AccessiLeaks: Investigating Privacy Leaks Exposed by the Android Accessibility Service

Mohammad Naseri, Nataniel P. Borges Jr., Andreas Zeller, Romain Rouvoy

Mohammad Naseri*, Nataniel P. Borges Jr., Andreas Zeller, and Romain Rouvoy

AccessiLeaks: Investigating Privacy Leaks Exposed by the Android Accessibility Service

Abstract: To support users with disabilities, Android provides the *accessibility services*, which implement means of navigating through an app. According to the Android developer's guide: *"Accessibility services should only be used to assist users with disabilities in using Android devices and apps"*. However, developers are free to use this service without any restrictions, giving them critical privileges such as monitoring user input or screen content to capture sensitive information. In this paper, we show that simply enabling the accessibility service leaves 72 % of the top finance and 80 % of the top social media apps vulnerable to eavesdropping attacks, leaking sensitive information such as logins and passwords. A combination of several tools and recommendations could mitigate the privacy risks: We introduce an analysis technique that detects most of these issues automatically, *e.g.* in an app store. We also found that these issues can be automatically fixed in almost all cases; our fixes have been accepted by 70 % of the surveyed developers. Finally, we designed a notification mechanism which would warn users against possible misuses of the accessibility services; 50 % of users would follow these notifications.

Keywords: Accessibility service, Android, privacy

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

Smartphones and mobile applications (apps) are widespread. From banking to social network, there are apps to assist most of our daily activities. To perform their tasks, apps frequently request or interact with sensitive user information, which raises privacy and security concerns. While one may expect app stores, such as Google Play Store, to filter out malicious apps on behalf of end-users, malicious apps are still widespread [4, 11].

A critical security issue of any software system is *authentication*. If, during this step, a user's information is intercepted by a malicious agent most, if not all, sensitive user information managed by the app could be collected. In addition, due to password reuse [13, 24], sensitive information captured on an unsafe app (weak-link) may compromise safe apps. Just as regular systems, Android devices are vulnerable to password interception techniques, such as key loggers and touch loggers [6]. While many malicious apps exploit security vulnerabilities, users with a disability, or those which prefer to use apps for enhanced accessibility, are vulnerable through an additional attack channel: the *Android accessibility service.*

The Android accessibility service is designed to provide alternative navigation feedbacks to the device user, such as converting text to speech or producing haptic feedback when the user hovers over an area of the screen. It is a standard feature of Android since version 1.6 (API 4). Once activated, this OS service allows other apps to monitor the current screen content, with restrictions, enabling developers to create apps which provide different accessibility features. By allowing developers to create new accessibility features, Android also enables the development of malicious software, which can monitor keyboard and screen for sensitive information such as login and password. Recently, Fratantonio et al. [10] demonstrated how a combination of enabling the Android accessibility service and controlling alert windows can enable a malicious app to control arbitrarily other apps.

In this paper, we investigate sensitive information leakage through the Android accessibility service, requiring nothing but the service to be enabled. We find that enabling accessibility services *exposes a large majority of apps to serious eavesdropping attacks, leaking passwords and other sensitive information.* We therefore design and introduce solutions that can mitigate the problem for app store moderators, for developers, and for end users. After introducing the Android acces-

***Corresponding Author: Mohammad Naseri:** Saarland University, E-mail: s8monase@stud.uni-saarland.de
Nataniel P. Borges Jr.: CISPA Helmholtz Center i.G., E-mail: nataniel.borges@cispa.saarland
Andreas Zeller: CISPA Helmholtz Center i.G., E-mail: zeller@cispa.saarland
Romain Rouvoy: Univ. Lille / Inria / IUF, E-mail: romain.rouvoy@univ-lille.fr

sibility service and how it can be maliciously exploited (Section 2), we make four contributions:

Extent of the problem. *Simply turning on the accessibility service exposes a large fraction of apps to vulnerabilities.* In our study (Section 3), we found that 72 % of the top 50 finance apps and 80 % of the top 50 social media apps in the Google play store are vulnerable to attacks via the accessibility service.

Detecting vulnerabilities. *App stores could detect most of the issues with automatic support.* Our AcDetect tool (Section 4) allows app store moderators to identify and flag apps which could access sensitive information from other apps, when accessibility services are enabled, and apps which are vulnerable to this problem.

Leak mitigation. *Developers benefit from tools that automatically fix the issue.* Our AcFix prototype (Section 5) was able to automatically fix the sensitive information leak in 40 open source apps tested. 70 % of the developers surveyed already have integrated the fixes suggested by AcFix.

User confidence and expectation study. *End users can (and should) be warned against abuse of accessibility services.* Our AcGuard tool (Section 6) allows end users to identify which apps could be monitoring its inputs and to choose how to behave when faced with this issue. In a user study, 50 % of all participants would follow the AcGuard warnings and not provide sensitive information.

After addressing threats to validity (Section 8), Section 9 discusses the related work, before we conclude in Section 10.

2 Attack Model

Why and how does the Accessibility Service introduce vulnerabilities? Figure 1 illustrates a typical vulnerability scenario—eavesdrop on passwords being typed in another app. When a device's accessibility service is enabled, both benign and malicious apps can register as listeners to be notified by the service. When a user interacts with a benign, yet vulnerable app, the OS service monitors this interaction and notifies all listeners, which can then leak it. Our attack model thus is composed of three major components: *enabled accessibility*, a *malicious accessibility service listener* and a *vulnerable app,* detailed below.

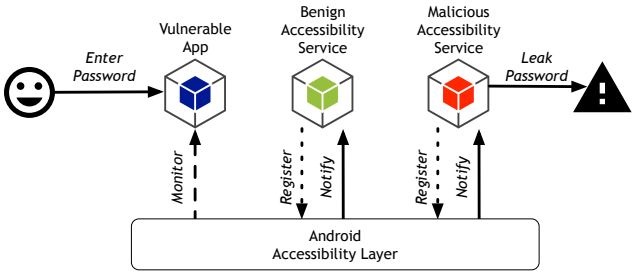


Fig. 1. Accessibility service password eavesdrop vulnerability exploitation. Malicious and benign services register and listen to interactions. Vulnerable apps broadcast passwords to all listeners.

2.1 The Enabled Accessibility Service

The first component of our attacker model is the accessibility service. By default, it is not enabled on Android and, for security reasons, requires explicit user consent to be activated, while manually accessing the *Accessibility* section of the device settings. Once enabled, it runs in background and works by an observer design pattern, where other apps can register themselves as listeners (to provide accessibility functionality). Once an event is triggered, all apps registered as listeners for that type of event are notified. There are currently 25 accessibility events and the information shared to each app is determined according to the listener's capabilities.

2.2 The Malicious Listener

The second component of our attacker model is an app which registers itself to the accessibility service. To register itself as an accessibility service listener (*bind*), an app must specify a configuration, which includes the accessibility events that it wants to listen (*e.g.*, clicks on views, text changes), as well as its capability level. While this configuration can be created dynamically, the required capabilities must be statically defined and cannot be changed at runtime. The major capabilities are: *key press events*, *retrieve window content*, *touch exploration mode* and *enhanced web accessibility*.[1] To exploit this vulnerability, our attacker model does not request any specific capability —thus getting access to the lowest and most restrictive level of information– we show

1 https://goo.gl/Y2Hkm3

that, even in this configuration, apps can behave maliciously since the *Text* attribute of *text changed* events is fired independently of the listener capabilities.

Apps can prevent information leaks through accessibility services by setting a single flag for sensitive inputs.

2.3 The Vulnerable App

In this work, we define a vulnerable app as an app which possess a password text input field whose data can be captured. In Android, when developers require a user input, they can use the standard `EditText` class or they can create their own custom class which should inherit from Android's one for backward compatibility.[2] An `EditText` is an overlay over *TextView* that configures itself to be editable. When an end-user enters a text in an `EditText` field, an accessibility event is generated and forwarded to all registered accessibility services. Among other data, the event's *Text* attribute includes the raw text that the user typed in the `EditText`, which can then be read by the registered services. Such information may reveal highly user-sensitive data, such as credentials or credit card numbers, even if developers define the input type as a password. In this case, the last character typed by the end-user is exposed by an accessibility event and, monitored in cascade, reveals the complete password.

Each text edit field can be configured to trigger or suppress events to the accessibility services, through their `importantForAccessibility` attribute, which can take four different values:

- *yes* indicating that the user input is important for accessibility and should be forwarded;
- *no* indicating that the user input is not important for accessibility and should not be forwarded;
- *noHideDescendants* indicating that the user input nor are any of its child views are important for accessibility and should be not be forwarded;
- *auto* allowing the system to determine whether the user input is important or not for accessibility (recommended by the Android documentation).

Configuring this value as *yes* or leaving it as *auto* (default and recommended value) does not prevent passwords from being sent to the accessibility service. Setting the attribute value to `no`, however, is sufficient to prevent the services retrieving the text value of the `EditText`.

2 https://developer.android.com/reference/android/support/v7/widget/AppCompatEditText

3 Extent of Accessibility Service Vulnerabilities

Motivation. We aim to measure how vulnerable to leaks through the accessibility services are popular apps. In this paper, we are focusing on Google Play Store, the main official Android market. With this goal we evaluated the top 50 apps[3] in *Finance* and *Social* categories in Google Play Store manually in order to assess its effectiveness. We specifically chose Finance and Social apps due to their possible impacts (transfers of funds and access to personal information). In addition, we chose top apps due to their widespread usage, with millions of users each, and focused exclusively on password leaks in log-in page of the app as they could be manually inspected without an app account.

Approach. We crawled the Google Play store for the top 50 finance and actively attempted to collect passwords while manually operating them. For this task, we used the open source app *Bee*[4], which uses the APISENSE mobile crowdsourcing platform[5] [12] to collect data from common smartphone sensors. We extended *Bee* to register to the accessibility service with no specific capability, thus receiving the least amount of information. Our version of *Bee* runs on the background and logs user inputs (including passwords) from apps running on the foreground. All input data is only stored in the device's file system and all passwords are hashed before storage. Eventually, *Bee* would report only privacy leak statistics, including the number of leaked passwords and the list of victim apps.

Results. On a per category analysis, from the Finance apps, we identified that 36 of top 50 apps (72 %) are vulnerable to the eavesdropping of a user's log-in passwords through the accessibility service, including the apps listed on Table 1, all with over 1,000,000 down-

3 We selected the top 50 apps with account management and money transferred capabilities, we discarded apps used only for stock price monitoring as they did not require any user account.
4 https://play.google.com/store/apps/details?id=com.apisense.bee
5 https://apisense.io

loads.[6] Password leak in finance category such as banking apps is a high crucial issue which emphasizes how important this vulnerability is.

For the Social apps, our manual evaluation confirmed the existence of the vulnerability on 40 out of 50 (80 %) apps, including the apps listed on Table 2, all with over 100,000,000 downloads.

We detected and exploited accessibility service vulnerabilities in the Google Play Store for ***72 %*** *for the top 50 Finance apps and* ***80 %*** *of the top 50 Social media apps.*

4 Detecting Accessibility Vulnerabilities

For widespread usage, developers publish their apps in an app store, which are then accessed by millions of users. In order to publish an app a developer needs only to submit a signed, binary of the app (APK), alongside some metadata information—*i.e.*, no source code is required. To assist the identification of vulnerable apps we developed AcDetect, a tool to analyze an Android app binary for accessibility service vulnerabilities.

4.1 The Technique

AcDetect starts by extracting all resources from the APK using Apktool. Among these resources are user defined strings and layout files, which define the *user interfaces* (UIs) and UI elements of an app. AcDetect then parses these resources to locate vulnerable input fields. An input field is considered vulnerable if it is *input field*, contains a *password* and is *important for accessibility*.

AcDetect considers as an *input field* all UI elements whose type is `EditText` or which extend `AppCompatEditText`. It then classifies input fields as a password or not, according to their `android:inputType` attribute.[7] If the field contains a password, AcDetect checks the field is vulnerable by inspecting its `importantForAccessibility` attribute. If the value is *yes*, *auto* or if no value is assigned to the attribute, it flags the app and the field as vulnerable to eavesdropping through the accessibility.

Commercial apps frequently have their source code and resources shrunk and obfuscated by tools, such as ProGuard. For this reason AcDetect does not inspect the apps compiled source code for elements created at runtime, as well as for *input type* and *important for accessibility* assignments, for the privacy leak in the decompiled source code of the app. Resource XML files are also obfuscated in this process, in a more limited way. According to the ProGuard documentation[8], if the attribute of an element includes a numeric value, then an obfuscation step will be applied on the attribute. Since `android:inputType` and `importantForAccessibility` attributes do not include any numeric value, AcDetect is not affected by the ProGuard obfuscation.

4.2 Evaluation

Motivation: We aim to evaluate that how effective AcDetect is in finding app vulnerable to leak through the accessibility services.

Approach: For this evaluation we re-use the previously evaluated dataset of top 50 *Finance* and *Social* apps. We used AcDetect to automatically detect vulnerabilities on all 100 apps and manually discarded vulnerabilities found outside of the apps main login screen, as they could not be accessed during exploration without an app account, and compere these results against the manual evaluation. We consider the results of our manual analysis as ground truth – as all vulnerabilities were concretely exploited.

Results: Figure 2 shows the evaluation of AcDetect as a confusion matrix, considering all 100 apps Finance and Social categories on the Google Play Store. Our results indicate that AcDetect correctly identified 71 out of 76 vulnerabilities and identified 9 false positives, resulting in a precision of 89 % and a recall of 93 %. On a per category analysis. AcDetect correctly identified the vulnerability in 31 out of 36 evaluated *Finance* apps and on all 40 vulnerable *Social* apps.[9]

6 The full list of apps is available at https://github.com/uds-se/AcTools/wiki/AcDetect

7 https://developer.android.com/reference/android/text/InputType

8 https://www.guardsquare.com/en/products/proguard/manual/usage

9 https://github.com/uds-se/AcTools/wiki/AcDetect

Table 1. Top-5 apps in the Finance category and their vulnerability to accessibility service eavesdropping according to AcDetect and manual checking

App	Downloads	AcDetect	Manual
com.google.android.apps.walletnfcrel	100,000,000+	Y	Y
com.starfinanz.smob.android.sfinanzstatus	5,000,000+	Y	N
com.paypal.android.p2pmobile	1,000,000+	Y	Y
at.paysafecard.android	1,000,000+	Y	Y
com.westernunion.moneytransferr3app.eu	1,000,000+	Y	Y

Table 2. Top-5 apps in the Social category and their vulnerability to accessibility service eavesdropping according to AcDetect and manual checking

App	Downloads	AcDetect	Manual
com.facebook.katana	1,000,000,000+	Y	Y
com.instagram.android	1,000,000,000+	Y	Y
com.snapchat.android	500,000,000+	Y	Y
com.pinterest	100,000,000+	Y	Y
com.linkedin.android	100,000,000+	Y	Y

	Classified as			
Input	**True**	**False**	**Total**	
True	TP = 71	FN = 5	76	*Precision* = 89 %
False	FP = 9	TN = 15	24	*Recall* = 93 %
Total	80	20	100	*Accuracy* = 86 %
				Specificity = 63 %

Fig. 2. Confusion matrix for presence (True) and absence (False) of accessibility service vulnerability among the top 50 apps in the Finance and Social categories on the Google Play Store.

AcDetect *detected accessibility vulnerabilities in the top 50* Finance *and* Social *apps with a precision of 89 % and a recall of 93 %.*

4.3 Discussion

We presented AcDetect to identify if an app can leak passwords through the accessibility service with approximately 90 % precision and recall. Based on this data, we decided to further analyze how users are currently warned by the Google Play Store, when apps require access to the accessibility services.

The Android developer's guide determines that *"accessibility services should only be used to assist users with disabilities in using android devices and apps"*.[10] The position of the Google Play Store regarding apps which request access to accessibility services is to email developers, asking them for a justification to access such a service.[11] If no acceptable answer is received, the app is removed from the Play Store. Despite the aforementioned policy, developers have used this service to provide distinct functionality and a quick search on the Play Store, reveals a large number of apps using accessibility services for, among other functionalities, password management, such as *LastPass* app[12], which is used by over a million users. In addition, our experimental *key logger* app, used in our evaluation, received no notification during the 4 months (January 2018 till April 2018) we kept it active in the Play Store.

The accessibility service is used by 2,815 apps out of 4,155,414 from the AndroZoo [3] repository, which is in turn crawled from the Google Play Store. To listen to accessibility events the developer must request the `android.permission.BIND_ACCESSIBILITY_SERVICE` permission in the app's manifest. If this information is requested at the application level, it is displayed in the app download page in the Google Play Store, as shown in Figure 3. If this information is requested only at the service level which is mandatory by Android framework, then no record of the permission is mentioned from Google Play, meaning that users are not notified beforehand that the app will be able to listen accessibility events.

10 https://goo.gl/hKmfFm

11 https://9to5google.com/2017/11/12/apps-android-accessibility-services-removed-google-play-store/

12 https://goo.gl/rgYqFX

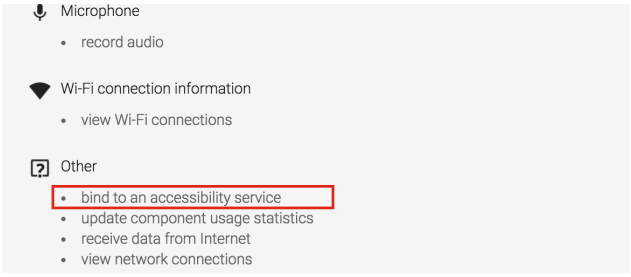


Fig. 3. Google Play Store screenshot informing about requested permissions when the accessibility service is declared in the application level

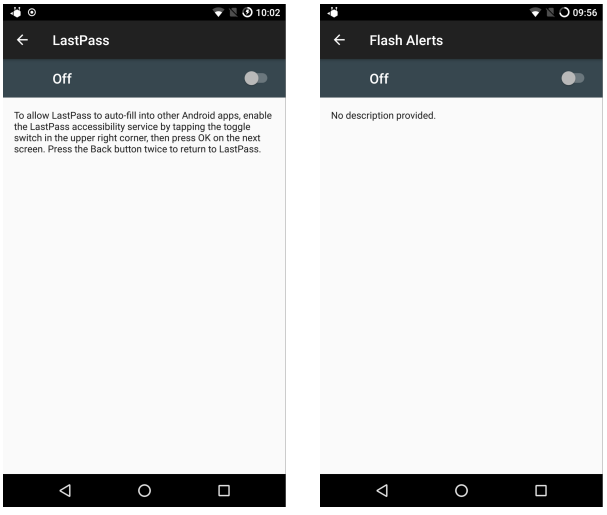
We then extended AndroGuard [7] to identify where apps declare the permission and we identified that only 1,247 apps (44.2 %) declare it the *application* level. This implies that 55.8 % of the accessibility apps do not notify the users about their accessibility service use and, while these apps are not necessarily malicious, the lack of notification does not allow the user to observe it.

To assess which other information are gathered by apps which use accessibility, we analyzed the permissions requested by the accessibility apps in addition to the accessibility service. Table 3 reports on the top 15 permissions that we observed in our data set. According to official documentation[13], 6 permissions out of top 15 permissions requested alongside the accessibility service have dangerous protection level, which again emphasizes the potential threats. For example, the `INTERNET` permission allows a malicious app to send any sensitive data captured with accessibility service to the Internet. An additional threat caused by the combination of Internet and Accessibility permissions are app updates. Since users were not properly notified on both permissions – as they are not categorized as dangerous – an app which currently does not leak data may be automatically updated to leak this information, without any user notification.

In addition to a Google Play Store notification, apps can provide a description about how they use the accessibility service. This service description is displayed to the user, when it enables or disables the service through Android settings menu, as illustrated in Figure 4. This attribute is, however, optional and can be omitted. Out of 2,815 apps, 271 (9.5 %) do not have a description for the accessibility service. Not having a description for the accessibility service does not characterize an app as malicious. However, the combination of a lack of notification on the Google Play Store with a lack of service use description does not allow a user to identify if and why an app would use this functionality.

Table 3. Top-15 requested permissions in apps with accessibility service

Requested permission	% of apps	Protection level
`INTERNET`	97.1 %	normal
`ACCESS_NETWORK_STATE`	92.6 %	normal
`WRITE_EXTERNAL_STORAGE`	81.6 %	dangerous
`READ_PHONE_STATE`	76.3 %	dangerous
`RECEIVE_BOOT_COMPLETED`	70.3 %	normal
`WAKE_LOCK`	69.3 %	normal
`SYSTEM_ALERT_WINDOW`	64.9 %	signature
`ACCESS_WIFI_STATE`	62.2 %	normal
`GET_TASKS`	62.0 %	deprecated
`VIBRATE`	61.9 %	normal
`GET_ACCOUNTS`	45.2 %	dangerous
`CAMERA`	43.9 %	dangerous
`READ_CONTACTS`	41.8 %	dangerous
`WRITE_SETTINGS`	41.1 %	signature
`READ_EXTERNAL_STORAGE`	40.5 %	dangerous



(a) With description **(b)** Without description

Fig. 4. Accessibility settings screen of an app with and without description of why it requires connection to the accessibility service. The text is provided by the user

The lack of information regarding the usage of accessibility service by apps hides accessibility service vulnerabilities. By notifying the users prior to the download of the app, app stores can play a role in mitigating the threat to the user, without removing benign apps.

13 https://developer.android.com/guide/topics/permissions/requesting.html.

5 Automatically Fixing Accessibility Vulnerabilities

While AcDetect can identify elements which can be potentially eavesdropped through the accessibility service, it processes *compiled* apps, in order to be effective at analyzing large amounts of apps. Developers, however, do not need to analyze several apps, but simply their own. In addition, they have access to the app's non-obfuscated source code, allowing for a more accurate detection. To assist developers in addressing these issues we developed AcFix, which 1. analyzes the source code of an Android app, including XML layout, 2. spots occurrences of such vulnerable code segments and 3. attempts to fix them.

5.1 The Technique

In Android development, all the views need to be bound to a layout, either at compile or at runtime. Additionally, all views which are referenced from source code require an `android:id` attribute. This identifier is unique for each view in the whole project. The binding between a layout and a UI element can occur in three locations:

- *XML layout file*: The developer defines an editable text field by creating a tag `EditText`. The declared text field can be retrieved from the source code by querying the *id* of that view;
- *Java source file*: The developer can define an editable text field by creating an instance of an `EditText` class and programmatically binding the object to a layout;
- *Annotation libraries*: The developer can annotate their source code and use libraries, such as Butterknife[14], to bind the view and the layout during runtime.

AcFix is developed as an extension to Spoon [22][15], a tool that parses Java source files and builds an *Abstract Syntax Tree* (AST) with analysis and transformation API. AcFix uses this AST to analyze the app source code and to detect the sensitive user input fields which can be eavesdropped through the accessibility service, supporting the detection of both statically and dynamically created objects. AcFix works as follows:

1. It generates the project's source code AST using Spoon;
2. It uses the AST to process all layout files and Java classes in order to find all the *password text fields*, covering all three forms of bindings.
3. reuses the *password text field* definition from AcDetect.
4. For each candidate field:
   (a) If the field is created in an *XML layout file*, AcFix checks if there is any assignment to the attribute `importantForAccessibility` in the XML layout file.
   (b) If the value is *yes*, *auto* or if no value is assigned to the attribute, it checks the the associated source code element, according to the AST, for an assignment.
   (c) If no assignment is found, AcFix automatically inserts an attribute `importantForAccessibility` with value *no* to the XML object.
   (d) If the field is created in the *source code* and it is not related to an XML layout object, AcFix searches the AST for a call to the method *setImportantForAccessibility*.
   (e) If the method input value is *yes*, *auto* or if no value is assigned to the attribute, it adds a new instruction in the source code, on the *onCreate* event callback of the activity triggering the *setImportantForAccessibility(no)* on the attribute associated to the field.

5.2 Evaluation

Motivation: We previously identified that many popular apps are vulnerable to eavesdropping through the accessibility service and proposed a tool (AcFix) to assist developers in automatically fixing this issue. We want to evaluate the effectiveness of such automated fixes, as well as the developers acceptance of them.

Approach: AcFix needs to be applied on the source code of the projects, thus, we could not apply it on the same dataset used to evaluate AcDetect. We instead targeted *open source* Android projects. We crawled the F-Droid[16] app repository and located 40 apps vulner-

14 https://jakewharton.github.io/butterknife

15 http://github.com/inria/spoon.

16 https://f-droid.org/en.html

able to the accessibility leak. We then applied AcFix on each project and manually evaluated its results. We issued a request to the maintainers of each project to merge the changes automatically made by AcFix into the apps main repositories.

Results: AcFix was able to automatically fix the problem in all 40 apps.

> AcFix *was able to automatically fix the sensitive information leak in all 40 apps tested.*

The fixes from AcFix also have been adopted by developers. At the time of writing, 28 out of the 40 merge requests (70 %) we submitted have already been merged. The maintainers of two other projects (5 %) questioned the impact of this change on people with disabilities.[17] The remaining ten merge requests (25 %) had no answer from the project maintainer, indicating the project is no longer under maintenance.

> *70 % of the developers already merged the pull request with the changes made by* AcFix.

5.3 Discussion

We introduced AcFix to help developers to fix the password privacy vulnerability in their projects. While the fixes make the password input secure, they hamper accessibility for people with disability. AcFix changes thus imply a trade-off that needs to be decided by developers.

From our 40 merge requests, we received only two replies questioning this issue and waiting for a Google provided solution.

> *Disabling accessibility services for a field improves security, but reduces accessibility.*

During our experiments we identified that some apps solved this issue manually, by implementing *biometric log-in.* This functionality, however, is only available from Android 6.0 (API 23) onwards[18], which would make apps unable to run on older devices. While fingerprint sensors become more and more common on handheld devices, they may not be a viable alternative to users with disabilities.

6 Warning End Users Against Accessibility Vulnerabilities

While using their phones, users are currently not notified when an accessibility app, malicious or not, is eavesdropping on another. They can only access this information through the device's configuration screen. To keep users informed about potential privacy risks, we developed AcGuard. It monitors the user interactions and notifies users about potential threats before they enter a password.

6.1 The Technique

To notify the user, AcGuard implements an accessibility service without any specific capability. AcGuard listens for accessibility events and waits until it identifies the user is about to enter a password. It identifies that the user is about to type a password by monitoring the sequence of accessibility events it receives. An event of type `TYPE_VIEW_TEXT_SELECTION_CHANGED`, on a field flagged as `isPassword`, immediately followed by an event of type `TYPE_VIEW_TEXT_CHANGED`, indicates that user clicked on an input field and is going to type the first password character.

At this moment, AcGuard queries the OS for the list of enabled accessibility services. If any service is found, it pushes a notification to the user, informing it about which apps could be accessing this information, as depicted in Figure 5. The user can then click on the notification and be redirected to the accessibility setting, where it can disable those services and prevent them from capturing the password for the moment. After entering their passwords, users can resume disabled services benefit again from their functionality. If no service is listening accessibility events, AcGuard informs the user, which can continue to input its password, as depicted in Figure 5. For improved usability, AcGuard possesses a *Trust* button, so that the user can white-list the application and not be notified about it again.

17 https://github.com/uds-se/AcTools/wiki/AcFix

18 https://developer.android.com/about/versions/marshmallow/android-6.0.html

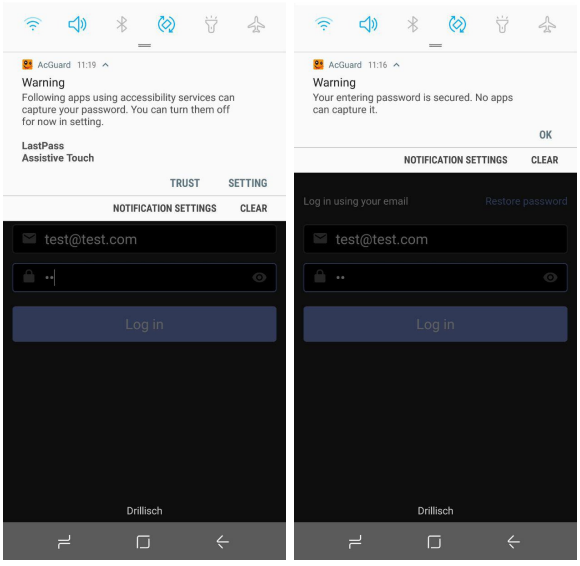


Fig. 5. Warning messages displayed by AcGuard for active accessibility services (left) and secure notification (right) when the user is entering a password.

6.2 Evaluation

Motivation: We proposed solutions to mitigate accessibility leaks for three stakeholders: users, developers and app stores. We aim to evaluate how the warning notifications provided by AcGuard affect the users, as well as identifying who the users would expect to fix the problem.

Approach: We performed an online survey with four questions on 50 Android users—between 18-30 years old university students from science, engineering and social sciences background—to evaluate how AcGuard's notification would impact their feeling of security on the app.

Regarding the ethical aspects of the user study. Each individual answering the study was allowed to cancel it at any point before submission and we considered only completed surveys for the results. Additionally, the surveys were answered anonymously, that is, not only no personal data is collected but it was also not possible for the researchers to link any specific user to a specific answer. Given our evaluation protocol, we discussed details of the user study with the Institutional Review Board (IRB) of Inria, who exempted us from an IRB review procedure as no ethical issue was raised.

We presented the users a scenario where the users would be using a finance app with banking capabilities (account checking, money transfer, etc.) on their own personal phones, running on the latest version of Android with all official security patches installed and with accessibility services enabled for enhanced usability.[19]

We then provided three screenshots of the finance app's login screen, the first without any notification, the second one with a *No apps can read your password* notification from AcGuard and a third with a warning informing the user that some apps on their phone (*LastPass and Assistive Touch* in our scenario) could be reading their passwords, as shown in Figure 5. For each screenshot we asked the user to how confident they felt—on a 10 point scale—about entering the password on the app, without it being stolen. The questions were presented one at a time and the users were only able to advance to the next question after answering the previous one. These questions aim to measure how AcGuard affected the user's confidence in the app.

After answering the question related to the last screenshot, we asked the users what they would do given the warning they just received. We used this question to identify how they expected the problem to be solved. The users were given the following options:

1. I will not enter my username and password on this app under any circumstance
2. I will follow the notification and deny, only when I'm using this app, access to the apps which can read my login data (LastPass and Assistive Touch)
3. I will uninstall the apps which can read my information (LastPass and Assistive Touch) because they can be reading this information on other apps as well
4. I will look for another finance app which is more secure
5. I will press "Trust" in the notification message and type my login data

Results: Figure 6 shows the three box plots of users' confidence levels for three different snapshots: no notification (Q_1), secure notification (Q_2) and warning notification (Q_3). The majority of the users had a confidence level between 5 and 6 points for Q_1, with 2 outliers with very high confidence level and 3 outliers with very low confidence level. For Q_2, the confidence level of most users stayed between 5 and 8 points, with some users having a confidence interval as low as 2 points or as high as 10. For Q_3, the majority of the users had a confidence level between 2 and 5 points, with values predominantly between 2 and 3 points. While there is an

19 Link to survey: https://goo.gl/Kz9eza

one outlier with confidence level 10, meaning that there was a user who trusted the displayed apps and was confident that its password could not be captured, confidence level decreased in most of the users with warning notification. The median confidence level value of users increased from Q_1 to Q_2 by 2 points and it decreased by 4 points when comparing Q_2 to Q_3.

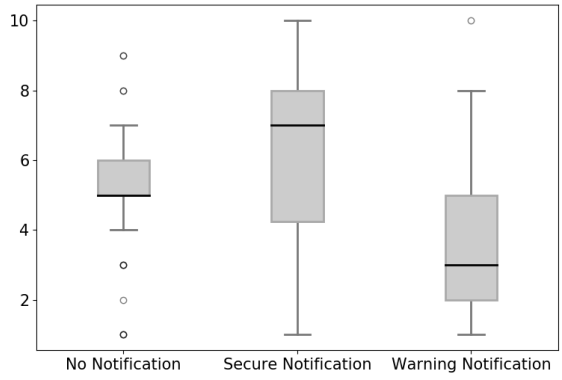


Fig. 6. Users' confidence level for three notification states

In Figure 7 we can observe the chart for our last question (users' response to the warning notification). 50 % of users intended to follow the notification provided by AcGuard and 36 % of the users in our study would either remove the apps which use the accessibility service or look for an alternative which is not vulnerable to it. It also shows that, given a warning regarding which apps can monitor their passwords, 14 % of the users would not enter their passwords on the study app under any circumstance. No user would white-list the apps in the notification and enter its password.

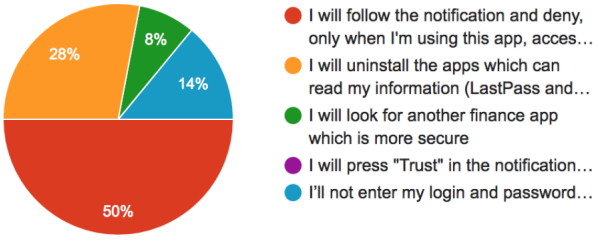


Fig. 7. User responses

Table 4. Average and standard deviation of confidence level values of the three user study questions results

Snapshot State	Median	Average	Standard Deviation
No Notification	5	5.3	1.8
Secure Notification	7	6.3	2.5
Warning Notification	3	3.8	2.3

We also extracted the average value and standard deviation for the studies' values in Table 4. We observe that, similar to the median, the average confidence level of users increased with secure notification and decreased with warning notification. Evaluating individual results, 32 users had a confidence level value of more than 5, when AcGuard notified them with a secure notification, while 22 users had confidence level value of more than 5 when there was no notification. 33 users had a confidence level of less than 5 when AcGuard notified them with a warning message. In order to support our experiment, we performed a Friedman test for repeated measures to check the statistical significance of our results. By comparing the answers for Q_1, Q_2 and Q_3 our experiments resulted in a p-value < 0.00001 being thus significant at $\alpha = 0.05$ (5%) and showing that AcGuard could affect users' confidence while entering the password.

With its notifications, AcGuard *could affect the users' confidence in an app. 50 % of users followed the* AcGuard *recommendation.*

6.3 Discussion

Our user study showed that 50 % of the users would follow the recommendation provided by AcGuard. Another 14 % of users from our study would not enter their password under any circumstance in the app with the accessibility vulnerability. The Android operating system can play an important role in addressing these concerns.

Native notification. While AcGuard helps users to identify and react against malicious apps using accessibility services, increasing or decreasing their confidence to enter their password in an app, it is still necessary that user installs and trusts it, as it exploits the accessibility service vulnerability to warn user of potential threats. A notification similar to AcGuard could be provided by the OS, for a widespread mitigation of the vulnerability.

User Interface. To enable our app's accessibility service, Android warns users with the message displayed

in Figure 8. However, this message—provided by the Android operating system—is uninformative regarding possible vulnerabilities, including the possibility of capturing user inputs while typing passwords. For most users, this notification seems legit, thus they usually confirm it. Studies, such as [15], illustrate the importance of meaningful messages.

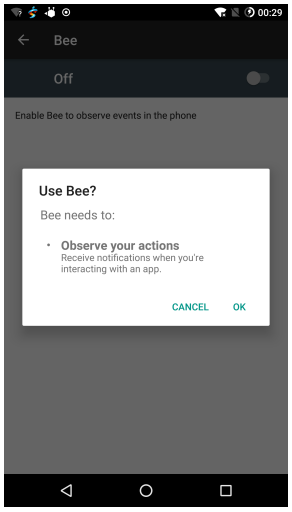


Fig. 8. Warning message displayed by Android when enabling services with no capability

Impact on Users with Disability. AcFix and AcGuard are the tools that we developed to mitigate the existing accessibility vulnerability. Nevertheless our solutions can impact people with disability. AcFix, which is specifically designed to fix password vulnerability, disables accessibility feature for password inputs. This may trouble users with disabilities, such as people who are blind or have low vision, who are unable to see screens and hence cannot use touchscreen keyboards or users with dexterity problems in using their fingers. That is the reason why we emphasized that AcFix is a quick solution for apps in the wild to be protected against such attacks. Google has provided accessibility services, such as *TalkBack*, *Select to Speak* and *Text-to-speech*[20] to help users with aforementioned disabilities. Such services are integrated in Android operating system. However, those services are not widely used by users. For instance, Rodrigues et al. [23] showed the usability of *TalkBack* is pretty daunting for users with visual impairments. Naftali and Findlater [19] found that users with motor impairments have challenges using services like *Select to Speak*. Although we can not generalize such experiments, it shows that the provided services are not widely used among users with disabilities and the current accessibility vulnerability in the wild is more severe. AcGuard does not impact users with disability as it does not enforce the user to disable any accessibility service. It just warns the user and usually people with disabilities are familiar with the services that they use occasionally. In the following, we would like to discuss the operating system changes that would achieve both accessibility and privacy of user data.

OS Changes. For the Android operating system, based on our studies, we can formulate some suggestions that would contribute to improve the security of the current model in terms of the accessibility services:

- **Permission Model.** In the current model, there is only one permission `BIND_ACCESSIBILITY_SERVICE` to request access to the accessibility service, which allows the app to receive all the accessibility events, no matter the app is in background or foreground. This issue is exploited by malicious apps, which listen and receive the events including information about other apps. A finer-grained permission model, with distinct permissions for accessibility services: *i)* a permission to receive accessibility notifications only from the foreground app, which can be considered as benign since it only receiving events regarding their own app, and *ii)* one permission for apps running in the background apps, which could be easily identified by users and would be required to provide a comprehensive description explaining their usage of the service before publication.
- **Predefined Accessibility Services.** As we explained before, Google has provided predefined set of accessibility services to help users with disability which developers can utilize in their application. The problem is that there is not difference between those services and new services, which are declared by programmers in current operating system structure. Once an event is fired, all enabled services receive the event. Since predefined services are hugely used by users with disabilities, a better solution would consist in separating the event's type of these two. In that case, developers can utilize predefined accessibility services in their app, such as password fields, without worrying about other services receiving triggered events.

20 https://support.google.com/accessibility/android/answer/6006564?hl=en

7 Extensions of Tools

In this paper, we focused and tuned our tools on password inputs as the main method to protect mobile phones. This section explains how aforementioned tools and techniques can be extended to cover additional types of sensitive data. The initial point of our attack model in Figure 1 is when the user inputs data, assuming accessibility service is enabled. Input controls can be text fields, buttons, checkboxes, radio buttons, toggle buttons, spinners and more. The common attribute of all the input control layouts is that they extend the Android *View* class. The *View* class has the attribute *importantForAccessibility*, so children classes inherit the attribute. In this way, the enabled accessibility service can receive callbacks regarding the input layout. The information that service can extract from the received callback depends on the type of the layout, but in general the content and user inputs can be retrieved from the callback event. The level of data privacy is not our concern here but the way that we can extend our tools. In the following, we provide a generic approach, which can be used to extend AcDetect, AcFix and AcGuard to cover other user inputs:

AcDetect *Extension*: In current state of AcDetect, the target UI layout is password *EditText*. Assuming the generic UI layout type is L, AcDetect can be extended to identify apps with accessibility vulnerability through L. First, it extracts the resources of the app. Then, it parses the resource files and finds all elements with type L. Afterwards, it checks the *importantForAccessibility* and marks the element vulnerable accordingly.

AcFix *Extension*: AcFix can also be extended to fix the vulnerability in the source code of the projects. It inspects both resource and source files and finds all defined variables with type L. It checks the accessibility attribute of fined elements. Accordingly, it changes the source code to provide the security for the elements.

AcGuard *Extension*: To extend AcGuard to cover UI elements with type L, we need to filter the receiving events corresponding to type L and then notify the user about the vulnerability.

8 Threats to Validity

The presented approach and experimental evaluation raise several limitations and threats to validity.

Regarding external validity, we have applied AcFix on 40 open source projects and we received 28 positive responses. We have not received any negative response from others yet, but to increase the confidence in AcFix, a larger dataset of apps should be used. Additionally, for the AcDetect evaluation, we selected 100 popular apps for our analysis, while this may not be representative of the app store as a whole, these apps possess access to critical information and, being developed and maintained by large companies, tend to adhere to up-to-date security practices of the industry.

In our user study for AcGuard, we gathered data from 50 users. The majority of these are 18–30 years old university students, which is not representative for the overall Android user population. A study with users from more diverse age and background, as well as users with currently use and those who do not accessibility services, may lead to different results. Users whom the Android accessibility service primarily aims at may have different opinions from those observed in our study.

Finally, the password input setting which displays the last typed character of a password is configurable. Changing this setting prevents accessibility services from accessing the password; however, it also impacts the user experience.

Regarding internal validity, dynamically created widgets in apps *(false negatives)* or widgets whose accessibility is defined in the source code *(false positives)* are not detected by AcDetect since it analyses only the embedded resource files. This problem is specially common on *web apps* whose content is dynamically generated by a Web view. Control and data flow analysis can be applied to the apps source code to improve the detection precision and recall; These techniques however may lead to problems in conjunction with app code obfuscation, which is a standard industry practice.

9 Related Work

Android keeps being exposed to various security and privacy attacks, which threaten the privacy of end users. We categorize the related work as follows.

9.1 Accessibility Services

Kraunelis et al. [17] were the first to identify Android accessibility services as a possible attack vector. They implemented a malicious app that used the Android ac-

cessibility service to masquerade as a legitimate application.

Jang et al. [14] studied different attacks that can be performed through accessibility in different operating systems, including Android. They claimed that password attacks cannot be done in Android. Nevertheless, in our study we revealed that, when considering the default Android settings, the last character of passwords is displayed to users while typing. Then, the received event includes this character, thus indirectly revealing the password.

Fratantonio et al. [10] is the most recent take on the subject, uncovering how two permissions `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE`, when combined, lead to new powerful attacks compromising the UI feedback loop. By modifying what a user sees, their approach can simulate interactions between user and device, exploiting screen overlays to steal and compromise user credentials.

The attack we study in this paper focuses on *eavesdropping* rather than generating inputs, and achieves its results *only by listening for standard accessibility events,* without requesting any specific capability. In contrast to the claim by Fratantonio et al. [10] "with only one of the two permissions, the user will very quickly discover the attack", this is not true for eavesdropping on user input and screen contents, as we show in this paper.

9.2 Dangerous Permissions

The new runtime permission model of Android allows users to grant or revoke dangerous permissions at any time. However, Alepis and Patsakis [2] illustrated several flaws in the new Android permission system to gain access to sensitive user data without users consent.

Other studies focused on identifying Android developers practices related to privacy, and helping them to write secure code. Felt et al. [8] analyzed 940 Android apps to study how developers proceed with permission requests. They found that about one-third of the apps are overprivileged—*i.e.*, require more permissions than its effective need. Felt et al. [8] manually analyzed 40 of the overprivileged apps to understand why developers asked for unnecessary permissions. They found that the main reason is insufficient API documentation that leads to confusion over permission names, related methods, deputies, and deprecated permissions.

9.3 Developer Support

Acar et al. [1] investigated the impact of information sources used by developers on the security of their code. They conducted a lab study where 54 Android developers were asked to write privacy and security relevant source code under time and information access constraints. They found that developers who were allowed to use only Stack Overflow produced a significantly less secure code than those accessing books or Android official documentation.

Nguyen et al. [20] developed the FixDroid tool to support developers in writing secure code. FixDroid highlights security and privacy related code problems, provides an explanation to developers, and suggests quick fix options.

Regarding users' perception, previous studies [9] have demonstrated that users are unaware of security implications in Android applications. Other researchers have found that users are often surprised by the capability of background applications to collect data [5, 16, 25].

Finally, previous researches have studied code smells in Android apps. Palomba et al. [21] proposed a tool, called aDoctor, which is able to identify a set of Android-specific code smells. Mannan et al. [18] performed an empirical study on Android code smells and they showed differences when compared to desktop applications. However, none of them addresses privacy code smells like the one we observed in the Android accessibility service.

10 Conclusion and Consequences

In this paper, we have demonstrated how to exploit the accessibility service in order to capture sensitive user information. We then have proposed a family of solutions to tackle these privacy leaks from different stakeholder perspectives. From the developer's perspective, we presented AcFix, to automatically detect and fix the issue on source code. AcFix correctly applied changes to all projects and 70 % of its changes were merged back to the main app repository by their developers.

From an app store perspective, we developed AcDetect, to identify vulnerabilities when no source code is available. We applied the tool on top apps in Google Play Store with access to highly sensitive information and correctly identified 93 % of the vulnerabilities, with only 5 % false positives.

From a user perspective, we presented AcGuard, which mitigates the privacy leak by warning users immediately before they input a password. Out user study showed that these warnings affect the user confidence positively (when the vulnerability cannot be exploited) and negatively (when the vulnerability can be exploited). Based on the results of the user study, we further examined how the Google Play Store current displays accessibility-related apps, as well as formulated possible improvements in the different level to further mitigate this problem.

This work can be further extended to acquire more sensitive information, such as payment data, in addition to passwords. It can additionally incorporate more advanced static analysis techniques to handle dynamically created and updated widgets.

The big final question our work leaves open, though, is this: How can a platform be open for accessibility services, and at the same time, be closed for possible abuses of the associated APIs? For people with disabilities, this is not a trade-off of convenience versus security, but a simple necessity of using specialized hardware as well as the software that drives it. Devising an inclusive *and* secure solution for this problem will need more efforts than just supplying a service that can be abused by anybody.

We submitted the security report number 119440283 to Google in order to notify them of this vulnerability. Additionally, to facilitate reproductivity of experiments, the tools and dataset used in the evaluation are available online:

https://github.com/uds-se/AcTools

Acknowledgements

This work is partially supported by the European Research Council proof of concept grant under grant number G514111401 – BOXMATE and partially funded by the German Research Foundation (DFG) grant number D514111409.

References

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 289–305. https://doi.org/10.1109/SP.2016.25

[2] Efthimios Alepis and Constantinos Patsakis. 2017. Hey Doc, Is This Normal?: Exploring Android Permissions in the Post Marshmallow Era. In *Security, Privacy, and Applied Cryptography Engineering*, Sk Subidh Ali, Jean-Luc Danger, and Thomas Eisenbarth (Eds.). Springer International Publishing, Cham, 53–73.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. 468–471. https://doi.org/10.1145/2901739.2903508

[4] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, Florence, Italy, 426–436. https://doi.org/10.1109/ICSE.2015.61

[5] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. 2017. Exploring decision making with Android's runtime permission dialogs using in-context surveys. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. USENIX Association, 195–210.

[6] Dimitrios Damopoulos, Georgios Kambourakis, and Stefanos Gritzalis. 2013. From keyloggers to touchloggers: Take the rough with the smooth. *Computers & security* 32 (2013), 102–114.

[7] Anthony Desnos. 2011. Androguard. *URL: https://github.com/androguard/androguard* (2011).

[8] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. New York, NY, USA, 627–638. https://doi.org/10.1145/2046707.2046779

[9] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12)*. ACM, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2335356.2335360

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

[11] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 1025–1035. https://doi.org/10.1145/2568225.2568276

[12] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. 2013. Dynamic Deployment of Sensing Experiments in the Wild Using Smartphones. In *Distributed Applications and Interoperable Systems - 13th IFIP WG 6.1 International Conference, DAIS 2013, Held as Part of the 8th International*

Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings (Lecture Notes in Computer Science), Jim Dowling and François Taïani (Eds.), Vol. 7891. Springer, 43–56. https://doi.org/10.1007/978-3-642-38541-4_4

[13] Blake Ives, Kenneth R Walsh, and Helmut Schneider. 2004. The domino effect of password reuse. *Commun. ACM* 47, 4 (2004), 75–78.

[14] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. 2014. A11Y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Scottsdale, Arizona, USA, 103–115. https://doi.org/10.1145/2660267.2660295

[15] L. Jeter and S. Mishra. 2013. Identifying and quantifying the android device users' security risk exposure. In *2013 International Conference on Computing, Networking and Communications (ICNC)*. 11–17. https://doi.org/10.1109/ICCNC.2013.6504045

[16] Jaeyeon Jung, Seungyeop Han, and David Wetherall. 2012. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 45–50.

[17] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2014. *On Malware Leveraging the Android Accessibility Framework*. Springer International Publishing, Cham, 512–523. https://doi.org/10.1007/978-3-319-11569-6_40

[18] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. 2016. Understanding Code Smells in Android Applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*. 225–234. https://doi.org/10.1145/2897073.2897094

[19] Maia Naftali and Leah Findlater. 2014. Accessibility in Context: Understanding the Truly Mobile Experience of Smartphone Users with Motor Impairments. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility (ASSETS '14)*. ACM, New York, NY, USA, 209–216. https://doi.org/10.1145/2661334.2661372

[20] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1065–1077.

[21] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. 2017. Lightweight detection of Android-specific code smells: The aDoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 00. 487–491. https://doi.org/10.1109/SANER.2017.7884659

[22] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[23] André Rodrigues, Kyle Montague, Hugo Nicolau, and Tiago Guerreiro. 2015. Getting Smartphones to Talkback: Understanding the Smartphone Adoption Process of Blind Users. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility (ASSETS '15)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2700648.2809842

[24] Elizabeth Stobert and Robert Biddle. 2014. The password life cycle: user behaviour in managing passwords. In *Proceedings of the Tenth Symposium on Usable Privacy and Security (SOUPS '14)*. Menlo Park, CA.

[25] Christopher Thompson, Maritza Johnson, Serge Egelman, David Wagner, and Jennifer King. 2013. When It's Better to Ask Forgiveness Than Get Permission: Attribution Mechanisms for Smartphone Resources. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS '13)*. Newcastle, United Kingdom, Article 1, 14 pages. https://doi.org/10.1145/2501604.2501605