# Lock the Door But Keep the Window Open: Extracting App-Protected Accessibility Information from Browser-Rendered Websites

Haichuan Xu
Georgia Institute of Technology
Atlanta, United States
haichuaxu@gatech.edu

Runze Zhang
Georgia Institute of Technology
Atlanta, United States
runze.zhang@gatech.edu

Mingxuan Yao
Georgia Institute of Technology
Atlanta, United States
mingxuanyao@gatech.edu

David Oygenblik
Georgia Institute of Technology
Atlanta, United States
davido@gatech.edu

Yizhi Huang
Georgia Institute of Technology
Atlanta, United States
raycloud@gatech.edu

Jeman Park†
Kyung Hee University
Seoul, Republic of Korea
jeman@khu.ac.kr

Brendan Saltaformaggio†
Georgia Institute of Technology
Atlanta, United States
brendan@ece.gatech.edu

## Abstract

The Android accessibility (a11y) service has been widely utilized by malware to abuse benign services. To prevent such abuse, developers need to secure a11y content access in both their apps and mobile websites. However, a misalignment of a11y protection mechanisms exists between them. Prior research has focused on attacking and defending a11y information embedded in native Android apps. However, our research found that a11y malware can retrieve app-protected a11y information in its mobile browser-rendered website counterpart, leaving mobile browser users more vulnerable to a11y attacks than app users. To help benign service developers vet this attack surface, we developed SOMBRA, an automated analysis pipeline to vet browser-side leakage of a11y information that is a11y-protected in apps. Using SOMBRA, we analyzed 294 benign services and found 29 of them deploy app-side a11y protection mechanisms to secure 256 views. SOMBRA discovered that 241, 402, 244, and 251 elements corresponding to their protected app-side views are a11y-exposed in their websites rendered by Chrome, Firefox, Brave, and Edge browsers, respectively. The leaked elements contain sensitive personal identifiable information. Finally, SOMBRA discovered that most developers do not adopt browser-side a11y protections because existing mechanisms either have ineffective protection or hinder the usability of their content.

## CCS Concepts

• **Security and privacy → Web application security**.

† Co-corresponding author.


ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744822

## Keywords

Android; Accessibility; Browser; Privacy

**ACM Reference Format:**
Haichuan Xu, Runze Zhang, Mingxuan Yao, David Oygenblik, Yizhi Huang, Jeman Park†, and Brendan Saltaformaggio†. 2025. Lock the Door But Keep the Window Open: Extracting App-Protected Accessibility Information from Browser-Rendered Websites. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744822

## 1 Introduction

Android's accessibility service [7], called a11y, although designed to help users better interact with their devices, has been widely utilized by malware to abuse benign services [46, 31, 34]. For developers to secure their content, they need to deploy defenses to all resources an a11y attacker can access, including both native Android apps and websites. However, an intrinsic misalignment exists between how a11y protection mechanisms are supported by native apps and browser-rendered websites. In particular, native apps can adopt stronger a11y protection mechanisms provided by Android, while websites still rely on browser-translated ARIA labels declared by developers alone.

Researchers have proposed a line of proof-of-concept (PoC) attacks [32, 44, 36, 42] abusing a11y to retrieve sensitive information from native Android apps. Malware analysis work [64] also has studied how real a11y malware can conduct in-app GUI attacks on benign services. Motivated by these attacks, Android has introduced several app-side a11y protection mechanisms to enable developers to hide their sensitive information from untrusted a11y services [5, 20]. Existing work also has proposed to use data-flow constraints [41, 35] to counteract several attacks that compromise the victim app's GUI by abusing the a11y service. However, no existing work has explored how a11y attackers can still steal sensitive information when benign service developers actively deploy these a11y protection mechanisms in their Android apps.

In our research, we discovered that an a11y attacker can retrieve app-protected a11y information in its mobile browser-rendered website counterpart due to the misalignment between native app's and website's a11y support. Specifically, Android allows native app developers to adopt both fine-grained static labels and dynamic a11y handlers to customize the a11y information exposed to on-device a11y services. On the other hand, mobile website developers can only rely on ARIA labels assigned to the elements to indicate their intended access level and announcement behavior. Since the labels are then translated by browsers before being interpreted by the Android OS, the same declaration by developers may result in different a11y output if rendered with different browsers. This leaves mobile browser users more vulnerable than app users to exposing their sensitive information to a11y attackers.

To help benign service developers vet this attack surface, we developed SOMBRA[1], an automated system to discover browser-side leakage of a11y information that is app-side a11y protected. SOMBRA first derives an app-side a11y model (§4.1) and a browser-side a11y model (§4.2) as a guide to its analysis. Given a service's Android app, SOMBRA conducts an app-side a11y-model-guided traversal (§4.3.1) to extract native fields protected by developers. SOMBRA then attributes each protected field to its app-side a11y protection mechanism (§4.3.2) by conducting static analysis. Using the traversal logic of the app, SOMBRA then guides the browser-side automation engine to discover the fields in the service's mobile website corresponding to the app-side protected components (§4.3.3). SOMBRA then extracts the embedded browser-side a11y information and checks whether any leakage has occurred. Finally, SOMBRA compares the ARIA labels declared by the website developers with app-side a11y protections declared by app developers (§4.3.4).

Using SOMBRA, we conducted a study of browser-side a11y leakage in real benign services' websites rendered by four different mobile browsers – Chrome, Firefox, Brave, and Edge. From the 294 benign services collected, SOMBRA discovered 29 services that utilized at least one a11y protection mechanism to secure 256 views in their Android apps. While matching the views to the ones rendered in their websites, SOMBRA found that 241, 402, 244, and 251 elements corresponding to their a11y-protected app-side counterparts are a11y-exposed in the four browsers, respectively (§5.2). The Firefox browser exposes more elements than the others because of its difference in a11y translation logic. For the 256 views protected on the app-side, SOMBRA found that their website developers deploy fewer browser-side a11y protection mechanisms. SOMBRA found that only 12 (4.7%) elements are hidden from all a11y services and only 48 (18.8%) elements have alternative a11y announcements. Most developers choose to declare no browser-side a11y protection because existing mechanisms either are ineffective at protecting their content or hinder the usability of their content (§5.4). The leaked a11y information on the browser side contains common sensitive personal identifiable information (PII). Among the 241 leaked elements in Chrome-rendered websites, 34.4% contain user account or credit card information and 7.5% contain user passwords (§5.3).

[1]**S**canner **O**f **M**obile **B**rowser **R**endered **A**ccessibility leakage

## 2 Background

**a11y Implementation in Android Apps.** Android's `a11yService` [7] allows developers to make their apps more accessible and usable. For each view [19] declared by developers in an app's GUI, the Android OS populates the view with `a11yNodeInfo` [6] representing its a11y properties. Whenever the view changes in the GUI, it will broadcast an `a11yEvent` [9] containing the changes of the view and its properties to the Android OS. The Android OS then redirects the `a11yEvent` to all `a11yServices` on the device that are registered and allowed to receive such a type of event. After parsing the `a11yEvents`, `a11yServices` can translate that information to other types of output to users such as audio, making the content more accessible. Similarly, `a11yServices` can also translate users' various input actions to text input or gestures on the app's GUI screen, realizing functionalities such as voice control and gesture recognition.

**a11y Implementation in Mobile Browsers.** A mobile browser's render engine is responsible for interpreting the HTML page and resolving dynamic content to show the web page to end users. Similarly, it is also responsible for translating and constructing the a11y information embedded in the HTML page to a11y constructs that are understandable and parsable by the Android OS. Specifically, a mobile browser render engine parses the Document Object Model (DOM) [52], translates ARIA labels [51], constructs view elements in the Accessibility Object Model (AOM) [33], and populates a11y node information within each view. With the constructed views interpretable by the Android OS, registered `a11yServices` can then parse `a11yEvents` generated by those views and read a11y information embedded in those views when the screen content changes.

Since each browser's render engine can have its own interpretation of the a11y labels and hierarchy, it can construct different AOMs. This is different from the native Android app's a11y support where the app's declared view hierarchy is directly interpreted by the Android OS, making the a11y events generated and their embedded a11y node information consistent across different devices. Additionally, the customizability of a11y information in mobile browsers is more limited than that in native Android apps. This is because mobile browsers rely entirely on ARIA labels declared in HTML pages to customize and render elements, while Android apps have access to multiple customizable `a11yEvent` [9], `a11yNode` [6], and `a11yDelegate` [20] methods to do so. As shown in §3.1, the inconsistency in the mobile browser's interpretation of a11y labels and its lack of customizability lead to failed a11y protection of sensitive information otherwise inaccessible to a11y attackers.

## 3 Exposing Mobile Browser Users' Sensitive Information

Due to the differences in a11y support between native Android apps and mobile browsers, the apps provide stronger a11y protections. This leaves users who access their accounts and services through mobile browsers more vulnerable to a11y attackers than users who

```
1 TextView accountNum = findViewById(R.id.accountNum);
2 // Android 14 (API Level 34)
3 if (Build.VERSION.SDK_INT >= 34) {
4     // Only accessible to a11y tools
5     accountNum.setAccessibilityDataSensitive(View.ACCESSIBILITY_DATA_SENSITIVE_YES);
6 }
```

**(a) a11yDataSensitive label protects a11y text.**

```
 1 TextView accountID = findViewById(R.id.accountID);
 2 // Bind an AccessibilityDelegate
 3 accountID.setAccessibilityDelegate(new View.AccessibilityDelegate() {
 4     @Override
 5     public void onInitializeAccessibilityNodeInfo(View host, AccessibilityNodeInfo info) {
 6         super.onInitializeAccessibilityNodeInfo(host, info);
 7         // Override and hide text
 8         info.setText("");
 9     }
10 });
```

**(b) a11yDelegate overrides and protects a11y text.**

```
1 TextView userBalance = findViewById(R.id.userBalance);
2 // Set a custom listener for accessibility events
3 userBalance.setOnPopulateAccessibilityEventListener(event -> {
4     // Override and hide text
5     event.getText().clear();
6     event.getText().add("User balance");
7 });
```

**(c) Customized a11yEvent handler protects a11y text.**

```
1 EditText password = findViewById(R.id.password);
2 // Set as password type
3 password.setInputType(android.text.InputType.TYPE_TEXT_VARIATION_PASSWORD);
4 // Mask input characters
5 password.setTransformationMethod(PasswordTransformationMethod.getInstance());
```

**(d) Password field with transformation protects a11y text.**

**Figure 1: Protections against a11y information leakage for native Android app users.**

do so in apps. Although there are fewer users of mobile browsers than users of native apps [29], they deserve the same attention and protection.

## 3.1 Leakage Types

Next, we illustrate the four types of a11y leakage against mobile browser users. App users are protected from these leakages because of app-side protection mechanisms, as illustrated in Figure 1. We provide real leakage examples SOMBRA discovered in two apps from the Google Play Store – Klarma (com.myklarnamobile) and Varo (com.varomoney.bank).

**App User Protection 1: a11yDataSensitive.** In the Klarna app, whenever a user binds a bank account to the app, the app displays a `ViewGroup` [21] for users to access the account. Within the `ViewGroup`, a standalone `TextView` [16] stores the bank account number. The `TextView`'s initialization routine [14] is set with the `a11yDataSensitive` [5] flag, as shown in Line 5 of Figure 1a. This ensures that whenever an `a11yService` not approved by the Google Play Store as an `a11yTool` [8] tries to access the view's a11y content, it will be displayed as null to protect its information.

**Browser User Leak 1: Absence of Fine-grained a11y Access Control.** The view that displays the bank account number on the Klarna website is directly focusable and visible while traversing the user profile page. Upon investigation, SOMBRA found that the HTML text field is declared with no labels that suggest its being hidden from a11y services not approved by Google. In fact, no fine-grained a11y access control label exists for website developers that only blocks untrusted `a11yServices`. As a result, SOMBRA found the view on the browser side by locating its `a11yNodeID` [12] in the top-level `ViewGroup` in the `window change event` [18]. After acquiring the view's `a11yNodeID`, SOMBRA inspected the a11y text field [11] of the view and found that the full account number is present and visible to a11y attackers.

**App User Protection 2: a11yDelegate Override.** In the Klarna app, a user can also check orders placed under the privacy and security tab on the user profile page. For each of the accounts bound, a `TextView` displays its account ID. SOMBRA found that when the view initializes, a custom `a11yDelegate` class is bound to the view, as shown in Line 3 of Figure 1b. The `a11yDelegate` class then sets the text field of the view to an empty string when it is called, as shown in Line 8 of Figure 1b. This ensures that the a11y text field of the view is never interpretable by an a11y attacker eavesdropping on the device.

**Browser User Leak 2: Absence of Element Delegate.** After locating the view showing the user account ID in the Klarna app, SOMBRA found that the view is traversable and contains the full account ID in the a11y text field of a11y events generated while focusing on the element. No labels or delegates exist for developers to achieve the same protection similar to the app-side delegates.

**App User Protection 3: Customized a11yEvent.** In the Varo app, a user can check the balance of the account in a top-level `TextView` on the user profile page. While the text field inside the view shows the aggregated numeral balance, the a11y text announcement of the field only contains the string "user balance," regardless of the actual balance. Upon investigation, SOMBRA found that the view customizes its a11y event broadcast by modifying the initialization of its `a11yNodeInfo` [14], as shown in Line 3 of Figure 1c. When the `a11yNodeInfo` initializes, the view overrides its existing text with a constant string, as shown in Lines 5 and 6 of Figure 1c, thus preventing a11y attackers from reading the sensitive information.

**Browser User Leak 3: Uncustomizable a11yEvent.** The view that displays the balance field in the Varo website is traversable by SOMBRA and contains the full numeral balance in the a11y text field. No customization is applied to the view's content.

**App User Protection 4: Password Field.** In the account login activity of the Varo app, the password box `EditText` [10] view is declared with a `textPassword` [15] flag, as shown in Line 3 of Figure 1d. The view is then applied with a customized transformation method to mask every user input character with a dot, including the most recently typed character, as shown in Line 5 of Figure 1d. This ensures that when a user types in a password, every a11y event generated from the view change contains only the masked dots. An a11y attacker eavesdropping on the field thus cannot piece together the typed-in password by concatenating the last visible characters from a sequence of a11y events.

**Browser User Leak 4: Inconsistent Password Input.** The password input box in the Varo website is focusable while traversing the main page by SOMBRA. While inputting characters in the `EditText` box, SOMBRA found that the most recently typed-in character is visible in the a11y text field of the view in window change events. Although the previously type-in characters are masked out, SOMBRA can piece together the user password by concatenating the last visible character in the a11y text fields. Upon investigation, SOMBRA found that although the input box is declared as a password type in the HTML page, the input box is applied with a JavaScript function that reveals the last typed-in character in an input event listener.

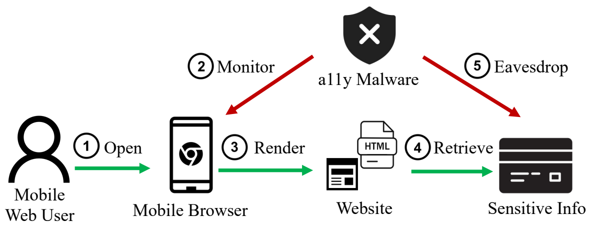



**Figure 2: An a11y malware's workflow to steal mobile browser users' sensitive information.**

## 3.2 Attack Workflow

Figure 2 shows an a11y malware's workflow to steal sensitive information from a mobile browser user. When the user opens a browser, as shown in ①, the a11y malware can monitor ther user's actions by parsing `WINDOW_STATE_CHANGE` events, as shown in ②. When the browser renders the website in ③ and the user is viewing sensitive information in ④, the malware can eavesdrop on the sensitive information by parsing `a11y events` generated by the views rendered by the unprotected browser-rendered elements.

## 3.3 Attack Prerequisites

To expose mobile browser users' sensitive information, as shown in §3.2, we assume an attacker has infected users' devices with malware that requests the a11y permission and the user has already granted the requested permissions. This is reasonable because a11y malware has been infiltrating the Google Play Store [45, 61] and can trick users into granting a11y permissions [62]. Furthermore, a11y malware remains the most popular mobile remote access trojan, targeting a wide range of benign services [64].

# 4 Pinpointing Browser-Side Leakage of a11y Information

To help developers vet a11y-exposed information for browser users that is protected for app users, we developed SOMBRA, an automated hybrid analysis pipeline that finds information exposed on mobile-browser-rendered websites but is protected in the app. Our study focused on services with both apps and websites because SOMBRA uses app-side protected a11y information as ground truth. That said, browser-side a11y leakage extends to websites that don't also have companion apps. We leave the analysis of these websites as future work since SOMBRA doesn't have ground truth for them.

The input to SOMBRA is the Android APK and the website URL of a service. After SOMBRA's automated a11y information scanning, SOMBRA outputs all elements in the mobile-browser-rendered websites that are accessible to a11y services but inaccessible in their Android app counterparts.

## 4.1 App-Side a11y Model

The first step for SOMBRA to vet the attack surface is to understand how a11y protection to native Android apps is implemented and formulate an app-side a11y protection model. We scoured the Android a11y service implementation [7] to find all mechanisms that allow app developers to modify a11y output or restrict a11y access to a11y services. Figure 3 shows the app-side a11y protection model available to app developers. Since developers can directly control the declaration and implementation of native views, they can adopt both static and dynamic a11y protection mechanisms. For static a11y protection, developers can declare `a11yDataSensitive` label to views. During app runtime, Android interprets this protection label and only broadcasts a11y information within the protected view to Google-approved a11y services. For dynamic a11y protection, developers can execute a11y output modification logic during runtime while views are constructed. Specifically, developers can modify the `a11yEvent` a view broadcasts, modify the static `a11yNodeInfo` property of a view, and assign an `a11yDelegate` to take over the a11y information broadcast. Given the app-side protection model, SOMBRA can further discover and attribute app-side protected information during the app traversal (see §4.3).

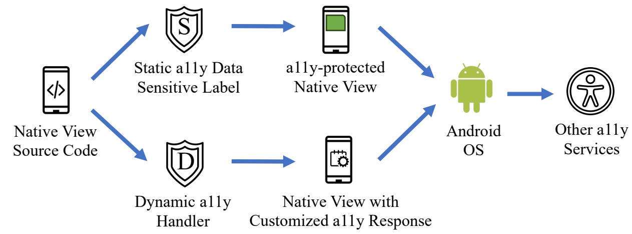



**Figure 3: App-side a11y protection model. Android native views can adopt both static protection labels and dynamic a11y modification handlers. They are directly interpreted by the Android OS.**

## 4.2 Browser-Side a11y Model

Next, to discover browser-side a11y leakage, SOMBRA needs to understand how a11y protection to browser-rendered websites is implemented in the web a11y standard. We studied the web a11y standard guideline [50] to extract all ARIA mechanisms that allow developers to change the a11y output or restrict a11y access to their website elements. Figure 4 shows the browser-side a11y protection model available to developers. With a given rendered web page, the only a11y-related fields in the DOM are static ARIA labels whether they are declared statically or assigned with JavaScript dynamically. In specific, the `aria-hidden` label allows developers to declare an element should be hidden from all a11y services. Alternative ARIA labels that allow developers to change the default a11y announcements of their fields include `aria-label`, `aria-labelledby`, `aria-describedby`, etc. However, since developers have no control over how browsers translate their ARIA labels, the actual exposed a11y information accessible to the Android OS in the AOM depends on each browser's implementation of a11y parsing logic. Given this extra layer of a11y translation, the same element declared by developers, if rendered with different browsers, may result in different a11y properties in views interpreted by Android and broadcast to a11y services. With the browser-side a11y protection model, SOMBRA is then able to extract exposed a11y information (see §4.3).

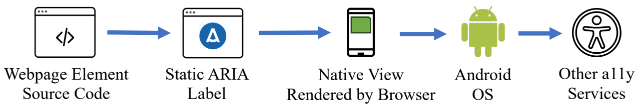



**Figure 4: Browser-side a11y protection model. Web page DOM elements can only contain static ARIA labels. The elements with labels need to be translated by browsers and then interpreted by the Android OS.**

## 4.3 Finding the Mismatch Between App-Side and Browser-Side a11y Protection

Given the misalignment of a11y protection between native Android app and browser-rendered web page components, SOMBRA next analyzes the Android app and mobile website of a given benign service and finds a11y leakage on the browser side.

*4.3.1 a11y-Model-Guided App-Side Analysis.* Pinpointing protected a11y information in the app is challenging because app developers use complex layered structures to organize the views shown to users in the frontend GUI. To reveal an app's underlying a11y information, a system should exhaustively traverse a given user GUI screen together with all redirections and child screens, as well as explore layered view structures within groups of views. To achieve this, SOMBRA adopts a customized depth-first strategy to guide the discovery and hierarchy breakdown of a11y information within the GUI.

After the manual page setup (details in §5.1), SOMBRA initializes the retrieval of a11y information from the user profile page. Algorithm 1 shows SOMBRA's traversal strategy for exhaustively recording the a11y information of GUI elements in the user account page. When the page is first initialized, SOMBRA captures the `TYPE_WINDOW_STATE_CHANGED` a11y event broadcast by the app that indicates the start of a new GUI screen, as shown in Line 3. SOMBRA then retrieves the source node (represented as an *a11yNodeInfo*) of the a11y event and marks it as the root node of the user account page. Given the root node, SOMBRA adopts a depth-first pre-order traversal to visit all descendants of the node. For each node visited, SOMBRA records the a11y text embedded and broadcast by the view, together with its properties such as `viewType` and `a11yDescription`, as shown in Lines 8 and 9. When encountering clickable elements that can redirect to a new GUI screen, such as a `Button` or `ViewGroup`, SOMBRA prioritizes the traversal of the new window state triggered by clicking the element, as shown in Lines 10 and 11. SOMBRA then captures the new a11y window state change event and continues the traversal. To keep a record of the layout hierarchy of the current page, given each GUI element, SOMBRA records all children of the node and pushes them into a stack for future processing, as shown in Lines 14-16. When all nodes within the current screen are traversed, SOMBRA issues `global_action_back` a11y command to navigate back to the previous screen and continues the traversal, as shown in Lines 18-20. Finally, the traversal ends when no elements remain in the node stack.

After the app traversal of a11y information concludes, SOMBRA records all a11y nodes with null text properties or empty strings.

**Algorithm 1:** SOMBRA's depth-first pre-order traversal of app-side a11y node information.

```
1  nodeStack = ∅;
2  Function onA11yEvent(event):
       // Capture the initial a11y event when navigating to a page
3      if event.type == TYPE_WINDOW_STATE_CHANGED then
           // Extract the top-level source node of the event
4          node = event.getSource();
5          nodeStack.push(node);
           // Depth-first traversal of the children of the a11y
              nodes
6          while !nodeStack.isEmpty() do
7              node = nodeStack.pop();
               // Record the a11y node's embedded text and type
8              record(node.text, node.viewType,
9              node.a11yDescription);
10             if node.isClickable() then
                   // Click buttons, expandable views to navigate
                      to new screen change
11                 node.actionClick();
12             end
13             else
14                 for childNode ∈ node.getChildNodes() do
15                     nodeStack.push(childNode);
16                 end
17             end
               // Traverse back to the previous screen when no
                  new nodes exist on the current GUI screen
18             if CurScreenNodeCount == 0 then
19                 global_action_back();
20             end
21         end
           // End the traversal
22         return;
23     end
24 end
```

We designed SOMBRA to ignore constant strings in a11y nodes that are different than the nodes' visible text. That said, SOMBRA will miss developers using constant strings to protect sensitive fields (e.g., hiding a user balance amount with the string "user balance"). SOMBRA users could enable this but will introduce higher false positives caused by developers' deliberate definition of different a11y node text (e.g. describing a "search" button as "search this app"). SOMBRA excludes an a11y node when it is a view type that originally shouldn't have a text property such as `imageView`, `layouts`, `switches`, `toggles`, `scrollView`, etc. Now, SOMBRA has a collection of elements whose a11y information is intended to be hidden from unwanted a11y services by app developers. This collection indicates the app-side a11y protected GUI elements by app developers and will be used to compare with the same elements on the browser-rendered website elements.

*4.3.2 Attributing App-Side Protection Mechanisms.* Since each type of browser-side leakage is caused by the inefficacy of providing a strong app-side a11y protection mechanism, SOMBRA first examines the app-side protected view and finds its implemented app-side protection mechanism. However, attributing the app-side protection mechanism given a dynamically captured a11y event is challenging. This is because multiple instances of the same view type can be instantiated by developers in the app, and multiple mechanisms, including both static XML declaration and dynamic view handler routine, can be used by developers to define and customize the a11y behaviors of a view type. To accurately pinpoint all a11y protection mechanisms declared by developers, a system must explore and capture all initialization characteristics involving a view. To achieve this, SOMBRA combines static a11y label scanning with a11y label data-flow analysis to resolve all a11y protection mechanisms used by a view.

Given an `a11yNodeInfo` property of a dynamically captured view that hides its a11y information, as discovered in §4.3.1, SOMBRA first captures its a11y node resource ID, as this ID indicates the static view type declared by developers. SOMBRA then searches all static view declarations in the APK (all XML files in res/layout) to pinpoint the view properties that match the same resource ID. SOMBRA then extracts static a11y labels declared by developers, in particular the `a11yDataSensitive` label that makes it inaccessible to untrusted a11y services. Then, to determine all dynamic a11y customization made to the view, SOMBRA searches for all view initialization routines in Activities. Under all functions that allow customization to a view component such as `setContentView`, `inflate`, etc. SOMBRA first taints the view data structure with the same Android ID that matches the resource ID found in dynamic a11y node capture. SOMBRA then propagates the taint and marks when it determines the tainted tag appears in any a11y label modification function such as `setA11yDelegate`, `seta11yNodeText`, etc. Given all a11y label customization routines specific to the view, SOMBRA finally maps the customization method to the four a11y protection routines discussed in §3.1.

*4.3.3 a11y-Model-Guided Browser-Side Analysis.* After obtaining the GUI elements in the app that are a11y-protected, SOMBRA next finds the same elements on the browser-rendered website and examines whether they are unprotected and contain a11y text information. However, locating the same GUI elements on the mobile browser-rendered website is challenging. A service's website and app, although intended to convey the same information to users, usually are developed by two teams of developers with different content layouts. Even for hybrid services whose apps are translated versions of the website, their GUI element hierarchy still differs between the two versions. Additionally, since each mobile browser implements its own interpretation of the website layout and adopts its own translation of a11y ARIA labels, the sequence for an a11y service to traverse the same website differs depending on the browser that renders it.

However, we found during our research that in addition to the DOM, the AOM browsers created while rendering a web page contain crucial a11y node structural information and that neighboring element properties can help adjust and calibrate the traversal to adhere to the app-side traversal sequence. To accurately pinpoint the same elements on a service website rendered by different browsers, SOMBRA adopts a dynamic app-side-guided AOM traversal strategy. After the manual setup (see §5.1), SOMBRA starts the traversal on the mobile-browser-rendered website in search of the corresponding app-side a11y-protected elements.

As shown in Algorithm 2, SOMBRA first acquires the AOM created by the browser at the initial state of the current page. For each of the traversal sequences on the app side that led to the

**Algorithm 2:** SOMBRA 's browser-side traversal for identifying elements corresponding to app-side a11y-protected Views. SOMBRA matches element types and aria labels in addition to app-side traversal sequence when browser layout is different than the app layout.

```
stepIdx = 0;
Function startTraversal(appA11ySequence):
    // Start from the root a11y node in AOM
    currentRoot = AOM.getRootA11yNode();
    while stepIdx < appA11ySequence.length do
        targetStep = appA11ySequence[stepIdx];
        matchingNode = null;
        // Match app-side traversal sequence
        nodeStack = ∅;
        nodeStack.push(currentRoot);
        while !nodeStack.isEmpty() do
            node = nodeStack.pop();
            // Match app-side element with text and type
            if node.matches(targetStep.text, targetStep.type)
             then
                matchingNode = node;
                break
            end
            for child ∈ node.getChildNodes() do
                nodeStack.push(child);
            end
        end
        // When browser layout differs from app layout
        if matchingNode == null then
            // Global page search with type and label
            nodeStack = ∅;
            candidates = [];
            nodeStack.push(currentRoot);
            while !nodeStack.isEmpty() do
                node = nodeStack.pop();
                if typeMatch(targetStep.type, node.role) then
                    candidates.push(node);
                end
                for child ∈ node.getChildNodes() do
                    nodeStack.push(child);
                end
            end
            // Match aria-label
            for node ∈ candidates do
                if node.ariaLabel ==
                  targetStep.contentDescription then
                    matchingNode = node;
                    break
                end
            end
        end
        // Click buttons, expandable views to navigate to new
           screen
        if targetStep.isClickable then
            matchingNode.actionClick();
            currentRoot = AOM.getRootA11yNode();
        end
        stepIdx++;
    end
    return matchingNode;
end
```

discovery of an app-side a11y protected element, SOMBRA matches the button and expandable view clicking sequences and searches for the element within the final landing page. If no clickable elements are present in the app-side traversal, SOMBRA matches the child

**Table 1: Element Type Rules SOMBRA Adopt To Match Web Page Elements To Their Android App Element Counterparts And Whether They Contain a11y Text.**

| HTML Element | Android Element | a11y Text |
|---|---|---|
| `<div>` | FrameLayout | ✗ |
| `<span>` | TextView | ✓ |
| `<p>` | TextView | ✓ |
| `<a>` | TextView | ✓ |
| `<h1>, <h2>, etc.` | TextView | ✓ |
| `<button>` | Button | ✓ |
| `<img>` | ImageView | ✗ |
| `<input>`$_{\texttt{text}}$ | EditText | ✓ |
| `<input>`$_{\texttt{button}}$ | Button | ✓ |
| `<input>`$_{\texttt{checkbox}}$ | CheckBox | ✗ |
| `<ul>, <ol>` | ListView | ✓ |
| `<select>` | Spinner | ✗ |

hierarchy of the a11y-protected view to the DOM hierarchy on the web page, as shown in Lines 7-18.

When the browser page layout differs from the app layout and no element is found according to the app-side traversal sequence, SOMBRA conducts a global page search to locate the element, as shown in Line 19. SOMBRA first narrows down the element candidates by finding all website elements that correspond to the app-side views according to Table 1, as shown in Lines 20-30 of Algorithm 2. For example, when an app-side a11y-protected `TextView` element's traversal sequence matches a `<span>`, `<p>`, `<a>`, or `<h1>`, etc., SOMBRA confirms that it is a valid candidate. Given these elements, SOMBRA further attributes the ARIA label to the app-side content description to infer the matching role of the element in Lines 32-37.

When a clickable element is matched, SOMBRA sends an *a11y action* to click it and continues the traversal on the updated page and its updated AOM, as shown in Lines 39-42. When a match of the app-side a11y-protected element is found in the DOM, SOMBRA then queries and records the *a11y role*, *a11y states*, and *a11y properties* fields within the element's AOM node and its *ARIA labels* declared in the DOM. SOMBRA finally compares the a11y text information accessible to any a11y services in the *a11y properties* fields with the a11y text field in the app-side element. If the browser-side information is not null or an empty string, SOMBRA confirms that the browser-side element leaks a11y information otherwise protected on the app side.

Since the discovered browser-side a11y information visibility can be caused by four different types of browser leakage as discussed in §3.1, SOMBRA next attributes the reason for each found browser-side a11y leakage.

*4.3.4 Attributing Browser-Side Leakage.* Since the development team of a service's app and website can be different, SOMBRA further needs to attribute the reason for the found browser-side leakage. If the app-side a11y-protected information is not declared with ARIA protection labels by website developers, the a11y leakage is caused by the deliberate inconsistency between a service's app and website development team. If the same app-side a11y-protected information is also declared with ARIA protection labels, the a11y leakage on the browser-side is inherent to the browser's translation of a11y information according to web a11y standards and cannot be avoided by website developers alone.

SOMBRA examines the *ARIA labels* declared in the DOM that correspond to the AOM element with the *a11y properties* leakage. If the element in the DOM does not contain either the `aria-hidden` label or alternative ARIA labels that can change an element's a11y announcement, as discussed in the browser-side a11y model §4.2, it is the inconsistency between the website development team and the app development team that caused the a11y leakage. Otherwise, if such a label is present, the browser's render engine reveals the a11y information to a11y services according to the web a11y standard and causes the a11y information leakage.

SOMBRA now has finished vetting benign services' a11y information leaked in mobile browser-rendered websites but protected in their native apps. We discuss the developer's defense as well as mitigation to the attack surface in §7.

## 5 Evaluation

We deployed SOMBRA to vet a11y-exposed information for browser users that are protected for app users in real services. App-side a11y information traversal is implemented in Java (1.1K lines) leveraging the Android a11y service [7]. Browser-side AOM retrieval and traversal is implemented in Python (0.5K lines) leveraging Appium [26], the SOTA mobile UI automation tool. Extracted a11y field categorization into common PII types is queried through Google Cloud natural language APIs [38]. Dynamic analysis of the applications and mobile browser-rendered websites is hosted on a Google Pixel 5 device running Android 14.

### 5.1 Dataset & Experiment Setup

**Dataset.** To collect a benign services dataset, we queried AppBrain [25], a state-of-the-art (SOTA) Android market intelligence service, for the top-150 free finance, shopping, and transportation Android applications in the U.S. This is selected according to the most abused categories of apps from the most recent Android a11y malware study [64]. For each application, we collected both the Android package name and its website URL (if it exists). To acquire their Android applications, we downloaded their most recent versions from AndroZoo [23], the SOTA Android application dataset used in top-tier research, resulting in a collection of 294 APKs, excluding duplicates, and have their service websites. We selected Chrome, Firefox, Brave, and Edge to render the services' websites.

**Experiment Setup.** We created test credentials for 226 services that support signing up with email/phone numbers. We used pre-existing personal accounts for 23 services. We asked for and received test credentials from four services that require real accounts that we did not have personal accounts for. We failed to acquire valid test accounts for 41 services. For each service, we logged into the app and browser web pages, filled in personal information fields, and bound one Chase credit card and one Chase banking account when possible to mimic normal users' sensitive information. We manually left the app and the website at the user account page for SOMBRA to start the traversal because this page contains the most sensitive information. SOMBRA users can pick any page to start the traversal. For services for which we failed to obtain valid login credentials, we manually left them at the login page for SOMBRA to start the traversal. This conforms to the experiment setup procedures from prior work [57].

## 5.2 Browser-Side a11y Leakage in Real Benign Services

Table 2 shows SOMBRA's findings of browser-side a11y information leakage in the Chrome, Firefox, Brave, and Edge browsers in real-world benign services. We manually verified and confirmed these results. We conducted additional validation of SOMBRA that shows SOMBRA can detect app-side a11y protection and match browser-side elements with low false positives and false negatives (shown in Appendix A due to space constraints).

As shown in the Total Row of Table 2, of 294 benign hybrid services we collected, SOMBRA discovered a total of 29 (9.9%) services that deploy at least one type of app-side a11y protection mechanisms in their Android apps. Upon further investigation, we extracted their Android app manifest information and found that all 29 services have updated their apps to target Android 14, which provides enhanced a11y protection mechanisms such as `a11yDataSensitive` declarations to help protect apps from non-Google-approved a11y services. We expect that more developers will gradually update and adopt the new app-side a11y protection mechanisms.

Columns 1 and 2 of Table 2 show the benign services' category and package name. Out of the 29 apps that adopt app-side a11y protection, nine (31.0%) are finance apps, seven (24.1%) are transportation apps, while the remaining 13 (44.8%) apps are shopping apps.

Columns 3 and 4 of Table 2 show the number of view elements that are a11y protected discovered by SOMBRA in the app-side dynamic analysis and the number of a11y protection types they adopted, respectively. As shown in the Total Row of Columns 3 and 4, a total of 256 views are a11y-protected, with an average of 8.8 views protected in each app. The number of views protected within each app varies significantly across different apps. For the largest numbers of app-side a11y-protected views, com.route.app contains 42 with two types of protection, namely `a11yDataSensitive` and customized `a11yEvent` handler. com.shopmium also contained 40 views protected by `a11yDataSensitive` fields. For the least number of a11y-protected views, com.acehardware.rewards only protects one view, which is the account login password `EditText` view, using the a11y password protection. As shown in Column 4, the majority of apps (18) adopt only one type of a11y protection, 10 apps adopt two different types of a11y protection, and only one app (com.puma.ecom.app) adopts three types of protection (`a11yDataSensitive` label, customized `a11yEvent` handler, and a11y password protection).

Columns 5 - 12 of Table 2 show the number of a11y-exposed elements discovered by SOMBRA in the benign services' websites rendered by four different mobile browsers (Chrome, Firefox, Brave, and Edge). The detailed exposed content type is discussed in §5.3). As shown in the Total Row of Table 2, the 29 benign services' websites rendered in Chrome, Firefox, Brave, and Edge mobile browsers exposed a total of 499, 893, 504, and 510 elements that correspond to the app-side a11y-protected views; that is 1.9x, 3.5x, 2.0x, and 2.0x more than their app-side counterparts. Upon further investigation, we found that the reason more elements are exposed on the browser side is that all elements within a `ViewGroup` element, including expandable views rendered in the browsers, inherit the a11y text information in the AOM. This means that for a single view in an Android app, its parent or child element in the browser-rendered element should there be any, all contain the exposed a11y information. To mitigate this duplication and avoid over-counting, we eliminated the duplicates and showed the unique elements exposed on the browser side in Columns 6, 8, 10, and 12. As shown in these columns, a total of 241, 402, 244, and 251 unique elements corresponding to the app-side protected views are exposed in the Chrome, Firefox, Brave, and Edge browsers, respectively. Averaging across all benign services, they exposed an average of 8.3, 13.9, 8.4, and 8.7 elements per service.

While rendering the same website, we found that the Firefox browser in particular contains more a11y-exposed views than the other three browsers. We studied the rendering logic and found that the Chrome, Brave, and Edge browsers have similar rendering logic because they all adopt the same Blink render engine, which is responsible for interpreting the website DOM structure and translating the ARIA labels into the AOM. The Firefox browser, on the other hand, adopts the Gecko render engine, which differs in AOM construction logic. Specifically, while both render engines implement the same `aria-hidden` label translation logic by excluding it from the AOM tree, they treat elements marked with alternative labels, such as `aria-describedby` and `aria-labelledby`, differently. For each DOM element that contains an alternative ARIA label, the Chrome, Brave, and Edge browsers utilizing the Blink render engine only expose the a11y text information in the alternative element, while the Firefox browser that utilizes the Gecko render engine exposes the a11y information in both the original and the alternative element.

**Takeaway.** SOMBRA identified a total of 256 app-side a11y-protected views across 29 benign services' Android apps. They adopted at least one and at most three types of Android a11y-protection mechanisms. While examining the benign services' websites, SOMBRA discovered that 241, 402, 244, and 251 elements matching their a11y-protected app-side counterparts are a11y-exposed in the Chrome, Firefox, Brave, and Edge browsers with 8.3, 13.9, 8.4, and 8.7 exposed elements per service. The Firefox browser in particular exposes more elements than the other three browsers because of the difference in adopted render engines. Specifically, the Gecko render engine adopted by Firefox has different interpretation logic for elements declared with alternative ARIA labels such as `aria-describedby` and `aria-labelledby`.

## 5.3 Security Impact

We categorized the types of information exposed to a11y services on benign services' websites that are a11y-protected in their apps. The exposed information causes compromised credit card/account/password and PII stalking to browser users. For each a11y-exposed field, SOMBRA extracts its a11y label and/or hint

**Table 2: SOMBRA's Discovered a11y Leakage In Benign Services' Websites Rendered In Chrome, Firefox, Brave, And Edge Browsers.**

| Category | Package Name | App-P.[1] | # P.[2] | Chrome | | Firefox | | Brave | | Edge | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Ex.[3] | w/o Dup.[4] | Ex. | w/o Dup. | Ex. | w/o Dup. | Ex. | w/o Dup. |
| Finance | com.varomoney.bank | 4 | 2 | 7 | 3 | 11 | 6 | 7 | 3 | 7 | 3 |
| Finance | com.DailyPay.DailyPay | 5 | 1 | 12 | 7 | 21 | 12 | 12 | 7 | 12 | 7 |
| Finance | com.syf.mysynchrony | 6 | 1 | 9 | 6 | 17 | 8 | 12 | 8 | 15 | 8 |
| Finance | com.usaa.mobile.android.usaa | 5 | 1 | 5 | 3 | 9 | 5 | 5 | 3 | 5 | 3 |
| Finance | com.propel.ebenefits | 3 | 1 | 3 | 2 | 6 | 5 | 3 | 2 | 3 | 2 |
| Finance | com.meetcleo.cleo | 2 | 1 | 10 | 7 | 25 | 19 | 10 | 7 | 16 | 11 |
| Finance | com.intuit.turbotax.mobile | 6 | 1 | 17 | 12 | 30 | 22 | 17 | 12 | 17 | 12 |
| Finance | com.squareup.cash | 4 | 1 | 6 | 3 | 15 | 10 | 9 | 6 | 6 | 3 |
| Finance | io.metamask | 7 | 2 | 9 | 7 | 16 | 13 | 9 | 7 | 9 | 7 |
| Transportation | com.yandex.yango | 2 | 1 | 8 | 3 | 13 | 8 | 8 | 3 | 8 | 3 |
| Transportation | com.coulombtech | 7 | 1 | 22 | 13 | 53 | 39 | 22 | 13 | 22 | 13 |
| Transportation | com.trailbehind.android.gaiagps.pro | 6 | 1 | 8 | 8 | 19 | 9 | 8 | 8 | 10 | 8 |
| Transportation | org.rajman.neshan.traffic.tehran.navigat | 10 | 1 | 15 | 8 | 21 | 12 | 15 | 8 | 15 | 8 |
| Transportation | net.sharewire.parkmobilev2 | 5 | 2 | 7 | 4 | 17 | 9 | 7 | 4 | 11 | 4 |
| Transportation | com.xatori.Plugshare | 3 | 2 | 7 | 5 | 12 | 8 | 7 | 5 | 7 | 5 |
| Transportation | com.ventrachicago.riderapp | 11 | 1 | 14 | 9 | 18 | 10 | 14 | 9 | 14 | 9 |
| Shopping | com.affirm.central | 3 | 1 | 3 | 2 | 3 | 2 | 3 | 2 | 3 | 2 |
| Shopping | com.puma.ecom.app | 8 | 3 | 26 | 17 | 42 | 28 | 26 | 17 | 28 | 17 |
| Shopping | com.route.app | 42 | 2 | 93 | 16 | 179 | 28 | 87 | 16 | 78 | 16 |
| Shopping | com.belk.android.belk | 12 | 2 | 11 | 7 | 16 | 9 | 11 | 7 | 11 | 7 |
| Shopping | com.dollargeneral.android | 8 | 1 | 18 | 12 | 27 | 21 | 16 | 8 | 18 | 12 |
| Shopping | com.myklarnamobile | 4 | 2 | 8 | 7 | 10 | 7 | 8 | 7 | 8 | 7 |
| Shopping | com.acehardware.rewards | 1 | 1 | 5 | 2 | 6 | 2 | 5 | 2 | 7 | 3 |
| Shopping | com.sneakerhotsapm.app | 30 | 2 | 43 | 27 | 77 | 40 | 43 | 27 | 43 | 27 |
| Shopping | com.cvs.launchers.cvs | 2 | 2 | 6 | 4 | 8 | 6 | 6 | 4 | 6 | 4 |
| Shopping | com.biglotsltds.biglotsam | 5 | 1 | 9 | 5 | 9 | 5 | 9 | 5 | 9 | 5 |
| Shopping | com.adidas.confirmed.app | 9 | 2 | 21 | 16 | 44 | 20 | 21 | 16 | 25 | 19 |
| Shopping | com.shopmium | 40 | 1 | 79 | 15 | 132 | 23 | 86 | 17 | 81 | 15 |
| Shopping | com.einnovation.temu | 6 | 1 | 18 | 11 | 37 | 16 | 18 | 11 | 16 | 11 |
| **Total** | 29 | 256 | 41 | 499 | 241 | 893 | 402 | 504 | 244 | 510 | 251 |

1: Number of app-side a11y-protected views. 2: Number of types of Android native a11y protection mechanisms.
3: Number of elements exposed with a11y information on the browser-rendered websites.
4: Number of unique elements exposed with a11y information, eliminating duplicates caused by `ViewGroup` handlers.

text fields and uses Google Cloud natural language APIs [38] to categorize them into common PII types.

Table 3 shows the extracted a11y information category exposed in Chrome-rendered websites while protected in their Android app-side counterparts. Column 1 shows the top 10 benign services with the most a11y-protected views in their apps. Column 2 shows unique elements protected in their apps but exposed in Chrome-rendered websites. As shown in the Total row of Table 3, a total of 241 elements are exposed out of the 29 apps that adopt app-side a11y protection mechanisms, an average of 8.3 elements per app.

Columns 3 - 7 show the a11y field types of the exposed elements such as passwords, account or credit card, key or identifiers, address or contact, etc. As shown in Column 3, a total of 18 password fields are a11y-exposed in the Chrome-rendered websites. The *puma* app, which adopts app-side a11y password protection (see §3.1), does not contain any browser-side protections, making the passwords accessible to a11y attackers eavesdropping on its mobile website. A total of 83 (34.4%) elements expose user account or credit card information fields in the Chrome-rendered websites. As shown in Columns 5 and 6, a total of 29 elements contain keys or identifiers such as tokens, wallet IDs, etc., while 40 elements expose address or personal contact information. In-depth examples of browser-side a11y leakage that causes compromised financial accounts and passwords to browser users are further illustrated in §6.

**Takeaway.** SOMBRA uncovered a total of 241 elements with exposed a11y information in Chrome-rendered services' websites. While their corresponding app-side views are a11y protected, the browser-side elements leak common sensitive PII information such

**Table 3: App-side Protected a11y Information Leakage Category In Top Services' Websites Rendered With Chrome.**

| Name | Chrome Ex. Elements[1] | Psw. | Act. / Card | Key / Ident. | Addr. / Contact | Others |
|---|---|---|---|---|---|---|
| route | 16 | 0 | 4 | 2 | 5 | 5 |
| shopmium | 15 | 1 | 5 | 0 | 0 | 9 |
| sneakerapm | 27 | 1 | 11 | 0 | 0 | 15 |
| belk | 7 | 0 | 4 | 1 | 0 | 2 |
| ventra | 9 | 0 | 0 | 3 | 0 | 6 |
| neshan | 8 | 1 | 4 | 1 | 0 | 2 |
| confirmed | 16 | 2 | 3 | 2 | 7 | 2 |
| puma | 17 | 1 | 4 | 0 | 4 | 8 |
| dollar general | 12 | 2 | 6 | 2 | 0 | 2 |
| metamask | 7 | 1 | 3 | 0 | 0 | 3 |
| Others | 107 | 9 | 39 | 18 | 24 | 17 |
| **Total** | 241 | 18 | 83 | 29 | 40 | 71 |

1: Number of unique elements protected in app but exposed in Chrome-rendered websites.

**Table 4: Developers' App-side And Browser-side a11y Information Protection Adoption Types.**

| Name | App-Side | | Browser-Side | | |
|---|---|---|---|---|---|
| | # Views | P. Types[1] | ARIA Hidden | ARIA Change | No Protection |
| route | 42 | ①, ③ | 0 | 4 | 38 |
| shopmium | 40 | ① | 0 | 0 | 40 |
| sneakerapm | 30 | ①, ② | 2 | 11 | 17 |
| belk | 12 | ②, ③ | 0 | 3 | 9 |
| ventra | 11 | ② | 3 | 0 | 8 |
| neshan | 10 | ① | 0 | 2 | 8 |
| confirmed | 9 | ①, ② | 0 | 0 | 9 |
| puma | 8 | ①, ②, ④ | 0 | 0 | 8 |
| dollar general | 8 | ① | 0 | 5 | 3 |
| metamask | 7 | ①, ② | 0 | 0 | 7 |
| Others | 79 | – | 7 | 23 | 49 |
| **Total** | 256 | – | 12 | 48 | 196 |

1: Android app-side a11y protection types. ①: a11yDataSensitive label, ②: custom a11yEvent, ③: custom a11yDelegate, ④: password protection.

as account or credit card, keys or identifiers, address or contact, and passwords. This exposure causes compromised financial accounts/passwords and PII stalking to browser users. Among the 241 leaked elements, 34.4% contain user account or credit card information, while 7.5% contain user account passwords.

## 5.4 Developers' App-Side and Browser-Side a11y Protection Adoption Comparison

With the extracted a11y information leakage in the browser-rendered websites, SOMBRA next compares the a11y protection adopted by the app and website developers. Table 4 shows the ARIA protection labels declared in benign services' websites corresponding to each app-side a11y-protected view.

Column 1 of Table 4 shows the 10 benign services that have the most views protected against a11y attackers in their Android apps. Columns 2 and 3 show the number of a11y-protected views in these apps, as well as the types of Android a11y protection mechanisms adopted in these views. As shown in the top row of Columns 2 and 3, the *route* app protected 42 of its views, which is the most among all 29 apps that adopted app-side a11y protection. The *route* app used two types of Android a11y-protection mechanism, declaring `a11yDataSensitive` labels and assigning customized `a11yDelegate` to views to alter their a11y exposure. The next apps that protected the most app-side views are *shopmium* and *sneakerapm*, containing 40 and 30, respectively. However, *shopmium* only utilized the `a11yDataSensitive` label to wrap all 40 views, while *sneakerapm* utilized both the label and customized `a11yEvent` handlers. Looking at Column 3, eight of the top 10 apps (80%) adopted the `a11yDataSensitive` label in Android 14 to protect them from being accessed by non-Google-approved a11y services. Four of the top 10 apps generate customized `a11yEvent` for the protected views, while two assign customized `a11yDelegate`. The puma app also protects its login password field with a11y-password protection §3.1.

While the benign services' app developers have the aforementioned methods to protect their app-side a11y information, the misalignment between app-side and browser-side a11y protection limits the ways to protect their website a11y information. Columns 4 - 6 show the ARIA labels declared by benign services' website developers. Only one browser-side element is counted for each matching app-side a11y-protect view to avoid duplication.

SOMBRA found from the DOM structures that only 12 out of 256 (4.7%) of app-side a11y-protected views are completely hidden on the browser side (Chrome, Firefox, Brave, and Edge all exclude `aria-hidden` elements from the AOM, thus making them inaccessible to any a11y services). Among the top-10 services that protected the most app-side views, only the *sneakerapm* app and the *ventra* app protected a total of five elements with `aria-hidden` label. Although the label provides the strongest protection on the browser side, it completely disallows any a11y services to access them, thus hindering the usability of the website for users of benign a11y utility apps. The inability to fine-grain the a11y access level to different a11y services makes this mechanism impractical for developers to adopt.

Column 5 shows the number of elements that are declared with `aria-label`, `aria-labelledby`, and `aria-describedby` labels to change the a11y field exposed and announced to a11y services. Looking at the Total row, of the 256 app-side protected views, 48 (18.8%) contain the label to alter the browser-side elements' a11y response. However, as discussed in §4.2, although the website developers can alter the default ARIA announcement, it is up to the browser's translation and interpretation to determine the actual exposed a11y information to the AOM and subsequently to the Android a11y services. As shown in §5.2, the Chrome, Firefox, Brave, and Edge browsers all still include the original and altered a11y information in the AOM, making them visible to any a11y services. Although some (18.8% elements) developers adopt the alternative labels, browsers' interpretation according to the website a11y model renders them ineffective at limiting the a11y exposure to a11y services.

Column 6 of Table 4 shows the number of elements that are completely free of any ARIA labels. As shown in the Total row, 196 (76.6%) of 256 elements protected in their app-side counterparts have no ARIA protections declared on their websites. The *shopmium* app, although having 40 app-side protected views, has no a11y protection for any of them on the browser-side. Similar behavior also exists in other apps such as *confirmed*, *puma*, and *metamask*. Not declaring ARIA protections ensures the usability of the website elements to all a11y services but at the same time makes them vulnerable to a11y malware.

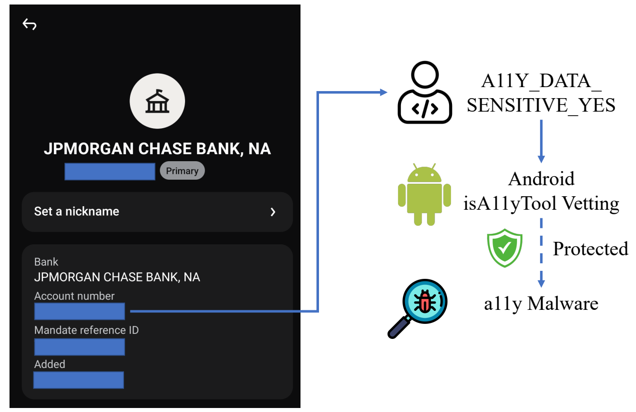
**Takeaway.** While SOMBRA discovered 256 a11y-protected views declared by Android app developers, few website developers declare ARIA labels to protect their browser-side a11y information. Specifically, 12 (4.7%) of the 256 elements declare `aria-hidden` labels to hide them from all a11y services. Since this protection makes the a11y content inaccessible to all a11y services, it hinders the usability of benign a11y utility apps. While 48 (18.8%) of the elements are declared with alternative ARIA announcement labels, browsers still expose the a11y information in the browser-side AOM and subsequently make them visible to a11y services due to the existing web a11y standard, making the protection ineffective. Most developers (196 / 256 elements) adopt no browser-side a11y protections, ensuring the functional usability of their content.

## 6 Case Studies

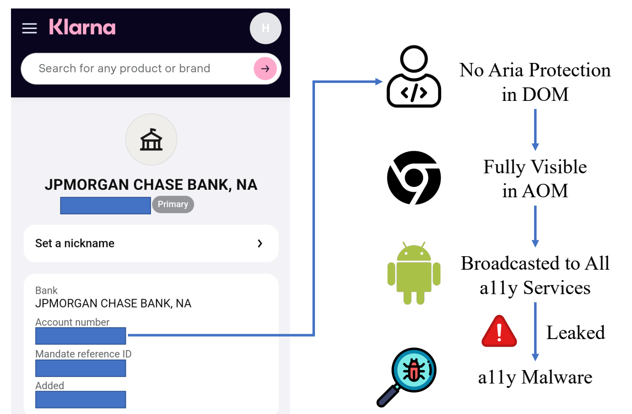
**Browser-Side a11y-Leaked Bank Account Number.** Klarna is a hybrid shopping service provider with access in both its Android app and their website. Its app is one of the most popular shopping apps in the Google Play Store with 10M+ downloads. The user setting page in both the app and website allows users to bind and view bank accounts for purchases. Figure 5 shows SOMBRA's a11y information extraction from both the Android app and mobile website rendered in Chrome while traversing the user payment method page. SOMBRA is able to extract users' sensitive bank account numbers through a11y access in the Chrome-rendered mobile website while unable to extract the same field in the Android app counterpart.

Figure 5a shows the screenshot of the payment method page of its Android app. During the a11y page traversal, SOMBRA discovered that the `ViewGroup` labeled as *Account Number* has a child element that has a null value in its a11y text field. After matching the child view's a11y resource ID with Klarna app's static view declarations extracted from the APK, SOMBRA found that the `TextView` representing the *Account Number* field is declared with the Android `A11Y_DATA_SENSITIVE_YES` property. This effectively forces the Android OS to only broadcast its a11y information to Google-approved a11y services with the *a11yTool* verification. Any non-Google-approved a11y services such as the one used by SOMBRA and the ones used by a11y malware cannot access this a11y-protected information.

Figure 5b shows the screenshot of the same payment method page of Klarna's mobile website rendered by Chrome. As shown in the figure, the structural hierarchy of views resembles that of




**(a) The account number field is a11y-protected in the Klarna app and inaccessible to a11y malware.**




**(b) The account number field is not a11y-protected in the Klarna mobile website rendered by Chrome and accessible to a11y malware.**

**Figure 5: Klarna's a11y implementation of the user bank account page in both the app and the mobile website.**

its Android app counterpart. Utilizing the same view hierarchy traversal sequence as SOMBRA did in the Android app, SOMBRA found an element with the same *Account Number* label. However, the account number is visible in the node's a11y text field and SOMBRA is able to retrieve the information. With access to the DOM tree during the traversal, SOMBRA found that no ARIA labels are declared for the *Account Number* field. As discussed in §5.4, most mobile website developers refrain from declaring `aria-hidden` protections to their sensitive information because it renders the field inaccessible to all a11y services, including benign utility ones, and hinders the usability of their websites. With no ARIA protection in the DOM, the Chrome browser then renders the a11y node fully visible in the AOM. Subsequently, Android extracts the field's a11y text from the AOM and broadcasts it to all registered a11y services, making them accessible to all a11y services registered on users' devices, including the ones controlled by a11y malware.

**Browser-Side a11y-Leaked Password.** As discussed in §5.4, since the `aria-hidden` protection renders the browser-side

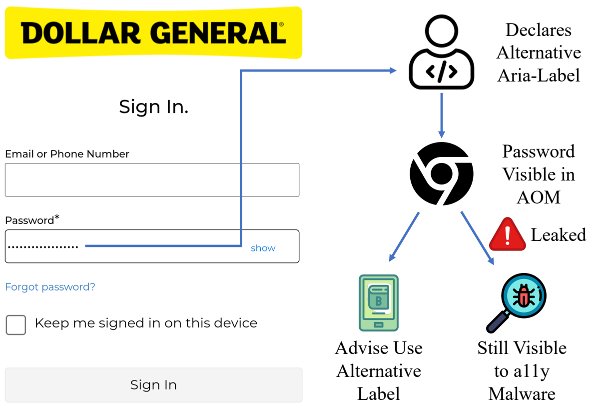



**Figure 6: Password leaked in Chrome-rendered Dollar General login page. Declaring alternative ARIA labels is ineffective at protecting browser-side a11y information.**

elements inaccessible to all a11y services, some developers seek other ways to protect their content by declaring ARIA alternative labels that customize the announcement of a field when accessed by an a11y service while making the elements still accessible. These alternative labels include `aria-label`, `aria-labelledby`, `aria-describedby`, etc. Although they are only intended by the web a11y standard to improve website usability, their ability to customize the a11y announcement of elements motivates developers to protect their browser-side elements.

Figure 6 shows the screenshot of Dollar General's Chrome-rendered website's login page. While SOMBRA's website traversal engine enters the registered password, SOMBRA's a11y service component is able to see and retrieve the newly entered digits by listening and parsing the screen change events. SOMBRA found that the password input box is declared with an `aria-label` field and set to a constant string "password." However, the browser still renders every input digit in the AOM. This is because the existing web a11y standard only recommends end-level a11y services to announce the alternative label while still advising browsers to make the field visible in the AOM. As a result, declaring alternative ARIA labels is ineffective at protecting against browser-side a11y leakage. Any a11y services on the device, including the ones used by a11y malware, can still access the fields declared with ARIA alternative labels.

## 7 Discussion

**Limitations.** Because Google implements fine-grained a11y protections on the app side, our study uses those protections as a baseline for what should be protected on the browser side. That said, if app developers accidentally misconfigure their app-side a11y protection for a view, SOMBRA will miss the corresponding browser-side element. Detecting misconfigurations in the app is out of scope, as we aim to align the existing protections. Additionally, SOMBRA requires valid credentials to test each service. However, SOMBRA users (service developers) should not face this challenge.

**WebViews and Custom Tabs.** During our study, SOMBRA found that 138 apps contain WebView [22] and 47 contain Custom Tab [13] elements. However, no a11y information is protected in those WebView and Custom Tab elements. This is because both use the Chromium-based render engine and Android's native a11y protection mechanisms are inapplicable to them.

**Developer's Defense.** As shown in §5.4, the existing browser-side `aria-hidden` mechanism sacrifices the usability of website content to benign a11y services by removing a11y content entirely from the AOM. Alternative a11y labels are also ineffective at hiding sensitive a11y information due to the existing web a11y standard. To allow developers to protect their browser-side a11y content while ensuring usability, we recommend they remove the sensitive content from their website and redirect the user to their Android app counterpart for access. For example, banking apps such as Chase advise users to install or redirect to the banking apps to conduct transactions.

**Mitigating the Attack Surface.** The existing Android native a11y support intends to both improve the usability and security of app content, while the existing web a11y standard only focuses on usability. To fundamentally mitigate this attack surface, the misalignment of app-side and browser-side a11y protection mechanisms needs to be eliminated. This would ideally consist of a three-party collaboration that involves redesigning the web a11y standard, enforcing browsers' interpretation of the web a11y model, and adapting Android's translation of browser-rendered AOM. At the base level, the web a11y standard should provide developers more freedom to fine-grain the level of a11y access to different a11y services. For example, mobile website developers can be allowed to delegate the screening of legitimate a11y services to Android and declare their content to be only accessible to Google-approved a11y services. At the browser level, a third-party should be introduced to enforce each browser to implement a consistent and correct interpretation of the same a11y model declared by website developers. This ensures that the a11y content access level does not differ among different browsers that the end users choose to use. Finally, the Android OS needs to adapt to the newly introduced web a11y standard with find-grained access to ensure each protection type is fully translated and broadcast to on-device a11y services.

**Disclosure and Open-Source.** We disclosed our findings to the Chrome, Firefox, Brave, and Edge teams. At the time of writing this paper, we received confirmation from the Firefox and Edge team, acknowledging the exposure of a11y information in browser-rendered elements. We also disclosed our findings to all 29 service developers who implement native a11y protections in their Android apps but leave their counterparts exposed on their websites. We recommended that they remove the sensitive content from their websites and redirect users to their Android app counterparts for access. Unlike addressing a traditional vulnerability, mitigating browser-side a11y leakage requires a long-term redesign of the web a11y standard, as the current standard lacks the find-grained a11y access control available in Android. We hope our findings can advocate for a three-party collaboration to mitigate this attack surface, as discussed earlier in §7. Finally, SOMBRA is open-sourced and available at https://github.com/CyFI-Lab-Public/SOMBRA.

## 8 Related Work

**Benign Misuse of a11y Service.** A11y services are often misused by benign applications such as anti-virus engines [27] and file system management apps [32] to achieve automated functionalities. Multiple works focus on dissecting the misuse of utility apps. Salehnamadi et al. [58] proposed a framework to assess mobile applications' a11y functionality correctness. Naseri et al. [53] introduced a study on how Android apps misuse the a11y service to achieve utility shortcuts. Chen et al. [30] proposed a dynamic traversal technique to extract a11y feature malfunctions in Android a11y apps. Instead of analyzing the benign misuse of the a11y service, we discovered an attack surface that can be abused by malicious a11y attackers.

**Attacks on a11y Service.** The a11y service is widely abused by malware to conduct automated phishing attacks [24, 28]. The powerful functionality of a11y service allows malware to launch attacks in an evasive manner [47]. Xu et al. [64] analyzed how real Android malware abuses the a11y service to conduct on-device fraud against mobile banking apps. Multiple works also proposed PoC attacks to exploit the a11y service [44, 43]. Fratantonio et al. [36] proposed an attack that enables malware to control the GUI of an Android device with the `SYSTEM_ALERT_WINDOW` and a11y permissions. Mehralian et al. [49] uncovered sensitive information leakage through overly accessible a11y elements in Android. Jang et al. [42] identified 12 a11y attacks on four different operating systems. Lei et al. [48] exposed an a11y side-channel attack that allows password leakage through guessing consecutive content queries. Unlike the attacks that target native Android apps, we uncovered an attack surface that allows a11y attackers to extract app-side inaccessible sensitive information in mobile-browser rendered websites. We found this attack surface impactful because mobile browser users of the same service are less protected from a11y attacks than app users.

**Defenses against a11y Attacks.** Malware and PoC a11y attacks have led to the development of multiple works to counteract malicious abuse of the a11y service [65, 40]. Fernandes et al. [35] introduced a technique to block all undeclared data-flows in Android apps by enforcing runtime restrictions. Huang et al. [41] proposed a more fine-grained Android a11y service design to enforce least-privileged data-flow constraints in runtime. Android also introduced new features to app developers to block app-side a11y access to untrusted a11y services [8]. With all the above defenses considered, we found the new attack surface introduced to still be feasible because it allows a11y malware to circumvent app-side protections. SOMBRA also helps benign service developers vet this attack surface and guides its mitigation.

**Program Analysis.** Prior work use API trace analysis [37, 55, 56, 3], network traffic analysis [71, 70, 4], symbolic analysis [69, 68] and forensic analysis [54, 60, 59] to reveal program behaviors. However, to discover app-side a11y-protected elements, SOMBRA uses a combination of dynamic app traversal and static attribution.

**Browser Instrumentation.** Prior work have instrumented the browser engine to collect activities of web pages from DOM [66, 1, 63, 2, 67]. SOMBRA is inspired by these techniques but focuses the traversal on the a11y tree to reveal unprotected elements.

## 9 Conclusion

We introduced SOMBRA, an automated analysis pipeline for benign service developers to vet browser-side leakage of a11y information otherwise protected in their Android app counterparts. Using SOMBRA, we analyzed 294 real benign services. SOMBRA found that 29 services utilized native a11y protection mechanisms to secure 256 views in their Android apps. However, SOMBRA discovered that 241, 402, 244, and 251 elements corresponding to the same fields are a11y-exposed in their websites rendered with Chrome, Firefox, Brave, and Edge mobile browsers. The leaked a11y information on the browser side contains sensitive PII information such as credit card information and user passwords. Finally, SOMBRA discovered that existing browser-side a11y protection mechanisms either are ineffective at protecting services' content or hinder the usability of the content.

## Acknowledgments

We thank the anonymous reviewers for their constructive comments and feedback. We also thank our collaborators at Netskope for their support throughout this research. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179 and N00014-23-1-2073; the National Science Foundation (NSF) under grant 2143689; and the Defense Advanced Research Projects Agency (DARPA) under contract N66001-21-C-4024. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors and collaborators.

## References

[1] Joey Allen, Zheng Yang, Matthew Landen, Raghav Bhat, Harsh Grover, Andrew Chang, Yang Ji, Roberto Perdisci, and Wenke Lee. 2020. Mnemosyne: an effective and efficient postmortem watering hole attack investigation system. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. Virtual Conference, (Nov. 2020).

[2] Joey Allen, Zheng Yang, Feng Xiao, Matthew Landen, Roberto Perdisci, and Wenke Lee. 2024. Webrr: a forensic system for replaying and investigating web-based attacks in the modern web. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, (Aug. 2024).

[3] Omar Alrawi, Moses Ike, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua, Taleb Hirani, Brennan Hill, and Brendan Saltaformaggio. 2021. Forecasting malware capabilities from cyber attack memory images. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Conference, (Aug. 2021).

[4] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Pai Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. 2019. The betrayal at cloud city: an empirical analysis of cloud-based mobile backends. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, (Aug. 2019).

[5] Android. 2025. Accessibilityevent: isaccessibilitydatasensitive(). (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent%5C#isAccessibilityDataSensitive().

[6] Android. 2025. Accessibilitynodeinfo. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo.

[7] Android. 2025. Accessibilityservice. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.

[8] Android. 2025. Accessibilityserviceinfo: isaccessibilitytool(). (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo%5C#isAccessibilityTool().

[9] Android. 2025. Accessibiltiyevent. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.

[10] Android. 2025. Edittext. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/widget/EditText.

[11] Android. 2025. Gettext(). (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getText().

[12] Android. 2025. Getuniqueid(). (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getUniqueId().
[13] Android. 2025. Overview of android custom tabs. (2025). Retrieved 2025-01-07 from https://developer.chrome.com/docs/android/custom-tabs.
[14] Android. 2025. Settext(java.lang.charsequence). (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/widget/TextView%5C#setText(java.lang.CharSequence).
[15] Android. 2025. Specify the input method type. (2025). Retrieved 2025-01-07 from https://developer.android.com/develop/ui/views/touch-and-input/keyboard-input/style.
[16] Android. 2025. Textview. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/widget/TextView.
[17] Android. 2025. Type_view_accessibility_focused. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent%5C#TYPE%5C_VIEW%5C_ACCESSIBILITY%5C_FOCUSED.
[18] Android. 2025. Type_window_state_changed. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent%5C#TYPE%5C_WINDOW%5C_STATE%5C_CHANGED.
[19] Android. 2025. View. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/View.
[20] Android. 2025. View.accessibilitydelegate. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/View.AccessibilityDelegate.
[21] Android. 2025. Viewgroup. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/view/ViewGroup.
[22] Android. 2025. Webview. (2025). Retrieved 2025-01-07 from https://developer.android.com/reference/android/webkit/WebView.
[23] Androzoo. 2025. Androzoo. (2025). Retrieved 2025-01-07 from https://androzoo.uni.lu/.
[24] Simone Aonzo, A. Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing attacks on modern android. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada, (Oct. 2018).
[25] AppBrain. 2025. Appbrain - everything you need for a successful android app. (2025). Retrieved 2025-01-07 from https://www.appbrain.com/.
[26] Appium. 2025. Appium. (2025). Retrieved 2025-01-07 from https://appium.io/docs/en/latest/.
[27] Marcus Botacin, Felipe Duarte Domingues, Fabrício Ceschin, Raphael Machnicki, Marco Antonio Zanata Alves, Paulo Lício de Geus, and André Grégio. 2022. Antiviruses under the microscope: a hands-on perspective. *Computers and Security*, 112, (Jan. 2022), 102500.
[28] Davide Bove and Anatoli Kalysch. 2019. In pursuit of a secure ui: the cycle of breaking and fixing android's ui. *it - Information Technology*, 61, 2-3, (Apr. 2019), 147–156.
[29] Andrew Buck. 2025. Mobile apps vs mobile websites: why people spend 90% of their time in apps. (2025). Retrieved 2025-01-07 from https://www.mobiloud.com/blog/mobile-apps-vs-mobile-websites.
[30] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2022. Accessible or not? an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 48, 10, (Oct. 2022), 3954–3968.
[31] CyberGhostVPN. 2025. New google play malware poses major threat to mobile banking. (2025). Retrieved 2025-01-07 from https://www.cyberghostvpn.com/en%5C_US/privacyhub/google-play-malware/.
[32] Wenrui Diao et al. 2019. Kindness is a risky business: on the usage of the accessibility APIs in android. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Beijing, China, (Sept. 2019).
[33] Alice Boxhall et al. 2025. Accessibility object model. (2025). Retrieved 2025-01-07 from https://wicg.github.io/aom/explainer.html.
[34] Threat Fabric. 2022. 2022 mobile threat landscape update. (2022). Retrieved 2025-01-07 from https://www.threatfabric.com/blogs/h1-2022-mobile-threat-landscape.html.
[35] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: practical data protection for emerging IoT application frameworks. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX, (Aug. 2016).
[36] 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 *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. San Jose, CA, (May 2017).
[37] Jonathan Fuller, Ranjita Pai Kasturi, Amit Sikder, Haichuan Xu, Berat Arik, Vivek Verma, Ehsan Asdar, and Brendan Saltaformaggio. 2021. C3po: large-scale study of covert monitoring of c&c servers via over-permissioned protocol infiltration. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Seoul, South Korea, (Nov. 2021).
[38] Google. 2025. Google cloud natual language api basics. (2025). Retrieved 2025-01-07 from https://cloud.google.com/natural-language/docs/basics.
[39] Google. 2025. Navigate your device with talkback. (2025). Retrieved 2025-01-07 from https://support.google.com/accessibility/android/answer/6006598?sjid.
[40] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: a programmable interface for extending android security. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA, (Aug. 2014).
[41] 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 *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Conference, (Aug. 2021).
[42] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. 2014. A11y attacks: exploiting accessibility in operating systems. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, AZ, (Nov. 2014).
[43] Anatoli Kalysch, Davide Bove, and Tilo Müller. 2018. How android's UI security is undermined by accessibility. In *Proceedings of the 2nd Reversing and Offensive-Oriented Trends Symposium (ROOTS)*. Vienna, Austria, (Nov. 2018).
[44] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2013. On malware leveraging the android accessibility framework. In *Proceedings of Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)*. Tokyo, Japan, (Dec. 2013).
[45] Ravie Lakshmanan. 2023. Goldoson android malware infects over 100 million google play store downloads. (2023). Retrieved 2025-01-07 from https://thehackernews.com/2023/04/goldoson-android-malware-infects-over.html.
[46] Ravie Lakshmanan. 2022. Teabot android banking malware spreads again through google play store apps. (2022). Retrieved 2025-01-07 from https://thehackernews.com/2022/03/teabot-android-banking-malware-spreads.html.
[47] Yonas Leguesse, Mark Vella, Christian Colombo, and Julio C HernandezCastro. 2020. Reducing the forensic footprint with android accessibility attacks. In *Proceedings of Security and Trust Management - 16th International Workshop (STM)*. Guildford, UK, (Sept. 2020).
[48] Chongqing Lei, Zhen Ling, Yue Zhang, Kai Dong, Kaizheng Liu, Junzhou Luo, and Xinwen Fu. 2023. Do not give a dog bread every time he wags his tail: stealing passwords through content queries (CONQUER) attacks. In *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, (Feb. 2023).
[49] 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 *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Rochester, MI, (Oct. 2022).
[50] Mozilla. 2025. Accessibility. (2025). Retrieved 2025-01-07 from https://developer.mozilla.org/en-US/docs/Web/Accessibility.
[51] Mozilla. 2025. Aria. (2025). Retrieved 2025-01-07 from https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA.
[52] Mozilla. 2025. Document object model. (2025). Retrieved 2025-01-07 from https://developer.mozilla.org/en-US/docs/Web/API/Document%5C_Object%5C_Model.
[53] Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. 2019. Accessileaks: investigating privacy leaks exposed by the android accessibility service. In *Proceedings of Privacy Enhancing Technologies Symposium (PETS)*. Stockholm, Sweden, (Apr. 2019).
[54] David Oygenblik, Carter Yagemann, Joseph Zhang, Arianna Mastali, Jeman Park, and Brendan Saltaformaggio. 2024. Ai psychiatry: forensic investigation of deep learning networks in memory images. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, (Aug. 2024).
[55] Ranjita Pai Kasturi, Jonathan Fuller, Yiting Sun, Omar Chabklo, Andres Rodriguez, Jeman Park, and Brendan Saltaformaggio. 2022. Mistrust plugins you must: a large-scale study of malicious plugins in wordpress marketplaces. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, (Aug. 2022).
[56] Ranjita Pai Kasturi, Yiting Sun, Ruian Duan, Omar Alrawi, Ehsan Asdar, Victor Zhu, Yonghwi Kwon, and Brendan Saltaformaggio. 2020. Tardis: rolling back the clock on cms-targeting cyber attacks. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. Virtual Conference, (May 2020).
[57] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug fixes, improvements, ... and privacy leaks - a longitudinal study of pii leaks across android app versions. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, (Feb. 2018).
[58] Navid Salehnamadi, Abdulaziz Alshayban, JunWei Lin, Iftekhar Ahmed, Stacy M Branham, and Sam Malek. 2021. Latte: use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 Conference on Human Factors in Computing Systems (CHI)*. Yokohama, Japan, (May 2021).

[59] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. Guitar: piecing together android app guis from memory images. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, (Oct. 2015).

[60] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G. Richard. 2016. Screen after previous screens: spatial-temporal recreation of android app displays from memory images. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX, (Aug. 2016).

[61] Gaurav Shinde. 2017. Malware on google play abusing accessibility service. (2017). Retrieved 2025-01-07 from https://www.zscaler.com/blogs/security-research/malware-google-play-abusing-accessibility-service.

[62] Bill Toulas. 2023. Cybercrime service bypasses android security to install malware. (2023). Retrieved 2025-01-07 from https://www.bleepingcomputer.com/news/security/cybercrime-service-bypasses-android-security-to-install-malware/.

[63] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. 2022. Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, (Nov. 2022).

[64] Haichuan Xu, Mingxuan Yao, Runze Zhang, Mohamed Moustafa Dawoud, Jeman Park, and Brendan Saltaformaggio. 2024. Dva: extracting victims and abuse vectors from android accessibility malware. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, (Aug. 2024).

[65] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and detecting overlay-based android malware at market scales. In *Proceedings of the 17th ACM International Conference on Mobile Computing Systems (MobiSys)*. Seoul, South Korea, (June 2019).

[66] Zheng Yang, Joey Allen, Matthew Landen, Roberto Perdisci, and Wenke Lee. 2023. Trident: towards detecting and mitigating web-based social engineering attacks. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, (Aug. 2023).

[67] Zheng Yang, Simon Chung, Jizhou Chen, Runze Zhang, Brendan Saltaformaggio, and Wenke Lee. 2025. Coindef: a comprehensive code injection defense for the electron framework. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, (May 2025).

[68] Mingxuan Yao, Jonathan Fuller, Ranjita Pai Sridhar, Saumya Agarwal, Amit K. Sikder, and Brendan Saltaformaggio. 2023. Hiding in plain sight: an empirical study of web application abuse in malware. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, (Aug. 2023).

[69] Mingxuan Yao, Runze Zhang, Haichuan Xu, Shih-Huan Chou, Varun Chowdhary Paturi, Amit K. Sikder, and Brendan Saltaformaggio. 2024. Pulling off the mask: forensic analysis of the deceptive creator wallets behind smart contract fraud. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, (May 2024).

[70] Runze Zhang, Mingxuan Yao, Haichuan Xu, Omar Alrawi, Jeman Park, and Brendan Saltaformaggio. 2025. Hitchhiking vaccine: enhancing botnet remediation with remote code deployment reuse. In *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, (Feb. 2025).

[71] Runze Zhang et al. 2025. Identifying incoherent search sessions: search click fraud remediation under real-world constraints. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, (May 2025).


# A Validation

We validated SOMBRA's accuracy in detecting app-side protected views and matching them with browser-rendered elements. From our dataset, we randomly selected an app and included it in the validation set if it had at least one app-side protected native element identified by SOMBRA, continuing this process until we found 10 such apps. This approach ensures the existence of app-side ground truth for a11y-protected views, enabling comparison with the corresponding browser-side elements. We then installed them on the Google Pixel 5 device running Android 14 and navigated to the Chrome-rendered websites of the 10 apps on the same device. We obtained the ground truth of app-side and browser-side protection utilizing Android's built-in TalkBack [39] service, manually traversing the app's and website's user profile screen, retrieving the text of each on-screen element and comparing them with the embedded a11y text broadcast in `Viewa11yFocused` [17] events.

**Table 5: Validation Of SOMBRA's Detection Of App-side Protected Views And Browser-side Exposed Views.**

| Name | App-Side | | | Browser-Side[1] | | |
|---|---|---|---|---|---|---|
| | SOMBRA [2] | FP | FN | SOMBRA [3] | FP | FN |
| MySynchrony | 6 | 0 | 0 | 9 | 0 | 0 |
| MetaMask | 7 | 0 | 0 | 9 | 0 | 0 |
| Varo Bank | 4 | 0 | 0 | 7 | 0 | 0 |
| ParkMobile | 5 | 0 | 2 | 7 | 0 | 2 |
| ChargePoint | 7 | 0 | 0 | 22 | 0 | 0 |
| Ventra | 11 | 0 | 0 | 14 | 0 | 3 |
| Klarna | 4 | 0 | 0 | 8 | 0 | 0 |
| CVS | 2 | 0 | 0 | 6 | 0 | 0 |
| Belk | 12 | 0 | 0 | 11 | 0 | 4 |
| Dollar General | 8 | 0 | 0 | 18 | 0 | 0 |
| **Total** | 66 | 0 | 2 | 111 | 0 | 9 |

1: Rendered by the mobile Chrome browser.
2: SOMBRA's detected app-side protected views.
3: SOMBRA's detected browser-side exposed views.

**Validation Results.** For app-side validation, as shown in Columns 2-4 of Table 5, SOMBRA has 97% accuracy in detecting app-side protected views. SOMBRA missed two (FN) while analyzing the Park Mobile app. Upon further investigation, we found that the Park Mobile app broadcasts fixed placeholder addresses for two views that represent users' addresses, acting as an effective a11y protection. We confirmed that this is a rare occurrence.

For browser-side validation, as shown in Columns 5-7 of Table 5, SOMBRA achieved 93% accuracy in matching and detecting browser-side exposed elements. Because of the two FN app-side views missed by SOMBRA in the Park Mobile app, SOMBRA also missed the two corresponding browser-side elements. For the Ventra app, SOMBRA missed three elements (FN) that match the app-side protected user account number and balance fields because of missing element labels by website developers. Similarly in the Belk app, SOMBRA missed four elements (FN) that match the app-side protected reward member number and contact detail because of empty element labels as well.