Device Administrator Use and Abuse in Android: Detection and Characterization

Zhiyong Shan
Wichita State University
Wichita, Kansas, USA
zhiyong.shan@wichita.edu

Raina Samuel
New Jersey Institute of Technology
Newark, New Jersey, USA
res9@njit.edu

Iulian Neamtiu
New Jersey Institute of Technology
Newark, New Jersey, USA
ineamtiu@njit.edu

ABSTRACT

Device Administrator (DA) capabilities for mobile devices, e.g., remote locking/wiping, or enforcing password strength, were originally introduced to help organizations manage phone fleets or enable parental control. However, DA capabilities have been subverted and abused: malicious apps have used DA to create ransomware or lock users out, while benign apps have used DA to prevent or hinder uninstallation; in certain cases the only remedy is to factory-reset the phone. We call these apps "Deathless Device Administrator" (DDA), i.e., apps that cannot be uninstalled. We provide the first systematic study of Android DA capabilities, DDA apps, DDA-attack resistance across Android versions, and DDA-induced families in malicious apps. To enable scalable studies of questionable DA behavior, we developed DAAX, a static analyzer which exposes potential DA abuse effectively and efficiently. In a corpus of 39,459 apps (20,467 malicious and 18,992 benign) DAAX has found 4,135 DA apps and 691 potential DDA apps. The static analysis results on the 4,135 apps were cross-checked via dynamic analysis on at least 3 phones, confirming 578 true DDAs, including apps currently on Google Play. The study has shown that DAAX is effective (84.8% F-measure) and efficient (analysis typically takes 205 seconds per app).

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Software security engineering**; • **Software and its engineering → Automated static analysis**.

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico

ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3345452

KEYWORDS

mobile applications; security; static analysis; mobile device management

ACM Reference Format:
Zhiyong Shan, Raina Samuel, and Iulian Neamtiu. 2019. Device Administrator Use and Abuse in Android: Detection and Characterization. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3345452

1 INTRODUCTION

To facilitate managing mobile device fleets, mobile OSes have introduced the concepts of Device Administration ("DA" on Android [11]) or Mobile Device Management ("MDM" on iOS [13]). DA/MDM give fleet device issuers control over a wide range of security capabilities according to company policy, e.g., enforce password strength/expiration, or lock-/wipe devices remotely. Other examples include the ability to restrict app behavior, e.g., an app used at a public kiosk; or for parental control (Section 2 provides an overview of DA while Section 3 characterizes DA use in benign apps).

These capabilities can be, and have been, abused. To the best of our knowledge, there are no tools or studies for understanding the DA ecosystem: detecting DA use and abuse, characterizing benign and malicious DA behavior, understanding consequences of malicious behavior and recovery strategies, grouping malicious behavior into families, etc. We fill this gap with (1) an automated, effective and efficient approach to detect, and (2) a study to characterize, DA-based abuse. While we target the Android platform, the potential for abuse is present in other platforms that offer DA/MDM.

One such widely-abused capability is leveraging active DA permissions to prevent app uninstallation. Specifically, apps employ techniques to (1) conceal their DA status so the user is unaware of these apps' privileges, or (2) prevent the DA status from being deactivated – a practice that compromises device security and can render the device unusable until a factory reset. We name such apps *Deathless Device Administrator* (DDA) apps. We have studied a wide range of malware and benign apps that have DA permissions, and defined three

DDA categories: DDA-Reset, DDA-Hide, and DDA-Expert (Section 4). We have discovered all three classes of DDA behavior in numerous apps that are still on Google Play.

Responsible Disclosure. We reported this attack vector to Google on 02/11/19. On 02/12/19, Google's Android Security team opened an Android External Security Report, and requested the standard 90-day disclosure window before making the findings public. On 02/22/19, Google rated the issue as "*moderate severity. Moderate severity issues are often fixed at the next appropriate opportunity.*" due to the multiple security implications, e.g., "*Local permanent denial of service (device requires a factory reset)*", or "*Local bypass of user interaction requirements*", etc.[1] We have been in contact with Google's Android Security Team and supplied the apps, environment information, etc. for reproducing the issues.

To permit automated, scalable DDA detection, we have developed DAAX (DA Abuse eXposer), a static analyzer that exposes potential DA abusive behavior (Section 5). We provide an evaluation of DAAX in Section 6. We started with a corpus of 39,459 sample apps (20,467 malicious and 18,992 benign). The 4,135 apps in the corpus that used DA were then analyzed dynamically (on the phone) and statically via DAAX. DAAX has detected 244 DDA-Reset apps, 7 DDA-Hide apps and 327 DDA-Expert apps. We manually verified DDAs on at least 3 phones running 3 different Android versions. We confirmed DDA behavior in 578 apps, including apps currently on Google Play. DAAX's precision was 83.7%; its recall was 86% (hence a 84.8% F-measure). The median per-app analysis time was 205 seconds.

Section 7 contains a longitudinal study on the effectiveness of DDA attack vectors across five Android OS versions: 5.1, 6.0.1, 7.1, 8.0, and 9.0. We found that more recent OS versions have added techniques that give users more control over apps, thus empowering users to combat DDA more effectively.

For malicious apps, we combined DAAX results with characteristics such as DDA icon, DDA name and DDA behavior to define 15 DDA families, described in Section 8. In Section 9 we outline solutions for eliminating or reducing the risk of DDA, and expose pre-installed hidden DA apps.

To summarize, we make the following contributions:

(1) An exposé of DA use/abuse, including 110 confirmed abuses in benign apps and 468 in malicious apps.
(2) DAAX, an automated approach and tool to detect (potentially abusive) DA behavior.
(3) A study that used DAAX on 20,467 malicious apps and 18,992 benign apps.
(4) A longitudinal study that compares the effectiveness of DDA attack vectors across five Android versions.
(5) A characterization of malware DA behavior (families).

[1] https://source.android.com/security/overview/updates-resources#severity

We envision DAAX being useful in a variety of settings. App markets can use DAAX to determine whether an app might contain DDA-Reset, DDA-Hide, or DDA-Expert code. Developers can use DAAX to find inadvertent violations of DA protocol. End users can find potentially harmful DA behavior before installing an app.

2 DEVICE ADMINISTRATOR OVERVIEW

DA – a set of "extra" app privileges allowing tighter control, or even remote control, over Android devices – was originally introduced in Android 2.2 to facilitate enterprise applications and management of device fleets. Starting with Android 5, an alternative, superior set of features, "Android for Work", was introduced, but apps have continued to use, and unfortunately, abuse, DA. We first discuss the DA timeline and then the lifecycle of a DA app on a device.

2.1 DA Timeline

Android 2.2 (May 2010): DA policies introduced. DA support was introduced, supporting the following policies:

(1) Enforcing password strength, reuse and expiration requirements, and forcing a password change.
(2) Enforcing inactivity locks.
(3) Forcing a certain storage area to be encrypted.
(4) Disabling the camera.
(5) Remote device locking.
(6) Remote device wiping.

Android 5.0 (Nov. 2014) – Android 8.0 (Aug. 2017): Android for Work introduced, expanded. Android for Work (later, Android Enterprise, "AE" [8]) introduced the concepts of managed devices, employed-owned devices, or work profiles – a set of features/policies somewhat similar to DA but broader, more secure, and with clearer roles.

Dec. 2017: Planned DA deprecation is announced. Google recommends that apps transition away from DA to AE and announces that DA will gradually be deprecated [35].

Android 9 (Aug. 2018): enterprise&soft deprecation. Starting with Android 9, DA was deprecated for enterprise use. For non-enterprise use, several policies (password expiration, disabling camera) were soft-deprecated, i.e., marked as deprecated but apps continue to function [10].

Android 10: hard deprecation. The aforementioned policies that were soft-deprecated will be hard-deprecated starting with Android 10, i.e., apps targeting 2019+ API levels that attempt to use the policy will trigger a `SecurityException`. Nevertheless, at least three policies – forcing a device lock, wipe, password reset – will continue to be supported [10].

DA features can benefit organizations, IT admins, or parents. Unfortunately, when abused, DA can be turned against users. To uninstall a DA app, the user must first deactivate the app's DA capabilities, and then attempt to uninstall the

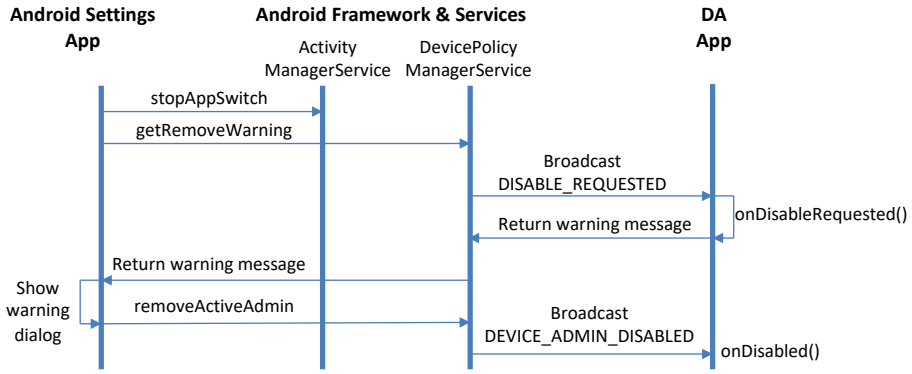


Figure 1: Message Sequence Chart for DA deactivation.

app. Therefore, if the app can prevent deactivation, it can prevent uninstallation. For benign apps, the consequences can be a nuisance (unless the app is buggy, which can render the device unusable). For malicious apps, preventing uninstallation can mean unlimited/unfettered access and opens the door to abuse, e.g., ransomware. The "hard" DA deprecation in Android 9 (enterprise) and Android 10 (apps with 2019 target API) mitigates this issue. However in the remaining cases (non-enterprise apps, apps with API target <2019) the DDA potential persists: the underlying cause is the assumption that apps will cooperate when asked to give up DA privileges. We discuss OS evolution's impact in Section 7 and some potential solutions in Section 9.

DA vs. Root. DA privileges do not require root access. "Rooting" a device (to gain root privileges) usually voids the device's warranty, whereas for an app to become DA the user simply needs to install the app and activate DA.

2.2 DA Status Lifecycle

We now explain the lifecycle of DA privileges in DA apps. Prior to deployment, developers have to claim DA capabilities (in the manifest) and then implement them. Implementation involves writing a DeviceAdminReceiver, which allows the app to receive intents sent by the system. The DeviceAdminReceiver class consists of a series of callbacks, triggered when particular DA-relevant events occur, e.g., onDisableRequested(), onDisabled(), onEnabled(). The lifecycle of DA contains three steps:

1. Activation/Pre-activation: Normally, DA is activated when the user performs an action that triggers the ACTION_ADD_DEVICE_ADMIN intent. However, some pre-installed apps such as Find My Device come with DA pre-activated.

2. Operation: Performing DA privileged operations, e.g., setting the device's password, locking or wiping the device.

3. Deactivation: Deactivating DA when the user sends ACTION_DEVICE_ADMIN_DISABLE_REQUESTED to the app. This gives the app a chance to supply a message to the user about the impact of disabling DA, by setting the extra field EXTRA_DISABLE_WARNING in the result Intent. If the field is not set, no warning will be displayed. If set, the message will be shown to the user before they disable DA. To uninstall an existing DA app, users need to first deactivate the app as a DA.

2.3 Deactivation Procedure

Per Android's official documentation "To uninstall an existing device admin app, users need to first unregister the app as an administrator" [11]. In other words, the DA status can be deactivated by using Android's Settings and then the app can be uninstalled. Figure 1 shows the Android protocol for deactivating DA. When the user proceeds to cancel DA privileges for a DA app, the Settings app will invoke stopAppSwitch() to restrict activity switches for a period of time (e.g., 5 seconds) and then requests the DA removal warning message from the app. The callback method onDisableRequested() is invoked and returns the warning text. The Settings app pops up a dialog showing the warning. Once the user presses the 'OK' button, DA privileges are deactivated by calling removeActiveAdmin().

An app can interfere with this procedure to prevent users from deactivating the app's DA privileges.

3 BENIGN DA CHARACTERIZATION

When installing a DA app, in theory the user should make an informed choice, and be familiar with DA app behavior or the policies the DA app enforces; in practice though, the user has close to zero knowledge of the consequences. Moreover, DA apps can come preinstalled (e.g., by the device issuer, device vendor, or mobile carrier – see Section 9). Therefore users may not understand entirely what these behaviors entail or when it is appropriate to grant an app such privileges.

Table 1: Most common DA behaviors.

Behavior	# Apps
Lock screen	68
Set password rules	34
Change the screen-unlock password	33
Monitor screen unlock attempts	32
Erase all data	31
Set lock screen password expiry	20
Disable cameras	19
Set storage encryption	16
Disable features in keyguard/screenlock	10

In this section we discuss the most prevalent "benign" DA behaviors we found via a separate analysis, focused on benign DA usage, on a sample set[2] of 151 benign DA apps from Google Play. First we characterize the behaviors, then study which Google Play categories contain the highest concentrations of DA apps.

3.1 DA Behaviors (Privileges)

The most common behaviors (by number of apps having that behavior) are provided in Table 1, and described next.

Lock Screen. This capability allows the DA app to control how and when the screen locks; it was most commonly found in screen lock apps, and also in parental control apps, antivirus apps, and enterprise management apps.

Set Password Rules. This controls the length and characters allowed in screen unlock passwords. There were 34 appearances of this behavior, in enterprise management apps, antivirus apps and remote phone security apps (which are used when a phone is lost or stolen).

Change Screen Lock. This behavior changes the screen lock password. There were 33 instances, primarily in enterprise and parental control apps. Unfortunately this capability can be detrimental if the implementation is buggy – the DA app can lock the device with a password or PIN unknown to the user, rendering the device useless until it is factory-reset.

Monitor Screen Unlock Attempts. This DA functionality monitors the number of incorrect passwords typed when unlocking the screen and will lock the phone or erase all the phone's data if too many incorrect passwords are typed. The functionality was used in 32 apps, mainly phone security apps (which is typical and the function of such apps), and enterprise management apps.

Erase Data. Phone data is erased without warning, by performing a factory data reset; there were 31 cases, mostly in security, antivirus, and enterprise management apps.

[2]To ensure a representative sample of popular apps, we chose apps from across all 34 categories on Google Play; median number of installs across the sample set: 1,000,000+.

Table 2: Popular apps with high # of DA behaviors.

App	Installs	Behaviors
BlackBerry UEM Client	5,000,000+	9
Malwarebytes Security	10,000,000+	8
Lookout Security&AV	100,000,000+	7
Kaspersky Mobile AV	50,000,000+	7
Bitdefender Mobile S.&AV	5,000,000+	6
Where's My Droid	10,000,000+	6
Avast AV	100,000,000+	5
McAfee Mobile Security	10,000,000+	5
AirDroid	10,000,000+	5
Microsoft Authenticator	10,000,000+	2

Disable Cameras. We found 19 instances, in enterprise management and security apps. As this is not a standard feature, disabling cameras might be puzzling or unsettling; based on the general nature of these apps, the user would expect the camera to work.

Storage Encryption. This encrypts the device's storage – a quasi-mandatory feature for enterprise management apps as well as phone security apps; we found 16 instances.

Disable features in keyguard/screenlock. This allows an application to disable the screen lock or any code that is involved with unlocking the device. Should it be misused, this is a potential major security breach for the device.

In Table 2 we show the apps with the 10 highest number of DA behaviors. Six apps are antiviruses (*Security* or *AV*); higher privileges are expected for such apps due to their nature. However, we were surprised to see so many privileges granted to the remaining 4 apps: BlackBerry UEM Client (9 DAs), Microsoft Authenticator (enterprise), Where's My Droid (device locator), and AirDroid (remote access/file transfer).

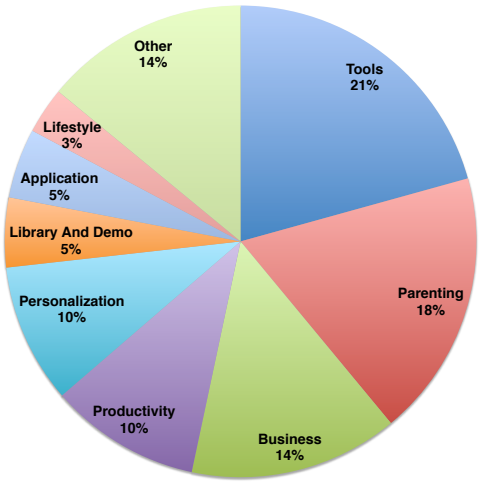


Figure 2: DA split across categories.

3.2 DA Across App Categories

As of March 2019, Google Play lists 34 app categories (a 35th, *Games*, category has its own subcategories). To find out which categories host the most DA apps, we performed a DA analysis on Top-600 apps in each category. Figure 2 shows the percentage of apps which fall into a certain category. While *Tools* and *Business* are expected to be close to the top, surprisingly, *Parenting* had the second highest prevalence of DA apps. As to the reasons for requesting DA privileges, we note that 21% of the DA apps were in the *Tools* category with Lock Screen as the most prevalent DA behavior. *Productivity* and *Personalization* each make up 10% of the DA apps and they both have Lock Screen as their top DA behavior; in fact, for *Personalization* apps, Lock Screen is their only DA behavior. This is expected, as many *Personalization* apps are often various types of themed lock screen launchers.

4 DEATHLESS DEVICE ADMINISTRATOR

Deathless Device Administrator (DDA) apps represent DA apps that *prevent the user from uninstalling the app*. To do so, DDAs exploit vulnerabilities or weaknesses in the procedures Android uses for handling DAs. Accordingly, we introduce three types of DDA and exemplify that behavior on actual apps. We derived these three types of behavior via a semi-automated process (installing, checking DA status, attempting to deactivate DA) on 4,135 apps that had DA permissions, as described in Section 6.2. Based on the observed behavior, we constructed DAAX (described in Section 5), a static analyzer designed to expose potential DDA behavior.

4.1 DDA-Reset

Definition. DDA-Reset apps prevent the user from disabling an app's DA capabilities; DDA-Reset is irrecoverable – the only way to remove an app that uses DDA-Reset is to restore the phone to factory settings.

Example 1. Sberbank_Online. The app disguises itself as an online banking app (for Russian bank Sberbank), to steal user credentials. The malware asks for administrator privileges upon installation, which, if permitted, can inflict serious harm to the victim's device. The app can also intercept SMS messages and incoming calls which could be a step to sidestep the bank's OTP (One Time Password) requirements.

The app becomes "deathless" by preventing users from deactivating the app's DA privileges. To prevent the pop-up (warning dialog), DDA apps add carefully designed code into method onDisableRequested(), as explained next.

Figure 3 shows Smali (Android bytecode) disassembled from the real malware; for clarity irrelevant code is removed. After the system notifies the app that its DA will be deactivated, the app's onDisableRequested() callback is invoked. This callback's body is shown on lines 1–19. The callback first prepares a new activity com.android. settings (lines 4–8), which will be used to dismiss the warning dialog. Note that this new activity is not in foreground yet, as stopAppSwitch() is still in effect (Figure 1, top left) for five seconds. Then, the callback locks the screen (lockNow() on line 13) and starts a new thread (start on line 18, thread body on lines 21–36) to lock (line 29) and sleep (lines 32) repeatedly, lasting more than five seconds. Thus, the system's warning window won't show, as the screen is locked. Five seconds later, the new activity switches to the foreground, thus dismissing the warning window.

As a result, the user *had no chance to see and heed the warning dialog and to permit DA deactivation.*

Example 2. Check Point Capsule Connect, a VPN app [20], was found by DAAX as DDA-Reset in December 2018, reported to Google in February 2019, and removed from Google Play around March 2019. As shown in Figure 4, when deactivating DA, the app pops a warning dialog and locks the phone with an unknown password. Unfortunately, to recover access to the device, the user must perform a factory reset.

Example 3. Mobile Tracker is a popular app (1,000,000+ installs) [32] that DAAX detected as DDA-Reset. We reported it to Google in February 2019; its DA capabilities have been removed while this paper was being prepared. The app can track device activity, delete files when the device is lost/stolen, etc. When disabling DA, clicking the deactivation button renders the Settings app unresponsive for extensive periods; eventually the app pops up a window finally asking the user to confirm the DA deactivation. Even if the user selects OK, the DA checkbox is still checked, and the Settings app crashes, which is shown in Figure 5. After restarting the phone, Mobile Tracker's DA checkbox remains checked. We found that (1) the app keeps verifying the "checked" status of the DA checkbox; if unchecked, a separate thread will turn it back on; and (2) the Settings crash is caused by Mobile Tracker continuously sending it the intent DEVICE_ADMIN_DISABLED.

4.2 DDA-Hide

Definition. Apps in the DDA-Hide category hide themselves from the DA list in Settings app, i.e., the user cannot even see that the app is operating as a DA.

Example. Bandwidth Meter monitors network connections and displays Internet speed. When the user attempts to uninstall the app, Android shows a message that the app cannot be uninstalled, as the app is DA. However, the app does not appear in Settings' DA app list. The Hidden Device Admin Scanner app by Trend Micro failed to find this app. While Trend Micro's Mobile Security & Antivirus flags Bandwidth Meter as a potential unwanted app, it does not remove the app (it can only find it).

This hiding behavior is caused by a security vulnerability in the Settings app, which omits to show a DA app in the list.

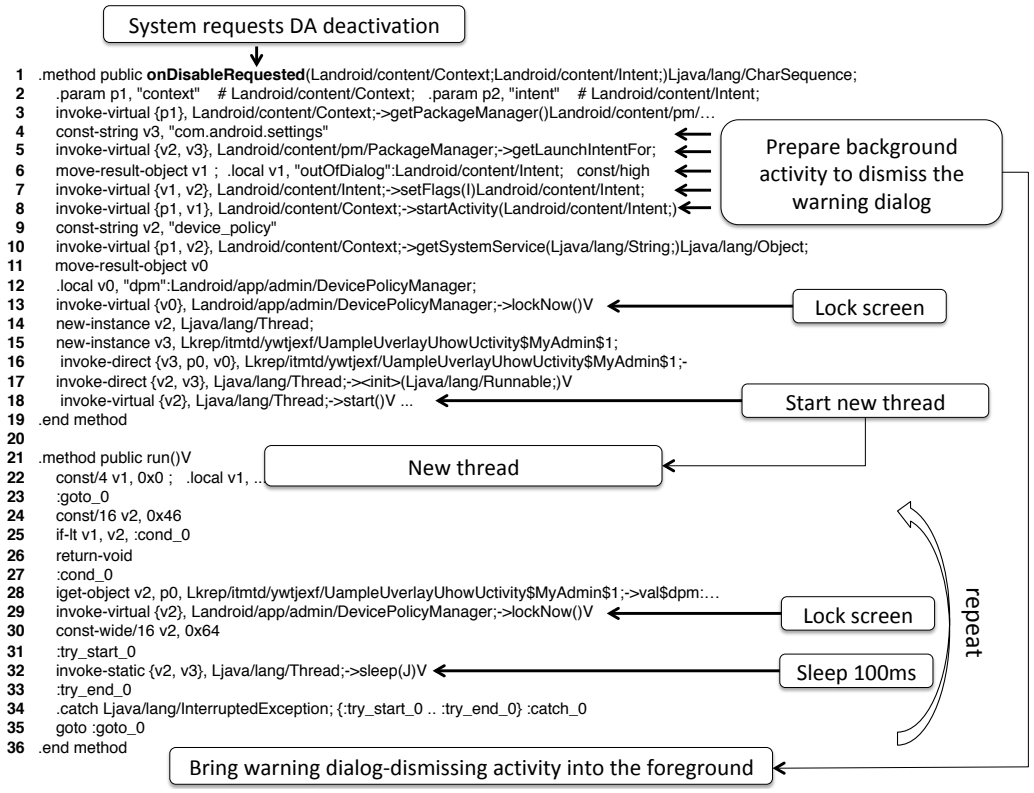
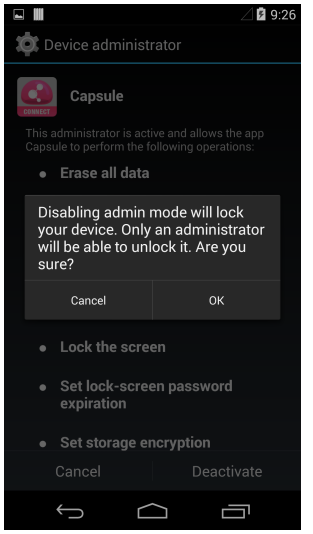


Figure 3: Smali code extracted from malware Sberbank_Online.



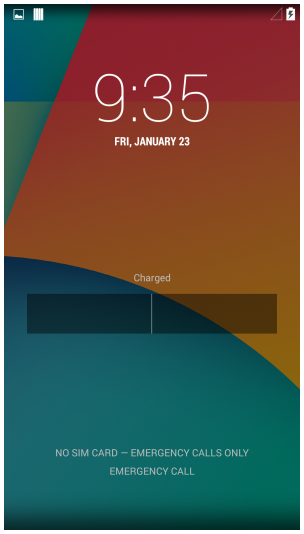
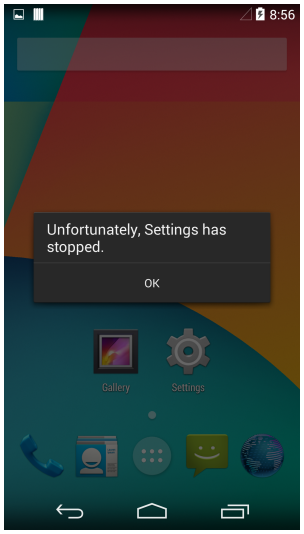


Figure 4: Capsule Warning (left); locked screen (right).



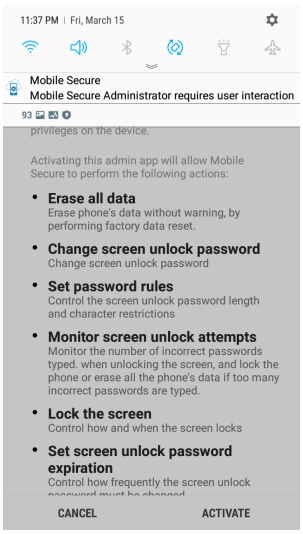


Figure 5: Mobile Tracker crashes Android Settings (left); SAP Mobile Secure app keeps popping up notifications and DA activation window with sounds (right).

Specifically, when updating the DA app list, the Settings app will first get the list of all activated DA apps and the list of all enabled DA. Only when an app is in *both the Activated and Enabled* lists, the Settings app will show it. However, some apps can be activated without being enabled, and use this artifact to hide from the Settings' DA app list.

4.3 DDA-Expert

Definition. DDA-Expert apps could in principle be deactivated but doing so is difficult for the average user. For example, malware Dowgin modifies the appearance of the check box in the DA list to disguise the fact that the app is still DA-active after deactivating the DA. The user can click the check box again to deactivate the DA. In actuality, this activates DA again. However, the user does not realize this trick and assumes that the DA cannot be deactivated.

Example 1. AppLock, one of the most downloaded screen lock apps on Google Play (100,000,000+ installs) [24], was detected as DDA-Expert by DAAX in December 2018 and reported to Google in February 2019. The issues have since been fixed. When the user attempted to deactivate DA, a password window popped up, demanding a password unknown to the user, caused by calling Android API's lockNow(). However, if the user had used the app to set up a password, the user can enter it and bypass the window, thus being able to deactivate the DA. According to our investigation, this is implemented by calling Android API method lockNow() within the callback function onDisableRequested().

Example 2. MaaS360 Mobile Device Management is a cloud-based mobile device management app (1,000,000+ installs) [31]; DAAX found DDA-Expert behavior which we reported to Google; the behavior has been corrected. Essentially when a user attempted to deactivate DA after multiple presses, Settings crashed, deactivating eventually. While it may not seem serious at a glance, many users have reported and vehemently complained of the inability to uninstall the app.[3] We found that Settings' crash is caused by MaaS360 sending intent DEVICE_ADMIN_DISABLED to Settings on a continuous basis.

Example 3. SAP Mobile Secure for Android is a device management client. As of March 17, 2019 the app is still available on Google Play [27], with DDA behavior still present. The app becomes "resistant" once the user attempts to deactivate DA. Specifically, as shown in Figure 5 (right), upon deactivating DA, the app keeps popping up a notification with sound and shows the DA activation dialog, which forces the user to re-activate DA. Even after restarting the phone, the notification and sound resume. Only when DA is re-activated, the notification and sound would stop.

Example 4. Habyts Agent [26] combines screen time management, parental controls, and a motivation/rewards system. After pressing the DA deactivation button, the app shows a warning window. Upon returning to the DA list, the DA status is still on. We found that the app creates a new thread to turn on DA in the background. As of March 17, 2019, the app is still on Google Play and exhibiting these issues.

[3]Excerpts from two recent reviews on Google Play: (1) *"This freaking app locked my phone completely cant even use my own home screen or my other apps is in emergency mode wont let me do a thing how in the hell do I remove it???"* (2) *"not letting me uninstall a single thing in my phone. I cant even deactivate this app to uninstall it lmao. Its dictating what i can and cant have on my phone... dont download it"* [31].

5 DETECTING DDA

This section presents our implementation, DAAX, which uses static analysis to recognize DDA apps effectively and efficiently. Figure 6 (left) shows DAAX's architecture. DAAX takes as input an APK (the format Android apps are distributed in) and outputs a list of potential DDA behaviors. We use Soot [16] to convert the app's Dalvik bytecode into the Jimple Intermediate Representation [16] and FlowDroid/IccTA [17] [29] to perform alias and call graph analyses. The module *DA Receiver Analysis* analyzes the app's XML manifest and then the bytecode to see whether a DA broadcast receiver administrator is claimed and implemented. The module *DA Activation Analysis* checks whether the app requests DA privileges. The module *Pre-Deactivation Analysis* checks whether the app prevents DA deactivation. The module *Post-Deactivation Analysis* checks whether the app forces the user into reactivating DA privileges.

5.1 DA Receiver Analysis

Detecting whether an app uses DA involves two checks: an XML analysis for the manifest and then a bytecode analysis. We first check the manifest for the BIND_DEVICE_ADMIN permission (this permission ensures that only the system can interact with the receiver, not other apps, to avoid this DA app from becoming a confused deputy); and the DEVICE_ADMIN_ENABLED intent filter (which allows the app to become a Broadcast Receiver).

However, checking the manifest only is not always sufficient for two reasons: (1) apps can declare the permission and filter, yet not use DA capabilities; (2) DDA-Hide apps can claim the permission but not declare the intent filter, as explained shortly. Therefore, our second step is a bytecode analysis: an app using DA has to implement a DA broadcast receiver (subclassing DeviceAdminReceiver and implementing a series of callbacks that are triggered when particular events occur). We analyze the bytecode to find the subclass of DeviceAdminReceiver and the callback methods. The most essential callback method that should be implemented is onEnabled(), which is called after the system has enabled DA for the app.

We define an app as DA only when it *claims and implements* the DA broadcast receiver, as well as *activates* DA capabilities. We have found apps that fail to take these actions – DAAX does not consider such apps as DA.

5.2 DA Activation Analysis

The user must explicitly *activate* DA capabilities for an app in order for DA privileges to be conferred. If the user chooses not to activate DA, the app will still be present on the device,

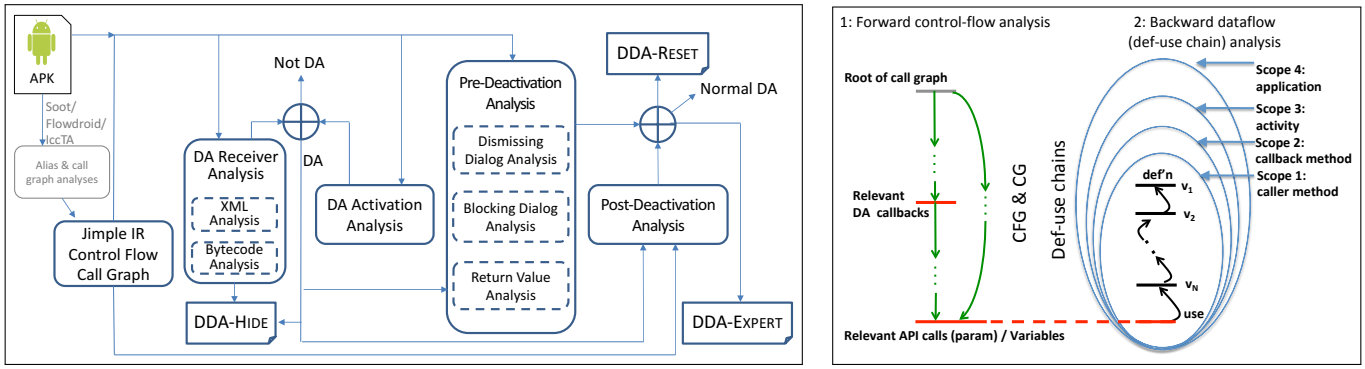


Figure 6: DAAX **architecture (left); analysis strategy (right).**

but policies that require DA will not be enforceable due to a lack of DA capabilities. The process of enabling DA begins when the user performs an action that triggers the ACTION_ADD_DEVICE_ADMIN intent.

```
1  // Launch the activity to prompt the user for DA activation
2  Intent intent = new Intent(DevicePolicyManager.
       ACTION_ADD_DEVICE_ADMIN);
3  intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN,
       mDeviceAdminSample);
4  intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
5  mActivity.getString(R.string.add_admin_extra_app_text));
6   startActivityForResult(intent, REQUEST_CODE_ENABLE_ADMIN)
```

The code above shows a typical DA activation procedure (for legibility, we only show the relevant parts). Our DA Activation Analysis finds such behavior; the algorithm (Figure 6 (right)), combines forward control-flow analysis with backward def-use chain analysis. Note that the static analyses used in subsequent modules follow a similar strategy.

The forward control-flow analysis searches the call graphs and control-flow graphs of each app activity to find all call sites of DA-relevant API methods. For the DA Activation Analysis, the relevant API methods are startActivity() and startActivityForResult(). The backward def-use chain analysis determines the methods' parameter values; in the case of DA Activation Analysis, the parameters are intent action and request code. We determine that the app is activating DA only if the intent is ACTION_ADD_DEVICE_ADMIN and at the same time the request code is REQUEST_CODE_ENABLE_ADMIN.

To improve performance, our algorithm is incremental, unwinding the backward analysis' scope to larger scopes (rather than analyze the whole app as a forward analysis would do). As shown in Figure 6 (right), the algorithm traverses def-use chains along control-flow paths within the current method, then in the top-level callback method, then in the activity, and finally in the whole app – until the chain ends. In practice most of the backward def-use analysis will end in the current method or the top-level callback method.

5.3 Pre-Deactivation Analysis

As discussed in Section 2, DDA apps implement callback method onDisableRequested() to prevent the user from deactivating DA. Our Pre-Deactivation Analysis detects this automatically by looking for the app's attempt to block and dismiss dialogs, as follows.

5.3.1 Return Value Analysis. According to the DA protocol, if onDisableRequested() returns an empty string, the Settings app will directly deactivate DA without showing a warning dialog (and hence without allowing the user to decide whether to proceed). Therefore we first find all exits of onDisableRequested() and its descendants, then perform backward def-use chain analysis to determine whether the method returns an empty string. The analysis returns (*Ret* in Section 5.5) TRUE for non-empty strings and FALSE for empty strings.

5.3.2 Dismissing Dialog Analysis. DDA apps need to create a new activity to dismiss the deactivation warning dialog (when switching to a new activity, the dialog of the old activity will be destroyed). Therefore, we check whether API methods startActivity() or startActivityForResult() are called within onDisableRequested() with argument FLAG_ACTIVITY_NEW_TASK (which will create a new activity).

5.3.3 Blocking Dialog Analysis. During the period when an activity switch is prohibited by the system, DDA apps need to block the warning dialog. This can be accomplished in several ways:

Lock screen + sleep. The DDA app calls lockNow() to lock the phone immediately and then sleep. Once the user unlocks the phone, the app locks it again and again. Thus, the warning dialog will not pop up. Our analysis checks whether lockNow() and sleep() are called, either in the method onDisableRequested() itself, or its descendants, or on paths to exits.

Transparent alert window + sleep. The DDA app creates a transparent system alert window to always stay in the foreground and cover the warning dialog. At the same time, an app thread sleeps for a period of time then removes the

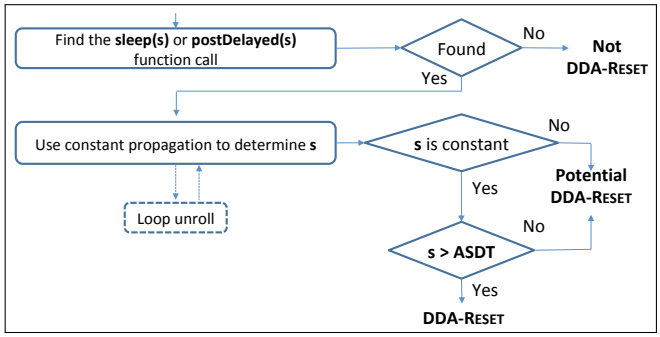


Figure 7: Timing analysis.

alert window. Our analysis detects when a new window is created within onDisableRequested() with certain characteristics: the type is "system alert", has no color (i.e., is transparent), and is not focusable. Via XML analysis of resource files, we also analyze the resource properties of the window to check its parameters (as they can be set via the resource files instead of in the bytecode).

Reset password + lock screen. The DDA app calls method resetPassword() immediately after the call to lockNow() within the callback onDisableRequested(). The resetPassword() changes the password for unlocking the device. When the user tries to unlock the device, the Settings app pops up an alert dialog to ask the user for a password that, of course, is unknown to the user. As a consequence, the callback method never returns and thus the warning dialog cannot appear. Our analysis searches for resetPassword() and lockNow() in the callback, its descendants, and on paths to exits. For example, malware log-amd-Jisut employs this mechanism.

Delay the return. To delay the return of onDisableRequested(), the DDA app calls sleep() or postDelayed() method. If the delay time is long enough, the warning dialog will not show. Hence our analysis checks for calls to sleep() or postDelayed().

If the app is using the above approaches (except the third one) we further perform an analysis to compute the sleep time **s** in the callback method onDisableRequested(). When the user is on the disable DA screen, Android calls stopAppSwitch() to prevent interaction/interference from other apps. But the blocking time is limited to the predefined time length APP_SWITCH_DELAY_TIME (**ASDT**). Meanwhile, the Settings app pops up the warning dialog for the user to disable DA. After the time limit, other apps can then move to the foreground. The DDA app thus has to block the warning dialog for a longer time than the time limit APP_SWITCH_DELAY_TIME (in other words, **s** > **ASDT**). Figure 7 shows the algorithm.

If sleep() or postDelayed() cannot be found we do not further pursue the app as DDA-RESET; while an app could theoretically time the sleeping via other means, those approaches are not portable. We use constant propagation (and loop unrolling if sleep() or postDelayed() are in a loop) to determine the value of **s**. If this value can be computed statically and **s** > **ASDT**, the app is DDA-RESET. In any other case, e.g., the value of **s** cannot be computed statically, we declare the app as potential DDA-RESET. Note that this approach could lead to a false positive due to over-approximation.

5.4 Post-Deactivation Analysis

According to Figure 1, the callback method onDisabled() will be invoked after the DA is deactivated. However, a DDA can lure the user into activating DA again by calling startActivity() or startActivityForResult(), as discussed in Section 5.2. A DDA can also call startActivityForResult() repeatedly to force the user to activate DA if the user has clicked 'Cancel' on the DA activation dialog, or has clicked 'Back'. Our analysis of onDisabled() determines whether the app requests to activate DA. Method startActivityForResult() returns a result code that is either RESULT_OK if the operation was successful or RESULT_CANCELED if the user backed out, or the operation failed for some reason. The app is effectively forcing the user to activate DA if startActivityForResult() is called repeatedly when startActivityForResult() returns RESULT_CANCELED.

5.5 Determining DDA

DDA-Reset and DDA-Expert. For each of the six static analyses in Figure 6 (left) we use the abbreviations *Rec* (DA Receiver Analysis), *Act* (DA Activation Analysis), *Ret* (Return Value Analysis), *Dis* (Dismissing Dialog Analysis), *Blo* (Blocking Dialog Analysis), *Pos* (Post-Deactivation Analysis) to indicate that the outcome of that analysis is TRUE. We can now define the DDA-RESET and DDA-EXPERT detection rules based on analyses' outcome:

$$\text{DA} = Rec\&Act$$
$$\text{DDA-RESET} = DA\&((Ret\&Dis\&Blo)|Pos)$$
$$\text{DDA-EXPERT} = DA\&Ret\&(Dis|Blo)\&!Pos$$

DDA-Hide. As discussed previously, if an app has permission BIND_DEVICE_ADMIN but without DEVICE_ADMIN_ENABLED in the manifest, we flag it as DDA-HIDE, i.e., Hiding DA. We also flag it as DDA-HIDE if its attribute DeviceAdmin_visible is false. Such apps can hide in the DA list and thus the user cannot deactivate the DA. We show the code from malware com.android.system.admin for such a scenario:

```
1  <receiver android:label="System" android:name=".OClICoO"
2  android:permission="android.perm.BIND_DEVICE_ADMIN">
3  android:name="android.app.device_admin"
4  android:resource="@xml/ccclocc"/> <intent− filter > <action
5  android:name="com.strain.admin.DEVICE_ADMIN_ENABLED"/>
6  </intent− filter > </receiver >
```

Please note the action name com.strain.admin.DEVICE_ADMIN_ENABLED is fake. A real action name should be android.app.action.DEVICE_ADMIN_ENABLED. Therefore, this app can hide from the DA list in the Settings app.

6 EVALUATION

We present the experimental setup and evaluate our approach along two dimensions. First, effectiveness: is the approach effective at identifying DDAs? What are the main causes of false positives/false negatives? Second, efficiency: does the analysis complete in a reasonable amount of time?

6.1 Dataset

We have analyzed 39,459 apps: 20,467 malicious and 18,992 benign. The benign apps are from Google Play, AndroZoo [7], and AppsApk [14]. The malware apps are from Drebin [15], DroidCat [2], Kharon [3], AndroMalShare [1], Malware Genome Project [40], Android Malware Dataset [36] and Offensive Computing [4]. We chose these apps using several criteria which we believe are necessary for making meaningful observations. The 20,467 *malicious apps* came from dozens of different families. The 18,992 *benign apps*: (a) cover different categories, e.g., Utilities, Email & SMS, Games, Health & Fitness, Wallpapers, Photography, Weather, News, Education, Browser, Map, Call & Contacts; and (b) have variety in terms of popularity, e.g., many apps have over 1 million installs while Facebook has over 1 billion installs. The entire dataset of 39,459 apps has variety in terms of size, from 2.5 KB (com.kharamly.tests) to 416 MB (com.netease.dhxy.mi).

6.2 Workflow and Ground Truth

We now describe our workflow, testbed, and Ground Truth procedure used for discovering and confirming DDA.

Step 1 (automatic). All 39,459 apps' manifests were checked for the BIND_DEVICE_ADMIN permission using two separate procedures: (1) using DAAX and (2) using `grep` on the Manifest extracted via Apkanalyzer[4]. We confirmed that we could check for the permission using at least one of these methods; this yielded 4,135 apps.

Step 2 (automatic). We used scripts to install the 4,135 apps that had DA permission and then to check, via the UI Automator[5] whether the app appears in the Settings DA list.

Step 3 (manual). We then manually:

- Attempted to deactivate DA, restart Settings and check whether DA is still deactivated. This reveals DDA-Reset and DDA-Expert apps.
- Checked those cases where an app had the DA permission but was not appearing in the Settings' list. This reveals DDA-Hide apps.

This split the 4,135 apps into 3,350 that were not DDA and 785 that were actual observed DDA, or reported DDA by DAAX.

[4]https://developer.android.com/studio/command-line/apkanalyzer
[5]https://developer.android.com/training/testing/ui-automator

Replication testbed. All behaviors, either revealed by DAAX or manually (in the 785 apps), were verified by the authors on at least three phones from a five-phone pool: two Google Nexus 5s running Android 4.4.4 and 6.0.1, respectively; an LG G4 running Android 5.1; a Google Pixel 3 running Android 9; and a Galaxy J7 Crown running Android 8.

Ground Truth. Ground Truth is essential for finding False Positives or False Negatives, hence determining effectiveness. We determined Ground Truth via the automated and manual process described above (steps 2 and 3): we manually checked all 4,135 apps with DA permission, which was efficient due to batch processing and the fact that 3,350 apps were not DDA. The remaining 785 DDA or reported as DDA by DAAX were subject to extensive analysis (step 3). The process yielded 578 true DDAs aka True Positives (with confirmed, replicated DDA behavior on multiple devices).

6.3 Effectiveness

From the 39,459 apps, DAAX could analyze 38,229 (1,230 apps failed due to timeouts or Soot instability, as explained shortly). Among these, DAAX reported 691 as DDA: 286 DDA-Reset, 7 DDA-Hide and 398 DDA-Expert.

Using the Section 6.2 process, we confirmed 244 DDA-Reset, 7 DDA-Hide and 327 DDA-Expert apps (i.e., 578 total). When DAAX reported 691 DDAs, it over-reported 113 apps (false positives) and under-reported 94. Table 3 shows the true positives, false positives (reported by DAAX but behavior not confirmed), false negatives (app with confirmed behavior but missed by DAAX), precision, and recall. Per Table 3 the precision is 83.7%, while the recall is 86%; hence the F-measure is 84.8%.

These results allow us to conclude that DAAX *is effective.*

Figure 8 breaks down the true number of DDAs: among the 578 true DDAs, 468 were in malware and 110 in benign apps. We found that DDA-Reset and DDA-Expert were the most prevalent DDAs for malware (241 and 220, respectively), while DDA-Expert was the most prevalent for benign apps (107). Note that effectively DDA-Reset and DDA-Hide apps cannot be uninstalled, thus these techniques are very rarely employed by benign apps.

False negatives. The major reasons for false negatives are: (1) the AXMLPrinter used in DAAX can not properly analyze the content of AndroidManifest.xml to determine whether the APK contains the permission BIND_DEVICE_ADMIN (specifically, the AXMLPrinter failed with an ArrayIndexOutOfBoundsException – an example is krep.itmtd.ywtjexf-1.apk). (2) Some apps are too big to finish analysis within a limited time (we set this limit to 20,000 seconds, i.e., 5.5 hours). (3) DAAX failed to analyze some apps because the apps are obfuscated, or the front-end analyzer Soot failed for various reasons.

Table 3: Effectiveness results.

	True	Over-reported (FP)	Under-reported (FN)	Precision	Recall	F-measure
DDA-Reset	244	42	35	$\frac{244}{244+42} = 85.3\%$	$\frac{244}{244+35} = 87.5\%$	$2 * \frac{85.3*87.5}{85.3+87.5} = 86.3\%$
DDA-Hide	7	0	1	$\frac{7}{7+0} = 100\%$	$\frac{7}{7+1} = 87.5\%$	$2 * \frac{100*87.5}{100+87.5} = 93.3\%$
DDA-Expert	327	71	58	$\frac{327}{327+71} = 82.2\%$	$\frac{327}{327+58} = 84.9\%$	$2 * \frac{82.2*84.9}{82.2+84.9} = 83.5\%$

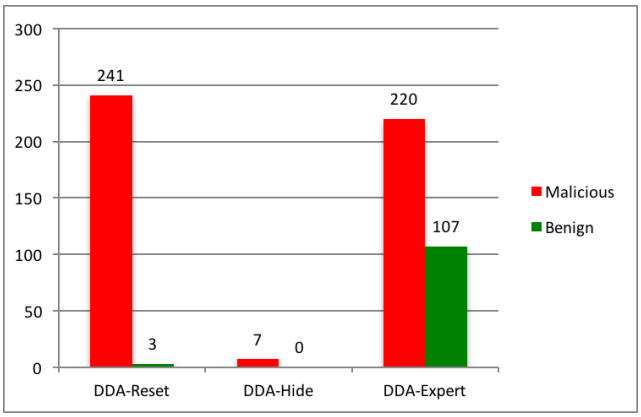


Figure 8: DDA prevalence by type.

False positives. The major reasons for the false positives are: (1) Some sophisticated DDA-Expert apps are reported as DDA-Reset. For example, in the Capsule app, upon deactivating DA the app will lock the screen and require the user to enter an unknown password. After rebooting the phone, the app requires the user to enter a cloud registration key. However, eventually it can be deactivated because of a bug in the app. Another example is Security Center (Chinese name): the app's DA cannot be deactivated on the first try, but it can be after restarting the phone. (2) Alias, data-flow, and control-flow analyses are over-approximating, which is inherent in static analysis.

In-store apps. Table 4 shows more Google Play DDA apps detected by DAAX. Six of them were detected in December 2018 and reported to Google at the end of February 2019; these were all fixed or removed from Google Play Store before 03/15/2019. For the 5 fixed apps, we compared them with the versions in December 2018, and found that DA behavior was fixed or removed. For the last four apps, detected in March 2019 and currently still on Google Play, the report is pending, so app status or its DA behavior might change in the future. Detailed app descriptions can be found in Section 4.

6.4 Efficiency

We measured efficiency by noting the wall-clock time for each app while we were running 7–10 DAAX instances in parallel[6] on an 8-core, 16-threads Xeon E5-2687W v2. In total, this took 30.1 CPU-core days (2,604,672 CPU-core seconds).

As the analysis times are significantly different, we divide the apps into two data sets: DA and non-DA. As explained in Section 5.1, DAAX considers as DA only those apps that both claim and implement the DA broadcast receiver. DAAX has separated the 39,459 apps into 4,135 DA apps and 35,324 non-DA apps. We show the detailed efficiency results in Table 5. The "Bytecode size" grouped columns show that the datasets had substantial variety in terms of app size, and some apps' bytecode size was as large as 416 MB. The "Time" grouped columns show running time statistics for each dataset. For DA apps, the median analysis time was 205 seconds while the mean was 348 seconds. Table 6 shows the stage-by-stage breakdown of the 348 seconds into percentages of total analysis time. For non-DA apps, the mean analysis time was 33 seconds while the median was 27 seconds; these lower times for non-DA times are expected, because DAAX stops the static analysis if a DA broadcast receiver is not found.

This allows us to conclude that DAAX *is efficient.*

7 DDA BEHAVIOR EVOLUTION ACROSS ANDROID VERSIONS

We performed a longitudinal study to measure the effectiveness of each DDA attack vector, and see how the Android version influences (permits or prohibits) DDA. While in Section 6.2 an app was deemed DDA if it could be confirmed on at least three devices/versions, here we only focus on apps that could be installed and run on *five* Android versions (5.1, 6.0.1, 7.1, 8.0, and 9.0), i.e., March 2015–August 2018, as this allows us to make more conclusive longitudinal observations. This stronger selection criterion reduced the number of apps to 301 malicious and 42 benign apps (compared to 468 malware and 110 benign apps for the three-version setup) for two main reasons. First, many apps were designed for late versions of Android hence failed to install on early versions; e.g., if the manifest specifies android:minSdkVersion=23, which corresponds to Android 6.0, the app will not install on Android 5.1. Second, old versions of apps that still run on Android 5.1 would immediately force an upgrade when started on

[6]Naturally, when running a single instance of DAAX at a time, per-app analysis time would be lower than what we report here.

Table 4: Google Play DDA apps.

App	DDA type	Detected	Reported	Status
Check Point Capsule Connect	DDA-Reset	Dec. 2018	Feb. 2019	App Removed
Mobile Tracker	DDA-Reset	Dec. 2018	Feb. 2019	DDA issues fixed
App Lock	DDA-Expert	Dec. 2018	Feb. 2019	DDA issues fixed
MaaS360 Mobile Device Management	DDA-Expert	Dec. 2018	Feb. 2019	DDA issues fixed
Safe & Found	DDA-Expert	Dec. 2018	Feb. 2019	DDA issues fixed
VMware WorkSpace ONE	DDA-Expert	Dec. 2018	Feb. 2019	DDA issues fixed
SAP Mobile Secure for Android	DDA-Expert	Mar. 2019	Report pending	Currently on Google Play
Habyts Agent	DDA-Expert	Mar. 2019	Report pending	Currently on Google Play
Digi Family Safety	DDA-Expert	Mar. 2019	Report pending	Currently on Google Play
Codeproof LG Mobile Security	DDA-Expert	Mar. 2019	Report pending	Currently on Google Play

Table 5: Efficiency results.

Dataset	Bytecode size (KB)				Time (seconds)			
	min	max	average	median	min	max	average	median
DA (4,135 apps)	11	49,747	2,860	1,311	71	19,693	348	205
non-DA (35,324 apps)	2.5	425,984	5,604	3,504	2	1,026	33	27

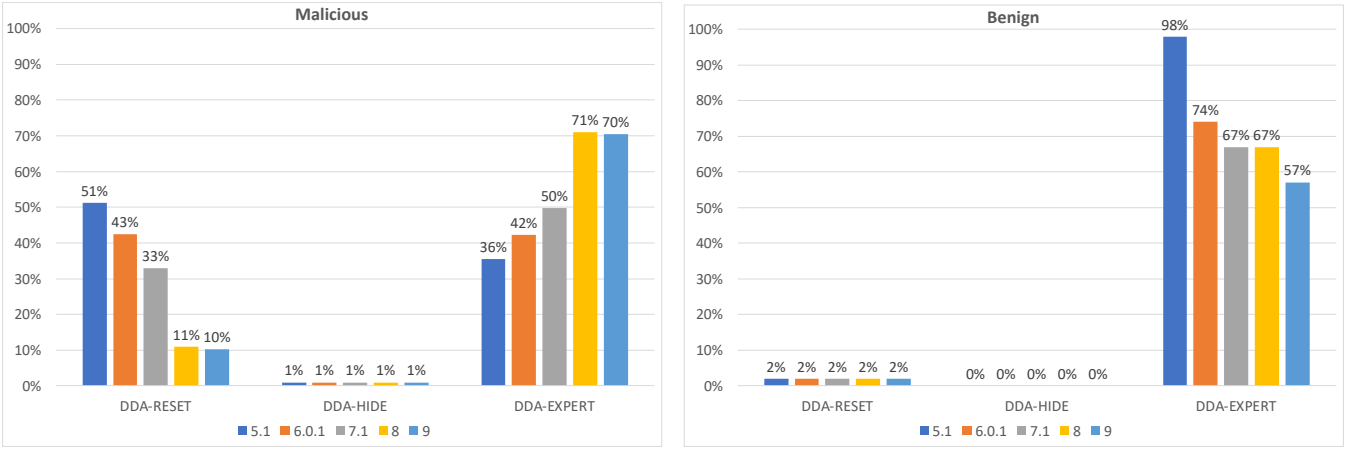


Figure 9: DDA prevalence across five Android versions in malicious apps (left) and benign apps (right).

Table 6: Breakdown of analysis time by stage.

Stage	Fraction of time
APK Unpacking	0.86%
DA Receiver Analysis	0.25%
DA Activation Analysis	8.39%
Pre-Deactivation Analysis	82.38%
Post-Deactivation Analysis	8.16%

8.0 or 9.0; however, allowing the upgrade would violate our requirement to run the same APK on all five OS versions.

Figure 9 shows DDA prevalence: percentage of apps exhibiting that behavior in a particular Android version. We begin with several general observations. First, DDA-Expert was the most prevalent behavior for both malicious and benign apps, respectively. Second, only three malicious apps employed DDA-Hide; no benign apps employed it. Third, as DDA-Reset apps cannot be uninstalled, these techniques are very rarely employed by benign apps (one app, Check Point Capsule Connect). We now make several longitudinal (evolution) observations.

Benign apps. Among benign apps with DDA-Reset, the behavior is the same regardless of the Android version. However, DDA-Expert applications have decreased with more recent Android versions. This is due to app behavior being ameliorated by the OS version, as explained shortly.

Malicious apps. DDA-Reset was prevalent in older versions of Android until Android 7.1 where the trend switched to DDA-Expert. This is due to the following reasons that are purely OS oriented. Apps with DDA-Reset in older versions often had an overlay that would prevent the user from, powering down the phone, accessing Settings, or the device itself, thus blocking the user from initiating any further activities and forcing the user to perform a factory reset. However, starting in Android 8.0, for some of these apps with an invasive overlay, a notification in the Notification Drawer [9] warns the user that the specified app has access to draw over other apps. When opening the notification, the system sends the user to the Settings option for the app hence allowing the user to disable the app's ability to draw over other apps and ultimately bypassing the invasive app overlay. Once that happens, the user is able to deactivate the DDA and uninstall the app. This is why in later versions of Android the DDA-Reset behavior for some apps becomes DDA-Expert.

In addition, starting in Android 7.1 the user has the ability to uninstall a DA app directly within Settings (not just disable DA). In older versions, some apps with DDA-Reset would incessantly keep popping up the Settings option to activate DA once the user would deactivate the DA. With no other way to remove the app and stop the harassment, the user would end up having to factory-reset the phone. In Android 7.1 and later, however, once the behavior manifests, the user can uninstall the app immediately once the Settings option pops up. Despite these improvements in the Android OS, DDA-Reset behavior still exists, albeit in smaller numbers.

8 MALWARE DDA FAMILIES

We discovered that DDA behavior in malware apps induces DDA families: a family of DDA samples shares similar DA icons or names in the DA list in the Settings app, or similar package names in the manifest files. More importantly, they share very similar behaviors when disabling DA in the Settings app. Finally, we inspected their bytecode (they share very similar DA handling code). Table 7 summarizes the 15 families we obtained. Entries ('Y' or 'N') in the "Deactivatable" column indicate whether a typical user would be able to deactivate DA for that family. The families with 'N' employ a range of maneuvers to prevent deactivation. While we were able to ultimately deactivate those apps, some measures we had to take were quite convoluted, beyond resetting to factory settings, e.g., removing the battery – well beyond the purview and expectations of a typical user. The name of each family is extracted from the name of the apps in Settings' DA list. These names are usually different from app names, and different apps can share the same name in the DA list. Some family names are in non-Latin script (e.g., Russian, Chinese), so we transliterated those. For brevity, we describe the behavior of a few 'N' families – the remaining families are essentially DDA-Reset or DDA-Expert.

Table 7: DDA Families.

DDA Family	Deactivatable	# Samples
Adobe Core Importing	N	1
Android_6_Update	N	1
Sberbank	N	2
Phone locator	Y	2
Administrator Identification	N	4
Firm module Update	N	9
Tele2 MMS-Centre	N	9
Dynamic screen lock	Y	11
Adobe Flash Player core	N	11
System update	N	15
Adobe Flash Player_P*rnDroid	N	21
P*rn Droid	Y	38
Hello World	N	66
Installation	N	102
Adobe Flash Player	Y	152

Administrator Identification. The app blacks out, locks the screen for 5 seconds, then shows a screen for entering a password – this password is unknown to the user. The user's only option is to restart the phone.

Firm module Update. The screen is locked and shows a web error. The user can thereafter no longer access the phone – this family is essentially ransomware.

Adobe Flash Player core. The app waits for several seconds and then pops up the message: "Please enter password of local storage for deleting Adobe Flash Player plugin"; again, this password is unknown to the user. The app cannot be uninstalled, including when attempting to force uninstall via adb. The only effective measure is a factory reset.

Hello World. This family is ransomware: apps cannot be uninstalled, including via adb. The screen is locked after activating DA, and the app asks for $100 to unlock the phone.

We also mapped the relationships between DDA families, defined by us, and AMD malware families, defined by Wei et al. [36]. For brevity we omit a detailed description, but make two observations. First, several DDA:AMD families map 1:1, e.g., Sberbank : Krep or HelloWorld : Jisut which strengthens the validity of our family identification. Second, AMD apps in the same family can use a variety of DDA techniques, e.g., Fusob can use System update and Firm module update to conduct their nefarious business.

9 POTENTIAL SOLUTIONS

Android runs on more than 2 billion devices – about 74% of mobile devices worldwide [33, 34]. To reduce the DDA risk,

Table 8: Number of "hidden" DA apps and their origin.

Phone	Android version	DA App Provenance		
		Google	Vendor	Other
LG G6	8.0	2	4	3
LG G4	5.1	2	4	1
Samsung Galaxy S9	8.0	2	8	2
Google Pixel 3	9.0	6	0	0
Google (LG) Nexus 5	6.0.1	3	0	0
Moto G5 Plus	8.1	2	1	0
Moto E 2nd Gen.	5.1	2	1	0
Huawei P Smart	8.0	3	0	0
Samsung Galaxy S3	4.4.2	2	0	1
Nexus (Huawei) 6P	8.1.0	2	1	0

we believe that developers following Google's recommendation to switch apps from DA to AE (Section 2) would be by far the most effective measure. Absent that, we foresee several measures the platform can take.

Remove untrusted app code from the critical path. Per the protocol in Figure 1, untrusted (and generally untrustworthy) app code is on the critical path between the user and completing actions such as disable DA or uninstall. The Android platform could rework this protocol to avoid trusting the app, e.g., by simply requiring that the DA deactivation message be a string in the manifest (which the platform can simply display, instead of executing app code).

Force-expose all DA apps in Settings. The immediate benefit would be to prevent DDA-Hide. However, this exposing can serve a broader purpose – make users aware of all installed apps which require DA privileges. In the process of analyzing DA behavior, we discovered that phones come with *up to a dozen preinstalled, hidden* DA apps that have DA privileges but do not appear in Settings. An analysis on 10 different phones is presented in Table 8. The last three columns show where these apps are coming from.

First, Google/Android: apps such as Find My Device and Google Pay are DA, but visible; interestingly, Gmail is also DA but invisible – users cannot tell without "pulling" the apps from the phone and unpacking the app.

Next, phone vendors can pre-install apps: LG pre-installs 4 apps on G4/G6 and Samsung pre-installs 8 apps on the Galaxy S9 – e.g., for diagnosis, debugging, email. Such apps might increase the attack surface/violate the principle of least privilege, e.g., the HTCLogger app [12].

Finally, "other" preinstalled DA apps, e.g., Vlingo, occasionally crash and leave the users puzzled (since they do not appear in the installed app list, they cannot be uninstalled) or worse, exfiltrate user data [6].

10 RELATED WORK

A variety of techniques have been proposed to detect and characterize malicious behavior. However, we were not able to find any approach that focused on Device Administrator. The only mentions of the malicious potential of DA apps were in the malware bulletins issued by mobile security companies [18, 21–23, 28, 30, 37]; note that those mentions refer to individual threats (apps), rather than broader issues.

Malware behavior characterization via program analysis. SmartDroid combines static and dynamic analysis to expose the behavior of Android malware in UI-based triggers [38]. MAST uses permissions, intent filters, native code, and zip files for multiple correspondence analysis which measures the correlation with qualitative data [19]. DREBIN [15] uses both static analysis and machine learning to optimize analysis and detection patterns. Apposcopy [25], DroidAnalytics [39], and DroidAPIMiner [5] use static analysis to analyze malware properties but do not analyze DA behavior. None of the aforementioned approaches touch on DA, though.

Malware families. Zhou and Jiang [41] have categorized 1,260 malware samples into 49 families. However their focus was on non-DA behavior: malware installation, activation, privilege escalation, turning the phone into a bot, etc.

11 CONCLUSIONS

We characterize and quantify both legitimate and nefarious use of DA capabilities in Android apps. Based on these observations we have constructed DAAX, a static analyzer that exposes potential DDA behavior in a given Android app. We ran static and dynamic analyses on large corpora of benign apps and malicious apps. DAAX has revealed potential issues in more than 500 apps; the confirmed issues have been reported to Google's Android Security team. Our study and tool can improve Android security by helping end-users, developers, and app marketplaces analyze DA behavior.

ACKNOWLEDGMENTS

We thank our shepherd Ardalan Amiri Sani and the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617584. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] 2017. AndroMalShare. http://sanddroid.xjtu.edu.cn:8080.

[2] 2017. The DroidCat Dataset. http://www.people.vcu.edu/~rashidib/Res_files/DroidCatDataset.htm.

[3] 2017. Kharon project. http://kharon.gforge.inria.fr/index.html.

[4] 2017. Open Malware. http://www.offensivecomputing.net/search.cgi?search=android.

[5] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*. Springer, 86–103.

[6] Aaron Tilton. 2012. Vlingo Privacy Breach: Data Sent to Remote Servers Without Consent. https://www.androidpit.com/Vlingo-security-flaw.

[7] 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)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[8] Android Open Source Project. 2019. Android Enterprise. https://developers.google.com/android/work/overview.

[9] Android Open Source Project. 2019. Android Notifications Overview: Status bar and notification drawer. https://developer.android.com/guide/topics/ui/notifiers/notifications#bar-and-drawer.

[10] Android Open Source Project. 2019. Device admin deprecation. https://developers.google.com/android/work/device-admin-deprecation.

[11] Android Open Source Project. 2019. Device administration overview. https://developer.android.com/guide/topics/admin/device-admin.

[12] Android Police. 2011. Massive Security Vulnerability In HTC Android Devices. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices.

[13] Apple, Inc. 2019. Managing Devices & Corporate Data on iOS. https://www.apple.com/business/resources/docs/Managing_Devices_and_Corporate_Data_on_iOS.pdf.

[14] AppsApk. 2019. AppsApk. https://www.appsapk.com.

[15] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.

[16] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The Soot-based Toolchain for Analyzing Android Apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, Piscataway, NJ, USA, 13–24. https://doi.org/10.1109/MOBILESoft.2017.2

[17] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[18] Lianne Caetano. 2013. Obad.a – What You Need to Know About the Latest Android Threat. https://securingtomorrow.mcafee.com/consumer/mobile-security/obad-a-what-you-need-to-know-about-the-latest-android-threat/.

[19] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. 2013. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. ACM, 13–24.

[20] Check Point Software Technologies. 2018. Check Point Capsule Connect. https://play.google.com/store/apps/details?id=com.checkpoint.CloudConnector.

[21] Check Point Software Technologies Ltd. [n.d.]. Charger Malware Calls and Raises the Risk on Google Play. https://blog.checkpoint.com/2017/01/24/charger-malware/.

[22] Android Central Community. 2018. Can't Remove Device Administrator. https://forums.androidcentral.com/general-help-how/50088-cant-remove-device-administrator.html.

[23] Santiago Cortes. 2013. Android.Obad. https://www.symantec.com/security-center/writeup/2013-060411-4146-99.

[24] DoMobile Lab. 2019. AppLock. https://play.google.com/store/apps/details?id=com.domobile.applock.

[25] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 576–587.

[26] Habyts Parenting. 2019. Habyts - Easier Screen Time. https://play.google.com/store/apps/details?id=com.habyts.agent&hl=en_US.

[27] Sybase Inc. 2019. SAP Mobile Secure for Android. https://play.google.com/store/apps/details?id=com.Android.Afaria.

[28] Swati Khandelwal. 2018. Facebook Password Stealing Apps Found on Android Play Store. https://thehackernews.com/2018/01/facebook-password-hacking-android.html.

[29] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 280–291. http://dl.acm.org/citation.cfm?id=2818754.2818791

[30] Robert Lipovsky and Lukas Stefanko. 2018. Android Ransomware: from Android Defender to Doublelocker. https://www.welivesecurity.com/wp-content/uploads/2018/02/Android_Ransomware_From_Android_Defender_to_Doublelocker.pdf.

[31] MaaS360. 2019. MaaS360 MDM. https://play.google.com/store/apps/details?id=com.fiberlink.maas360.android.control.

[32] Naveeninfotech. 2019. Mobile Tracker. https://play.google.com/store/apps/details?id=com.nav.mobile.tracker.

[33] Ben Popper. 2017. Google announces over 2 billion monthly active devices on Android. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users.

[34] Statcounter GlobalStats. 2019. Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[35] Tom Watkins. 2017. Why it's time for enterprises to adopt Android's modern device management APIs. https://www.blog.google/products/android-enterprise/why-its-time-enterprises-adopt-androids-modern-device-management-apis/.

[36] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, Bonn, Germany, 252–276.

[37] Martin Zhang. 2016. Android ransomware variant uses clickjacking to become device administrator. https://www.symantec.com/connect/blogs/android-ransomware-variant-uses-clickjacking-become-device-administrator.

[38] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 93–104.

[39] Min Zheng, Mingshen Sun, and John CS Lui. 2013. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE, 163–171.

[40] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 95–109. https://doi.org/10.1109/SP.2012.16

[41] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 95–109.