Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android

Wenrui Diao*†, Yue Zhang‡, Li Zhang‡, Zhou Li§, Fenghao Xu¶, Xiaorui Pan‖, Xiangyu Liu♮, Jian Weng‡, Kehuan Zhang¶, and XiaoFeng Wang‖

*Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, diaowenrui@sdu.edu.cn

†School of Cyber Science and Technology, Shandong University

‡Jinan University, {zyueinfosec, zhanglikernel, cryptjweng}@gmail.com

§University of California, Irvine, zhou.li@uci.edu

¶The Chinese University of Hong Kong, {xf016, khzhang}@ie.cuhk.edu.hk

‖Indiana University Bloomington, xiaopan@umail.iu.edu, xw7@indiana.edu

♮Alibaba Inc., eason.lxy@alibaba-inc.com

Abstract

The assistive technologies have been integrated into nearly all mainstream operating systems, which assist users with disabilities or difficulties in operating their devices. On Android, Google provides app developers with the accessibility APIs to make their apps accessible. Previous research has demonstrated a variety of stealthy attacks could be launched by exploiting accessibility capabilities (with `BIND_ACCESSIBILITY_SERVICE` permission granted). However, none of them systematically studied the underlying design of the Android accessibility framework, making the security implications of deploying accessibility features not fully understood.

In this paper, we make the first attempt to systemically evaluate the usage of the accessibility APIs and the design of their supporting architecture. Through code review and a large-scale app scanning study, we find the accessibility APIs have been misused widely. Further, we identify a series of fundamental design shortcomings of the Android accessibility framework: (1) no restriction on the purposes of using the accessibility APIs; (2) no strong guarantee to the integrity of accessibility event processing; (3) no restriction on the properties of custom accessibility events. Based on these observations, we demonstrate two practical attacks – installation hijacking and notification phishing – as showcases. As a result, tens of millions of users are under these threats. The flaws and attack cases described in this paper have been responsibly reported to the Android security team and the corresponding vendors. Besides, we propose some improvement recommendations to mitigate those security threats.

1 Introduction

The assistive technologies have been integrated into nearly all mainstream operating systems, which assist users with disabilities in operating their devices. It is not only the kindness of OS vendors but also the requirements of federal law [20]. On mobile platforms, tremendous efforts have been spared into developing assistive technologies. For example, Android provides TalkBack [7] for users who are blind and visually impaired. They could perform input via gestures such as swiping or dragging on the screen and listen to the feedback in an artificial voice. Other supports include Switch Access (an alternative to using the touchscreen), Voice Access (control device with spoken commands), etc.

Besides the built-in accessibility features provided by Android OS, to support the development of accessible apps, Google also provides app developers with the *accessibility APIs* to develop custom accessibility services. The mission of these APIs is to provide user interface enhancements to assist users with disabilities, or who may temporarily be unable to fully interact with a device [11]. With the accessibility APIs, an app could observe user actions, read window content, and execute automatic GUI operations, which improves the interactions between users and apps. Since these APIs are quite powerful, as a restriction, the accessibility service must be protected by the `BIND_ACCESSIBILITY_SERVICE` permission to ensure that only the system can bind to it.

On the other hand, the powerful capabilities of the accessibility APIs can be exploited for malign purposes. Previous research demonstrated a variety of stealthy attacks could be launched with the accessibility capabilities [35, 37, 43] and investigated the inadequate checks on accessibility I/O paths [36]. However, previous works focused on exploring what kinds of attacks could be achieved through a malicious app with the `BIND_ACCESSIBILITY_SERVICE` permission, and none of them touched the design of the Android accessibility framework.

Our Work. Motivated by the significant security implications of the accessibility service, in this work we perform the first comprehensive study to evaluate the usage of the accessibility APIs and the design of their supporting architecture in Android. In particular, we first conducted a large-scale study on 91,605 Android apps crawled from Google Play to measure the accessibility APIs usage in the wild. The result shows the accessibility APIs have been misused widely. Most assis-

tive apps utilize them to bypass the permission restrictions of Android OS, which deviates from the original mission.

Then, we reviewed the Android accessibility framework to investigate the fundamental reasons for misuse and the potential security risks. Finally, we identify a series of fundamental design shortcomings that can lead to severe security threats. (1) Specifically, we find that there is no restriction on the purposes of using the accessibility APIs. Any app can invoke the accessibility APIs even when the purpose is not for helping the disabled users. (2) Also, we notice that the Android accessibility architecture is event-driven. Under such design, the event receivers do not communicate with the event senders directly. The execution logic of accessibility services only could rely on the received accessibility events. However, the information contained in the events does not provide a strong guarantee to the integrity of event processing. (3) Even worse, Android allows zero-permission apps to inject arbitrary custom accessibility events into the system, which brings the possibility of constructing fraudulent activities.

Exploiting these design shortcomings, we demonstrate two real-world attacks as showcases. That is, a malicious app without any sensitive permission can hijack the execution logic of assistive apps installed on the same phone to *install arbitrary apps* and *send phishing notifications*. Note that, different from the previous works [35, 37, 43], in our attacks, we consider a more general model and do not assume the malicious app has been granted with the `BIND_ACCESSIBILITY_SERVICE` permission.

Following the responsible disclosure policy, we reported our findings to the Android security team and the corresponding vendors. At present, Google has confirmed our discovery and rewarded us with $200 under the Android Security Rewards Program. The latest update could be tracked through `AndroidID-79268769` and `CVE-2018-9376`.

As mitigations, we also propose some improvement recommendations for each shortcoming. However, to thoroughly address the current security threats, a new accessibility architecture may be needed. How to trade off the security and usability is still an open question.

Contributions. We summarize the contributions of this paper as follows:

- *Data-driven Analysis.* We perform the first large-scale study to measure the usages of the accessibility APIs in the wild (based on 91,605 app samples from Google Play). Our study shows the accessibility APIs have been misused widely.
- *Discovery of New Design Flaws*. After reviewing the design of the Android accessibility supporting architecture, we identify a series of fundamental design shortcomings which may bring serious security risks. We also propose improvement recommendations.
- *Demonstration of Proof-of-Concept Attacks*. We demonstrate two concrete attacks exploiting the design shortcomings of the accessibility framework as showcases: installation hijacking and notification phishing.

Roadmap. The rest of this paper is organized as follows. Section §2 provides the background of the accessibility service on Android. Section §3 introduces the threat model and methodology of this paper. In Section §4, we measure the usage status of the accessibility APIs on a large-scale app dataset. Section §5 summarizes the discovered design shortcomings. Section §6 demonstrates two practical attacking exploiting these shortcomings. Section §7 discusses some attack conditions and limitations. Section §8 proposes some improvement recommendations. Section §9 reviews related works, and Section §10 concludes this paper.

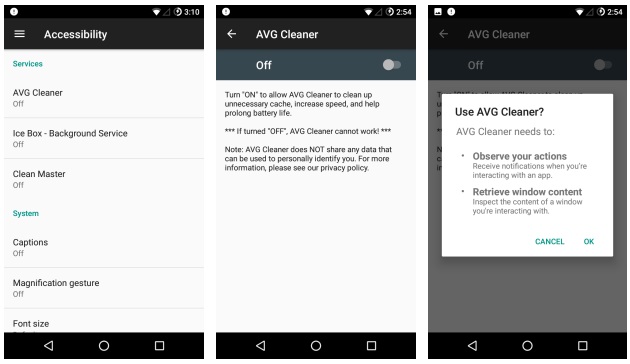
2 Accessibility Service on Android

The accessibility service was introduced by Google starting from Android 1.6 (API Level 4) and gained significant improvement since Android 4.0 (API Level 14). It is designed to implement *assistive technology* with two main functionalities: (1) receiving input from alternative input devices (e.g., voice into microphone) and transforming it to commands accepted by OS or apps; (2) converting system output (e.g., text displayed on screen) into other forms which can be delivered by alternative output devices (e.g., sound through a speaker).

Capabilities. To this end, Android provides a set of capabilities for the accessibility APIs, and they can be grouped into eight categories [5], as listed below:

- C0: *Receive* `AccessibilityEvents`. This is the default capability ensuring the accessibility service can receive notifications when the user is interacting with an app.
- C1: *Control display magnification*.
- C2: *Perform gestures*, including tap, swipe, pinch, etc.
- C3: *Request enhanced web accessibility enhancements*[1]. Such extensions aim to provide improved accessibility support for the content presented in a WebView.
- C4: *Request to filter the key event stream*, including both hard and soft key presses.
- C5: *Capture gestures from the fingerprint sensor*.
- C6: *Request touch exploration mode*. In this mode, a single finger moving on the screen behaves like a mouse pointer hovering over the user interface.
- C7: *Retrieve interactive window content*. An interactive window is a window that has input focus.

[1] This capability was deprecated in Android 8.0 (API level 26).



(a) Accessibility settings (b) AVG Cleaner (c) Security warning

Figure 1: Turn on accessibility service for AVG Cleaner.

Building An Assistive App. Android provides standard accessibility services (such as TalkBack), and developers can create and distribute their own custom services. An app providing the accessibility service is called *assistive app*. To build an assistive app, developers need to create a service class that extends `AccessibilityService`.

Permission. For security reasons, the accessibility service must be protected by the `BIND_ACCESSIBILITY_SERVICE` permission (protection level: signature) to ensure that only the system can bind to it. An extra requirement is that the user must manually turn on the accessibility switch and confirm the security implications for every accessibility service (assistive app), as demonstrated in Figure 1(c). The listed items in this picture are the capabilities declared by this service.

Service Interaction. The internal mechanism of the accessibility framework is quite complicated, and here we focus on how accessibility events are processed. As illustrated in Figure 2, three components are involved in the Android accessibility service framework: topmost app, system-level service `AccessibilityManagerService`, and multiple assistive apps with custom accessibility services. A typical and simplified invocation process of accessibility service is illustrated as below:

1. *Generate & Send Events*: An accessibility event [2] is fired by the topmost app which populates the event with data for its state (e.g., the changes in the UI) and requests from its parent to send the event to interested parties. The event delivery is based on the binder IPC mechanism via `IAccessibilityManager`.

2. *Dispatch Events*: All generated accessibility events will be sent to the centralized manager of the Android OS – `AccessibilityManagerService`. After some basic checkings (such as event types), it dispatches events to each bound accessibility services through binder `IAccessibilityServiceClient`.

3. *Receive & Handle Events*: Through the callback function `onAccessibilityEvent`, assistive apps obtain the dispatched events and further process them following their own programmed logics. If the assistive app needs to inject actions (e.g., clicking), it could reversely lookup the view hierarchy from the source contained in an event, locate a specific view node (e.g., a button), and then perform actions on the topmost app.

Accessibility Events[2]**.** As described above, the accessibility architecture is event-driven. The `AccessibilityEvent` is generated by a view and describes the current state of the view. The main properties of an `AccessibilityEvent` include [2]: `EventType`, `SourceNode`, `ClassName`, and `PackageName`. Note that, each event type has an associated set of different or unique properties.

3 Threat Model and Methodology

According to whether the assistive apps are malicious, the security threats related to the accessibility APIs could be classified into two groups. In this paper, we focus on the normal use cases of the accessibility APIs and, therefore, consider the security threats assuming benign assistive apps.

It has been well-studied that malicious apps can exploit the powerful capabilities of the `BIND_ACCESSIBILITY_SERVICE` permission to launch attacks. Previous research [35,37,43] has demonstrated a variety of stealthy attacks could be launched, even complete control of the UI feedback loop (see Section §9 for more details). Such kind of attacks is built on the dominant feature through the accessibility APIs which could achieve the cross-app operations. As shown in Figure 1, the key point of a successful attack is how to induce victim users to turn on the accessibility service.

Threat Model. In our study, we consider a more general model: the attacker only could control a malicious app installed on the victim's phone without any sensitive permissions. Also, there is a benign assistive app installed on the same phone. We assume the malicious app attempts to hijack the execution logic of the assistive app to perform malicious activities. In this process, the assistive app becomes the confused deputy, and its assistive capabilities are abused.

Methodology. In our study, we employed the following two-step methodology:

- *Measuring the usage of the accessibility APIs in the wild.* With the data collected by ourselves, we could answer whether the accessibility APIs are used correctly by developers as expected.

- *Reviewing the design of Android accessibility supporting architecture.* If the answer of the first step is "no", we

[2] We use "`AccessibilityEvent`" and "accessibility event" interchangeably in this paper.

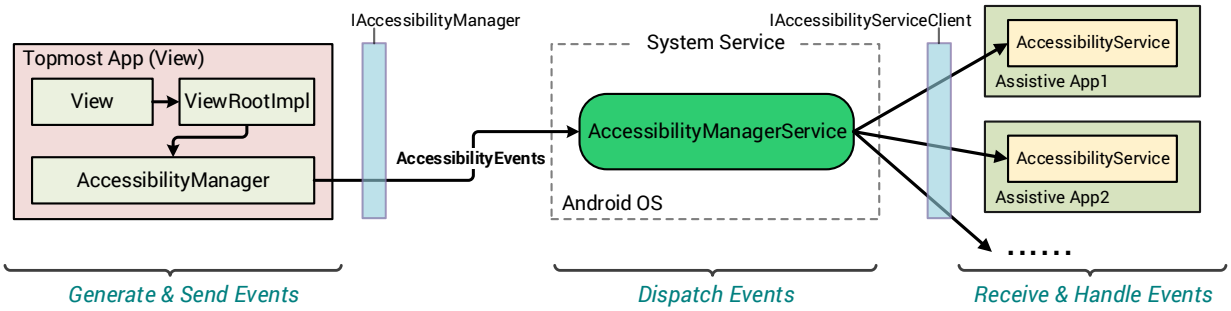


Figure 2: Android accessibility service framework.

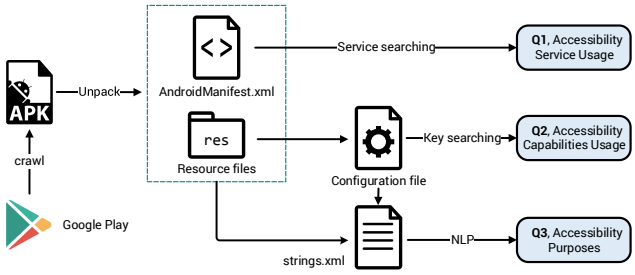


Figure 3: App analysis.

need to investigate the fundamental reasons of misuse and the potential security risks.

4 Accessibility APIs Usage

As the first step, to understanding the accessibility APIs usage status, we carried out a large-scale study on Android apps in the wild. In particular, we try to answer the following three questions.

Q1: *How many apps use the accessibility APIs?*
Q2: *What kinds of accessibility capabilities are used?*
Q3: *What are the purposes of using accessibility APIs?*

Since we are interested in the usage status of legitimate apps, an APK sample dataset (91,605 samples, around 1.12 TB) crawled from Google Play was used in our experiment. These samples were collected in 2018, covering most popular apps in each category except for games. As preparation, we used Apktool [9] to disassemble them and obtained the corresponding manifest and resource files (the `res` folder). To each question, we deployed different analysis approaches, and Figure 3 illustrates the overall analysis process.

4.1 Accessibility APIs Usage

To **Q1**, we wrote a shell script to search the services protected by the `BIND_ACCESSIBILITY_SERVICE` permission in manifest files.

Result. The result shows around 0.37% apps (337 / 91,605) from Google Play use the accessibility APIs. Also, these 337 assistive apps provide 342 accessibility services[3]. Though the percentage looks quite low, it does not mean these assistive apps receive little attention. On the contrary, more than half of them (56.7%) have over 1 million installations.

4.2 Accessibility Capabilities Usage

To **Q2**, accessibility services must declare the needed accessibility capabilities (listed in Section §2) in advance and ask users to confirm, like Figure 1(c).

In particular, every assistive app must prepare a configuration file for its accessibility service, which declares some meta information, such as needed capabilities, expected event types, and timeout. Here we take the configuration file of Network Master (`com.lionmobi.netmaster`) as an example, as shown in Listing 1. The key-value pair `[android:canRetrieveWindowContent="true"]` indicates it needs to invoke the capability of retrieving the active window content. Note that the default capability of receiving accessibility events will always be granted automatically.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <accessibility-service android:description
    ="@string/
    boost_tag_acc_kill_service_description
    " android:accessibilityEventTypes="
    typeWindowStateChanged"
    android:accessibilityFeedbackType="
    feedbackGeneric"
    android:notificationTimeout="100"
    android:canRetrieveWindowContent="true
    "
3 xmlns:android="http://schemas.android.com/
    apk/res/android" />
4 </service>
```

Listing 1: Accessibility configuration of Network Master.

Based on this observation, we obtain the capability usage data by analyzing the accessibility service configuration files.

[3]Note that, one app could provide multiple accessibility services.

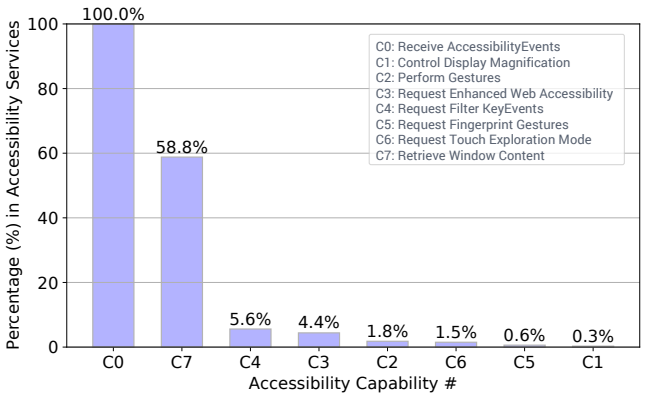


Figure 4: Statistics of capabilities usage.

Among the collected 342 accessibility services, we did not obtain the configuration files from 8 of them. The reasons for the failed retrieval are two-fold: (1) some apps declare the accessibility services in manifest files but do not implement them in code; (2) the other apps deploy anti-analysis protections (e.g., packer), and the resource files cannot be disassembled successfully.

Result. The statistical result is plotted in Figure 4. It shows, besides the default capability of receiving accessibility events (C0), the capability of retrieving the active window content (C7) is the most popular one, say 58.8%. The use cases of the other capabilities are not common. Also, 128 accessibility services (37.4%) only use the default capability.

Our assessment: Most assistive apps only use the accessibility APIs to receive accessibility events (C0) and execute automated clicking operations (C7). Also, the deployment scenarios of some accessibility APIs (C1 and C5) are extremely rare, of which design may be ill-considered.

4.3 Purposes of Using Accessibility APIs

To **Q3**, it is indeed a non-trivial task. An intuitive solution is to analyze the disassembled code of assistive apps. Through building a context-sensitive call graph, we could track the accessibility APIs invocation and related code executions. However, the challenge is how to identify the ultimate purpose of the accessibility code. For example, we could identify the assistive app injects some clicking actions to the foreground app, but the purpose of the injection operations is not easy to identify, especially for the various nonstandard implementations. To achieve it, we have to manually analyze several implementation samples and build a series of models. When facing obfuscated apps, it will become a tough process. Therefore, static code analysis is not practical for this task.

NLP-based Analysis. As an alternative, we designed a lightweight solution based on natural language processing (NLP) techniques. We notice that every accessibility service must provide a description to explain why it needs the accessibility service, as shown in Figure 1(b). If the assistive app is legitimate (and appears on Google Play), it has no incentive to provide a fake description in most cases. Also, since this description should be understood by ordinary users, it is usually written in plain languages, like Listing 2 from Network Master. Also, we give more examples in the Appendix.

```
<string name="
    boost_tag_acc_kill_service_description
    ">"Turn it on will help Network Master
     stop apps and extend your battery
    life. Network Master uses
    accessibility service to optimize your
     device only. We will never use it to
    collect your privacy information. If
    you receive warnings about privacy,
    please ignore."</string>
```

Listing 2: Service description of Network Master.

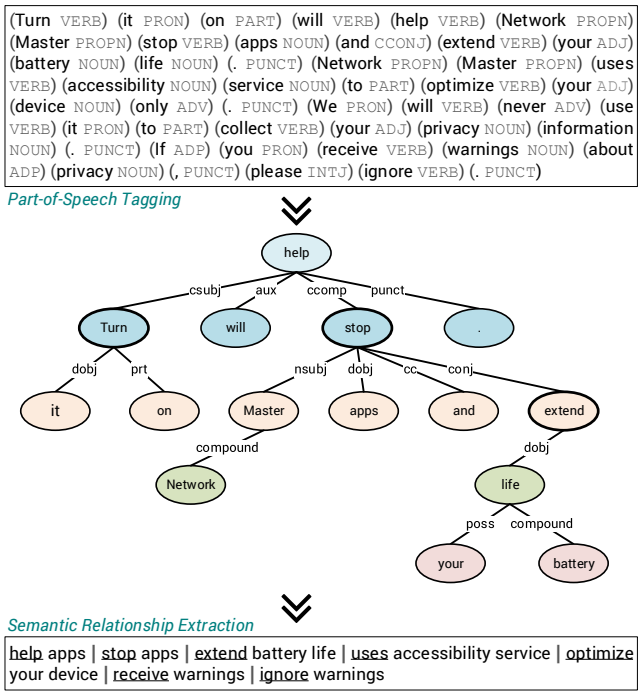
Motivated by this observation, the purposes of accessibility APIs invocations could be extracted through analyzing their usage descriptions. Here we describe our four-step approach:

1. *Service Description Crawling.* Through analyzing the service configuration file, we could locate and extract the service description from `res/values/strings.xml` (or similar paths). If the description is not in English, we translate it to English through the Google Translate API.

2. *Part-of-Speech Tagging.* Part-of-speech (PoS) tagging is the operation of tagging a word in a text as corresponding to a particular part of speech, based on both its definition and context [17]. Since we are concerned with the actions mentioned in the description, we need to extract the contained action phrases, like "stop apps". With PoS tagging, we could obtain all verbs in a sentence as preparatory knowledge. In the implementation, we use spaCy [21] to complete this step, and the result is shown in Figure 5 (based on Listing 2).

3. *Semantic Relationship Extraction.* In this step, we extract the [action + object] relationship from the sentences. Our method is to build a semantic relationship tree for each sentence based on spaCy. Then, through breadth-first searching from each verb, we could get the [action + object] relationships. As illustrated in Figure 5, we obtain "help apps", "stop apps", and "extend battery life" from the first sentence. Note that, the negative statements have been excluded in this step because the contained actions will not happen. Therefore, "(never) collect private information" is not extracted from the third sentence.

4. *Matching and Classification.* The last step is to build a series of matching rules for classifying the [action +



Remarks: The other three relationship trees are omitted due to space limitations. The definitions of grammatical relations (like csubj, aux, ccomp, and prt) are based on the Stanford typed dependencies [31].

Figure 5: [action + object] relationship extraction.

object] relationship sets. We apply a heuristic method to build rules[4]. That is, when an app is not matched by any rule, we will check its accessibility service description and add new rules. If an app is classified incorrectly, we will adjust the existing rules. The formats of rules are [*v*] for matching a single verb, [*n*] for matching a single noun, and [*v n*] for matching action phrases. For example, the rules for the usage of killing processes contain:

[*v* kill *n* app; *v* stop *n* app; *v* block *n* app; *v* kill *n* applic; *v* stop *n* applic; *v* block *n* applic; *n* batteri; *n* cach; *n* acceler; *n* power]

Note that, we have applied the stemming in matching to avoid the interference of inflected words. Therefore, in this example, "applic" could match "application" and "applications". Similarly, "stop" could match "stop", "stops", "stopping", and so forth.

After the first step, we obtained 321 descriptions from 342 accessibility services. Among the failure samples, 8 of them lacked available configuration files (the reason has been given in Section §4.2), and 13 of them did not provide the service descriptions.

Result. Finally, we classified the descriptions into 10 categories, and the result is plotted in Figure 6. Among them,

[4]We did not use machine learning-based algorithms (like k-means) in this step because they do not work well on short text due to insufficient features.

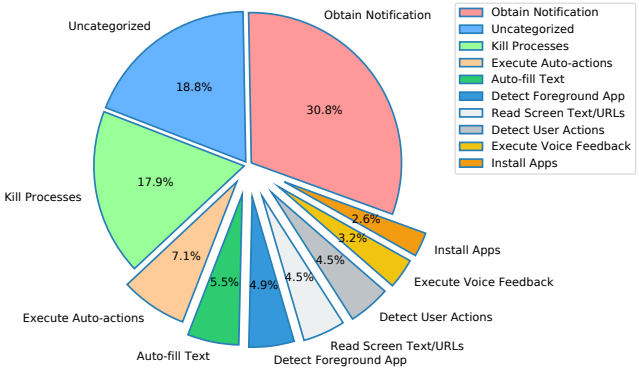


Figure 6: Statistics of purposes.

58 accessibility services (18.8%) are labeled "uncategorized" because their descriptions do not provide useful information[5].

Among the collected descriptions, only 11 apps mention they are designed for users with disabilities, i.e., 3.2%. It means most accessibility service invocation behaviors are suspicious to some extent. Also, according to Android developers documents: "*accessibility services should only be used to assist users with disabilities in using Android devices and apps* [4]." Here we define that if an app uses the accessibility APIs not for helping the disabled people, it should be treated as a **misuse** behavior. Note that, even such usage is not for malicious purposes, it also could be classified into misuse behaviors.

Through the categorical data analysis and manual confirmation, we identify some typical misuse implementations.

(1) Around 30.8% of assistive apps use the accessibility APIs to *obtain system notifications*, which occupies the most significant share. Most of them belong to the launcher, lockscreen, or status bar apps. Though Android has provided the notification reading APIs and the `BIND_NOTIFICATION_LISTENER_SERVICE` permission in Android 4.3, several assistive apps still keep the accessibility-based approach to avoid compatibility issues.

(2) Another significant category is the purpose of *killing background processes*, say 17.9%. The apps falling in category use the accessibility service to click the "FORCE STOP" button on the app info menu in the system setting. This method could terminate a background process and prevent it from restarting again. Also, the regular `KILL_BACKGROUND_PROCESSES` permission cannot achieve preventing apps restart. The `FORCE_STOP_PACKAGES` permission could achieve it but not available for third-party apps. Therefore, such implementation is popular in battery saver or system booster apps.

[5]For example, "*Tap on the top right hand toggle to enable CM Launcher. Attention: You may receive standard privacy warnings. There's no need to worry, no personal data will be collected.*" from CM Launcher (`com.ksmobile.launcher`).

(3) Executing auto-actions (7.1%) means accessibility services could *automatically complete a series of clicking actions* without user operations. A typical case is that input method apps use it to send GIFs. Users with motor impairments also could benefit such usage.

(4) Auto-filling text (5.5%) is mainly implemented to *automatically fill username and password*. It has become the standard feature in nearly all password manager apps.

(5) Around 4.9% of assistive apps use the accessibility APIs to *detect foreground apps*, such as measuring game playing time. The information about which app is running in the foreground is sensitive because it may be abused for phishing attacks. Therefore, Google has replaced the `GET_TASKS` permission by the system-level permission `REAL_GET_TASKS` in Android 5.0 to block accessing from third-party apps. However, with the accessibility service, assistive apps could still obtain the foreground app information.

Our assessment: The accessibility APIs have been misused widely. Most assistive apps utilize them to bypass the permission restrictions of Android OS, which deviates from the original mission.

5 Design Shortcomings

Motivated by the less optimistic results of app scanning, we further reviewed the design of Android accessibility supporting architecture. Finally, we identify a series of design shortcomings lying in the Android accessibility framework, which may bring serious security risks.

Design Shortcoming #1[6]**.** The accessibility service is designed for users with disabilities and, therefore, enhances the user interactions (i.e., input and output). However, *there is no restriction on the purposes of using the accessibility APIs*. Any app can invoke the accessibility APIs even it is not designed for disabled users. Naturally, in practice, how to use these APIs depends on the developers' understanding and users' demand.

Since the accessibility APIs are very powerful, the assistive apps can know the current foreground app, displayed texts, and user's actions, and even operate arbitrary other apps. On the other hand, due to the increasingly strict restriction of the Android permission system [51], some apps need to find a new code implementation approach to meet the requirements of their function designs. As a result, through combining the accessibility APIs and programming tricks, several dangerous permissions could be bypassed, as summarized in Table 1.

Design Shortcoming #2. The accessibility architecture of Android is event-driven. The execution logic of accessibility services only could rely on the received accessibility events. The assistive app extracts the event properties (like `EventType`, `ClassName`, `PackageName`) and further judges what happens in the foreground. However, *the information contained in the events cannot provide a strong guarantee to the integrity of event processing*. The current design cannot guarantee that two events with the same properties are definitely generated by the same view, i.e., uniqueness guarantee.

In the accessibility framework, the event receivers (assistive apps) do not communicate with the event senders (topmost app) directly. Such a design ensures the centralized management and efficient event dispatching. On the other hand, the integrity of event processing flow only relies on the checkings implemented by the assistive apps themselves. The unreliable provenance information may confuse the checkings.

Design Shortcoming #3. Android allows zero-permission apps to inject custom `AccessibilityEvents` into the system. This function is provided to developers to make their custom view components accessible [12]. However, *there is no restriction on how to set the properties of a custom `AccessibilityEvent`*, which brings the possibility of constructing fraudulent events. Also, though Android OS requires the `AccessibilityEvent` only could be sent by the topmost view in the view tree [3], this restriction is not enforced.

Any app could implement the following code to construct and inject a custom `AccessibilityEvent`:

```
1  AccessibilityManager manager = (
       AccessibilityManager) getSystemService
       (ACCESSIBILITY_SERVICE);
2  AccessibilityEvent event =
       AccessibilityEvent.obtain();
3
4  event.setEventType(...);
5  event.setClassName(...);
6  event.setSource(...);
7  event.setParcelableData(...);
8  ... // Other properties are omitted
9
10 manager.sendAccessibilityEvent(event);
```

Listing 3: Inject custom AccessibilityEvent.

6 Attack Case Studies

In this section, we discuss how to exploit the discovered design shortcomings to launch real-world attacks. Specifically, we present installation hijacking and notification phishing as showcases. The attack demos are available at https://sites.google.com/site/droidaccessibility/.

6.1 Case Study: Installation Hijacking

In this case, a malicious app without sensitive permission could hijack the execution logic of assistive apps to *install arbitrary apps silently*.

[6] **DS#1** for short. Similarly, we have **DS#2** and **DS#3**.

Table 1: Bypassed permissions through the accessibility APIs.

Usage	Bypassed Permission	Protection Level
Obtain notification	BIND_NOTIFICATION_LISTENER_SERVICE†	Signature
Kill processes	FORCE_STOP_PACKAGES	Not for third-party apps
Execute auto-actions	INJECT_EVENTS	Not for third-party apps
Auto-fill text	BIND_AUTOFILL_SERVICE†	Signature
Detect the foreground app	REAL_GET_TASKS	Not for third-party apps
Install / Uninstall apps	INSTALL_PACKAGES	Not for third-party apps
	DELETE_PACKAGES	Not for third-party apps

†: The corresponding service must be protected by this permission to ensure that only the system can bind to it.

The popularity of Android is primarily due to a wide variety of apps provided by Google Play – the official Android app store. However, due to the policy restriction, the Google service framework (including Google Play) is not available in some countries. Also, some apps on Google Play are region-locked. Therefore, third-party app stores (*store app* for short) become an alternative choice, such as 1Mobile [1], Amazon Appstore [6], and APKPure [8].

In Android, the INSTALL_PACKAGES permission is designed to prevent the apps from unknown sources to be installed silently. Also, it is a system-level permission and not available for third-party apps. As a result, third-party store apps have to work as APK downloaders and ask the user to click the "INSTALL" button of the Installer by themselves, as shown in Figure 7(a). However, with the accessibility APIs, store apps can achieve the automatic installation by clicking the "INSTALL" button programmatically, which saves user clicks and bypasses the INSTALL_PACKAGES permissions. Such implementation improves the user experience but disobeys the mission of the accessibility APIs [**DS#1**].

Logic Analysis. Here we describe the logic implementation of the automated installation of store apps, as illustrated in Figure 7(c). After the target APK file has been downloaded to the device, the store app utilizes the Intent mechanism [13] to load this APK file. Then Android OS will invoke a proper program (i.e., PackageInstaller [16] in this case) to process it. After that, the Installer requests the user to confirm the installation and required permissions. Note that, during this process, the store app continuously monitors the change of foreground UI through filtering AccessibilityEvents. When it finds that the PackageInstaller is launched to process the APK file just downloaded, it will invoke the accessibility service to click the "INSTALL" button automatically.

However, we find that, before deciding whether to click the "INSTALL" button, the checking logic (Step ③ in Figure 7(c)) of the store app is vulnerable. This step checks four parameters of incoming AccessibilityEvents:

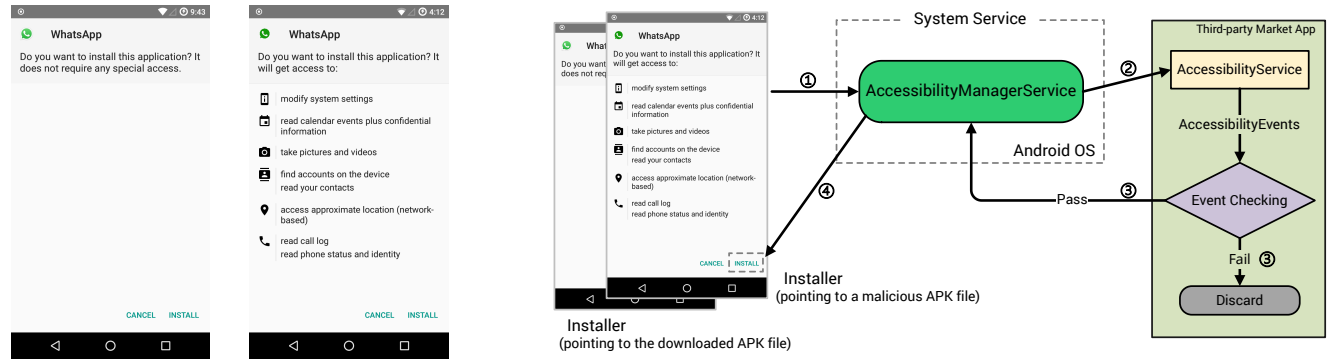
1. SourceNode != null? (If null, the store app cannot retrieve the window content, locate the "INSTALL" button, and execute the clicking action.)
2. EventType == TYPE_WINDOW_CONTENT_CHANGED? (This type of events is usually triggered by adding or removing views.)
3. PackageName == com.android.packageinstaller? (This ensures that the app running in the foreground is the PackageInstaller.)
4. Text is the name of the downloaded app (e.g., "WhatsApp" in Figure 7(a))? (This ensures that the app being installed is the one just downloaded.)

If all four conditions pass, the store app will believe the PackageInstaller is processing the downloaded APK file [**DS#2**]. Unfortunately, this AccessibilityEvent-based checking is not complete, and a malicious app installed on the same phone can construct a scenario passing the checking conditions to hijack the work-flow of store apps.

Attack. In this attack, the malicious app only declares the READ_EXTERNAL_STORAGE permission, a very common permission. Its payload contains a repackaged Trojan APK file disguised as a popular app, like WhatsApp. This Trojan app can execute various malicious operations with many dangerous permissions, like Figure 7(b).

First, the malicious app running in the background monitors the download folder of the victim store app. In general, since the downloaded APK files are not sensitive data, this folder is usually located in the public storage of the device [19]. Therefore, any app with the READ_EXTERNAL_STORAGE permission could access it. Note that, if the store app keeps the downloaded APK files in its private folder, the malicious app will not be able to monitor the file downloading status and further launch the hijacking attack. (Un)fortunately, using the public storage for saving temporary data is a widespread operation in Android apps [34, 40], including at least 11 popular store apps as listed in Table 2.

During the monitoring, if a new cache file appears in this folder, it means the store app starts to download a new APK file. The code implementation could be based on the Runnable interface [18] for periodic file existing checking. Through identifying the name of the cache file, the malicious app could know what app is being downloaded because the file



(a) Installer - WhatsApp (b) Installer - Trojan (c) Attack overview

Figure 7: Installation hijacking attack.

name is usually the hash value (MD5 or SHA-1) of the being downloaded APK file or contains the package name. Taking APKPure as an example, the default path for saving APK files is `/Download` on the external storage, and the format of the cache file name is `WhatsApp_Messenger_[xxx].apk.tmp`, e.g., `WhatsApp_Messenger_2a1417b0.apk.tmp`.

Next, if the malicious app finds the store app is downloading the target app (i.e., WhatsApp in our case), the hijacking attack will be launched. When the downloading completes (`WhatsApp_Messenger_[xxx].apk.tmp` becomes `WhatsApp_Messenger_[xxx].apk`), the malicious app utilizes the same Intent mechanism as the store app to load its Trojan APK file immediately, as shown in Figure 7(b). The PackageInstaller pointing to the Trojan APK file (*Installer-A*) will happen to cover the PackageInstaller pointing to the downloaded APK file (*Installer-B*), as illustrated in Figure 7(c). Note that, this step creates a race condition (*Installer-A* vs. *Installer-B*), and the attack may fail if the Intent from the malicious app is not processed by the OS at the right time. In practice, the success rate of attacks could be significantly improved through adjusting the point in time of launching *Installer-A*. For example, on Motorola Moto G3 (the device used in our attack demo), when the downloading completes, the malicious app will wait 400ms before launching *Installer-A*. Following this trail, in experiments, we achieved nearly 100% success rate (*Installer-A* covering *Installer-B*). Also, it is an empirical time value and may be different on other devices with varying computing performance.

At this moment, the `AccessibilityEvent` from the *Installer-A* is almost the same as the one from *Installer-B*, which meets all four conditions listed previously. As a result, the store app will be deceived into thinking the *Installer-A* is processing the APK file it just downloaded, so it decides to click the "INSTALL" button. Finally, the repackaged Trojan app prepared by the attacker is installed on the phone.

Summary. The checking logic of store apps entirely depend on the information contained in `AccessibilityEvents`. However, the verified factors cannot guarantee which APK file is being processed, which results in the possibility of creating a race condition.

Table 2: Vulnerable third-party app stores.

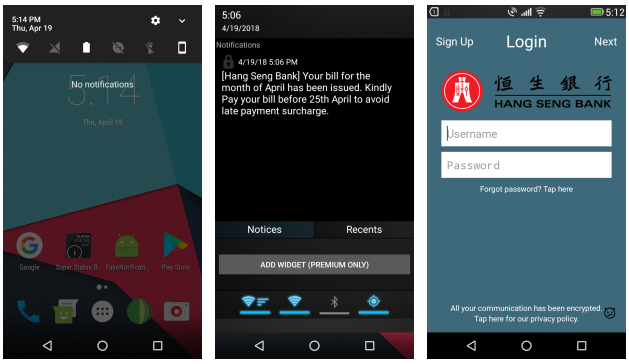
Store	Package Name	Version
APKPure	`com.apkpure.aegon`	2.12.2
1Mobile Market	`me.onemobile.android`	6.8.0.1
360 Mobile Assistant	`com.qihoo.appstore`	7.1.90
Baidu Mobile Assistant	`com.baidu.appsearch`	8.5.1
Sogou Mobile Assistant	`com.sogou.androidtool`	6.7.2
MoboMarket	`com.baidu.androidstore`	4.1.9.6222
PP Assistant	`com.pp.assistant`	6.0.8
AppChina	`com.yingyonghui.market`	2.1.62716
Lenovo Le Store	`com.lenovo.leos.appstore`	9.8.0.88
2345 Mobile Assistant	`com.market2345`	5.6
Wandoujia	`com.wandoujia.phoenix2`	5.74.21

Scope of Attacks. We checked popular third-party store apps and found at least 11 of them (with tens of millions of users [23, 24]) suffer from the security risk of installation hijacking, as listed in Table 2.

6.2 Case Study: Notification Phishing

In this case, a zero-permission malicious app could exploit the execution logic of assistive apps to *send phishing notifications to users*. Also, this attack is different from the direct notification abuse attack [47]. Even the attack app has been blocked for sending system notifications, this attack still works.

Here we consider the apps with the function of notification management, such as status bar app. On Android, the system default status bar could be replaced by third-party status bar apps for a better experience. They could provide several advanced features, like replaceable theme styles, customized fonts, spams filtering, and so forth. On Google Play, there are several popular status bar apps with over one million in-



(a) Original status bar (b) Phishing notification (c) Phishing Activity

Figure 8: Notification phishing attack.

stallations, such as Super Status Bar[7] (`com.firezenk.ssb`), Status (`com.james.status`), and Material Notification Shade (`com.treydev.msb`).

As an essential function, status bar apps need to obtain system notifications and notify the user. After Android 4.3 (API level 18), third-party apps could obtain system notifications through the `NotificationListenerService` [15] with the `BIND_NOTIFICATION_LISTENER_SERVICE` permission. However, for the devices equipped with old Android versions, the only method of obtaining system notifications is to utilize the accessibility service. Due to the Android fragmentation problem and the consideration of backward compatibility, this accessibility-based notification obtaining method is still very popular. This observation also has been confirmed by our study in the accessibility purpose analysis, say 30.8% usage (see Section §4.3). Again, such implementation is not designed for disabled users and disobeys the mission of the accessibility APIs [**DS#1**].

Logic Analysis. Here we describe the execution logic of accessibility-based notification obtaining. First, the status bar app filters the received accessibility events for notifications. If the `EventType` is `TYPE_NOTIFICATION_STATE_CHANGED`, it will believe the system just dispatches a new notification [**DS#2**]. Then the status bar app further extracts the properties of this event, and parses the necessary information, like the notification title, content, parcelable data. However, we find this process is vulnerable. A zero-permission malicious app could construct a custom `AccessibilityEvent` with phishing information and cheat the event receiver.

Attack. As preparation, our zero-permission malicious app has been installed on the user's phone and runs in the background. Taking Super Status Bar as an example (Figure 8), the attack app intends to send a phishing notification disguised as a message from a bank app. Therefore, it needs to construct and inject a custom `AccessibilityEvent` with the following properties [**DS#3**]:

1. `EventType` = `TYPE_NOTIFICATION_STATE_CHANGED`.
2. `ClassName` = `android.app.Notification`.
3. `PackageName` = `com.hangseng.rbmobile`, notification sender, a bank app.
4. `SourceNode` = `null`, there is no source for the type of `TYPE_NOTIFICATION_STATE_CHANGED` [**DS#2**].
5. `ParcelableData` is set as a `Notification` [14] instance which contains the phishing message and an Intent pointing to a phishing Activity (prepared by the malicious app) disguised as the bank app.

When Super Status Bar receives the custom (phishing) `AccessibilityEvent`, it will think the bank app just posts a new notification to the system. Then it parses the properties of this event and displays the phishing notification in its status bar, like Figure 8(b). After the user notices this new notification and clicks it, the status bar app will load the contained Intent. Finally, the phishing Activity is launched and induces the user to fill her credentials, as shown in Figure 8(c).

Summary. As mentioned in Section §2, different types of accessibility events may have different properties. To some specific types, like `TYPE_NOTIFICATION_STATE_CHANGED`, there is no `SourceNode` property, which results in that assistive apps cannot identify the original senders.

Impact. All assistive apps implementing the accessibility-based notification obtaining suffer from the security risk of notification phishing. This attack method and the design shortcoming on custom `AccessibilityEvent` have been reported to the Android security team and assigned tracking id `AndroidID-79268769`. At present, they have acknowledged our report and rewarded us with $200. They mentioned it was also "*reported by an internal Google engineer*". In other words, they confirmed we discovered the problem independently. Also, a CVE-ID has been assigned – `CVE-2018-9376`. Besides, after our report, the vulnerable app Super Status Bar (`com.firezenk.ssb`) was removed from Google Play.

7 Discussions and Limitations

Here we discuss some attack conditions and the limitations existing in our experiment and analysis.

Attacks without accessibility services. To the installation hijacking attack, if the accessibility service is not presented, the user will be involved in the installation process. To an experienced user, she may notice the unusual permission requests (see Figure 7(b)) and rejects the installation. To the notification phishing attack, if the accessibility service is not presented, how to select the time of showing the phishing

[7] At present, this app is unavailable on Google Play, see the "Impact" part of this subsection for more details.

Activity will become a problem. This is the primary technical challenge of Activity phishing attacks on Android.

APK dataset. The dataset for app scanning experiment contains 91,605 samples, and 337 assistive apps (containing 342 accessibility services) were identified. Also, due to anti-analysis protection and legacy code, only 334 service samples could be used for subsequent analysis. Our dataset could be extended to obtain more apps for analysis.

Dishonest descriptions. In Section §4.3, our analysis is based on the accessibility service descriptions provided by assistive apps. Though these apps are legitimate, it is still possible that their descriptions are not honest. They may conceal (parts of) their true intentions for some reasons. Such a situation may affect the accuracy of our purpose analysis.

Misuse Identification. In some cases, it is difficult to judge whether the usage behaviors are misuse, especial executing auto-actions. For example, Automate (`com.llamalab.automate`) could help users create their automations using flowcharts. The supported actions include automatically sending SMS or E-mail, copy files to FTP or Google Drive, play music or take photos, etc. According to the introduction on its website [10], we believe this app is not designed for disabled users, but it is difficult to judge based on its usage descriptions or behaviors.

8 Recommendations to More Secured Accessibility APIs and Framework

In this paper, we systematically analyze the usage and security risks of the accessibility APIs. Given the design shortcomings of Section §5, we propose some possible improvements to mitigate these security risks.

At the high level, the accessibility APIs are very special because they are designed for the users with disabilities. Therefore, the usability is essential in the framework design. It cannot be too complicated for disabled people. The trade-off between security and usability is still an open question. The shortcomings (**DS#1**, **DS#2**, and **DS#3**) discovered in this paper are the fundamental design issues of the event-driven accessibility framework. A new architecture may be needed to completely solve them. At this moment, it is out of the scope of this paper, and here we discuss some targeted improvements for each shortcoming.

To **DS#1**, ideally, if an app is not designed for disabled users, it should not invoke the accessibility APIs. The problem is that some assistive apps belong to the killer apps with millions of installations, and (parts of) their core functionalities rely on the accessibility service, such as LastPass (`com.lastpass.lpandroid`) and Universal Copy (`com.camel.corp.universalcopy`). On November 2017, Google required the assistive app developers must explain how their apps are using the accessibility APIs to help users with disabilities, or their apps will be removed from the Play Store [32]. However, according to our observation, this plan was not be executed smoothly, and Google gave up due to the public outcry about favorite apps will stop working [27]. The lesson to be learned here is that whether something is a "misuse" is mostly determined if the users are happy with how that something is used.

We recommend designing new APIs for the requirements of misuse cases. The existence of misuse cases reflects the current Android APIs cannot meet the requirements of developers. New APIs and permissions could be added to make developers give up using the accessibility APIs. Such an improvement will be once and for all. Google also has made such an attempt. On Android 8.0, a new permission `BIND_AUTOFILL_SERVICE` is added, and password manager apps could utilize this new permission to achieve the auto-fill feature [41]. Due to Android fragmentation, it may take a long time before all relevant issues are fixed. On the other hand, the introduction of new APIs will bring some compatibility problems inevitably. For example, the apps developed with the new APIs cannot run on old devices directly. As a result, the developers have to use the Android Support Library [22] to achieve backward compatibility. Even so, due to the limitations of the host device platform version, the full set of functionality may still be unavailable.

To **DS#2**, under the current architecture, it is nearly impossible to fix this design shortcoming. Since the accessibility event senders and receivers do not interact directly, it is difficult for an assistive app (receiver) to identify the event sender.

We recommend improving the execution logic of assistive apps as short-term mitigation. For example, in the case of installation hijacking, the store app should save the downloaded APK files to its private data folder (i.e., internal storage) [19], which would significantly reduce the chance of being identified what APK file is being downloaded. Also, in the case of notification hijacking, the status bar app should not launch the (unreliable) Intent contained in the received `TYPE_NOTIFICATION_STATE_CHANGED` events.

To **DS#3**, the basic information of custom accessibility events should not be filled by third-party apps, including `SourceNode`, `ClassName`, and `PackageName`. Only the OS could fill such information. This restriction ensures the sender information cannot be tampered with.

On the other hand, a new permission could be added for restricting sending custom accessibility events. Since this functionality is provided to developers to make their custom views accessible, it should not be used by any app without restrictions. At least, more restrictions should be applied to the allowed types and numbers of custom events.

9 Related Work

Assistive technologies do not come at no cost. In this section, we review the related works on the security issues of accessibility techniques.

Jang et al. [36] presented the first security evaluation of accessibility support for four mainstream platforms (Microsoft Windows, Ubuntu Linux, iOS, and Android). Their study demonstrated that inadequate security checks on I/O paths make it possible to launch attacks from accessibility interfaces. It is the closest work to us. The difference is that this study focused on the accessibility module I/O and did not touch the underlying design of Android accessibility framework.

On the Android platform, Kraunelis et al. [37] first noticed the possibility of attacks leveraging the Android accessibility framework. More recently, Fratantonio et al. [35] designed the "cloak and dagger" attack. Their attack combines the capabilities of the `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions, which achieves the complete control of the UI feedback loop. Aonzo et al. [28] uncovered the design issues of mobile password managers and mentioned the misuse of the accessibility service (though it is not the focus of this work). Naseri et al. [43] investigated the sensitive information leakage through the accessibility service. They found 72% of the top finance and 80% of the top social media apps are vulnerable. Different from our work, previous works focused on exploring what kinds of attacks could be achieved through a malicious app with the `BIND_ACCESSIBILITY_SERVICE` permission. Our study focused on evaluating the usage of the accessibility APIs and the design of their supporting architecture. Also, the demonstrated attacks do not need any sensitive permission.

To the security risks of voice control, Diao et al. [33] first discovered the Android built-in voice assistant module (Google Now) could be injected malicious voice commands by a zero-permission app. Some subsequent improved attacks are designed, like hidden voice commands [25, 29, 46, 48] and inaudible voice commands [45, 49]. The corresponding defense mechanisms also have been proposed, like articulatory gesture-based liveness detection [50], tracking the creation of audio communication channels [44], using the physical characteristics of loudspeaker for differentiation [30], utilizing the wireless signals to sense the human mouth motion [42].

In this paper, we present installation hijacking and notification phishing as showcases. Some other works also achieve similar attacks on Android with different approaches or adversary models, such as abusing the notification services [47], exploiting push-messaging services [39], ghost installer attack [38], and UI redressing attacks [26].

10 Conclusion

In this paper, we systematically studied the usage of the accessibility APIs and the design of their supporting architecture. Through code analysis and a large-scale apps scanning study, we identified a series of fundamental design shortcomings that may bring serious security risks. As showcases, we presented two concrete attacks exploiting these shortcomings: installation hijacking and notification phishing. As mitigations, we also propose improvement recommendations. We believe the security threats reported in this paper are not just isolated incidents. A new accessibility architecture may be needed to completely solve these flaws.

Acknowledgements

We are grateful to our shepherd Jason Polakis and the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61902148, No. 61572415, Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14217816, and Qilu Young Scholar Program of Shandong University.

References

[1] 1Mobile. http://www.1mobile.com/.

[2] AccessibilityEvent. https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html.

[3] AccessibilityEvent.java. https://android.googlesource.com/platform/frameworks/base/+/android-8.1.0_r27/core/java/android/view/accessibility/AccessibilityEvent.java.

[4] AccessibilityService. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.

[5] AccessibilityServiceInfo. https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo.html.

[6] Amazon Appstore. https://www.amazon.com/androidapp.

[7] Android accessibility overview. https://support.google.com/accessibility/android/answer/6006564.

[8] APKPure. https://apkpure.com/.

[9] Apktool. https://ibotpeaches.github.io/Apktool/.

[10] Automate. https://llamalab.com/automate/.

[11] Building Accessibility Services. https://developer.android.com/guide/topics/ui/accessibility/services.html.

[12] Building Accessible Custom Views. https://developer.android.com/guide/topics/ui/accessibility/custom-views.html.

[13] Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters.

[14] Notification. https://developer.android.com/reference/android/app/Notification.html.

[15] NotificationListenerService. https://developer.android.com/reference/android/service/notification/NotificationListenerService.

[16] PackageInstaller. https://developer.android.com/reference/android/content/pm/PackageInstaller.

[17] Part-of-speech tagging. https://en.wikipedia.org/wiki/Part-of-speech_tagging.

[18] Runnable. https://developer.android.com/reference/java/lang/Runnable.

[19] Save files on device storage. https://developer.android.com/training/data-storage/files.

[20] Section 508 Law and Related Laws and Policies. https://section508.gov/content/learn/laws-and-policies.

[21] spaCy. https://spacy.io/.

[22] Support Library. https://developer.android.com/topic/libraries/support-library/.

[23] 30 Best Google Play Store alternative as of 2018. https://www.slant.co/topics/2175/~google-play-store-alternative, 2018.

[24] Top 20 Chinese Android App Stores. https://www.appinchina.co/market/, January 2019.

[25] Hadi Abdullah, Washington Garcia, Christian Peeters, Patrick Traynor, Kevin R. B. Butler, and Joseph Wilson. Practical Hidden Voice Attacks against Speech and Speaker Recognition Systems. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-27 , 2019*, 2018.

[26] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android Case. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, 2017.

[27] Ron Amadeo. Public outcry causes Google to rethink banning powerful "accessibility" apps. https://arstechnica.com/gadgets/2017/12/google-pauses-android-accessibility-app-crackdown-after-public-outcry/, December 2017.

[28] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. Phishing Attacks on Modern Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19, 2018*, 2018.

[29] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David A. Wagner, and Wenchao Zhou. Hidden Voice Commands. In *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016*, 2016.

[30] Si Chen, Kui Ren, Sixu Piao, Cong Wang, Qian Wang, Jian Weng, Lu Su, and Aziz Mohaisen. You Can Hear But You Cannot Steal: Defending Against Voice Impersonation Attacks on Smartphones. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, June 5-8, 2017*, 2017.

[31] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, 2008.

[32] Steve Dent. Google cracks down on apps that misuse accessibility features. https://www.engadget.com/2017/11/13/google-cracks-down-accessibility-features/, 2017.

[33] Wenrui Diao, Xiangyu Liu, Zhe Zhou, and Kehuan Zhang. Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM), Scottsdale, AZ, USA, November 03 - 07, 2014*, 2014.

[34] Shaoyong Du, Pengxiong Zhu, Jingyu Hua, Zhiyun Qian, Zhao Zhang, Xiaoyu Chen, and Sheng Zhong. An Empirical Analysis of Hazardous Uses of Android Shared Storage. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[35] 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 *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 22-26, 2017*, 2017.

[36] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, AZ, USA, November 3-7, 2014*, 2014.

[37] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On Malware Leveraging the Android Accessibility Framework. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services - 10th International Conference, MobiQuitous 2013, Tokyo, Japan, December 2-4, 2013, Revised Selected Papers*, 2013.

[38] Yeonjoon Lee, Tongxin Li, Nan Zhang, Soteris Demetriou, Mingming Zha, XiaoFeng Wang, Kai Chen, Xiao-yong Zhou, Xinhui Han, and Michael Grace. Ghost Installer in the Shadow: Security Analysis of App Installation on Android. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, June 26-29, 2017*, 2017.

[39] Tongxin Li, Xiao-yong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, AZ, USA, November 3-7, 2014*, 2014.

[40] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. An Empirical Study on Android for Saving Non-shared Data on Public Storage. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, 2015.

[41] Joe Maring. Accessibility Services: What they are and why Google is cracking down on their misuse. https://www.androidcentral.com/android-accessibility-services, 2017.

[42] Yan Meng, Zichang Wang, Wei Zhang, Peilin Wu, Haojin Zhu, Xiaohui Liang, and Yao Liu. WiVo: Enhancing the Security of Voice Control System via Wireless Signal in IoT Environment. In *Proceedings of the Nineteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), Los Angeles, CA, USA, June 26-29, 2018*, 2018.

[43] Mohammad Naseri, Nataniel P. Borges Jr., 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.

[44] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), Los Angeles, CA, USA, December 7-11, 2015*, 2015.

[45] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. Inaudible Voice Commands: The Long-Range Attack and Defense. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, WA, USA, April 9-11, 2018*, 2018.

[46] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine Noodles: Exploiting the Gap between Human and Machine Speech Recognition. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT), Washington, DC, USA, August 10-11, 2015*, 2015.

[47] Zhi Xu and Sencun Zhu. Abusing Notification Services on Smartphones for Phishing and Spamming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT), Bellevue, WA, USA, August 6-7, 2012*, 2012.

[48] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, Xiaofeng Wang, and Carl A. Gunter. CommanderSong: A Systematic Approach for Practical Adversarial Voice Recognition. In *Proceedings of the 27th USENIX Security Symposium (USENIX-SEC), Baltimore, MD, USA, August 15-17, 2018*, 2018.

[49] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. DolphinAttack: Inaudible Voice Commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[50] Linghan Zhang, Sheng Tan, and Jie Yang. Hearing Your Voice is Not Enough: An Articulatory Gesture Based Liveness Detection for Voice Authentication. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[51] Yury Zhauniarovich and Olga Gadyatskaya. Small Changes, Big Changes: An Updated View on the Android Permission System. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, 2016.

Appendix

Here we give ten additional examples of accessibility service descriptions.

1. `com.yahora.ioslocker15` ⇒ "*Turn it on to receive notifications such as unread messages, unread mail, missed*

call and so on, it doesn't collect any data except receive notifications."

2. `arun.com.chromer` ⇒ "*This service will be used by Chromer to scan text links on your screen and load them in the background. No other information is processed by Chromer. You can still use Chromer in light mode if you choose to not grant this. For more information, launch Chromer app.*"

3. `com.parentsware.ourpact.child` ⇒ "*OurPact Jr. requires accessibility permissions in order to set healthy screen time limits for your children and avoid excessive or compulsive device usage. Please ENABLE accessibility permissions, then press Back twice to return to set up. Note: No personal information is collected through permissions.*"

4. `pl.damianpiwowarski.navbarapps` ⇒ "*Navbar Apps will use this service to detect active running app, which can be used to color navigation bar by active app color. This can be useful for users with disabilities as it makes Navigation Bar more distinguishable and visible.*"

5. `com.cootek.smartinputv5` ⇒ "*Turning on Accessibility makes sending GIFs easier. Flip the switch to enable GIF Keyboard.*"

6. `com.lenovo.anyshare.cloneit` ⇒ "*Open CLONEit install (Accessibility), help you click on the button when install applications from old phone. Open CLONEit install service in accessibility, confirmation dialog will pop up. Cloneit agreement, this feature is only available as install application. No other permissions, Android 4.1 or later.*"

7. `com.companionlink.clusbsync` ⇒ "*Enables DejaOffice to respond to various voice commands.*"

8. `com.joaomgcd.touchlesschat` ⇒ "*Touchless Chat allows users with disabilities to interact with several chat apps to automatically send and reply to messages without ever needing to touch the device. This can be of tremendous help for people who have trouble handling their devices with their hands. Permissions: Observe your actions: this permission is requested by default by all accessibility services. Touchless Chat doesn't need it. Retrieve window content: Touchless Chat needs to know what's on the screen so it can find text fields to paste written messages on behalf of the disabled user.*"

9. `com.ace.cleaner` ⇒ "*Turn on the above button to activate Ace Cleaner Power Mode for maximum acceleration! Ace Cleaner use accessibility features to help stopping notusing apps. Please don't worry if you see the privacy risk reminder, that's just a regular informative warning for any accessibility service. We promise NOT to collect ANY information.*"
10. `com.callpod.android` ⇒ "*KeeperFill allows you to securely and quickly fill your login credentials on websites and mobile apps. On the next screen, enable the KeeperFill keyboard.*"