Spring 5-31-2018

Detecting and characterizing self hiding behavior in android applications

Raina Samuel

New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

DETECTING AND CHARACTERIZING SELF HIDING BEHAVIOR IN ANDROID APPLICATIONS

by
Raina Samuel

Applications (apps) that conceal their activities are fundamentally deceptive; app marketplaces and end-users should treat such apps as suspicious. However, due to its nature and intent, activity concealing is not disclosed up-front, which puts users at risk. This study focuses on characterization and detection of such techniques, e.g., hiding the app or removing traces, known as "self hiding" (SH) behavior. SH behavior has not been studied per se – rather it has been reported on only as a byproduct of malware investigations. This gap is addressed via a study and suite of static analyses targeted at SH in Android apps.

SH behavior ranges from hiding the app's presence or activity to covering an app's traces, e.g., by blocking phone calls/ text messages or removing calls and messages from logs. Using static analysis tools on a large dataset of 9,452 Android apps (benign as well as malicious) the frequency of 12 such SH behaviors is exposed. It has revealed that malicious apps employ 1.5 SH behaviors per app on average. Surprisingly, SH behavior is also employed by legitimate ("benign") apps, which can affect users negatively in multiple ways. The approach has high precision and recall (combined F-measure = 87.19%). This approach is also efficient, with analysis typically taking just 37 seconds per app.

DETECTING AND CHARACTERIZING SELF HIDING BEHAVIOR IN ANDROID APPLICATIONS

by
Raina Samuel

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Software Engineering

Department of Computer Science Ying Wu College of Computing

May 2018

APPROVAL PAGE

DETECTING AND CHARACTERIZING SELF HIDING BEHAVIOR IN ANDROID APPLICATIONS

Raina Samuel

Dr. Iulian Neamtiu, Thesis Advisor — Date
Associate Professor of Computer Science NJIT

Dr. Qiang Tang Committee Member — Date
Assistant Professor of Computer Science NJIT

Dr. Xiaoning Ding, Committee Member — Date
Assistant Professor of Computer Science NJIT

BIOGRAPHICAL SKETCH

Author: Raina Samuel

Degree: Master of Science

Date: May 2018

Undergraduate and Graduate Education:

- Master of Science in Software Engineering,
  New Jersey Institute of Technology, Newark, New Jersey, 2018

- Bachelor of Science in Information Technology
  Southern New Hampshire University, Manchester, New Hampshire, 2016

Major: Software Engineering

ACKNOWLEDGMENT

I thank Dr. Iulian Neamtiu for providing me the resources and motivation to complete this research. I thank Dr. Xiaoning Ding and Dr. Qiang Tang for taking the time to be part of my research committee. I also thank Dr. Zhiyong Shan for his dedication and research in this paper.

TABLE OF CONTENTS

TABLE OF CONTENTS

(Continued)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Objective

Mobile security research has mostly focused on malware activation, malicious payloads, permission abuse, or leaking sensitive data. Little attention has been paid to deceptive mechanisms that are essential for the success of malware, i.e., how malware manages to get installed, and continues operating on the phone without the users noticing anything suspicious. To do so, malware uses a range of SHB, e.g., hiding the app, hiding app resources, blocking calls, deleting call records, or blocking and deleting text messages. Surprisingly, extremely popular "benign" apps such as Airbnb, Truecaller, and Waze also employ certain SH techniques in the name of user convenience. We believe that SHB is fundamentally deceptive and that having tools that perform accurate and early detection of SHB is key. First, app marketplaces, e.g., Google Play or Apple Store, should be able to detect SHB, so that SHB can be considered in the decision to publish an app or not. Even when an app with SHB is published on the marketplace, users should be forewarned about the SHB so they can decide whether to install the app on their phone or not. We address these problems on the Android platform via several advances: (1) we shine a light on SHB via detailed characterization, (2) we construct an SHB-detecting tool based on static analysis, and (3) we show how our approach for identifying SHB can be very effective at exposing malicious apps as well as deceptive practices in benign apps.

CHAPTER 2

SELF HIDING BEHAVIOR

In this section, we provide a comprehensive description of SH behaviors. We define as SH a behavior meant to hide the app or its actions from being viewed (or heard!) by the user. Note that we exclude those behaviors meant to evade security mechanisms, e.g., anti-malware tools or access control mechanisms – they have been studied thoroughly and are outside the scope of this paper. Our characterization is based on manual analysis of about 200 malicious apps and automated analysis of about 3,000 other malicious apps. We found 12 SHBs; few of these are even mentioned in the research community, let alone characterized thoroughly, and some, including "Hide icon" and "Hide activity", are not mentioned at all. Users could employ three main approaches for identifying the presence of malicious apps: inspecting app objects (icon, app, activity), analyzing remote communication (SMS, MMS, and phone calls) or checking system reminders (system dialogs, sound, system logs, notifications, recent apps list, etc). There are two main issues with this approach, though: (1) it requires a highly knowledgeable user who performs such inspections periodically, and (2) malware actively attempts to escape (by hiding itself) from such identification.

2.1 App Objects

After installation, benign apps add their icon to the home screen. To hide itself, a malicious app removes the icon so the user cannot notice the app's presence. There are two methods for hiding the icon: (a) Modifying the app's manifest file to remove the app from the default launcher, i.e., home screen. This can be done by deleting category android.intent.category. LAUNCHER from the app's main activity section in the manifest file. For example, malware Fake-skype camouflages as the popular app Skype and runs in the background without an icon in the home screen. (b) Calling an

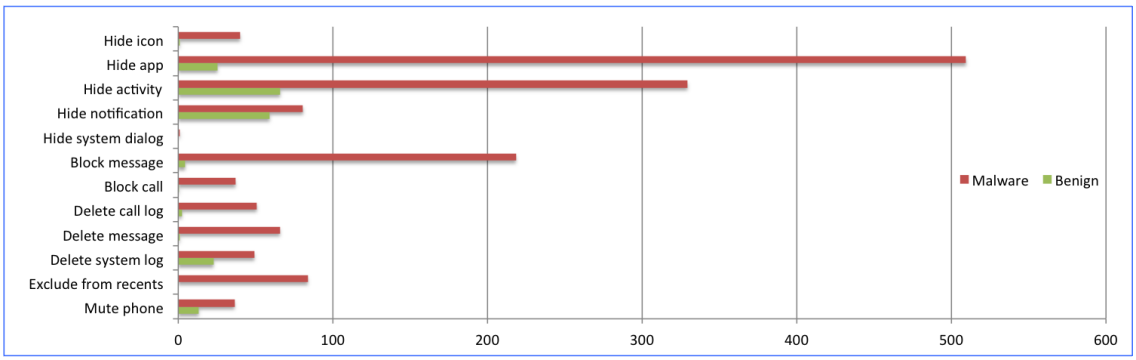

Figure 2.1 The numbers of SHBs in two sample sets of 1,000 malware apps and 1,000 benign apps, respectively.

Android API method to disable the icon at runtime. This can be done by invoking method setComponentEnabledSetting(). For example, malware Facebook-otp (full package name: jgywwv.jvyjsd.sordvd), masquerades as the Facebook app but disables its icon immediately after installation. When benign apps are running, they typically show up in the running app list. In contrast, a malicious app can run as a service, in the background, hence does not show up in the list.

2.2 Remote Communication

Sending SMS/MMS messages furtively, in the background, is a common behavior in malware. Therefore, several anti-malware products focus on this to recognize malware. After sending or receiving SMS/MMS in the background, Android saves a copy of the SMS/MMS in the outbox or inbox, respectively. To cover its tracks, malware needs to delete this copy. The malware usually calls delete() on a content URI, i.e., "content :// sms/inbox/" and "content :// sms/outbox/", respectively. Furthermore, malware can also delete SMS/MMS associated with a certain message ID, time, or phone number. An example is malware XTaoAd.A that deletes a message upon receipt. After a malicious app sends SMS/MMS to sign up for a premium-rate service in the background, it will receive a confirmation SMS/MMS sent from the

service provider. To prevent users from knowing this, the malware has to filter the received SMS/MMS by calling abortBroadcast().

2.3 System Reminders

System dialogs could reveal the presence of malware by displaying alarms, user account balances, or other abnormal behaviors to the user. To avoid this, malware has to dismiss the system dialog by broadcasting the intent ACTION CLOSE SYSTEM DIALOGS.

Apps can send alerts to the user by generating a notification on the notification bar. But the malware can delete notifications by calling NotificationManager's methods cancel() or cancelAll() when receiving notifications. To cover their presence, malicious apps often resort to muting the phone or disabling the vibrate function, to prevent the user from hearing the sound of alarms, notifications, phone calls or incoming SMSs. This can be accomplished in a variety of ways: switching to silent mode, calling the vibrator service, setting the phone to mute, or adjusting the volume to the lowest level. After an app has run, the system puts its activities into the recent apps list. To prevent this, malware can set the flag excludeFromRecents in the manifest file, or by calling ActivityManager.setExcludeFromRecents(). Android saves system activity into the system log, which can be viewed via the logcat. Malware can call adb logcat -c to delete the logs if the phone is rooted or if the Android API is lower than 16.

CHAPTER 3

DETECTING SH BEHAVIORS

Our approach relies on a suite of static analyses to detect SHBs. Figure 2 shows an overview of our tool's design. The input is an APK file (APK is the format Android apps are distributed in). We pass the bytecode to Soot [3]/FlowDroid [4] which perform basic tasks such as alias analysis, call graph analysis, as well as fixpoint computations to deal with loops and recursion. Next, we perform our core analyses (described shortly) on both bytecode and XML files. Finally, a report detailing the potential SHBs is produced.

3.1 Static Analysis

Our first analysis finds whether SH API calls are invoked (we name this SAPI analysis for short). We present the analysis in Figure 3. Specifically, the analysis starts at an Origin (app or activity start). In the first stage, we use control-flow and call graph analysis to find whether a certain SAPI call is invoked , green nodes and edges on the left represent methods and call graph edges, respectively. In the second stage, we use backward dataflow analysis to find if the call is invoked with certain SH-indicating parameters; more precisely, we walk the def-use chains backwards (shown in black) until we can find the parameter definition, e.g., a constant or an alias. For example, to detect the "Hide app" SHB, our analysis will check whether the call to Context.startService() is reachable when starting in BroadcastReceiver.onReceive().To check for "Delete message" on the other hand, we start tracking from BroadcastReceiver . onReceive(SMS RECEIVED ACTION VIEW) to see if we can reach ContentResolver.Delete(); next, we walk the def-use chains backwards to see if the argument is "content :// sms". For certain behaviors, e.g., "Hide activity", we use all the app's entry points as origin; app entry points

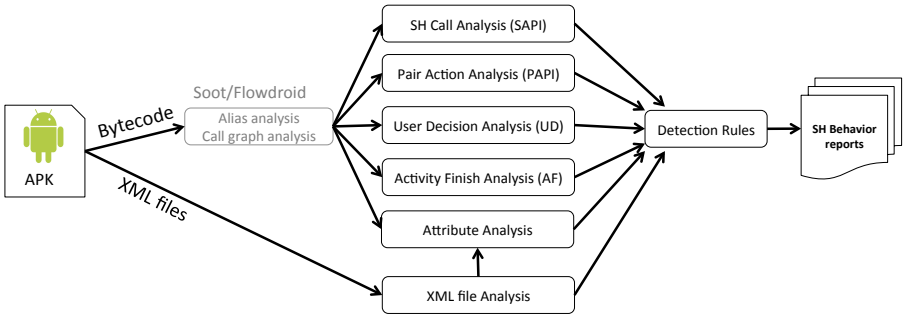

Figure 3.1 Tool overview.

are provided by FlowDroid. Another broad self-hiding category consists of pair actions, where an app first performs a malicious action then deletes traces of this action, e.g., deleting a text message after sending it. Our analysis (we name this PAPI analysis for short) detects six types of pair actions: send message/delete, message log, receive message/delete message log, receive/block message, make phone call/delete call log, receive phone call/delete call log, receive/block phone call. Our pair action detector uses data flow analysis in a manner similar to taint analysis to see if data flows from a pair start to a pair end. To reduce potential false positives in cases where SAPI methods are also used by benign apps, we perform a user-decision analysis that checks whether an API method invocation is the result of a user decision. We name this UD analysis for short. The user's GUI actions can be decision-related or decision-unrelated, as explained next. Decision-related actions include clicking a button, checking a checkbox or selecting a menu item; in other words, the user takes decisions (and acts accordingly) in a way meant to change the app state. Examples of decision-unrelated actions include scrolling down a window or changing focus. If an SAPI is invoked by a decision-related action, we rule that call as legitimate, rather than an SH attempt. However, if invoked by a decision-unrelated action, it can be an SHB. Note that existing research can only detect whether an API is invoked by a GUI , whereas we further consider whether the GUI can reflect decisions. In order to present the user-decision analysis approach,

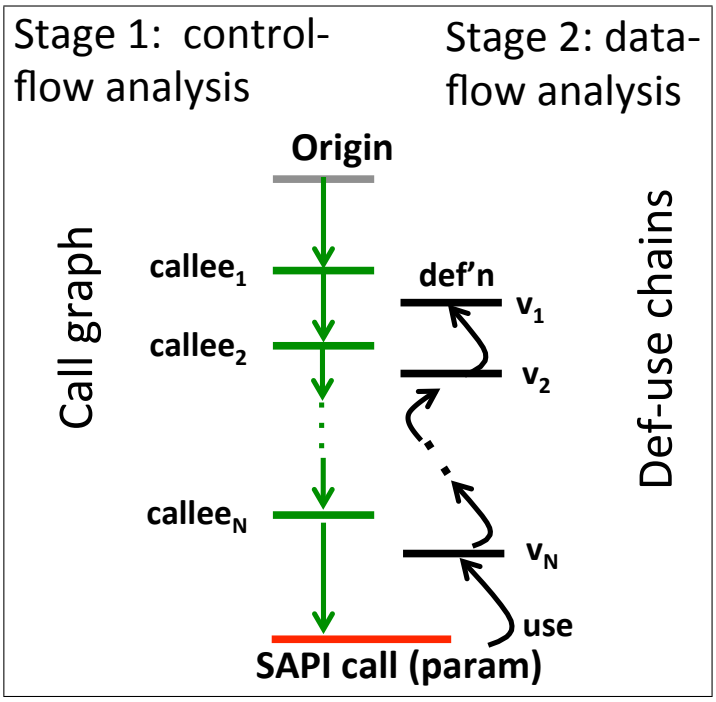

Figure 3.2 SAPI analysis.

we introduce several definitions. User-Decision-GUI (UDG) is an interactive GUI element, e.g., Button, Checkbox, Radio Button, Toggle Button, Spinner, Picker, or menu. User-Decision-Callback (UDC) is a top-level callback method directly invoked as a result of the user action, e.g., onClick(), onCheckedChanged(). In contrast, some callback methods are due to decision-unrelated actions, e.g., onBackPressed(), onScroll(), onEditorAction(). Android offers two ways for creating a correspondence between a callback method and a GUI element: statically defining the call-back as the handler of an event in the GUI element's layout file or dynamically defining a callback for the GUI element by registering a listener object – we handle both. We determine that a given callback is an UDC if either of these two conditions is satisfied: (1) The corresponding GUI of the callback is an UDG, and the event to be handled by the callback is a decision-related event, e.g., click. Note that there exist decision-unrelated events, e.g., scroll and focus change. (2) The corresponding GUI of the callback is an UDG, and the listener of the callback is decision-related,

e.g., onCheckedChangeListener . Note that there exist decision-unrelated listeners, e.g., onCreateContextMenuListener and onFocusChangeListener. Finally, we infer that an API invocation is user-decided if all of its callbacks are UDC, which includes callbacks within the same component and callbacks in other components. If any callback is not UDC, we infer that the API call is not invoked by the user. Further, if this call is an SAPI, it is potentially an SHB. This analysis detects activity hiding, i.e., whether an activity is terminated prematurely, before being displayed. To achieve this, the activity calls finish () within onCreate() , onStart() , or onResume() (or their descendants in the call graph). Therefore, our analysis starts at the beginning of these three callback methods. We perform a control flow analysis to check whether there exists a path from the beginning of the callback to the callback's end that includes finish(); if such a path exists, it indicates potential activity hiding. We name this AF analysis for short. The purpose of this analysis is to check whether the app attempts to manipulate activity attributes in order to deceive the user. The analysis checks both the XML manifest file and the attribute-related API methods. For example, the liner layout of an activity has an attribute "background color". If the attribute value is (hex value)00000000, the activity is transparent. An app can set the value of an attribute in the manifest or layout files, or by calling certain API methods, e.g., setBackgroundDrawable(), setBackgroundColor() or setBackgroundResource(). Another example is the attribute excludeFromRecents which can be specified in the manifest file, or set via the API methods setFlags() and addFlags() . We use SAPI, PAPI, UD, AF, and Attribute to denote the five static analyses. We now explain each rule. Rule 1 reports "Hide icon" when the main activity is removed from the home screen without user involvement. Rule 2 detects "Hide app" if starting an app as a service without user involvement. Rule 3 reports "Hide activity" when the activity finish analysis returns true or the main activity is transparent. Rule 4 infers behavior "Delete message" when deleting occurs after receiving or sending a message. Rule

5 reports "Delete call log" when deleting occurs after making or receiving a phone call. Rule 6 detects "Block message" if blocking received or sent messages. Rule 7 reports "Block call" when blocking incoming or outgoing phone calls. Rule 8 reports "Hide alert" if the app closes a system dialog without user involvement. Rule 9 infers behavior "Hide notification" when canceling a notification without user intervention. Rule 10 detects "Mute phone" when muting the phone surreptitiously. Rule 11 finds SHB "Exclude from recent apps list" if the attribute EXCLUDE FROM RECENTS is set without user's involvement. Rule 12 reports "Delete system log" if that specific shell command is not launched by the user.

3.2 Implementation

We implemented our tool on top of the Soot and FlowDroid static analysis frameworks. These frameworks only analyze bytecode, so we added modules to analyze XML files (e.g., categories and attributes in AndroidManifest.xml, style.xml, etc). Our static analysis modules use both data-flow and control-flow analyses. Finally, the analysis results are produced using the detection rules.

3.3 Limitations

Our tool has several analysis limitations. First, if an SHB is invoked by GUI interaction but the GUI text does not reflect the invocation of the SHB, the tool will not report it; Huang et al.'s idea of finding mismatches between user interface and app behavior could be used to address this limitation. Second, there were a few apps that, due to obfuscation, could not be analyzed, e.g., TripAdvisor (com.tripadvisor.tripadvisor.apk) and KCLS (com.bibliocommons.kcls.apk). Our analysis, built on top of FlowDroid, is based on over-approximation, and handles reflection/native code conservatively, this can be a source of false positives. Also, the SAPI functions with zero parameters tend to have more false positives, a more

precise alias and flow analysis would improve precision. There could be other classes of SHBs, beyond the ones we have discovered. Nevertheless, our list of SHBs: (1) is effective at malware discrimination, and (2) exposes questionable practices in benign apps. Finally, our approach cannot recognize specific malware families: certain SHBs might span multiple malware families. This is expected, as our design goal was at a lower level, automatic SHB identification, rather than clustering malware by family.

CHAPTER 4

EVALUATION

In this section, we present an evaluation of our approach along several dimensions: Is the approach effective at identifying SHBs? Is the approach efficient? What are the main causes of false positives/false negatives? We begin by describing the two datasets used in our evaluation. Our first dataset, which we name MA-198, contains 198 malware samples that were decompiled and analyzed manually, in detail. The 198 samples come from the Malware Genome Project [5], Drebin [2], and AndroZoo [1]. The second dataset, which we name ALL-9452, consists of 6,233 benign apps and 3,219 malicious apps. These apps were analyzed automatically. To ensure that the benign set does not contain malware, we sent all the apps in this set to VirusTotal, a public malware scanning service. If an app is reported by at least one common anti-virus tool as malicious, we removed it from the benign set. For the malware samples, we performed a quick and simple static analysis to eliminate the samples without any possibility to have SHBs. This is done by searching requested permissions, major SAPI calls and intent actions. For example, if an app does not have permissions SEND SMS and RECEIVE SMS, it is impossible to have the SHB "Delete message". Moreover, in order to make sure that the samples are malware, we sent them to VirusTotal. If an app was reported malicious by less than two scanners, we removed it from the malware set. The static analysis tool ran on an 8-core Intel Xeon i7-4770 (8MB Cache, 3.4 GHz) with 32GB of RAM. The system ran Ubuntu 14.04.1, Linux kernel version 3.13.0-32-generic.

4.1 Effectiveness

The test for evaluating effectiveness consists of two steps: SHB detection validation (manual) and large scale measurement (automatic). As there is no existing

oracle to determine SHB, we manually verified each static analysis report. Specifically, we reverse-engineered each app, decompiled the app (to source code) via the JADX decompiler. Note that decompilation is not always possible due to obfuscation, so some of our manual analysis was based on source code inspection, some on Dalvik bytecode inspection. This yields an F-measure of 85.71%, indicating that our tool is quite effective. A sample is identified as malicious if it exhibited any one of the SH behaviors. The tool missed (false negatives, or 'FN') 311 samples from the malware set, hence the recall value is 90.62%. The tool also reported 996 benign apps as having SHB (false positives, or 'FP') from the benign set, hence the precision is 84.02%. While these 996 apps were not malicious, their use of SHB is questionable – we discuss such uses at length in Section 5. Finally, the F-measure is 87.19%; the malware set exhibited 1.5 SHB per sample on average5 while the benign set exhibited only 0.2 SHB per sample. We believe that the high F-measure value and the per-app figures of 1.5 SHB (malicious) vs 0.2 (benign) indicate that our approach is effective for detecting SHB (and perform SHB-based triaging) in Android apps. To better understand the causes of false positives, in Figure 4.1 we have grouped them by SHBs. Five SHBs, "hide activity", "hide notification", "hide icon", "hide app" and "delete system log", generated the most false positives. We investigated this and found that the false positives were due to several reasons: (1) Certain apps employ SHB, such as running in the background without the user having started the app, or without the user being able to see that running app, in the name of improving user experience (see Section 5). (2) Static Analysis: alias, data-flow, and control-flow analyses are over-approximating, which is inherent in static analysis. We have categorized the false negative sources as follows: (1) Parameters of an SHB are dynamically sent from a remote-control server, hence our static analysis cannot identify the behavior. For example, spyware Saveme has a remote server that sends the id, time or phone number through the network to delete certain SMS/MMS messages. (2) SHBs are

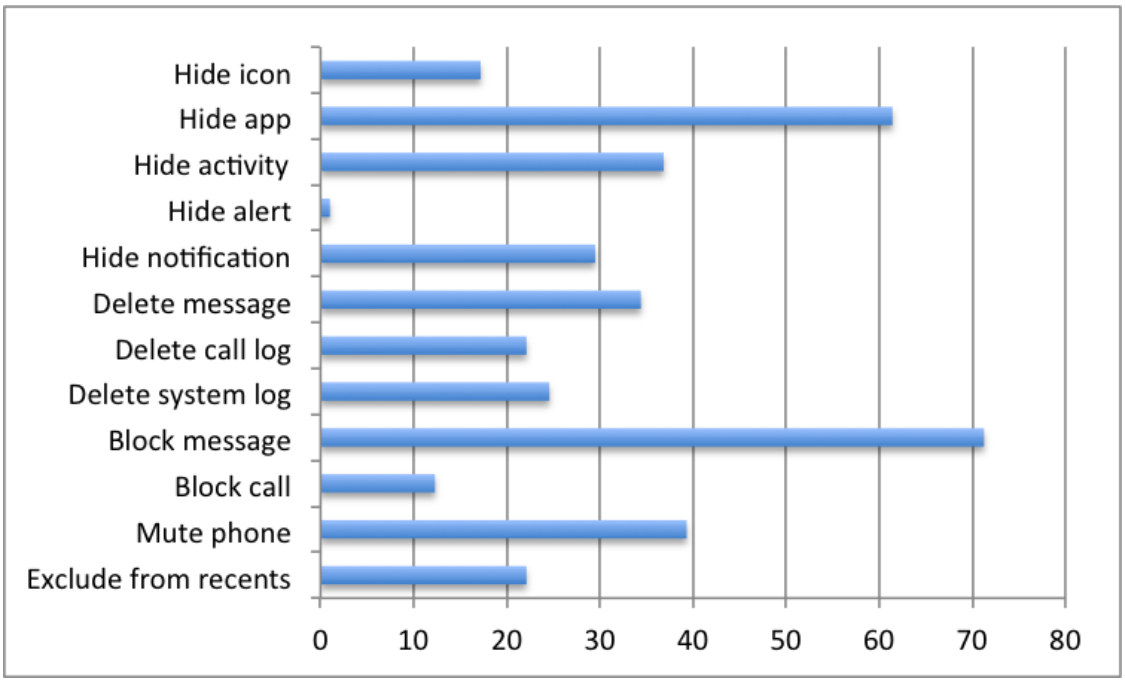

Figure 4.1 FPs generated by each SHB.

launched by GUI interaction, but the behavior mismatches the content shown on the GUI. For example, apps Pure girl and iCalendar employ this behavior. (3) Some malware samples do not have SHB, though they do invoke SAPI calls, e.g., Towelroot and FakeCMC. Our tool did not identify these samples as malicious.

4.2 Efficiency

Running our tool on the 9,452 apps took about 10 days. The datasets had substantial variety in terms of app size, and some apps' bytecode size was as large as 24 MB. The "Time" grouped columns show running time statistics for each dataset. We focus on AA-9452 as it is larger, hence more representative. The mean analysis time was 84 seconds while the median was 37 seconds, which shows that our analysis is practical. Finally, we believe that even the maximum analysis time of 15,290 seconds (i.e.,4 hours 15 minutes) is acceptable for a static analysis. To conclude, with a median analysis time of 37 seconds on a median app size of 2.4MB we believe that our approach is efficient at SHB analysis.

CHAPTER 5

SELF-HIDING BEHAVIOR IN BENIGN APPS

For each SHB category our tool has found in benign apps, we performed a two-part targeted manual investigation: first, we analyzed the disassembled bytecode, and then ran the app with instrumentation to confirm the SHB. We focused this investigation on two categories of apps: (1) apps that are very popular, e.g., with more than 100 million installs; or (2) less popular apps which displayed severe cases of SHB. Ultimately, we aimed to answer the questions "Why does this SH behavior occur and what are the consequences for the user?" This section summarizes some of our findings; we limit the discussion to 8 SHBs for brevity.

5.1 Hide App

Many popular benign apps, such as Airbnb and BBM start themselves as an automatic service after receiving the BOOT COMPLETED event. This event, which requires the permission RECEIVE BOOT COMPLETED, notifies the app that the system has rebooted. In conjunction with this event and permission, there is a function which launches the auto-start service. Our tool reports this as "Hide app" SHB. Apps employ this technique as a means to initialize app-specific information and functions upon startup. While it could be argued that the app is not hiding in the malicious sense (rather it is running in the background to have access to certain types of data, most commonly, location services), we believe that users should know when such apps are running: (1) so they understand why the battery is draining, and (2) so they understand the privacy implications of apps accessing and transmitting sensitive information (e.g., location) in the background.

5.2 Hide Notification

Certain apps, such as Waze, Truecaller, All in One Toolbox, Quick Heal Mobile Security and MiniFetion use NotificationManager. cancel() or NotificationManager-.cancelAll() to block notifications without user intervention. As a result these apps have been marked as having the "Hide notification" SHB. This is due to the nature of cancel() and cancelall() , which cancel all previously-shown notifications. Apps employ this technique as a means to update the user to the most recent notification or to consolidate notifications, especially in communication apps such as MiniFetion and TrueCaller. Lately, many "clean up" and "device maintenance" apps have started to exhibit this behavior for the same reasons. Consolidated notifications may appear convenient to the user; however the app does not have a means to show high-priority notifications first (other than through chronological order). Therefore, users might prefer to receive notifications for all messages to reduce the risk of missing an important notification. However, in the case of Waze, the app blocks certain notifications using Vanagon Notification Manager which cancels all app notifications when the user is not driving. While the app might be trying to appear helpful, notification cancellation and blocking without user's consent/awareness is questionable at best.

5.3 Mute Phone

Our tool discovered the use of AudioManager.setRingerMode() in the benign app Camera360.As its name states, this is a camera app which edits and takes photos; it has more than 100 million installs and was "Best App of 2016 on Google Play in several countries". Many camera apps use volume controls when recording audio. Our tool also discovered the "Mute phone" SHB in certain benign popular apps like Smart Truck Route and All in One Toolbox due to the use of Vibrator.cancel() and AudioManager.setRingerMode(). Regarding Smart Truck Route, the app directly

checks and manipulates the device's audio settings, including its ringer mode. As for All in One Toolbox, the app mutes the phone based on the SDK version of the device. This is dubious behavior for a utility app aimed at optimizing the Android device. To sum up, even though it seems reasonable in some of these cases, we believe that muting the phone should be done by the user through a system-wide control rather than silently by the app.

5.4 Block Message

As the BroadcastReceiver is usually a dormant app component, it is not surprising that its methods can be categorized as SHBs, especially abortBroadcast(). As a result, many benign apps can exhibit this behavior. Interestingly, these apps are not limited to those which rely heavily on BroadcastReceiver. For example, the popular navigation app Waze uses abortBroadcast() which can be construed as the "Block Message" SHB. The abortBroadcast() method is used to prevent other receivers from obtaining the broadcast, thus blocking the communication. It might be justified that Waze employs this tactic as a means to prevent itself from getting location-based alerts that may be irrelevant or annoying to the user. While the intentions of message-blocking apps might appear benign, such blocking removes decision-making from the user and can interfere with usability.

5.5 Block Call

Apps which use ITelephony.endCall() are considered to have the "Block Call" SHB. The benign app Truecaller has the sole purpose to identify and block spam calls, hence it was obviously marked to have this behavior. Despite explicitly stating that it automatically blocks calls, an app which decides for the user which calls are spam can be maliciously manipulated against the user's interest.

5.6 Hide Icon

"Hide icon" achieves its goal by deleting an activity's category. LAUNCHER from the Android manifest. While this deletion merely indicates that that activity should appear as an initial activity of a task, it is evident that a deceitful app can use hide-icon to promote other activities, masking the deceitful app beneath. Many popular benign apps such as ES File Explorer and Next Launcher 3D Shell Lite have this behavior. For example, app Next Launcher 3D Shell Lite is a premium launcher for Android's home screen, but one of its key features is that it draws 3D icons and widgets over their original counterparts. App ES File Explorer has permissions to draw over apps, which is surprising and might be regarded as excessive for a file manager. By having the ability to promote certain activities and controlling the launcher's top level apps, apps with this SHB should be treated with caution.

5.7 Delete Call Log

The app Quick Heal Mobile Security exemplifies this SHB. The app uses Content Resolver Delete() to delete the call logs on the device. The app has call filtering capabilities and has explicit permissions to read and write call logs on the user's device. Nevertheless, (1) users may not be aware of the security implications of log deletion, and (2) the user does not initiate call deletion. These two factors make this particular SHB instance quite problematic.

5.8 Delete System Log

MiniFetion, an app from the Baidu app marketplace, sends free SMS to the user's contacts. Despite the seemingly straightforward nature of the app, we found two highly questionable behaviors. First, the app deletes the system log via "logcat -c". Second, the app has an activity MobClickAgent which uploads device logs to a third party server. Thus the app is able to manipulate, as well as exfiltrate, the

system logs without the user's awareness. While not many popular apps have this SHB, users need to be extremely suspicious of any app which send device logs and user information to third-party servers.

CHAPTER 6

CONCLUSION

Motivated by the common tendency of Android malware to self-hide in order to deceive users and cover malicious traces, we define a set of self-hiding behaviors and construct a suite of static analyses to reveal such behavior. Our experiments indicate that the presence of self-hiding behavior is strongly associated with malice in a given app. Nevertheless, we also found plenty of benign, widely-popular apps that employ hiding techniques, which suggests that end-users and marketplaces would benefit from using an approach like ours to shed light on potential nefarious behavior in Android apps and improve user experience.

BIBLIOGRAPHY

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 13–24, Piscataway, NJ, USA, 2017. IEEE Press.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 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, pages 259–269, New York, NY, USA, 2014. ACM.

[5] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.