A11y and Privacy don't have to be mutually exclusive: Constraining Accessibility Service Misuse on Android

Jie Huang, Michael Backes, and Sven Bugiel

CISPA Helmholtz Center for Information Security
{jie.huang, backes, bugiel}@cispa.saarland

Abstract

Accessibility features of Android are crucial in assisting people with disabilities or impairment to navigate their devices. However, the same, powerful features are commonly misused by shady apps for malevolent purposes, such as stealing data from other apps. Unfortunately, existing defenses do not allow apps to protect themselves and at the same time to be fully inclusive to users with accessibility needs.

To enhance the privacy protection of the user while preserving the accessibility features for assistive apps, we introduce an extension to Android's accessibility framework. Our design is based on a study of how accessibility features are used in 95 existing accessibility apps of different types (malware, utility, and a11y). Based on those insights, we propose to model the usage of the accessibility framework as a pipeline of code modules, which are all sandboxed on the system-side. By policing the data flows of those modules, we achieve a more fine-grained control over the access to accessibility features and the way they are used in apps, allowing a balance between accessibility functionality for dependent users and reduced privacy risks. We demonstrate the feasibility of our solution by migrating two real-world apps to our privacy-enhanced accessibility framework.

1 Introduction

Accessibility features, also known as *a11y services*[1], are meant to assist people with disabilities or impairment in using their computer systems. Android includes an accessibility framework since Android v1.6 that allows authorized third party apps to act as a11y apps, such as a screen reader app or alternative navigation via voice commands and head gestures.

Since accessibility apps necessarily have to be exempted to a certain extent from the usual isolation between apps, access to the accessibility framework is restricted with a dedicated permission (`BIND_ACCESSIBILITY_SERVICE`). Only after obtaining this permission from the user, apps can retrieve information from the accessibility framework about other apps or send events to other apps (e.g., UI interactions). However, this permission is coarse-grained and very powerful. Once an app is granted access to the accessibility framework, it has the privilege of accessing private data from all other apps, including sensitive data normally protected by other permissions or user entered data, or to mimic human users' actions (like button clicks). According to Google's guidelines, the accessibility features are supposed to be used only by a11y apps that help disabled and impaired users to operate their devices and apps. Despite this guidelines, there exist a lot of apps that use those powerful features for their own purposes, for example, automatization of tedious user actions (e.g., easy uninstallation of apps via injected button clicks that navigate the *Settings* app) or auto-filling of credentials by password manager apps. Given the power of accessibility apps and the wide range of usage of the accessibility features, it is correct to assume that not all apps use this power appropriately [18] and currently the accessibility framework comes with an inherent threat to the users' privacy. Even worse, various samples of malicious apps [1, 3, 4] have already been reported to utilize the accessibility features to monitor and mimic user interactions with third-party apps in order to steal sensitive data, like user credentials or bank information, and also academic research [22, 29] has highlighted the risks of a11y features.

What should be clear by today is that the current restrictions to access the accessibility framework are not sufficient to protect user data and defend against malicious intents. The burden to establish any defense today rests on the shoulders of the app developers that might fall victim to misuse of a11y features. App developers can pro-actively exempt components or UI elements of their apps from being monitored by the accessibility framework in an effort to protect sensitive data or prevent misuse of UI elements. Unfortunately, not only do many app developers abstain from those defenses [34], but even worse, those defenses defeat the very purpose of the accessibility services. For example, an app developer of

[1] a11y is the abbreviation of *accessibility*.

a mobile banking app that exempts the input field for the account number to avoid leakage via a11y services would also exclude screen readers or voice command apps from reading back or writing that number. What is needed to not make accessibility and privacy mutually exclusive is an accessibility framework that supports a more fine-grained control over how its features can be used.

In this work, we propose an extension to Android's default accessibility framework that enables configuration of a more fine-grained control over how accessibility features are used by accessibility apps. We start by investigating the integration and usage of the accessibility framework in 95 real-world apps that are either benign a11y apps, apps repurposing a11y features (e.g., automatization), or malware abusing accessibility features in order to better understand what kind of policy enforcement such a solution has to provide and which potential limits exist. Our results exhibit a clear tendency of how malware is currently misusing the accessibility features. However, our results also raise the challenge that malicious behavior and benign behavior are not distinguishable at the API boundary (e.g., which accessibility data and features are being accessed) and that a suitable solution has to control the data flows within accessibility apps.

Noticing parallels between our setting and that of IoT and augmented reality apps, we take inspiration from the ideas of *data processing pipelines* for AR apps [28] and of *quarantined code modules* with *opaque data handles* for IoT apps [21]. Transferring those ideas to our problem setting for a11y, accessibility apps access certain information from the framework and process them in a particular way, or they trigger certain accessibility actions as reaction to certain triggers. For example, a screen reader accesses text information and outputs an audio stream, or a virtual mouse app tracks eye movement and clicks buttons. The key idea of our solution is to make the single steps in such processing pipelines explicit and sandbox them in least-privileged service components. Accessibility apps then build their pipelines by chaining those services together and orchestrate their interactions. We enforce policies at their input/output boundaries to govern to which data and features each module has access. By keeping the overall pipeline in mind, those policies control how data can propagate within a single pipeline—sources to sinks—or under which circumstances a pipeline can trigger (accessibility) actions.

Although our study of existing malware and a11y apps indicates that a policy that universally maximizes functionality for benign apps while simultaneously eliminating the potential for misuse seems unlikely, our solution allows configuration of a trade-off between functionality and protection according to users' needs (e.g., disabling accessibility features that are not necessary for the desired a11y apps). This is a clear benefit over stock Android's all-or-nothing protection against misuse of the accessibility framework. Since our design only changes the public APIs of the default accessibility framework (e.g., apps needs to register and orchestrate their modules), only

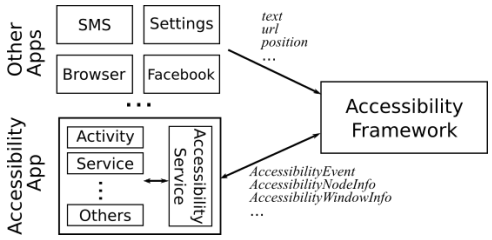

Figure 1: Accessibility Communication Channel

developers of a11y apps need to adapt their code to the new setting but no other app developers are affected. We demonstrate this by porting two open-source accessibility apps to our enhanced accessibility framework.

Contributions. We make the following contributions:

1) Systematization of accessibility service integration. We study the actual usage of accessibility features in real-world benign, utility[2], and malware apps. Our results reveal patterns and behaviors how the accessibility API is misused. We believe those results contribute to a deeper understanding of how a11y features are being (mis-)used and can help future work in creating better defenses against a11y attacks.

2) Privacy-enhanced accessibility framework. Based on the results of our systematization, we propose a privacy-enhanced accessibility framework. Privacy here means that data retrieved via the accessibility framework should not leak without authorization and that all accessibility actions should be authorized or triggered by the user or at most be inefficiently misused. Our framework separates a11y logic of apps into sandboxed code modules and allows enforcement of privacy policies at the input/output boundary of those modules. This enables a more fine-grained control over how accessibility features are used, how data propagates in the pipelines formed by those modules, and, hence, offers a more effective protection against misuse of the accessibility framework than stock Android.

3) Real-world app migration and evaluation. We migrate two real-world open-source accessibility apps to our privacy-enhanced framework to demonstrate how our framework provides better protection in those cases. Further, micro-benchmarks show that the performance overhead imposed by our solution is acceptable.

2 Android Accessibility Service

We provide technical background knowledge on the accessibility framework in Android and building accessibility apps.

[2] We refer in the context of this paper to apps that repurpose the a11y features for user desired but by Google unintended use-cases as *utility* apps.

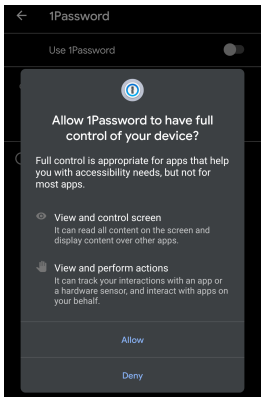

Figure 2: Example for explicitly authorizing an AccessibilityService, here of the *1Password* app, in the system settings

2.1 Accessibility Service Overview

Android supports accessibility features since API level 4 through its Accessibility Framework [11]. Figure 1 provides an overview of how the framework works. The accessibility framework acts as an intermediary between applications and accessibility (or a11y) apps. It monitors relevant events within applications and forwards them to accessibility apps, which in turn can use the framework to retrieve certain information (e.g., UI content) from those applications or inject events into those applications (e.g., inserting text or clicking a button).

`AccessibilityService` is the key component for an accessibility app to use the accessibility [7] features. Each accessibility app has to register an `AccessibilityService` to listen for accessibility events. Through `onAccessibilityEvent()` callbacks, the app receives accessibility events that are wrapped as `AccessibilityEvent` objects. The app can then perform custom logic to consume and react to those events. For instance, a screen reader could read aloud a button description that was contained in a received event for a user's UI interaction. To register the `AccessibilityService` in the system, the developer of the accessibility app should declare it as such in the `AndroidManifest.xml` of the app. To ensure that only the system's accessibility framework can bind to this service of the app, the service declaration should require the system permission `android.permission.BIND_ACCESSIBILITY_SERVICE` from any caller. Since no third party app can successfully request this permission, the accessibility app is ensured that any caller to the `AccessibilityService` is the system. Lastly, since access to the accessibility framework is highly critical for user privacy, Android requires the user to explicitly grant this access via the *Settings* app. Figure 2 gives an example of this explicit activation for the *1Password* app in the system setting. Only after those steps, the accessibility app is able to assist the user (or attack them and other apps).

```
1  public class MyAccessibilityService extends AccessibilityService {
2    @Override
3    public void onAccessibilityEvent(AccessibilityEvent event) {
4
5      // global action: back to home screen
6      performGlobalAction(AccessibilityService.GLOBAL_ACTION_HOME);
7      // global action: show activity history
8      performGlobalAction(AccessibilityService.GLOBAL_ACTION_RECENTS);
9
10     AccessibilityNodeInfo node = event.getSource();
11     if (isTargetButton(node)) {
12       // local action: click the target button
13       node.performAction(AccessibilityNodeInfo.ACTION_CLICK);
14     } else if (isTargetEditText(node)) {
15       // local action: input string "android" to an EditText
16       Bundle arguments = new Bundle();
17       arguments.putCharSequence(
18           AccessibilityNodeInfo.ACTION_ARGUMENT_SET_TEXT_CHARSEQUENCE,
19           "android");
20       node.performAction(AccessibilityNodeInfo.ACTION_SET_TEXT, arguments);
21     }
22   }
23 }
```

Listing 1: Code example for using the accessibility service

2.2 Accessibility Communication Channel

Lastly, we have to zoom in to the communication channel between an `AccessibilityService` and the accessibility framework. Accessibility objects, such as `AccessibilityEvent`, `AccessibilityNodeInfo`, or `AccessibilityWindowInfo`, carry other apps' sensitive data, e.g., screen text, to enable the `AccessibilityService` in doing its intended job, e.g., reading screen text aloud. However, the accessibility app can also invoke *global* or *node* actions. Listing 1 provides a toy `AccessibilityService` to illustrate this. *Global* actions are not targeting any specific app and include, for instance, invoking the device's home button or opening the recents screen (or recent task list screen) showing recently accessed apps (see Lines 6 and 8 in Listing 1). *Node* actions target a particular element in another app, for instance, a button or text field (see Lines 13 and 20).

Sensitive data exposure via accessibility features: To be able to provide assistive functionality, an `AccessibilityService` is very powerful and can access a great amount of sensitive data within other apps. Different from sensitive data that is usually protected by Android's permission model and UID-based sandboxing from unauthorized access by apps, the accessibility framework can easily leak such protected data across application boundaries to an accessibility app. For example, an accessibility app without `READ_CONTACTS` permission can still get contact information stored in the *Contacts* app through reading the text fields in `AccessibilityEvents` from the *Contacts* app.

To inform the accessibility framework and the user about which events an `AccessibilityService` is interested in, the developer can specify an `AccessibilityServiceInfo` [8] that lists the capabilities and accessible `AccessibilityEvents`. Thus, the `AccessibilityServiceInfo` informs about which data is exposed to an `AccessibilityService` and what the service could do. For example, by limiting the `packageNames` attribute of `AccessibilityServiceInfo` to `com.android.settings`, the service will only receive

events for the *Settings* app. This configuration can be set either statically as meta-data inside an `xml` file or dynamically at runtime through the `setServiceInfo` interface of the accessibility framework. However, this configuration relies on the incentives of accessibility app developers.

Developers of other applications can further communicate to an `AccessibilityService` that certain UI elements are not important for accessibility. This is in two different ways fallible: first, every app developer has to become active and, second, this forces app developers to choose between writing an app that is protected or that is inclusive. Moreover, this is merely an indication by the app developer and an `AccessibilityService` can decide to ignore this attribute and operate on all UI elements in a targeted app [10].

3 Study of Accessibility Service Usage

Considering the high privileges of an accessibility service and the diverse ways to use it—for a11y as intended, as a user-desired utility, or for malevolent purposes—we are interested in how real-world apps make use of this service and whether there exist distinguishing features in the usage patterns between a11y, utility, and malicious apps. Prior work [18] evaluated the usage of accessibility services in *normal*[3] apps based on natural language processing of the app descriptions. This approach highly relies on the accurate (and honest) developer documentation. A missing, ambiguous, or dishonest description could hide the actual usage of the accessibility features from the results. To gain a more comprehensive and reliable understanding of the usage of accessibility features, we base our study of accessibility (mis-)usage directly on the apps' code, including utility and malware samples. By collecting each sample app's access to the accessibility framework and then comparing the integration between each app's components and their accessibility services, we discover patterns how accessibility apps actually make use of the a11y framework and we gain an overview how accessibility can undermine the users' privacy in practice.

In the remainder of this section, we look at the different ways how an `AccessibilityService` is configured (e.g., which events are being subscribed), which APIs are being used, and which behavioral patterns can be detected in accessibility apps. The key question we want to answer is ***if the different types of accessibility apps—a11y, utility, and malware—are distinguishable in their configuration, API access, or use of accessibility services?***

3.1 Accessibility App Sample Set

We differentiate between three classes of accessibility apps: *malicious*, *utility*, and *a11y*.

[3] Here *normal* refers to apps in official markets.

Table 1: Accessibility service configuration in sample apps

Attribute	#M (57)	#U (36)	#A (8)
events from all apps[1]	**49** (86%)	**24** (67%)	**8** (100%)
canRetrieveWindowContent	**42** (74%)	**30** (83%)	**6** (75%)
∪	**57** (100%)	**34** (94%)	**8** (100%)
∩	**34** (60%)	**20** (56%)	**6** (75%)

M: Malware; **U:** Utility; **A:** A11y
[1] Service does **not** define an allowlist of package names

Malicious apps take advantage of accessibility service to attack users, e.g., logging sensitive user input, mounting phishing attacks, stealing private data from other apps, or surreptitiously granting permissions and installing apps. To collect a representative and timely set of malicious apps for our investigation, we turn to renown malware repositories on GitHub. From GitHub, we collected 608 reported Android malware samples from top ranking malware repositories [12, 27, 38, 39]. After filtering out samples without an `AccessibilityService`, we obtained 55 *malicious* accessibility app samples (57 `AccessibilityService` implementations).

In contrast, *a11y* apps use specific accessibility features to assist people with disabilities or impairments. The use of the accessibility service in those apps meets the intended purpose by Google. For example, a screen reader app reads aloud the text label on a touched button to assist users with visual impairments in using the device. By keyword search on Google Play, we gathered 5 *a11y* sample apps. Lastly, *utility* apps are neither typical assistive apps nor malicious. They ignore Google's accessibility developer guide [11] by using accessibility features for user-desired functionality beyond supporting people with disabilities, such as optimizing user experience (e.g., automatization of tedious tasks or password auto-fill). Google once announced to remove apps that use accessibility features for purposes other than the intended way [2], but this ban was paused after Google realized the popularity of accessibility features in supporting non-accessibility functionality. We crawled 2,751 top Google Play apps in December 2018 and found 36 accessibility apps of this kind, which we use as our *utility* app samples. To check that both the *utility* and *a11y* apps are not malware in disguise, we scan those two sample sets with VirusTotal [41].One app, *Avira*, was reported as malware by VirusTotal. Considering it was flagged by only 1 of 60 engines, we conservatively removed it from our set but did not think this significant enough to report to Google Play. Our non-malicious app sets finally consist of 5 *a11y* apps with 8 `AccessibilityService` and 35 *utility* apps with 36 `AccessibilityService`.

In total, we collected 95 accessibility app samples (101 `AccessibilityService` implementations) for our investigation.

Table 2: Allowlisted package names in service configurations. No a11y app configured an allowlist.

Package	#M (8)	#U (12)
com.android.settings	0	5
com.android.packageinstaller	0	2
browser*	0	3
communication*	2	1
shopping*	2	0
transportation*	2	0
tool*	2	0
self*	6	3

M: Malware; **U:** Utility; ***:** Category of apps since multiple packages of this type are monitored

3.2 Accessibility Service Configuration

As introduced in Section 2.2, app developers can control the capabilities and types of events that their `AccessibilityService` will receive by customizing the `AccessibilityServiceInfo` configuration. This configuration provides a statement about which sensitive data from other apps is potentially exposed to the `AccessibilityService` via the accessibility framework. Among the different available configuration attributes, `packageNames` and `canRetrieveWindowContent` effectively constrain the accessibility app's access to other apps. Attribute `packageNames` allow-lists the source packages for `AccessibilityEvents` the `AccessibilityService` will receive. If this attribute is not set, the `AccessibilityService` will receive events from *all* other packages. If developed with least privilege principle in mind and if applicable, the `AccessibilityService` should specify all the necessary source app packages here. The attribute `canRetrieveWindowContent` controls if the accessibility app can access the window content of other apps, including sensitive data contained within those windows. Obviously, this access to window content is a great way to steal data.

We compare the accessibility service configurations for those two highly sensitive attributes within our app samples to understand the extent of sensitive data to which different accessibility apps have access to. Since this configuration can be set both statically and dynamically, we extract the static configuration file from the apps and combine this with runtime information from tracing the `setServiceInfo` system API. Table 1 shows the number of packages that do **not** declare a package name (i.e., monitor broadly) and that are able to inspect the window content of other apps. The results show that all malware and a11y apps in our sample set monitor broadly, i.e., every malware and a11y app is at least able to inspect window content or receive events of all other apps, while 34 (60%) of the malware and 6 (75%) of the a11y services can do both. While this is intuitive, given the nature of those apps, also 34 (94%) of the utility services make use of those features, where 20 (56%) utility services use both features. For those services that specified an allowlist of package names, we also check the package name details. Of all malware apps, 8 services set an allowlist and receive only events from listed packages, while 12 utility services set an allowlist. None of the 8 a11y services set an allowlist and all of them monitor broadly. The distribution of the (types of) allow-listed packages by the malicious and utility apps can be found in Table 2. Those results show that while utility apps listen primarily to events from system apps, like settings, installer, or browser, malware targets specifically packages in certain categories, such as communication or shopping.

Summary: From those results, we conclude that the currently available constraints on accessibility service do not prevent the risk of abuse of a11y features, since all app types, including legitimate accessibility apps, configure a broad monitoring. Further, the similarity between the configurations makes it hard to distinguish purely on the configurations between a targeted attack and compliance to the least privilege principle.

3.3 Accessibility API Usage

Since the accessibility service configuration does not show a distinguishable pattern between different app types, we further investigated the accessibility framework API usage within accessibility services. After a review of the accessibility framework documentation, we categorize the accessibility API into three categories: 1) *retrieve information*, 2) *perform node action*, and 3) *perform global action*. *Retrieve information* APIs refer to interfaces that request information about other apps, including on-screen text, window position and so on. *Perform node action* API refers to interfaces that perform an action on a specified UI element (*node*), e.g., clicking a button. *Perform global action* API refers to interfaces for issuing a global operation, like clicking the "home" button or showing the recent task list.

Based on this categorization, we analyzed the types of accessibility APIs that are used in our sample apps and with which goal they were used by the apps (i.e., scenario). To this end, we manually interacted with the app UI and pinpointed possible usages based on the service descriptions and hints of UI elements. Since malicious apps by nature might mislead the user in those descriptions, we further collected accessibility-related behavior descriptions from technical reports by malware analysts and reverse engineers. For each discovered usage scenario, we manually inspected one app in depth through either reverse engineering or source code analysis where possible to find patterns how accessibility services are integrated into their apps.

In the end, we found four common patterns for the usage of accessibility methods:

Table 3: Patterns of accessibility API Usage

	Scenario	Patterns P1	P2	P3	P4
Malicious	Content Eavesdropping	✓	✗	✗	✗
	Phishing	✓	✗	✗	✗
	Process Persistence	✓	✗	✗	✗
	Silent Installation	✗	✓	✓	✗
	Silent Privilege Elevation	✗	✓	✓	✗
	E-Banking Fraud	✗	✓	✓	✗
Utility	Fingerprint Gesture	✓	✗	✗	✗
	App Locker	✓	✗	✗	✗
	App Usage Tracing	✓	✗	✗	✗
	Browser Usage Tracing	✓	✗	✗	✗
	TextView Mapping	✓	✗	✗	✗
	Notification Replay	✓	✗	✗	✗
	Smart Reply	✓	✗	✗	✗
	Auto Permission Grant	✗	✓	✗	✗
	Password Auto Fill	✗	✓	✗	✗
	Web Control	✗	✓	✓	✗
	(Un)Installation Protection	✓	✗	✗	✗
	Auto Uninstallation	✗	✓	✗	✗
	Deep Clean	✗	✓	✓	✗
	Battery Save	✗	✓	✗	✗
	Global Menu	✗	✗	✗	✓
A11y	Screen Reader	✓	✗	✗	✗
	Speech to Text	✓	✗	✗	✗
	Facial Access	✗	✓	✓	✗
	Gesture Access	✗	✓	✓	✗
	Voice Access	✗	✓	✓	✗
	Switch Access	✗	✓	✓	✗

✓: uses pattern; ✗: does not use pattern

*Pattern **P1**: retrieve information $\Longrightarrow$ accessibility app operation*. The accessibility apps digest the retrieved information about other apps locally, but do not trigger any global/local accessibility action. For example, a screen reader app gathers screen texts and then processes this information in a separate `TextToSpeech` component to read it aloud.

*Pattern **P2**: retrieve information $\Longrightarrow$ node action*. Here, first a node is selected based on information retrieved from the accessibility framework (e.g., locating a specific button) and then an action is triggered on that specific node (e.g., clicking). For instance, a facial access app that allows controlling the device via facial and head gestures can perform a click on a button to which the users points with such a gesture.

*Pattern **P3**: retrieve information $\Longrightarrow$ global action*. Different from pattern P2, information gathered from the accessibility framework about another app is used to trigger a global action. One typical scenario is a switch access app that captures the "home" key event from an external keyboard and then performs the global action to go back to the home screen.

*Pattern **P4**: accessibility app operation $\Longrightarrow$ global action*. In this pattern, the app triggers a global action purely based on app-internal results but without first retrieving any information about other apps from the accessibility framework. For instance, a soft key mapping is one example for this pattern.

Summary: Table 3 shows the mapping between usage scenarios and the integration patterns for different types of apps. From those results, we can see that scenarios from different categories can have the same API integration pattern. For instance, silent app installation, deep clean, and voice access share patterns P2 and P3. This makes a static detection of accessibility API misuse based on the integration pattern infeasible. Thus, also heuristics based on which APIs are being used—a common technique for malware detection—cannot sufficiently distinguish the different app types purely based on the observed API usage patterns.

3.4 Complete Accessibility Pipelines

Table 3 shows the high-level API-based patterns for interacting with the accessibility framework, which contain both retrieving data (*Patterns 1,2,3*) and triggering actions (*Patterns 2, 3, 4*). Since different app types cannot be distinguished at that abstract level, we now take app-specific contexts around those patterns into consideration and zoom in to apps to investigate the various events that trigger access to the accessibility framework, how data retrieved from the accessibility framework is used, and to which sinks such data flows. For simplicity, we call those app-specific combinations of triggers and usage the apps' *accessibility pipelines*. By comparing the pipelines of malicious applications and benign applications of the accessibility framework, we can pinpoint further similarities and differences between different app categories. The results of investigating the accessibility pipelines for different app types and scenarios are summarized in Table 4. We explain this table in the following, when we discuss the similarities and differences between malicious apps and a11y apps after comparing their triggers and intentions.

Similarities: 1) Although the triggers of the two app categories vary a lot, the commonality is that all apps determine trigger events themselves. Here, *target app operation* means that an app that is monitored with the help of the accessibility framework performs a specific operation (e.g., comes to foreground on screen), while in the remaining triggers the accessibility app reacts to specific stimuli from the user (e.g., finger or facial gestures) or it reacts to arbitrary custom logic (e.g., auto-start when service is registered). In any case, evaluation whether a trigger condition is met resides entirely within the apps. **2)** We found that 2 out of 4 prominent intended operations in a11y apps overlap with the intended operations in malicious apps: voice access provides voice controlled *text editing* support, which overlaps with the *text input* in malware that mimics user interactions in e-banking

Table 4: Accessibility pipelines for different app types and scenario

	Scenario	Trigger	Intention
Malicious	Content Eavesdropping	Auto enabled	Send to remote
	Phishing	Target app operation	Load a phishing page
	Process Persistence	Target app operation	Back home
	Silent Installation	Ad Click	Click specific buttons in specific app
	Silent Privilege Elevation	Auto enabled	Click specific buttons in specific app
	E-Banking Fraud	Auto enabled	Text input & click specific button in specific app
A11y	Screen Reader	Finger Select	Read text aloud
	Speech to Text	Auto enabled	Enable shortcut button
	Facial Access	Camera detection	Screen navigation
	Gesture Access	Finger gesture	Screen navigation
	Voice Access	Microphone detection	Screen navigation & text editing
	Switch Access	Hardware Keyboard	Screen navigation & text editing

fraud; and facial access provides screen navigation through a camera-based mouse that performs *button clicks*, which are also used by malware for, e.g., silent package installation and granting permissions. **3)** Although the intentions of screen reader, voice access, and content eavesdropping are not the same, all of them require raw data processing within the app. Hence, the raw data usage is opaque without precise data flow analysis and constraints. This also affects utility apps. For instance, McAfee Safe Family transmits user web and app usage tracking data to their server to support multi-device parental control—behavior that uses the accessibility framework similarly to content eavesdropping malware.

Differences: 1) We noticed that although some apps from different categories share the same intentions, a11y apps usually require *more* powerful accessibility functions. For example, the silent installation scenario requires clicking specific buttons in the settings app, while facial access supports users in clicking *any* button in *any* app on screen for navigation. That means, in fact, malicious apps can be easily over-privileged without raising immediate suspicion. **2)** Both benign and malicious apps require raw content processing within other components of the apps, but their *final* data destinations are different. For example, we found audio as data sink for screen reader, UI as the destination for voice access text editing, and network interface as sink for malicious content eavesdropping. **3)** By comparing the triggers of the pipelines, we found that malicious apps are more likely to perform operations silently or against users' intentions. Three of the malicious pipelines are *auto enabled* after accessibility service activation. No user involvement is needed. The other three triggers react to specific user operations on itself or third-party apps (similar to a11y apps), however, the reaction violates the users' expectations (e.g., a phishing page is shown). In contrast, triggers in benign apps are more likely to be user-explicit and the corresponding reactions are always in conformity with user intentions. For example, switch access clicks the same buttons as silent installation, but this click action is explicitly triggered by the user through a keyboard press.

Summary: The fact that a11y apps need a more general access to accessibility features (e.g., being able to press any button in any app) prevents a simple least-privilege policy on access to the accessibility framework in order to constrain misuse of accessibility features. Further, the comparison shows that the pipelines of different app categories share similar triggers and actions, thus, like API patterns (see Section 3.3), differentiation of app types purely on only concrete triggers or concrete actions is not feasible. The crucial difference between the app categories that we find is that driven by the category of the app, the complete pipeline is distinguishable when being able to detect the combination of which trigger lead to which action or data leak. For instance, a screen reader has full access to all screen content but only needs the audio API as a data sink to read discovered texts and labels. Or, a facial access app needs to click an arbitrary position that was determined from the user's head movement in the camera feed. Unfortunately, all apps evaluate their trigger conditions themselves and the accessibility pipelines in stock Android are opaque to any fine-grained enforcement of control and data flows.

This leads us to our key insight for our solutions.

4 Key Idea and Threat Model

From our study, we learned that benign accessibility apps distinguished themselves from malicious ones through different data destinations in combination with explicit user-consented node actions, both of which are dependent on the purpose of the a11y app. Benign accessibility apps usually gather user intentions through either on-device sensors or peripherals. Then they take advantage of accessibility features to either perform specified UI operation based on user intention (e.g., clicking a button) or collect necessary user-requested information from

the application framework and other apps. These gathered sensitive information may finally be consumed by components that provide feedback to the user (e.g., audio output). Thus, we define *privacy* in the context of our work as *data retrieved via the accessibility framework should not leak without user authorization and all node actions should be authorized or triggered by the user or at most be inefficiently misused.*

In consideration of those insights, a potential privacy-enhanced accessibility framework should 1) associate the UI operations by an `AccessibilityService` with user intentions to avoid (covert) malicious node actions or at least withhold crucial information for efficient, malevolent node actions; and 2) prevent the on-screen information of apps that is gathered by an `AccessibilityService` from being misused by malicious accessibility apps (e.g., unauthorized leakage of sensitive information). To illustrate, consider Figures 3 and 4. The accessibility app in Figure 3 acting as a supposed screen reader can consume textual information from the accessibility framework as input and can write arbitrary output streams to audio sinks. To avoid the screen information from leaking or the app from issuing malicious node actions, it should not be allowed to issue node clicks or use any other output channel (i.e., least-privilege). Thus, it can work as intended as a screen reader while preventing sensitive data leakage or surreptitious interactions with other apps. Similarly, the accessibility service in Figure 4, acting as a facial access app, can receive arbitrary input from other components of the accessibility app (e.g., results of processing video data or motion sense API for gesture recognition), but should only issue clicks "blindly" to certain UI elements or global events. That means the coordinates for button clicks should come from the video processing component, which in turn can only consume camera feeds, but the app should not be able to analyze the screen content otherwise. Then, since then the app cannot efficiently explore other apps' UI since it lacks feedback about screen hierarchy, misusing accessibility features for maliciously installing apps or granting permissions is impeded.

Key idea: The key idea of our solution, whose implementation we present in the following Section 5, is **1)** to treat the accessibility pipeline of accessibility apps as a sequence of steps, such as trigger detection, local processing, and output streams or node actions; and **2)** to redesign the accessibility framework such that those steps are made explicit and each step's privileges and I/O can be individually governed by a least-privilege privacy policy, however, the content of each step is treated as a blackbox. By keeping the overall pipeline in mind when authoring the privacy policy, we establish a control over the possible data flows of accessibility apps. With a suitable policy that allows benign flows to proceed while preventing potentially malicious flows, privacy protection and enabling accessibility services do not need to be mutually exclusive anymore.

Threat model: We assume that an accessibility app is malicious, meaning that all code, even when divided into individual sandboxed steps, can *collaboratively* still be malicious and steps be tailored to each other to use their individual access rights and I/O to implement an attack (i.e., unauthorized action via the accessibility framework or leakage of data obtained via the accessibility framework). Picking up the example in Figure 4, although the video processing component cannot inspect the screen content anymore to detect buttons, it could send hard-coded coordinates for click events that are independent of the camera feed in order to trigger clicks at coordinates desired by the attacker. We discuss the efficiency of our solution under this threat model in Section 7.

Figure 3: Example sandboxing for screen reader

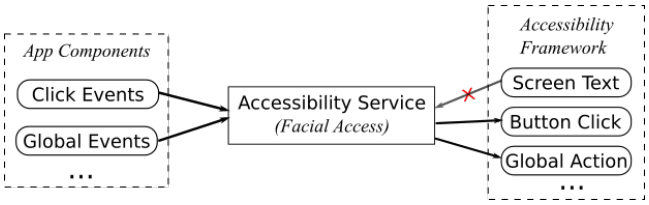

Figure 4: Example sandboxing for facial access

5 Privacy-Enhanced Accessibility Framework

In the following, we present the design concepts and implementation to realize our idea for constraining misuse of the accessibility framework.

5.1 Overview and Design Concepts

The key idea in our solution is that we treat the accessibility pipeline as a sequence of connected, individual steps and apply flow constrains to control the data flows along the pipeline to prevent unauthorized data leaks or actions. We categorize the steps of those pipelines into three types of code modules that are chained (see Figure 5): a *frontend module (optional)* to gather user intentions (e.g., from sensors or peripherals), an *accessibility module* to perform UI operations or retrieve sensitive information via the accessibility framework, and a *backend module (optional)* that creates the output of the pipeline (e.g., audio or text output). Those types have been directly derived from our previous observations about how a11y apps operate and we find them sufficient to implement the ac-

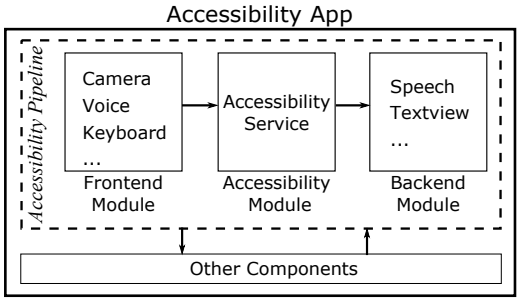

Figure 5: Accessibility pipeline with sandboxed modules.

cessibility pipelines with minimum-function, least-privilege steps. Although such a logical pipeline exists in real accessibility apps, clearly distinguishable modules do not necessarily exist in current apps and their logic is commonly mixed together in app components. We demonstrate in Section 6 how accessibility apps can be retrofitted to our solution. To prevent misuse of the accessibility API in this pipeline, we transfer design concepts for privilege separation and information flow control to our solution. In particular, we found strong parallels to *opacified computation* for IoT apps [21] and to *recognizers* in augmented reality data processing pipelines [28].

Privilege separation: We implement privilege separation of the involved modules, a common practice in other areas of privacy protection on Android, such as constraining third party libraries [26, 35, 37]. By default, all code running within the same app sandbox (i.e., under the same UID in Android) would share the same privileges. Thus, to privilege-separate untrusted code, it is moved into a separate sandbox in form of another UID under which it executes with a distinct set of permissions and access rights. This establishes a clear boundary between sandboxed (or *quarantined* [21]) code modules and allows access control at the process boundaries. Further, it allows control over the interactions between modules that are in separate sandboxes. We transfer this idea to the accessibility framework by composing the pipeline of actual, distinguishable code modules in their own sandboxes. Thus, we can control to which resources or APIs each module has access and designating each module for a certain step in the pipeline makes the overall process more transparent. No module by itself should have enough privileges to conduct the malicious operation. For instance, if a backend module of a text-to-speech app has to produce audio output, we allow this module to only access Android's audio API but not leak any data to the filesystem, other modules, or network sockets.

Information flow control: To build the pipeline, modules must interact with each other in a coordinated fashion. For instance, an accessibility module could accept screen coordinates as input and will output the on-screen information of the *node* (reference to UI element) at this particular location, which another module might receive to operate on (e.g., read out text elements of the UI element). One way to build these pipelines would be with direct IPC connections between modules. However, this would necessitate that the I/O interfaces of modules are tailored to each other, which would make the setup inflexible (e.g., if a frontend module could provide data to several kinds of accessibility modules) at no apparent security benefit. Instead, in our design, components of the accessibility app that are outside the pipeline connect modules and orchestrate the pipeline (e.g., forward the data between modules), which only requires each module to expose their own IPC interfaces to those app components via newly introduced I/O functions.This creates the risk that private data can leak to the components of the accessibility app that are not sandboxed or that those components can modify or counterfeit data exchanged between modules. To solve this problem, we take inspiration from *opaque handles* [21]: hidden references to raw data that are associated with a taint label and that can only be dereferenced within a sandboxed module. By only releasing handles to the orchestrating components outside the pipeline we prevent leakage of potentially private data to code that is not sandboxed and protect the integrity of that data from modifications by code outside the pipeline. By tainting the handles with the tag of the code module that output the data and checking those taints when handles are given as input to another module, we can ensure the authenticity of the received data and, further, can enforce simple flow constraints that govern how the modules have to be chained together. Originally [21], the taint labels also propagate to the taint label sets of module sandboxes and are forwarded to outgoing handles. That was necessary, since multiple flows might converge at a module and the context of the sandbox and of its outgoing data need to be distinguishable. Our design is simpler, since we have only a single flow in the pipeline and hence do not need to keep taint sets on sandboxes. Moreover, in contrast to the original work, we noticed that in some pipelines non-privacy-critical data could be released to the host app to allow, for instance, customizations (see Section 6.2 for such a scenario). Thus, our policy supports specifying that raw data can be released to components outside the pipeline by dereferencing the handle. To ensure the integrity and origin of all data, our solution allows only handles to be passed as input arguments to other modules.

Recognizers: Lastly, we borrow the concept of *recognizers* used to limit sensitive data sharing in augmented reality data processing pipelines [28]: in place of getting raw video data, augmented reality apps subscribe to the output of certain trustworthy video processors (e.g., object recognizers) and only receive the minimal amount of data necessary for their operation. We noticed a similar setting in accessibility apps. Accessibility apps can depend on a pre-processing of raw data from sensors or peripherals, e.g., the camera. For

instance, a facial mouse detects with the camera the user's head movements and gestures, and maps this to screen coordinates and click events. This would be done in our solution in the frontend modules (see Section 6.2). Although our threat model assumes all modules can be malicious and the output of the frontend module is generally not trustworthy, it is not unreasonable to assume that also scenarios exist in which the frontend module could be pre-installed or be coming from a trusted source, similar to recognizers in AR data pipelines that move common pre-processing to trusted system-provided component. A crucial benefit of a trusted frontend module in accessibility pipelines is that it provides a trusted source for detecting user intentions. In our design, we use this concept of recognizers by recording the outputs of frontend modules and later comparing them against the parameters of node actions. If the frontend is a trusted recognizers, this allows verification of node actions and to link user intentions with node actions.

5.2 Implementation

Figure 6 gives an overview of our implemented solution. We extend Android with a new service *PASManagerService* and its corresponding UI application *PASServer*. *PASManagerService* is the core component in our implementation. It provides accessibility apps with a new set of APIs to orchestrate their accessibility pipelines. It works as a central accessibility event dispatcher that bridges between Android's original accessibility system service `AccessibilityManagerService` and the accessibility modules of client apps, i.e., client apps that want to make use of a11y features use our *PASManagerService* instead of the default `AccessibilityManagerService`. The *PASManagerService* itself is a system-side client (*AMSBridge*) to the `AccessibilityManagerService`. We implement the flow control for accessibility pipelines within the *InfoFlowController* of the *PASManagerService*.

In the following, we will first introduce the details of the system-side *PASManagerService*, *PASServer*, and their components (Section 5.2.1). Then the new accessibility APIs and client-side integration will be introduced (Section 5.2.2).

5.2.1 System-Side Components

PASManagerService consists of three key components: *CommunicationManager*, *AMSBridge* and *InfoFlowController*.

CommunicationManager is the IPC communication hub between host apps, the modules in their pipelines, the `AccessibilityManagerService` (via the *AMSBridge*), and a new settings app *PASServer*. We use the standard Android proxy-stub concept for Binder IPC, where every client to the *CommunicationManager* uses a *PASManager* to call the *CommunicationManager* and to receive callbacks from *CommunicationManager*.

The host app and its modules also exchange data via the *CommunicationManager* with each other. If the host app and its modules could communicate directly with each other, this would necessitate that all opaque handles are set as part of the IPC communication (e.g., within Binder). This would be a very invasive change to a fundamental component of Android. By prohibiting direct communication between the host app and its modules as part of the modules' sandboxes, using a custom permission[4] unavailable to third party apps, and ensuring policies that prevent modules from leaking data to locations readable by the host app or other modules (e.g., SD card), we force modules and their host app to communicate via *CommunicationManager* with each other. Hence, while modules can be chained together, this solution ensures that they can only be chained through the *CommunicationManager* as a channel controlled by our framework. This places *CommunicationManager* in the position of a reference monitor to enforce information flow control (see further down).

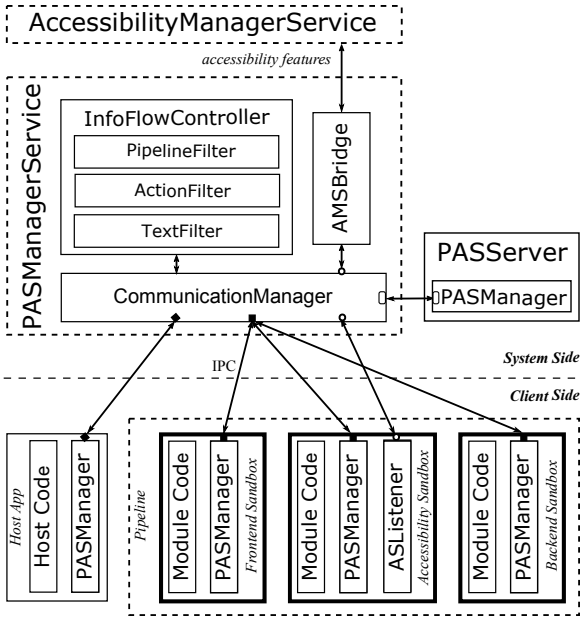

Figure 6: Privacy-enhanced Accessibility Framework

Accessibility modules additionally need to communicate with the `AccessibilityManagerService` to make use of accessibility features. Instead of direct access to the `AccessibilityManagerService`, the *CommunicationManager* together with the *AMSBridge* bridges this communication. They provide to the accessibility module in a new manager class *ASListener* as much of the vanilla `AccessibilityManagerService` API as possible in order to reduce the effort of migrating apps from the original framework to our solution. In turn, *AMSBridge* is registered as an event listener to the original `AccessibilityManagerService` and dispatches these events to registered modules or forwards requested actions

[4] A future version of our solution could also use new SELinux types.

from the modules to the `AccessibilityManagerService`. This puts *AMSBridge* into a great position to enforce access control on the accessibility features used by modules.

Lastly, *PASServer* is a new settings app for our solution to assist users with accessibility feature management. Users can en-/disable a pipeline or (de-)activate the centralized accessibility service through this app.

InfoFlowController realizes flow constraints on the communication passing through the *CommunicationManager*. It implements three types of flow constrains: *PipelineFilter*, *ActionFilter* and *TextFilter*. *PipelineFilter* implements the opaque handles and taint-based flow control. It maintains a set of unique identities for all modules as well as the host apps, and it keeps a mapping between module outputs and their handles and taints (i.e., identity of module that created the output). Thus, when a module sends an output to the host app, the data will be replaced by a new handle and the data be stored in *PipelineFilter*. The host app cannot use the handle to modify the referenced data. Every time the host app sends a handle to a module as input and the corresponding data is supposed to come from a specific other module as output, *PipelineFilter* uses the stored taint to validate this claim or otherwise abort the release of the data as input to the receiving module. Similarly, it releases raw data to the host app if the policy allows this and the host app requests dereferencing a handle. However, only a handle can be passed as input to another module, hence, integrity and origin of released data is always ensured when being further processed in the pipeline.

TextFilter implements a similar control but for textual elements within `AccessibilityNodeInfos` returned from *AMSBridge* to modules. It replaces the plain text with a random UUID before sending the node info back to the host app. This UUID and plain text pair is stored in a map in *TextFilter* and only on input to an authorized module *TextFilter* releases this text. Thus, if `AccessibilityNodeInfos` is released to a module, *TextFilter* can decide whether that module is authorized to also receive textual content that could be privacy sensitive. Modules that are not in need of such information, e.g., because they only need the node for information about screen layout, can thus operate with lower-privileges. This can be easily applied to other content besides textual information, however, we have not encountered the need to hide other content yet.

ActionFilter validates the user intentions for action events when the frontend module is trusted, i.e., is a trusted recognizer component. *ActionFilter* records the output of frontend modules, e.g., the coordinate of a UI element that the user wants to click. Once *AMSBridge* receives a call to perform an action from a module, it asks *ActionFilter* to validate if the target UI conforms to the user intention recorded before. If the frontend module is trusted, a successful validation links the action to the user intention. Thus, a pipeline with a trusted frontend module is hindered in issuing actions that were not authorized (triggered) by the user. If the frontend module is not trusted, *ActionFilter* cannot help, since the frontend module and the accessibility module could be colluding to issue malicious actions.

5.2.2 Client-Side Integration

Modules are started by the *PASManagerService* very similarly to regular app sandboxes and their launch establishes a bi-directional communication between a module's process and the *PASManagerService*. When *PASManagerService* launches a module's application sandbox, it already receives a Binder reference to this process from Android, which allows *PASManagerService* to send messages to the module. After the module's application sandbox has been started and the module's code been loaded, it requests a Binder reference to the *PASManagerService*, which is encapsulated in a *PASManager* and allows it to send messages to *PASManagerService*. With this two Binder references a bi-directional communication is established. Modules that make use of accessibility features additionally register an *ASListener* with *PASManagerService* through which they can receive accessibility events and issue actions. The host app also has a *PASManager* that allows it to issue commands to *PASManagerService*, e.g., invoke modules and pass/receive data handles via *CommunicationManager*.

For a full-fledged implementation, we envision that accessibility apps carry their modules as payload (separate dex files) and register them during installation in the privacy-enhanced framework, similar to how prior works proposed sandboxing third party libraries [26, 37]. Alternatively, modules could be provided as standalone packages on a market and accessibility apps declare which ones should be retrieved and installed into the pipeline of the app, similar to emerging app-in-app ecosystems [13, 32]. In any case, the host app declares the modules in its manifest, where it also states their required privileges and the flow policy, which can hence be inspected and approved (e.g., by the user during app installation). For our prototypical implementation, we create the module sandboxes as dedicated apps as a fixed part of our modified Android image in order to test functional correctness and evaluate our solution in terms of performance overhead (see next Section 6).

6 Evaluation

In this section, we take two open-source accessibility apps, TalkBack [23] and EVA Facial Mouse [16], as examples to test the performance of our solution and show how to enhance the privacy protection in accessibility services.

6.1 Case Study: TalkBack

We use Google's official screen reader app for visually impaired users, TalkBack, to evaluate the protection of on-screen text against leakage. This app has been installed more than

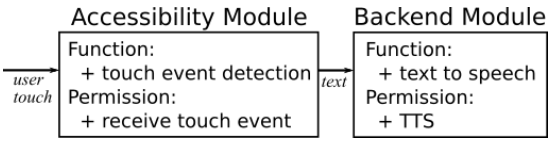

Figure 7: Screen Reader accessibility pipeline

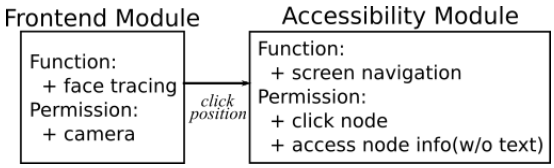

Figure 8: Facial Mouse accessibility pipeline

5 billion times according to Google Play (see Table 7 in Appendix A). TalkBack is a complex app containing multiple modules and multiple preference settings. We focus on its core module—touch-based screen reader with default settings. The accessibility pipeline for this module can be seen in Figure 7: the app has an accessibility module and a text-to-speech backend module. Once the user touches the screen, accessibility module collects the textual information about the touched node from the accessibility framework. That information is passed to the text-to-speech component that reads the text aloud via Android's TTS service.

Migration: We build the accessibility module by moving the touch detection logic, which includes accessibility event processing and cursor controls, to an accessibility module in the pipeline. When a touch event is detected, the module outputs the textual information about the UI element at the touch coordinates. Similarly, we establish the backend module here by moving TalkBack's original text-to-speech code to a backend module and exposing the necessary interfaces like *isSpeaking(), speak(String)* and *shutdown()* to the host app. To orchestrate this pipeline from the host app, we replace the original local calls in the host app with calls to the API exposed by the two modules (i.e., callbacks for text output from the accessibility module and calls to, e.g., *speak()*). Thus, the host app can forward the text from the touch detection to the text-to-speech logic, each executing in their own sandbox. We made 3k+ LOC changes on a code base 27k+ LOC for this migration.

Privacy enhancement: In our design, all modules and host app are running in their own sandbox with distinct permission sets. The accessibility module has the privilege to receive touch events but nothing else, thus, it is unable to scavenge through another app's screen content and leak it. The backend text-to-speech module can only access the TTS API of Android to play the result of the text processing, but cannot leak the text to another sink (e.g., network socket or filesystem). Neither module has the privilege to issue node actions, e.g., pressing buttons in an unauthorized way. By only releasing handles for the output of the accessibility module to the host app, the host app cannot inspect the textual content, which might be privacy-sensitive. Using flow control on those handles, we ensure that the backend module only receives data as input that was generated by the accessibility module.

6.2 Case Study: EVA Facial Mouse

We use EVA Facial Mouse app to confirm the feasibility of restricting the misuse of node actions. The app provides a virtual mouse that is controlled by facial movements, e.g., if the user cannot use their hands. The accessibility pipeline in Figure 8 contains a frontend that uses the camera to capture user intentions and an accessibility module to perform user-intended clicks. The frontend module has access to the device camera and when it detects a head gesture that indicates a click, the coordinates of the virtual mouse on screen will be output. Based on the coordinates, the accessibility module can retrieve the target node from the accessibility framework and perform the actual click on this node.

Migration: We put the app's original camera-tracing code to the frontend module and expose necessary callback interfaces, like *onMouseEvent(location, click)*, to the host app. We also allow the necessary accessibility features for node detection based on coordinates and performing click actions on nodes to the accessibility module. As for TalkBack, we replace the original calls to the camera and accessibility features inside the host app with calls and callbacks to/from the two modules, such that the host app orchestrates the pipeline and forwards data between the modules. The frontend module traces the user's head movements and outputs the corresponding mouse tracing events, i.e., coordinates of the mouse cursor. To maintain the look and feel of a mouse cursor the host app can in this case dereference the handle to the coordinates data to draw a mouse cursor on screen and also easily allow the user to customize the cursor (e.g., size, color). When a click event is detected by the frontend, the host app invokes the accessibility module with the coordinates for which to retrieve the UI element and to which to issue a click. We changed 1k+ LOC on a code base 9k+ LOC for this app.

Privacy enhancement: Again, the modules and host app are in separate sandboxes with distinct permission sets. The frontend module has access to the camera, but nothing else. The accessibility module can retrieve nodes from the accessibility framework based on screen coordinates and issue click actions to those nodes. Applying the text filter to the node infos released to the accessibility module, we prevent that this module learns the content of the UI element (e.g., button label or content of a text view). Further, neither module can investigate the screen content and hence produce targeted clicks, e.g., to navigate the settings app without user approval

Table 5: Performance test results

Application	Original (ms)	Migrated (ms)	Δ
TalkBack	10.75±1.35	18.55±2.26	7.80 (73%)
EVA Facial Mouse	15.60±1.26	29.50±3.55	13.90 (89%)

Intervals for 95% confidence

to grant permissions or install new apps silently. However, the modules could issue node actions "blindly" and without feedback, e.g., the coordinates are hard-coded in either module, which could succeed in navigating the device surreptitiously when the coordinates fit to the current screen-size and the device screen was in a well-known state (e.g., home screen). A countermeasure to this would rely on our Action Filter, i.e., assuming that the frontend module is trusted and that the coordinates output by this module can be validated against the coordinates of a node when the accessibility module issues a node action. In that case, forging or manipulating coordinates would not succeed.

Further, it should be noted that this app could misuse the camera permission to spy on user input. The trace of cursor coordinates and click events allows the app to monitor where on screen the user clicked. While our solution prevents the app (concretely, the accessibility module) from misusing the accessibility framework to learn *and* leak the information about clicked UI elements, the app can use side-channels to infer this information independently of the accessibility features. For instance, if the host app has a valid assumption about the screen content (e.g., an onscreen keyboard), coordinate trace together with click events would allow the host app to derive which input the user gave (e.g., mapping coordinates with click to the screen position of keys of the soft-keyboard). However, this is purely an abuse of the camera permission and **not** of the accessibility framework. Although being outside of our threat model, our solution could offer a potential solution in this concrete case as well by moving the cursor rendering to a module that cannot leak the derived information and only releasing non-dereferencable handles to the host app.

6.3 Performance Overhead

Our framework is deployed on Android v8.1 on a Pixel 2XL device. We use the two migrated apps to estimate the performance impact of our framework. We utilize microbenchmarking to measure the overhead. Since the runtime of an accessibility operation is affected by complex user interfaces (e.g., time to find a specific node), we develop a dedicated test app with only one `TextView` and one `Button`. Thus, our measurements approximate the upper bound for the overhead, since we minimize the runtime for common operations and thus give more weight to the overhead. We run the test 20 times for the original and migrated versions of the TalkBack and Facial Mouse app. Table 5 summarizes the results.

TalkBack Result: We measure the time the screen reader module needs to read the `TextView` text aloud after a user touched on it. We start the measurement as soon as a touch event is detected. The measurement completes when the text-to-speech's `speak` instruction is executed. The average overhead for the migrated app is 7.80ms or about 73%.

EVA Facial Mouse Result: We measure the time from click generation in the frontend module until the `onClick()` callback of the target button is triggered. The result shows that the induced overhead is 13.90ms or about 89%.

Summary: Although the relative overhead is high, we want to a) note again that this is an upper bound since our test app optimizes the common operations and weights the overhead higher and b) point out that those affected operations occur in many cases with low frequency and the absolute overhead in our measurements is well below the average human perceptible latency. Thus, while overhead due to the additional IPC between modules and host app was expected, we think the overall overhead is still in an acceptable range.

7 Discussion

7.1 Limits and challenges

Sandboxing the modules in the pipelines and controlling to which APIs (sources and sinks) they have access together with the opaque, tainted handles for data exchanged between modules provides control over data flows in the same fashion as in similar solution in IoT settings [21]. Thus, we are facing some similar challenges as well as new challenges in protecting the users' privacy.

Indistinguishable data flows: Like other solutions, we treat the modules as blackboxes and control the data flows to and from modules. But we cannot control how the modules generate their outputs, and we have only limited means to control the exchanged data (e.g., text filters). As a result, if the data flows including sources and sinks are indistinguishable between a11y and malware apps, our solution can likely not prevent unauthorized leakage—although, we did not find an example in our study of malicious accessibility apps where this was case, as shown in Section 3.

Authorized node actions: Further, we face the additional requirement that not only the unwanted leakage of data should be prevented but also unauthorized node actions. The challenge is to connect a node action with a user action. Our current solution tries to validate the parameters of actions (i.e., action filter) but at least limits the effectiveness of malicious node actions by limiting the data on which actions are based (e.g., preventing the reconnaissance of the screen content, see Section 6.2).

Off-device processing: Accessibility apps can depend on off-device services, for instance, for image or audio processing. As for other information flow control solutions, like [19, 21, 25], the device boundary is a hard boundary for

our enforcement. However, by strengthening the sandbox (see below) our solution can provide control over the network destinations (e.g., URL) to which modules can connect and, hence, ensure that only trusted, user-approved services are used as part of the pipeline.

Side-channels: We cannot exclude side-channels that can be used by modules to secretly exchange data or that modules use to conduct reconnaissance (e.g. [15]).

Summary: While the ideal result would be to prevent all potential leakage of private data and all malicious node actions as described in Section 3.4 while at the same time upholding all benign, legitimate a11y app functionality, there currently exist potential cases in which malware and a11y apps are not distinguishable for our policies. However, compared to stock Android's all-or-nothing protection, our solution provides a trade-off where required assistive apps can function while the potential for misuse of accessibility features is drastically reduced. For instance, a user that requires a facial mouse and allows the corresponding policy might still fall victim to "blindly" injected click events but none of the other malware could operate as usual, such as e-banking fraud and content eavesdropping.

7.2 Strengthening the sandbox and IFC

An obvious improvement to our solution would be better information flow control along the *entire* data flow even *within* sandboxes. This could help to validate that node actions indeed depend on input generated by user actions or that leaked data does not depend on private data. On Android, taint tracking [19, 25, 40] techniques have been proposed to this end. Unfortunately, taint tracking suffers from a hard to defend reference monitoring. A malicious app can always win by dropping the taint (e.g., native code, indirect control flows) and currently these solutions would only apply to curious-but-honest modules. Further, we currently consider every module to be used only in one pipeline, which makes the information flow control between modules (e.g., the tainting of the handles) and the non-interference within a single module trivial. A more advanced scenario could allow the re-use of modules in different, simultaneously executing pipelines (e.g., re-using pre-installed modules) and in that case an integration of distributed information flow control (e.g., [33] or [21]) would be necessary to ensure non-interference.

Our sandboxes rely on the stock Android mechanisms (i.e., UIDs with permissions). However, those only provide a very coarse-grained access control to the application framework or filesystem. Since the way how we start modules from the *PASManagerService* resembles the procedure how virtualized apps are started [13], we could integrate "module virtualization" in the future where the *PASManagerService* takes the role of the broker and puts modules into an isolated process (the least privileged execution environment that stock Android supports). Similarly, frameworks [24] for more fine-grained and context-sensitive access control policies could be integrated to provide better control over the functionality and data each module can access from the Android API.

7.3 User approval

In our current prototype, we assume that the user approves the policies via the *PASServer* app or writes custom policies that satisfy their individual privacy preferences via this app. We are aware that the history of permission prompts on Android has shown that users are not capable of this and that there is ongoing research into improving the user experience (e.g., seminal work by Porter Felt et al. [20]). Since the user in our solution even has to approve *flows* and the goal of this work is to show that a11y and privacy protection do not have to be mutually exclusive *per se*, we defer the question of how to improve the user experience in approving or configuring policies in our solution to future work.

7.4 Threats to Validity

We specifically looked for collections of malware samples in actively maintained, popular GitHub projects. However, we cannot guarantee that those collections are the most representative ones for malicious *accessibility* apps. Further, we searched for supposedly benign a11y and utility apps by keyword search among the top apps on Google Play. Thus, we think our collection of utility apps is representative. Unfortunately, the number of a11y apps is limited and many of the top apps are written by Google. Thus, there might be a bias in our collection of a11y apps towards Google's software engineering practices.

7.5 Utility apps

Our solution reduces the chances for misuse of the accessibility API while preserving the functionality of *a11y apps*. However, *utility* apps might depend on pipelines that differ from those of a11y apps when providing innovative usages of the accessibility framework. For example, password managers take advantage of it to fill-in passwords. Since this is an abuse of the accessibility features, Google introduced the auto-fill framework [9] as an alternative to support password managers. However, for cases in which no alternative framework or API exists in Android, it remains an open question how to support those utility apps while maintaining a high level of privacy protection or whether those use-cases can be generally implemented in accessibility pipelines as well. For instance, the pipeline for a utility app that automates tedious user actions through sequences of automated button clicks might not be distinguishable enough from malware secretly navigating user interfaces.

7.6 Other attacks and privacy issues

Apart from the attacks we analyzed in Section 3, other attacks might leverage the accessibility framework as a building block or stepping stone. For instance, the accessibility API might be used for reconnaissance. A typical example is a phishing attack, in which a malicious app uses the accessibility framework to monitor the name of the foreground activity and time the launch a phishing activity. However, in such cases, the accessibility framework is often just the path of least resistance to gather information and alternatives exist (e.g., foreground activities can also be identified via side-channels [15]), thus we did not separately study and evaluate those attacks in our work. Further, our defense relies on proper policies, thus, if the user is involved in setting and granting them, we exclude attacks against the user from our threat model, such as deceptive overlays [22, 42].

Lastly, there exist apps to support impaired or disabled users via crowdsourcing instead of relying on the accessibility framework. For example, camera-based assistive apps to support visually impaired users. Those apps outsource the users' questions, e.g., about their physical surroundings, to volunteers with whom the users have to share sensitive information, such as photo or video stream. Prior work [6] investigated the privacy concerns raised in using camera-based assistive apps under different scenarios. Their results confirm the request by dependent users for privacy protection in using assistive technologies, which we take as further motivation for our research although those particular cases of sharing camera data with volunteers are not covered by our work. Similarly, other data stealing attacks, such as taking screenshots or recording audio that do not rely on the accessibility framework are out of scope of what we can defend against.

8 Related Work

We briefly present and compare related works to our work on enhancing the privacy of Android's accessibility framework.

Security and privacy concerns from accessibility frameworks. Already in 2013, Kraunelis et al. [31] demonstrated a malware that utilizes Android's accessibility framework. Jang et al. [29] studied the security of assistive technologies and identified multiple vulnerabilities on four popular platforms. Their result shows that the trade-off between security, compatibility, and usability is the root cause of these vulnerabilities. Kalysch et al. [30] assessed the weakness of a11y features and proposed corresponding *developer side* countermeasures. Diao et al. [18] evaluated Android's accessibility APIs with an analysis of the framework as well as a large-scale app analysis. Their result reveals the intrinsic shortcomings in Android's current design and confirms the broad misuse of the accessibility APIs. Fratantonio et al. [22] present attacks when combining Android's `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions, thus, further highlighting the shortcomings in privacy protection of the a11y framework. Follow-up work [42] demonstrated the usage of these permissions by malicious apps. Those works highlight the existing privacy concerns in the accessibility framework, but did not create an appropriate defense. Our solution is the first work that not only looks in-depth into accessibility API usage by different types of real-world apps but also proposes a system side privacy enhancement.

Defenses against malicious accessibility apps. Naseri et al. [34] proposed a *developer side defense* against eavesdropping through accessibility features. In their work, multiple tools are implemented to detect apps that are vulnerable to eavesdropping, to automatically fix discovered vulnerabilities, and to notify users of potential accessibility service misuse. Unfortunately, this solution requires every developers' effort and makes accessibility and privacy an "either-or" choice that sacrifices the user experience of people with impairment for privacy gains. Different from it, our approach provides a better balance between privacy and user experience.

Process-based privilege separation on Android. A few solutions separate sensitive or untrusted components into isolated processes to mitigate privacy violations. Works focusing on advertisement libraries [26, 35, 37] demonstrated different solutions to isolate said libraries from their host apps and privilege separate them. Roesner et al. [36] sandboxed untrusted UI components in isolated processes to support secure UI embedding. Davidson et al. [17] provided a dedicated `WebView` service app to protect host apps from untrusted web content. Starting with Android O, Google officially put the `WebView` renderer into an isolated process [5]. Other works privilege separate entire apps, e.g., Backes et al. [13] create a virtualized environment for untrusted apps and, similarly, Bianchi et al. [14] demonstrated an approach that sandboxes an untrusted app inside a separate non-privileged context to enforce privacy and security policies. Our work transfer those concepts to the accessibility framework by sandboxing the code modules that form an accessibility pipeline.

Information flow control in IoT applications. Closest to our work is *FlowFence* [21], which introduced information flow control for IoT apps to prevent unwanted data leakage. It introduced the concepts of quarantined modules and opaque handles that we also used in our implementation. In contrast to FlowFence, our flows are very simple and linear. For instance, in FlowFence multiple flows might converge on the same module, necessesitating taint sets for modules, and modules can set custom taints on output to prevent their data from reaching certains sinks. Our modules only consume output from a single predecessor module within a short pipeline for which the user sets the policy. Thus, while we could support the same taint arithmetic and taint sets as FlowFence, this is currently not necessary and simplifies our setup, avoiding the issue of overtainting module sandboxes. On the other hand, our solution has to additionally deal with the problem of authorizing actions by modules. We addressed this by adopting

the concept of recognizers [28] and using data flow control to limit the information needed for (effective) malicious actions.

9 Conclusion

Android's accessibility framework is a powerful service intended to allow assistive apps in supporting impaired and disabled users in navigating their devices. Unfortunately, the service is also a popular building block for utility and malevolent apps that do not apply accessibility features as originally intended and might violate the users' privacy. Existing defenses in stock Android force users and app developers to choose between inclusiveness and privacy protection. To improve on this situation, we propose a privacy-enhanced accessibility framework forward. By representing a11y logic as pipelines, sandboxing every code module in a pipeline, and enforcing flow contraints, our solution allows a more fine-grained control over accessibility features and reduces the attack surface while upholding functionality of a11y apps. We showcase the feasibility of our solution by migrating two a11y apps. We also discuss shortcomings of our approach and hope this work will raise further interest into building solutions that protect a particular dependent user group.

Acknowledgments

We thank our anonymous reviewers for their insightful comments and suggestions which have helped us improve our paper.

References

[1] Android trojan steals money from paypal accounts even with 2fa on. https://www.welivesecurity.com/2018/12/11/android-trojan-steals-money-paypal-accounts-2fa/. Accessed: 2021-02-22.

[2] Google pauses removal of apps that want to use accessibility services. https://www.zdnet.com/article/google-pauses-crackdown-of-accessibility-api-apps/. Accessed: 2021-02-22.

[3] Mobile malware continues to plague users in official app stores. https://securityintelligence.com/anubis-strikes-again-mobile-malware-continues-to-plague-users-in-official-app-stores/. Accessed: 2021-02-22.

[4] Skygofree — a hollywood-style mobile spy. https://www.kaspersky.com/blog/skygofree-smart-trojan/20717/. Accessed: 2021-02-22.

[5] What's new in webview security. https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html. Accessed: 2021-02-22.

[6] Taslima Akter, Bryan Dosono, Tousif Ahmed, Apu Kapadia, and Bryan Semaan. "i am uncomfortable sharing what i can't see": Privacy concerns of the visually impaired with camera based assistive applications. 2020.

[7] Android Developer Docs. Accessibilityservice. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService. Accessed: 2021-02-22.

[8] Android Developer Docs. Accessibilityserviceinfo. https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo. Accessed: 2021-02-22.

[9] Android Developer Docs. Autofill framework. https://developer.android.com/guide/topics/text/autofill. Accessed: 2021-02-22.

[10] Android Developer Docs. View: Attribute importantforaccessibility. https://developer.android.com/reference/android/view/View.html#attr_android:importantForAccessibility. Accessed: 2021-02-22.

[11] Android Developer Guide. Build more accessible apps. https://developer.android.com/guide/topics/ui/accessibility. Accessed: 2021-02-22.

[12] ashishb. android-malware. https://github.com/ashishb/android-malware.

[13] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (SEC'15)*, 2015.

[14] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'15)*, 2015.

[15] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (SEC'14)*, 2014.

[16] cmauri. Camera based mouse emulator for android. https://github.com/cmauri/eva_facial_mouse.

[17] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. Secure integration of web content and applications on commodity mobile operating

systems. In *12th ACM Symposium on Information, Computer and Communication Security (ASIACCS'17)*, 2017.

[18] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. Kindness is a risky business: On the usage of the accessibility apis in android. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[19] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[20] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *7th USENIX Workshop on Hot Topics in Security (HotSec 12)*, 2012.

[21] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (SEC'16)*, 2016.

[22] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *38th IEEE Symposium on Security and Privacy (SP'17)*, 2017.

[23] Google. Talkback app. https://github.com/google/talkback.

[24] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A programmable interface for extending android security. In *23rd USENIX Security Symposium (SEC'14)*, 2014.

[25] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM Conference on Computer and Communication Security (CCS'11)*, 2011.

[26] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *24th ACM Conference on Computer and Communication Security (CCS'17)*, 2017.

[27] hxp2k6. Android-malwares. https://github.com/hxp2k6/Android-Malwares.

[28] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J Wang, and Eyal Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *22nd Usenix Security Symposium (SEC'13)*, 2013.

[29] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. A11y attacks: Exploiting accessibility in operating systems. In *21st ACM Conference on Computer and Communication Security (CCS'14)*, 2014.

[30] Anatoli Kalysch, Davide Bove, and Tilo Müller. How android's ui security is undermined by accessibility. In *2nd Reversing and Offensive-oriented Trends Symposium*, 2018.

[31] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On malware leveraging the android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2013.

[32] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, Xiaofeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *27th ACM Conference on Computer and Communication Security (CCS'20)*, 2020.

[33] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *25th USENIX Security Symposium (SEC'16)*, 2016.

[34] Mohammad Naseri, Nataniel P Borges, Andreas Zeller, and Romain Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. *Proceedings on Privacy Enhancing Technologies*, 2019(2):291–305, 2019.

[35] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *7th ACM Symposium on Information, Computer and Communication Security (ASIACCS'12)*, 2012.

[36] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *22nd Usenix Security Symposium (SEC'13)*, 2013.

[37] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *21st Usenix Security Symposium (SEC'12)*, 2012.

[38] sk3ptre. Androidmalware_2018. https://github.com/sk3ptre/AndroidMalware_2018.

[39] sk3ptre. Androidmalware_2019. https://github.com/sk3ptre/AndroidMalware_2019.

[40] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *23rd ACM Conference on Computer and Communication Security (CCS'16)*, 2016.

[41] Virus Total. Api scripts. https://support.virustotal.com/hc/en-us/articles/115002146469-API-scripts. Accessed: 2021-02-22.

[42] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and detecting overlay-based android malware at market scales. In *17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019.

A App Sample Sets

Table 6: Utility App Sample Set (Google Play)

Package	#Installed
com.amazon.tahoe	1,000,000+
com.antivirus	100,000,000+
com.antivirus.tablet	10,000,000+
com.apusapps.emo_launcher	100,000+
com.apusapps.launcher	100,000,000+
com.avast.android.cleaner	10,000,000+
com.avast.android.mobilesecurity	100,000,000+
com.avg.cleaner	50,000,000+
com.bitdefender.security	5,000,000+
com.bitspice.automate	500,000+
com.cleanmaster.mguard	1,000,000,000+
com.eset.ems2.gp	10,000,000+
com.eset.parental	100,000+
com.gau.go.launcherex	100,000,000+
com.italia.autovelox.autoveloxfissiemoibli	1,000,000+
com.kaspersky.safekids	500,000+
com.kms.free	50,000,000+
com.ksmobile.launcher	100,000,000+
com.lastpass.lpandroid	5,000,000+
com.lionmobi.battery	50,000,000+
com.mcafee.security.safefamily	100,000+
com.microsoft.launcher	10,000,000+
com.oneapp.max.cleaner.booster	10,000,000+
com.piriform.ccleaner	50,000,000+
com.pleco.chinesesystem	1,000,000+
com.server.auditor.ssh.client	500,000+
com.teslacoilsw.launcher	50,000,000+
com.wsandroid.suite	10,000,000+
dreamy.earth.theme.natural.launcher	1,000,000+
galaxy.iphone.hd.wallpaper.live.screen.lock	10,000,000+
ginlemon.flowerfree	10,000,000+
mobi.infolife.appbackup	10,000,000+
org.malwarebytes.antimalware	10,000,000+
panda.keyboard.emoji.theme	100,000,000+
red.love.rose.theme.valentine.launcher	1,000,000+

Table 7: A11y App Sample Set (Google Play)

Package	#Installed
com.google.android.marvin.talkback	5,000,000,000+
com.sesame.phone_nougat	10,000+
com.crea_si.eviacam.service	1,000,000+
com.google.audio.hearing.visualization.accessibility.scribe	50,000,000+
com.google.android.apps.accessibility.voiceaccess	1,000,000+

Table 8: Malicious App Sample Set (Github)

MD5	Classification (VirusTotal-Alibaba)
007ae04ac52f17d5d637f2c41658f824	TrojanSpy:Android/Banker.a30eb151
03e5d8ece783245b28cb97373e739842	Trojan:Android/Agent.3fc9b0c7
042f2f3a0df4aef0460d1ee74f1df033	Backdoor:Android/Agent.8f28ba9e
09b60aa78291e7ef8b0ddfc261afb9f9	TrojanDropper:Android/Skeeyah.a026644f
10f8097ef0db6adbed3b314055491ca4	Trojan:Android/Rootnik.efbca116
1512c3fa688ca107784b3c93cd9f3526	TrojanDropper:Android/Hqwar.657ae279
18a3c09ce58b3db05cf248730adb6bd0	TrojanDropper:Android/Hqwar.9e0b0668
2254002370c03cf14c3eabb27b3b826d	TrojanBanker:Android/Anubis.58e63764
2f07c9b2a67104f8bc08d831c8922b6a	TrojanBanker:Android/Riltok.32dfd36e
31ba565fcc1060ad848769e0b5b70444	Trojan:Android/Agent.4c52deda
39fca709b416d8da592de3a3f714dce8	Trojan:Android/Skygofree.355eb294
3b07862da0b78632d8e4486444adbbfd	Backdoor:Android/Agent.3ebcdecc
3ffedf4759a001417084c64db48b549a	TrojanSpy:Android/AndroRAT.afe389c9
4aea3ec301b3c0e6d813795ca7e191bb	TrojanSpy:Android/Donot.60880405
519018ecfc50c0cf6cd0c88cc41b2a69	TrojanSpy:Android/ZooPark.436e912a
51f388f9ca606812d7fb4d5330e42ce7	TrojanBanker:Android/Anubis.75cc2361
55366b684ce62ab7954c74269868cd91	Trojan:Android/Ztorg.6ff5f73b
5cc953f25deeff951c38a5c118a81fe9	Trojan:Android/Agent.008476da
63a56f3867ef4b4a3cf469e81496aee7	TrojanDropper:Android/Agent.ac60e49b
647f6b2503205dd1f1da5ea490b6c71f	Trojan:Android/Rootnik.977d3960
64e374807d87102cfc27489a91f8a13d	AdWare:Android/Batmob.cbf4dda9
6a3ae5a916bc109e0186b40093084a78	Backdoor:Android/Agent.63d66ab9
6c39197bb2c2fd5fc9253ed18467d0d7	Backdoor:Android/Brata.7b8ffc88
70a937b2504b3ad6c623581424c7e53d	Trojan:Android/Skygofree.f9b277e6
71b80c162001f9d2f4872f2efb7431fe	TrojanSpy:Android/AndroRAT.a2734e43
75f1117cabc55999e783a9fd370302f3	TrojanBanker:Android/Banker.4650457b
880540d10cca559f68db96314f206225	Trojan:Android/Rootnik.1fce124c
8a266e277c61ffd6afa3d15b8691b9fb	Backdoor:Android/Agent.48e611ae
8a9540fa5541054074d1efdc7729da43	Backdoor:Android/Lucbot.5aac9302
8d1f5637dc0bc76064d6f3283482a7c5	Trojan:Android/Agent.7c517cfb
8df5b22cabc10423533884da7648e982	TrojanBanker:Android/Asacub.3fc31d6b
91f0daa8cb837a9d3e815da8db999a08	TrojanSpy:Android/Banker.35c71d45
957ce53315496083a43c6765f5ed9e42	TrojanSpy:Android/AndroRAT.d9b0b7c8
9ae53ef2a4f2d40b06cc85e5c3778f48	Trojan:Android/Agent.14c930cd
a287a434a0d40833d3ebf5808950b858	Trojan:Android/Skygofree.639ce6ec
a2a921c0e8a9171300a805c5b1df78b8	TrojanSpy:Android/Banker.f9398502
a384a27681df0ed1732d4346f6c52d0a	TrojanBanker:Android/Generic.ba1d86be
a44a9811db4f7d39cac0765a5e1621ac	Trojan:Android/Agent.34c921b3
a8a8479dab8fbdee1fb058b8de97e89b	Trojan:Android/Agent.a87db02a
a962759a71f899a9bbe4d27790e91b00	Backdoor:Android/Lucbot.69116e3e
a97eb28853eeeecffb749bf49b68af55	TrojanSpy:Android/AndroRAT.6fd591ab
ac613a7dee1ee8c47321403ab8fa1372	Trojan:Android/Agent.f483d3f4
ac67f1b22d6c7812003609de284a9ad9	TrojanDropper:Android/Hqwar.20f8d210
ac92258ff3395137dd590af36ca2d8c9	Trojan:Android/Agent.c1ade2d1
c148c63c974e2312d8f847d07242a86b	TrojanBanker:Android/Anubis.65b2e27e
c580e7807fbd18106d2659af3cc58f8d	AdWare:Android/Gibdy.2f426bf5
cdf10316664d181749a8ba90a3c07454	TrojanSpy:Android/Donot.d27afe4a
d0641eb22198c346af6c22284fca38a6	TrojanBanker:Android/Riltok.b7c88ed6
d3f53bcf02ede4adda304fc7f03a2000	TrojanSpy:Android/Donot.ecf77e96
d6ef4e16701b218f54a2a999af47d1b4	TrojanBanker:Android/Banker.a4cbd698
dc74daf70afc181471f41fd910a0dec0	TrojanDropper:Android/Hqwar.ef5f2c4a
e105b0fd0eadc5db26bf979e4e96007c	Trojan:Android/Rootnik.1c8d7a29
e4187a74e6bef1a8cd30116500ed10f8	TrojanBanker:Android/Banker.3457c734
ef8493089deecbef6e459434ec7fee0b	TrojanDropper:Android/Hqwar.04dccff
ffacd0a770aa4faa261c903f3d2993a2	TrojanBanker:Android/Banker.522c0eb4