Automated Dynamic Detection of Self-Hiding Behavior

Luke Baird
Embry-Riddle Aeronautical University
Department of Computer, Electrical and Software Engineering
Email: bairdl1@my.erau.edu

Zhiyong Shan
Wichita State University
Department of Electrical Engineering and Computer Science
Email: zhiyong.shan@wichita.edu

Vinod Namboodiri
Wichita State University
Department of Electrical Engineering and Computer Science
Email: vinod.namboodiri@wichita.edu

***Abstract*—Certain Android applications, such as but not limited to malware, conceal their presence from the user, exhibiting a self-hiding behavior. Consequently, these apps put the user's security and privacy at risk by performing tasks without the user's awareness. Static analysis has been used to analyze apps for self-hiding behavior, but this approach is prone to false positives and suffers from code obfuscation. This research proposes a set of three tools utilizing a dynamic analysis method of detecting self-hiding behavior of an app in the home, installed, and running application lists on an Android emulator. Our approach proves both highly accurate and efficient, providing tools usable by the Android marketplace for enhanced security screening.**

I. Introduction

Malware is ubiquitous in the Android marketplace. For instance, in 2019, Avast released an article presenting a set of malware applications on the Google Play store that had over 30 million installs combined [1]. The proliferation of Android malware continues each day. As a part of malicious behavior, malware attempts to conceal itself from the user, exhibiting a self-hiding behavior as defined in [2]. With respect to Android applications, this can be manifest as apps hiding themselves from certain Android application lists, such as the home screen app list, the installed app list, and the running app list. Malicious apps hiding in the home list can therefore be installed, but cannot be launched by the user. An app hiding in the installed app list cannot be uninstalled by a normal user, as the installed application list is where most users uninstall an app. Finally, if an app is hiding from the running app list, the app can run without the user ever knowing.

It would be beneficial if apps exhibiting this behavior could be caught prior to their uploading to the Android marketplace in a timely manner. Although not all self-hiding behavior is malicious, as discussed in section five of [2], automatically detecting self-hiding behaviors allows for further vetting to insure those behaviors are not malicious. Previous research has produced static analysis methods of detecting self-hiding behaviors. However, these static analysis methods suffer from code obfuscation, preventing reverse-engineering of the source code [2], [3].

We propose a set of three tools—AutoSHBHome, AutoSHBInstalled, and AutoSHBRunning—that perform dynamic analysis to detect self-hiding behavior in the home, installed, and running application lists respectively on an Android device. These tools could be used to analyze apps when they are uploaded to the Android marketplace prior to those apps being made available to the general public, thereby detecting self-hiding behaviors before a hiding app reaches a user's phone. Each of our tools complete on average in less than two minutes per app analyzed with an F-measure of greater than 97%.

A video demonstration of AutoSHBHome, AutoSHBInstalled, and AutoSHBRunning can be found at the following links, respectively:

- https://youtu.be/AYC839XoMlY
- https://youtu.be/jdkPtFrhnMc
- https://youtu.be/QrwX51Tla7Q

The two main contributions of the research presented in this paper are:

- The first dynamic analysis tool set to detect self-hiding behaviors
- An analysis on a set of 77 benign and malicious applications, revealing their self-hiding behaviors

II. Background

Our tools seek to detect self-hiding behavior within the lifecycle of an app on an Android phone. This lifecycle covers the existence of an app on a user's Android phone from its installation to its uninstallation. When the user first installs an app, the app should appear in the user's home screen as an icon until the user decides that they want to launch it. After the app is launched, it should appear to the user that it is running. Eventually, when the user wishes to uninstall the app, the user navigates to the install application list and deletes the app from the phone.

We see then that the android app lifecycle includes the following events for a proper app:

- Installation of the app
- Inclusion of the app in the home app list, after installation
- Inclusion of the app in the running app list, after launch
- Inclusion of the app in the installed app list
- Deletion of the app from the device

Consequently, if the app is hiding from the home, running, or installed app lists, this directly interferes with this lifecycle,

978-1-7281-4121-3/19/$31.00 ©2019 IEEE
DOI 10.1109/MASSW.2019.00024

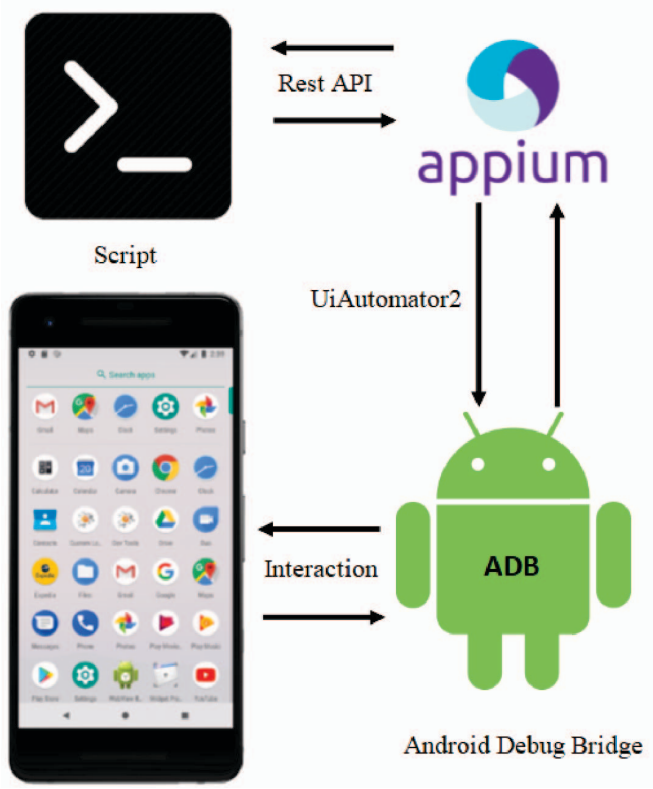

Fig. 1. Interface between a script, Appium REST API, Android Debug Bridge (ADB), and a target emulator or device.

preventing a normal user experience with the app. Furthermore, because the app is hiding, the user may be unaware of the app's presence, threatening the user's privacy and security. Our tools address this by uncovering self-hiding behaviors in the home, running, and installed app lists.

III. Tool Design

The centerpiece of this set of three tools is the Appium Framework. Appium is designed as a REST API server, capable of receiving JSON requests from a script and returning HTTP status codes. For each tool, Appium is used as an interface between a Python script and an emulator interface. Appium directly interacts with the Android Debug Bridge (ADB), which relays Appium's commands to the emulator. This is shown in figure 1.

Appium requires the use of a driver in order to send commands to ADB. UiAutomator2 is one such driver built into modern versions of Android. UiAutomator2 provides a set of automated testing tools suitable for interacting with a device. UiSelector is a tool built into UiAutomator2 for fetching certain elements from the emulator's foreground activity.

In order to connect to the device, Appium creates a session with a set of desired capabilities which describe, among many specifications, which application to load. The three tools developed by this research navigate either through the home screen, the Settings app, or through both. For an application already installed on a device, our script sends the app's package and the app's launchable—or starting—activity to the desired capabilities in Appium. Appium and its uses are further discussed in [4], [5].

Our initial script accepts command line argument flags signifying which of AutoSHBHome, AutoSHBInstalled, and AutoSHBRunning need to be run along with a directory containing APK files to analyze. The script promptly uses ADB commands "adb shell pm list packages" and "adb uninstall <package>" to loop through each package on the device and attempts to uninstall it. If the command fails to uninstall a package, it is assumed that the package is mandated by the system. This is done to minimize the number of apps that the tools have to search through in different lists. At this point, the script runs each tool that is flagged on individual apps in succession.

A. AutoSHBHome

The process for detecting self-hiding behaviors in the home app list is illustrated in figure 2. AutoSHBHome begins by creating a new Appium session for the home screen application. On a Pixel 2 emulator, the package for the home application is com.google.android.apps.nexuslauncher, and the launchable activity is .NexusLauncherActivity. Upon creating a new session, AutoSHBHome uses Appium to navigate from the main home screen to the list of all apps in the home screen. Appium is then used to count all of the displayed apps on the screen before closing the session to avoid memory leaks. At this point, the target APK file for this iteration of AutoSHBHome is installed with a simple adb install <apk>command. If this command fails, the self-hiding behavior of the app is reported as unknown. Finally, Appium is again used to create a new session with the home application to count the number of applications in the home app list. If the number of applications is different (increased by one), then the application is not hiding in the home application list. Otherwise, it is.

Each application icon in the home screen is of the class android.widget.TextView. The label associated with the application is the content description: "content-desc". In order to count the number of applications, the content description of each TextView is added to a saved list in Python. An attempt is made to simulate an interaction with the device where the user drags their finger from the last icon displayed in the lower right to the first icon displayed in the upper left—a scrolling action. If the last application in the list after scrolling is different from the last application in the list prior to scrolling, then all content descriptions not already in the saved list are added to it. This continues until the other condition is true, when the last application displayed both before and after scrolling are the same. At this point, the procedure terminates, returning the length of the saved list.

In order to tap the correct element to reach the home application list, Appium's inspector tool is used to manually navigate through the home screen and reach each element. This reveals the accessibility id of the up arrow leading to the list of all applications in the home screen, along with the class

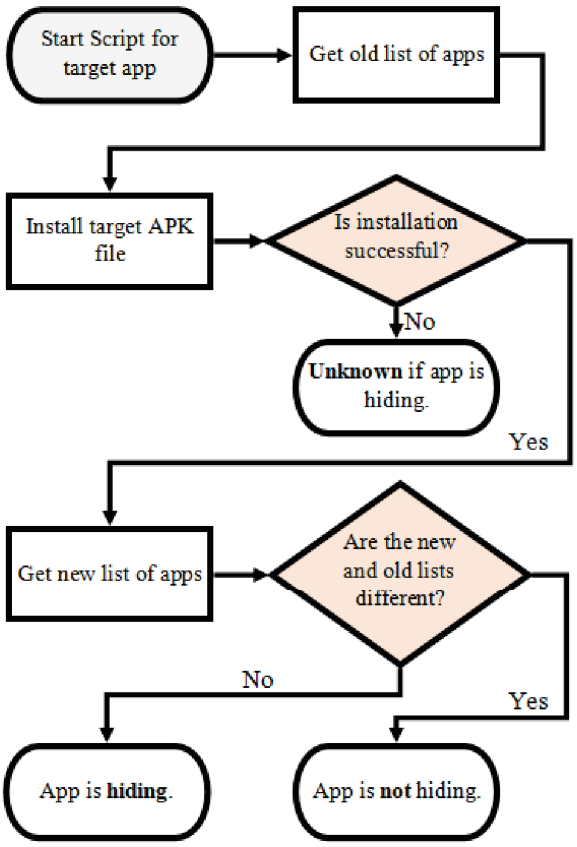

Fig. 2. Flowchart illustrating AutoSHBHome and AutoSHBInstalled's algorithm.

and layout of the individual icons. The inspector tool is used to discern the right target element types and hierarchies for the other two tests as well.

B. AutoSHBInstalled

AutoSHBInstalled works similarly to AutoSHBHome, using the same algorithm shown in figure 2. It too begins by starting a new Appium session, this time for the settings application. On the Pixel 2 emulator, the package for the settings application is com.android.settings and the launchable activity is .Settings. After navigating to the installed application list within the settings app, the number of apps are counted and the session is closed. The target APK file is installed on the device with adb install <apk>and the process is repeated.

Navigation to reach the installed application list involves repeated use of the UiAutomator2 Google-provided testing platform. UiAutomator2 has a method that allows for Appium to search for an element by its text. Appium requires the Python script to pass in a Java code snippet using the Java UiSelector class that contains a selector element. Upon reaching the installed application list, AutoSHBInstalled scrolls and counts elements using a similar methodology to that of the home application list test. All clickable elements currently visible on the screen are pulled. If the element is of the class android.widget.RelativeLayout or android.widget.LinearLayout, then the first of two shown TextViews within that parent element is added to the list of found applications if and only if there are two TextViews. In the installed application list, these elements happen to be LinearLayout, although for the sake of code re-usage with the running application list test, both types of layouts are valid for pulling elements in a list. A swipe action is then completed, scrolling by the number of pixels equal to six times the height of an individual element in a list. This is repeated until the last element shown on the screen after a swipe action is equal to the same element prior to a swipe action.

Testing the positive case of this procedure—where an application hides itself from the installed application list—proved difficult as very few applications, even malware applications, hide themselves in the installed application list. Neither the benign nor malicious apps in the dataset used in the development of this tool contained such an application. However, testing the positive result can be done without installing any APKs, as a hidden app should add zero apps to the installed list. Running the test twice, therefore, should return that there is self-hiding behavior.

C. AutoSHBRunning

Unlike the prior tools, the running application list self-hiding behavior detection tool AutoSHBRunning uses two different Appium sessions to execute different tasks. The algorithm is shown in figure 3. AutoSHBRunning must first install the target APK file, open the application, and only then navigate to the running application list within the settings app and attempt to find the application. If the application is found, then the app is not hiding, otherwise, it is reported as hiding. If the app either failed to install or failed to launch, then the self-hiding behavior of the app is undetermined. The first Appium session navigates through the home application list to tap the icon of the target app, while the second Appium session navigates through the settings app to the running app list.

In order to find the correct app in the home app list, the "aapt" command, provided by the Android build tools, is used to parse the application label from the target APK file. This label is the same text that appears in the content description of the home application list icons and is the same text that appears with the target application in the running app list.

Like AutoSHBHome, Appium is used to navigate first to the list of all applications. Then, the same sequence is used to scroll through the list until a label is found equal to the label of the target application. This session is closed prior to starting the second session for the settings app.

Upon opening the settings app session, the sequence designed to scroll through the list of options in the installed application list is used until the system settings are found, under which the developer options reside on our emulator. Finding this is done with the UiAutomator2 driver as before with AutoSHBInstalled, searching for specific text. If the

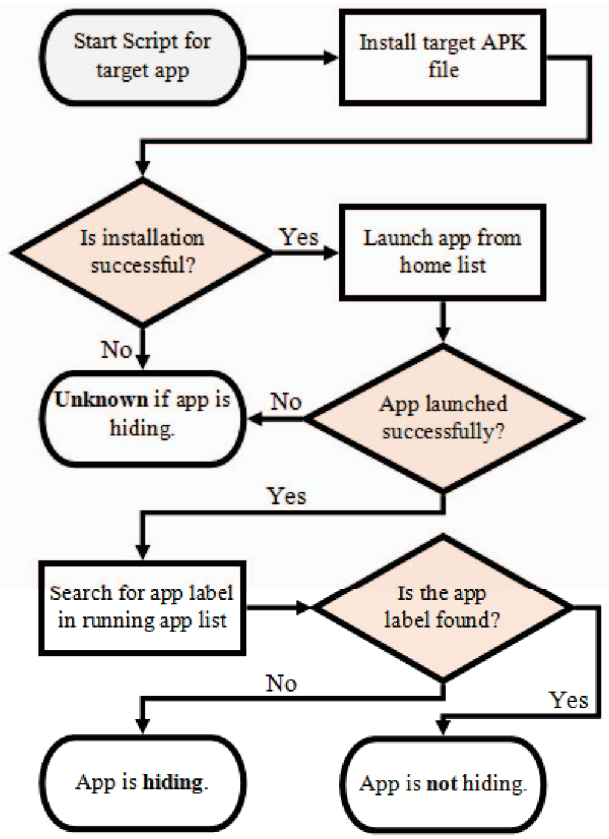

Fig. 3. Flowchart illustrating the algorithm for AutoSHBRunning.

developer options cannot be found within the system settings, the script navigates to the build number, again using UiAutomator2. After tapping the build number seven times, the script navigates up one level and attempts to find the developer options again. Within the developer options is the running services list. If the label is found in the running services list while scrolling and searching, the application is reported as not hiding. Otherwise, to display non-foreground processes, the text "show cached processes" is tapped, and AutoSHBRunning searches and scrolls through the new list. If the label is found, the target app is reported as not hiding, otherwise, it is reported as hiding in the running app list.

One major limitation of AutoSHBRunning is that applications are loaded based on their label in the home application list. If the application under test has already proven itself to be hiding from the home application list, AutoSHBRunning will fail. Another issue is present when testing certain malicious APK files. Loading certain apps will cause the malware to dump its payload, impacting the performance of or altogether crashing the emulator. Some malware that requests device admin permissions will force the user to either reject or accept a device admin request, but will continually reload the request until the user taps "accept."

IV. Evaluation

The tools were run on a Pixel 2 AVD Android Emulator running an x86 image of Android Oreo 8.1. Certain applications failed to install in testing due to use of an x86-imaged emulator. An x86 emulator runs much faster than an ARMv8 emulator on a Windows platform, which is also compiled for x86. These applications are removed from the dataset. The testing host platform ran Microsoft Windows, version 10.0.18362.239. Our dataset included 20 benign representative APK files and 57 malicious APK files.

The tools are highly UI-dependent. Upgrades to the Android version of the emulator or the use of a different emulator would require modifications to be made to the scripts in order to use them. However, for analyzing an application in a controlled environment prior to its release to the Android marketplace, this is an acceptable limitation.

Manual analysis was performed on each of these APK files by hand, installing and opening each list (home, installed, and running) to visually check if the target application was hiding. The results of this manual analysis are used as the standard by which false positives and false negatives are judged. Our results are listed in Table I.

As mentioned before, AutoSHBRunning returns an error for apps that are hiding in the home app list, resulting in a larger number of errors. An application was not found to be hiding in either manual or automated analysis of the installed application list. Based on this data, self-hiding behavior in the home application list is the most common hiding behavior in this dataset.

If the emulator is unstable from running malware applications, or if the Appium server loses connection to ADB, random errors can occur. This is the reason for the three errors associated with AutoSHBHome and the single error with AutoSHBInstalled. This also explains the errors in the running application list that are not due to home app list self-hiding behavior. Malware sometimes fails to run on the emulator completely, hence decreasing the valid sample size from 77 to 63 for the running application list between ignoring apps that failed to run and ignoring apps hiding in the home app list.

A false positive can occur if Google Play protect is enabled on the emulator while a test is run on a malicious APK. Google Play protect recognizes certain malicious APKs and can remove them before detection tools have time to navigate to their target lists.

The efficiency data is listed in Table II. Out of all of these three tests, AutoSHBInstalled took the most amount of time per app and overall. This is due to the fact that the installed application test has the most navigation steps out of any of the tests.

AutoSHBRunning took an average time of less than two minutes per application. If an application is invalid—such as hiding in the home application list—the tool returns a result in as little as 5 seconds. However, if an app results in the emulator hanging, or an unexpected navigation event, the tool

Test	# analyzed	# hiding	# not hiding	False positives	False negatives	Precision	Recall	F-Measure	Errors
AutoSHBHome	77	12	62	2	0	97.47%	100%	98.72%	3
AutoSHBInstalled	77	0	76	0	0	100%	NA	100%	1
AutoSHBRunning	63	3	40	0	0	100%	100%	100%	20

TABLE I
ACCURACY OF ANALYSIS

Test	Total time	Average time per app	Median time per app	Maximum time per app	Minimum time per app
AutoSHBHome	8569 seconds	85.9 seconds	84 seconds	139 seconds	30 seconds
AutoSHBInstalled	14712 seconds	149.3 seconds	156 seconds	188 seconds	76 seconds
AutoSHBRunning	10982 seconds	111 seconds	96 seconds	1373 seconds	5 seconds

TABLE II
EFFICIENCY OF ANALYSIS

can take a long time to time out and continue working through the applications, as seen by the maximum time per app for AutoSHBRunning.

V. Related Work

Since the creation of the Android operating system, a plethora of tools have been developed to detect and analyze malware. Research in certification of apps to hamper malware was investigated as early as 2009 [6]. Other papers work in classifying malware based on its behavior [7]. In [7], the implementation of different sets of malware is carefully dissected and classified, though this requires that a target piece of malware not have obfuscated code. RiskRanker [8] tries to proactively find malware when it is first uploaded to the Android marketplace.

[9] discusses covert channels being used to secretly share a user's data. These channels are designed to transfer control information or other metadata, but instead, malware can use these channels to transmit private data. However, this research does not explicitly seek out self-hiding behaviors for their own sake. [2] does exactly this using a static analysis approach. This research classifies self-hiding behavior into twelve different categories and is the first to attempt to detect self-hiding behavior specifically.

Dynamic analysis techniques have been used with the Android operating systems in prior research [3], [10], [11]. DREBIN [3] uses static analysis whenever possible prior to dynamic analysis to minimize usage of a phone's resources, but then uses dynamic analysis and machine learning to build behavioral patterns for a user. TraintDroid [10] monitors applications in flight to see how applications use their private data in ways that are not visible to a user. Finally, Stowaway [11] combines both dynamic and static analysis techniques to find apps that are overprivileged. Static analysis shows the APIs that are needed by the decompiled target APK's code, while dynamic analysis reveals which APIs map to which permissions requested by an application's manifest file.

VI. Conclusion

This research has successfully developed a set of three tools—AutoSHBHome, AutoSHBInstalled, and AutoSHBRunning—for automated dynamic detection of self-hiding behavior in the home, installed, and running app lists on an Android emulator. Based on the high F-measure rates and the time that it took to run these tools, we conclude that the tools are both effective and efficient. Future work extending these tools to a wider variety of emulators could be pursued. Another area in which these tools could be developed further is in regard to the running application list self-hiding behavior detection tool, AutoSHBRunning. Methods of detecting and restarting a frozen or hanging emulator could be implemented in the tool better. Potentially, these tools can be used to analyze an application in a controlled environment prior to its publishing to the Android marketplace, saving time and money by avoiding having to deal with malware on a device.

References

[1] "Adware plagues google play store," Apr 2019. [Online]. Available: https://blog.avast.com/adware-plagues-google-play

[2] Z. Shan, I. Neamtiu, and R. Samuel, "Self-hiding behavior in android apps: detection and characterization," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 728–739.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[4] G. Shah, P. Shah, and R. Muchhala, "Software testing automation using appium," *International Journal of Current Engineering and Technology*, vol. 4, no. 5, pp. 3528–3531, 2014.

[5] S. Singh, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on appium," *International Journal of Current Engineering and Technology (IJCET)*, vol. 4, no. 5, pp. 3627–3630, 2014.

[6] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[7] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 95–109.

[8] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[9] J.-F. Lalande and S. Wendzel, "Hiding privacy leaks in android applications using low-attention raising covert channels," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 701–710.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[11] Z. Aung and W. Zaw, "Permission-based android malware detection," *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.