissions to the apps and evaluate two main functionalities of lockers: 1) locking the locker app itself, and 2) locking another third-party app that they are to protect. Table 3 summarizes the test set and results. The table contains three groups of 5 app lockers — most popular, randomly selected and least popular — from a list of 125 lockers we got from the Google Play store. The list includes the app locker from our empirical study, marked by '*' in the table. 'x' indicates that the protection could be bypassed by an AT, while '✓' indicates no OA element was detected by OverSight. We also manually confirmed

Table 2: Accuracy of OverSight in running on 30 apps.

App	N	Smells		DP	TalkBack		SAT	VP	VR
		P	A		R	A	A		
...domobi...	47	0	2	0.00	0	2	2	1.00	1.00
	26	9	6	1.00	8	3	6	1.00	0.95
...alpha...	12	0	0	1.00	0	0	0	1.00	1.00
	8	0	0	1.00	0	0	0	1.00	1.00
...sp.pro...	42	0	0	1.00	0	0	0	1.00	1.00
	26	9	6	1.00	8	5	6	1.00	0.90
...thinky...	18	0	0	1.00	0	0	0	1.00	1.00
	17	1	0	1.00	0	0	0	1.00	0.50
...litetoo...	73	6	7	1.00	6	7	7	1.00	1.00
	73	0	0	1.00	0	0	0	1.00	1.00
...nevways...	55	1	1	0.00	0	1	1	1.00	1.00
	6	0	0	1.00	0	0	0	1.00	1.00
...ammy.a...	16	0	0	1.00	0	0	0	1.00	1.00
	26	9	6	1.00	8	6	6	1.00	0.95
...gsmobile...	53	0	0	1.00	0	0	0	1.00	1.00
	37	0	0	1.00	0	0	0	1.00	1.00
...cd.app...	12	0	0	1.00	0	0	0	1.00	1.00
	12	0	0	1.00	0	0	0	1.00	1.00
...saeed.ap...	13	0	0	1.00	0	0	0	1.00	1.00
	16	0	1	0.00	0	1	1	1.00	1.00
...c51	83	4	1	0.00	4	1	1	1.00	1.00
	29	0	1	0.00	0	0	1	1.00	1.00
...fatsec...	41	0	1	1.00	0	0	1	1.00	0.50
	147	14	5	1.00	2	0	5	1.00	0.70
...colpit...	18	2	0	1.00	0	0	0	1.00	0.50
	67	2	1	1.00	0	0	1	1.00	0.50
...tripit	202	55	20	0.91	6	1	20	1.00	0.54
	270	68	23	1.00	4	4	23	1.00	0.55
...contex...	52	0	26	1.00	0	0	5	1.00	0.50
	57	1	0	0.00	1	0	0	1.00	1.00
...yelp.an...	66	0	0	1.00	0	0	0	1.00	1.00
	129	15	9	0.86	0	0	6	1.00	0.50
...devhd.f...	71	19	4	1.00	4	0	4	1.00	0.60
	138	44	21	1.00	8	0	21	1.00	0.67
...ziprecr...	36	0	0	1.00	0	0	0	1.00	1.00
	63	5	5	1.00	0	0	2	1.00	0.50
...diction...	102	8	5	1.00	2	1	4	1.00	1.00
	177	98	5	0.89	98	1	1	1.00	1.00
...and...	110	16	10	0.95	0	0	8	1.00	0.50
	69	16	10	0.53	16	9	9	1.00	1.00
...airbnb...	42	0	1	1.00	0	0	0	1.00	0.50
	56	0	3	1.00	0	0	1	1.00	0.50
...carfax...	30	0	0	1.00	0	0	0	1.00	1.00
	20	0	0	1.00	0	0	0	1.00	1.00
...expedi...	53	0	2	1.00	0	0	1	1.00	0.50
	98	6	0	0.33	4	0	0	1.00	0.83
...houzz	22	0	0	1.00	0	0	0	1.00	1.00
	169	37	27	1.00	9	3	5	1.00	0.85
...mcdona...	42	1	0	1.00	0	0	0	1.00	0.50
	126	35	10	0.32	35	5	5	1.00	0.94
...meditat...	22	3	2	1.00	0	0	2	1.00	0.50
	46	15	1	0.75	14	0	1	1.00	0.94
...pinterest	32	1	1	1.00	1	1	1	1.00	1.00
	24	0	0	1.00	0	0	0	1.00	1.00
...popular...	36	5	3	1.00	0	0	3	1.00	0.50
	158	40	19	1.00	21	0	2	1.00	0.68
...theathl	20	0	1	1.00	0	1	1	1.00	1.00
	77	35	5	1.00	34	5	5	1.00	0.99
...weawow	32	2	2	0.00	0	0	2	1.00	1.00
	656	280	52	1.00	194	3	5	1.00	0.87
Average:				84.23%				100%	83.27%

Table 3: Over accessibility issues in app lockers.

App	Version	#Installed	Rate	State 1		State 2	
				TB	SAT	TB	SAT
com.netqin.ps	293	+100M	4.3	✓	✓	✓	x
com.domobile....	2021052001	+100M	4.2	✓	✓	x	x
com.alpha.app...	412	+50M	4.7	✓	✓	✓	✓
com.sp.protec...	231	+50M	4.4	✓	✓	x	x
com.thinkyeah...	166	+10M	4.6	✓	✓	✓	✓
com.litetools...	91	+10M	4.3	x	x	✓	✓
*com.gamemalt...	108	+5M	4.3	x	x	x	x
com.nevways.a...	92	+5M	4.3	✓	✓	✓	✓
com.ammy.app....	151908296	+1M	4.6	✓	✓	x	x
com.gsmobile....	34	+500K	4.5	✓	✓	✓	✓
me.ibrahimsn....	134	+50K	4.0	✓	✓	✓	x
com.cd.applo....	2	+10K	4.5	✓	✓	✓	✓
com.saeed.app...	4	+10K	4.4	✓	✓*	✓	✓*
com.applockli...	8	+10K	4.0	✓	✓	✓	✓
com.mms.applo...	1	+5K	4.0	✓	✓	✓	✓
app.lock.hide...	6	+5K	3.5	✓*	✓*	✓*	✓*

see the lock screen. On the other hand, interestingly, the over-access problem is not as common in the last 5 apps. We realized that app lockers utilize different strategies in providing a lock screen. For example, in the last app in Table 3, we found that the app locker first puts the app in the background and then displays the lock screen. In this way, OA elements still exist, yet they will not endanger the target app as they are the nodes on the home screen. Such strategy is time and energy consuming and would be less appealing to users. Among other apps, some inflate a full-screen overlay on the locker without creating a new Activity such as "com.gamemalt.ap..." or "com.litetools...", while the other ones such as "com.sp.protec...", "com.domobile...." and "com.ammy..." create a new Activity for the lock screen. Android provides mechanisms for both approaches to manage the hierarchy of nodes for the UI elements. The default behavior in inflating an overlay on the same Activity results in appearance of all the elements of the Activity, including those that should not be accessible, in the UI hierarchy. Thus, developers need to take proper actions to avoid that by setting their nodes not important for accessibility for example. However, by default, the hierarchy of nodes for a new activity only incorporates nodes that are specified in this activity and will not leak the elements from prior activities.

Developer's decision in utilizing these strategies can impact app stability, robustness and usability. We strongly encourage them to consider security threats of OA elements, resulting their design decisions as well as the other app qualities.

5.4 RQ3. Qualitative Analysis of OA Elements

We manually examined all reported OA elements by OverSight in Table 2 and categorized them based on their impact on disabled users and app developers in terms of app accessibility, app security, and work flow violations.

5.4.1 App Accessibility. Both over perceivable and over actionable elements degrade app accessibility, hindering disabled users' ability to explore the app conveniently. For example, in "30 days workout" app, Figure 6(a), a blind user has to navigate through the covered elements, highlighted in green. Although these OA elements, requiring paid subscription to access, are not actionable, a user who wants to understand the app content would be confused of what is shown on the screen. Moreover, if she wants to reach a specific button, e.g., Profile, she has to pass through all OA elements, resulting in a less optimal user interaction. A similar scenario happens in the

the automatically diagnosed over-access problem by OverSight and reported all the issues to the developers.

As Table 3 shows, 13 cases out of 30 different states have over-access problems not only by SAT, but also by TalkBack, the standard screen reader. 5 of these issues belong to the most popular apps, endangering the security of hundreds of millions of users.

We also observed that apps with lower rating and installation number are not as robust as popular ones. For example, we had to reopen the locked app using "com.saeed..." multiple times to finally
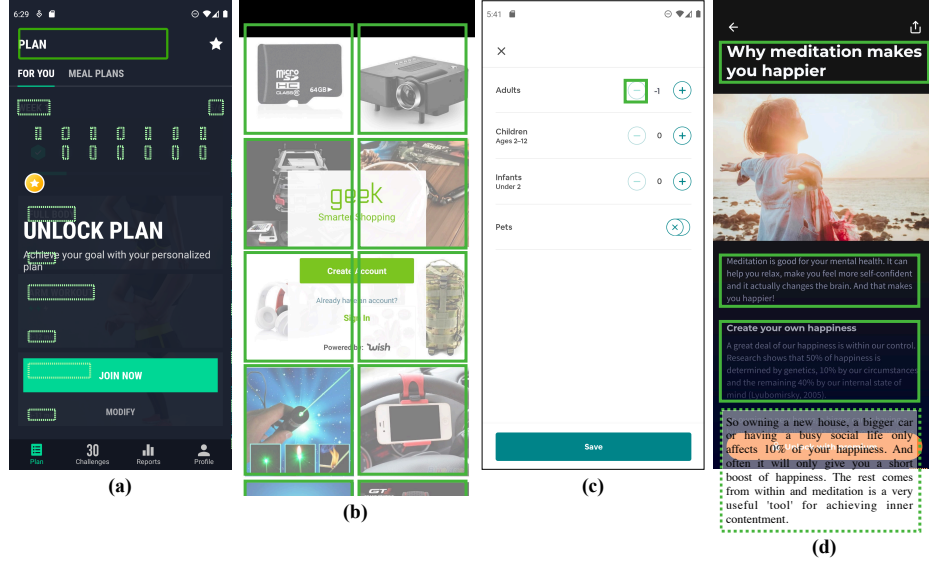
Figure 6: Impacts of OA elements. (a) Accessibility issue of overly perceivable elements. (b) Accessibility issue of overly actionable elements. (c) Workflow violation, giving access to premium content. (d) Workflow Violation, breaking app logic.

welcome page of "iSaveMoney" app. The intended use-case is for the user to follow the introductory steps; however, the information from next steps are available to AT user from the beginning, making the introduction complicated. School Planner, ZipRecruiter, and McDonlads have a similar issue. It is worth mentioning that OA elements in app lockers discussed in RQ2 not only undermine app functionality for AT users but also complicate their interaction with apps. When they explore app by swipe, there is no lock preventing their access. However, app exploration by touch will not activate the OA elements that supposedly exist on the screen.

In some cases, OA elements provide actions to AT users. Case in point, background images in Geek, shown in Figure 6(b), are not accompanied with any textual data but are actionable. Although none of them are associated with any functionalities, i.e., they do not change the screen content when triggered, they complicate app exploration for AT users who believe there are real buttons on the screen. Interestingly, this app was also diagnosed with under-access problem in a prior work [42] because of a rolling dynamic widget in the background. The AT user gets stuck in an infinite loop and cannot login if she wants to explore the screen by swiping.

5.4.2 App Security. In RQ2, we extensively explained the critical impact of OA elements on the security of app lockers. OA elements in such apps can reveal the screen content of other apps that they are designed to protect. They can also provide access to the settings page, where the AT user can disable the lock totally. As the app lockers are mainly responsible for protecting app content, OA elements put a vulnerable app's reputation at stake. The issue is, however, not limited to app lockers. For example, parental control apps, which provide a mechanism to lock specific apps on the child's device, or variety of built-in locks in apps such as banking are also vulnerable to OA elements.

5.4.3 Workflow Violations. Developers design a workflow by which users interact with apps. Violating such workflows can 1) break app logic, 2) provide unauthorized access to premium content.

Developers restrict access to some functionalities to avoid false inputs and gather required information from users. For example,

in the Airbnb app depicted in Figure 6(c), when the number of passengers is zero, the decrease button for the number of travelers is disabled. However, using AT one can decrease the number of passengers to less than zero. Similarly, in Expedia and FatSecret apps, the continue button is disabled until the user enters the required information at each step. Using ATs, users can pass invalid inputs, which can result in the app malfunctioning or crashing.

In some cases, the workflow violation targets developer's revenue model. Figure 6(d) illustrates an article in a meditation app which is only available fully for the subscribed users. However, TalkBack announces the whole content of this article and scrolls through it without asking for a subscription. The same issue exists for the premium articles in the "The Athletics" app. While these examples are related to the restricted scroll functionality, the same issue threatens any other blocked functionalities that are intended to be available to subscribed users.

5.5 RQ4. Performance

The time-consuming component of OVERSIGHT is OA Verifier which needs to interact with the device and perform actions on elements. On average, it takes 54 seconds for OA Verifier to perform an action. The execution time varies in different apps as their number of nodes and OA smells are different. For the apps in our test set, the average execution time of OVERSIGHT is 571 seconds, which can be effectively used in practice. Any dynamic analysis tool, including OVERSIGHT, is costly in time compared to simple static checkers. The OA Detector runs very fast, under one second. By identifying the OA smells, OA Detector reduces the number of nodes that need to be verified by 84% on average. Without OA Detector, an expensive verifier would need to assess every single node on the screen.

6 THREATS TO VALIDITY

External validity. An important threat is the completeness of OA elements' conditions, extracted from examining 100 different states of 20 randomly selected apps. To mitigate this issue, we carefully selected a diverse set of app states considering the limitations of

manual exploration. The extracted conditions were organized under two main classes, Over Perceivability and Over Actionability, inspired by accessibility guidelines. Although this process gives us confidence that the conditions provide a good coverage for different variations of app states, having a larger set of app states would increase the validity of generalization of our findings.

Another threat is the generalizability of the reported results of OverSight on real world apps. Our evaluation dataset for RQ1 consists of 60 screens from 30 apps. While including more apps and screens would increase the validity of this experiment, we have attempted to mitigate this threat by selecting the apps from three different sources: (1) apps with confirmed under-accessibility issues, (2) apps with an intention to make users' information secure, and (3) a diverse selection of popular apps – 30 apps in 16 different categories in total. The first two groups are intentionally selected, since they are related to under- and over-accessibility, respectively.

While Oversight mainly relies on XML layout of a screen to detect OA conditions, for detecting camouflaged elements it requires a screenshot of the screen, which is not possible for apps that restrict the ability to capture screenshot. This tends to be the case for apps displaying copyrighted content. In such situations, Oversight may miss OA elements with the camouflaged condition.

Internal validity. Our implementation of OverSight is built on top of several tools, like *ADB* and *AccessibiltiyService*, which can introduce bugs in the process of OA element detection. Moreover, it is possible there are defects in our implementation of the prototype. To address these threats, we used the latest versions of third-party tools, conducted code review on our implemented program via Github, and extensively tested the prototype in a set of apps (with no intersection with the empirical study or the evaluation data sets).

7 RELATED WORK

Accessibility Analysis of Mobile Apps. Analyzing mobile app accessibility has been an active research area with the focus on proposing accessibility guidelines [46], empirical study [3, 41, 44], automated testing [6–8, 17, 24, 40], and repair techniques [2, 14, 33, 53]. Although accessibility principles [49] have implications for both under-access and over-access problems, there is no guideline regarding over accessibility and prior studies and tools are merely aimed at analyzing inaccessible functionalities in apps. OverSight is the first work in introducing the over-access problem and the first attempt to detect them.

The biggest challenge in detecting these issues is that OA elements manifest themselves in interactions involving ATs. However, the majority of accessibility testing tools are AT-agnostic. Static analysis tools like Lint [8] parse screen content and configuration files upon compilation to identify accessibility violations in code. To find issues that are undetectable in code, dynamic approaches [6, 7, 17, 26, 40] have been developed that analyze the rendered UI components on the screen, either after manual navigation to the target state of the app [6, 7, 26] or with an automated crawler [17]. These techniques, however, do not consider the use of ATs like screen readers and external keyboards in app exploration.

A prior technique, called Latte [42], utilizes ATs to evaluate if an app's functionalities, generated from its UI test cases, can be performed by disabled users. However, test cases are not always available, without which assessing UI elements with ATs is a time and

memory intensive process. To mitigate this issue, Groundhog [43] proposes an optimized app exploration approach for accessibility testing. However, both Latte and Groundhog only focus on inaccessible elements. OverSight's contribution is in taking advantage of characteristics of OA elements to detect and verify over accessibility issue and its impacts on mobile apps.

Security Studies on Accessibility. Accessibility has also been studied in a security context, considering how accessibility APIs on mobile platforms can be exploited by attackers [27]. Kraunelis et al. [32] showed how malicious apps can abuse accessibility service to detect app launches and bypass security measures [45]. Researchers have also investigated the potentials of using accessibility APIs in designing attacks such as ClickJacking attacks [4, 29, 52]. These targeted the `BIND_ ACCESSIBILITY_SERVICE` permission to take full control of the UI, as demonstrated by Cloak and Dagger technique [18]. Studies have devised defense schemes [51] and solutions to this attack [25, 38, 39].

These security studies approached accessibility from a malware's perspective, designing potential attacks, analyzing the framework and proposing solutions based on the assumption that the assistive app is malicious. Conversely, we introduced OA elements as a new vulnerability that can be exploited using a benign app or standard ATs such as TalkBack. The privacy implications of such elements for users as well as their negative impact on developers' revenue and reputation has remained unnoticed in security studies. While prior security studies have shed light on what attackers can do through exploitation of accessibility APIs, our paper aims to provide software engineers with a tool to detect and eliminate vulnerabilities due to over-access in their apps.

8 CONCLUSION

Assistive Technologies help disabled users have equal access to mobile apps by providing alternative modes of interaction. An inconsistency between different interaction modes may result in both under-access as well as over-access problems. The former has been extensively studied in prior works, concerning inaccessible data and functionality. However, in this study, we presented the latter and discussed the threats of overly accessible elements, enabling an assistive-technology user to get access to app content or functionality that is not available otherwise. We also studied the characteristics of overly accessible elements and proposed OverSight to automatically detect them in mobile apps with high accuracy. Our evaluation reveals overly accessible elements have severe impacts on both disabled users and developers. They can degrade app accessibility, endanger app security, and put developers' reputation and revenue at stake. To avoid such issues, in the future, we investigate the application of automatic program repair techniques in resolving the over-access problem in mobile apps.

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 *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[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, 730–742.

[3] 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.

[4] Yair Amit. 2018. Accessibility Clickjacking–Android Malware Evolution.(2016).

[5] Android. 2020. About Switch Access for Android. https://support.google.com/accessibility/android/answer/6122836?hl=en.

[6] Android. 2020. Accessibility Scanner - Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US.

[7] Android. 2020. *Espresso : Android Developers*. Google. Retrieved August 20, 2020 from https://developer.android.com/training/testing/espresso

[8] Android. 2020. *Improve your code with lint checks*. Google. Retrieved August 20, 2020 from https://developer.android.com/studio/write/lint?hl=en

[9] Android. 2022. *Android Debug Bridge*. Google. Retrieved March 15, 2022 from https://developer.android.com/studio/command-line/adb

[10] Apple. 2022. Apple Accessibility Scanner. https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html.

[11] Meditation Moments B.V. 2022. *Meditation Moments*. Retrieved March 10, 2022 from https://play.google.com/store/apps/details?id=com.meditationmoments.meditationmoments&hl=en_US&gl=US

[12] Mikey Campbell. 2021. Lock screen bypass enables access to Notes in iOS 15. Retrieved February 26, 2022 from https://appleinsider.com/articles/21/09/20/lock-screen-bypass-enables-access-to-notes-in-ios-15

[13] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.

[14] 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.

[15] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination?. In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1202–1214.

[16] Bettye Rose Connell. 1997. The principles of universal design, version 2.0. *http://www. design. ncsu. edu/cud/univ_design/princ_overview. htm* (1997).

[17] 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.

[18] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *2017 IEEE Symposium on Security and Privacy (SP)*. 1041–1057. https://doi.org/10.1109/SP.2017.39

[19] Google. 2020. *AccessibilityNodeInfo*. Retrieved March 6, 2022 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo#isVisibleToUser()

[20] Google. 2020. Get started on android with talkback - android accessibility help. https://support.google.com/accessibility/android/answer/6283677?hl=en.

[21] Google. 2022. *AccessibilityFlags*. Retrieved March 16, 2022 from https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo#attr_android:accessibilityFlags

[22] Google. 2022. *AccessibilityInteractionController.java*. Retrieved May 3, 2022 from https://android.googlesource.com/platform/frameworks/base/+/80943d8/core/java/android/view/AccessibilityInteractionController.java#680

[23] Google. 2022. *AccessibilityNodeInfo*. Retrieved March 12, 2022 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo

[24] 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*. 204–217.

[25] Jie Huang, Michael Backes, and Sven Bugiel. 2021. A11y and Privacy don't have to be mutually exclusive: Constraining Accessibility Service Misuse on Android. In *30th USENIX Security Symposium (USENIX Security 21)*. 3631–3648.

[26] IBM. 2020. *IBM Accessibility Checklist*. Retrieved September 14, 2020 from https://www.ibm.com/able/guidelines/ci162/accessibility_checklist.html

[27] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. 2014. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 103–115.

[28] KewlApps. 2022. *AppLock*. Retrieved March 10, 2022 from https://play.google.com/store/apps/details?id=com.gamemalt.applocker

[29] Swati Khandelwal. 2017. New ransomware not just encrypts your Android but also changes Pin Lock. https://thehackernews.com/2017/10/android-ransomware-pin.html

[30] Filip Koroy. 2018. Another BAD iOS 12 Passcode Bypass! 12.1/12.0.1 (Works on XS). Retrieved February 26, 2022 from https://www.youtube.com/watch?v=7CyiouCv6Kk

[31] Filip Koroy. 2018. iOS 12 Passcode Bypass! Photos & Contacts (Works on XS). Retrieved February 26, 2022 from https://www.youtube.com/watch?v=YYucGhyOjUE

[32] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2014. On Malware Leveraging the Android Accessibility Framework. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Ivan Stojmenovic, Zixue Cheng, and Song Guo (Eds.). Springer International Publishing, Cham, 512–523.

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

[34] Microsoft. 2022. *Accessibility Insights for Android*. Retrieved March 13, 2022 from https://accessibilityinsights.io/docs/en/android/overview/

[35] Matt Miller. 2017. Monetization Insights from App Professionals. https://www.data.ai/en/insights/app-monetization/app-marketers-developers-survey-2/.

[36] Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. 2019. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. (2019).

[37] OverSight. 2022. OverSight. https://github.com/seal-hub/Oversight.

[38] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1120–1136.

[39] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android.. In *NDSS*.

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

[41] 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.

[42] 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*. Association for Computing Machinery, New York, NY, USA, 11 pages. https://doi.org/10.1145/3411764.3445455

[43] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. GroundHog: An Automated Accessibility Crawler for Mobile Apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[44] 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, 41–52.

[45] Dinesh Venkatesan. 2016. "Malware may abuse androids accessibility service to bypass security enhancements.

[46] W3. 2020. *Web Content Accessibility Guidelines (WCAG) Overview*. World Wide Web Consortium. Retrieved August 20, 2020 from https://www.w3.org/WAI/standards-guidelines/wcag/

[47] W3. 2022. *Principle 1: Perceivable*. World Wide Web Consortium. Retrieved March 15, 2022 from https://www.w3.org/TR/WCAG20/#perceivable

[48] W3. 2022. *Principle 2: Operable*. World Wide Web Consortium. Retrieved March 15, 2022 from https://www.w3.org/TR/WCAG20/#operable

[49] W3. 2022. *Understanding the Four Principles of Accessibility*. World Wide Web Consortium. Retrieved March 15, 2022 from https://www.w3.org/TR/UNDERSTANDING-WCAG20/intro.html#introduction-fourprincs-head

[50] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 470–483.

[51] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for Android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*. 55–63. https://doi.org/10.1109/CNS.2016.7860470

[52] Martin Zhang. 2016. Android ransomware variant uses clickjacking to become device administrator.

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