Multimedia-Based Testbot for Detecting Malware-Hiding Behaviors

Zhiyong Shan
School of Computing
Wichita State University, USA
zhiyong.shan@wichita.edu

Sathyanarayanan Vimalan
School of Computing
Wichita State University, USA
vxsathyanrayan@shockers.wichita.edu

***Abstract*—Less attention has been paid to the deceptive mechanisms of malware on smart devices. Smart device malware uses various techniques to conceal itself, e.g., hiding activity, muting the phone, and deleting call logs. In this work, we developed a novel approach to semi-automatically detect malware hiding behaviors. To more effectively and thoroughly detect malware hiding behaviors, our prototype checks multiple mediums, including vision, sound, vibration, phone calls, messages, and system logs. Our experiments show that the approach can detect malware hiding behaviors. The F-measure is 87.7%, indicating that our approach is quite effective.**

***Keywords*—Android, Hiding Behavior, Testbot**

I. Introduction

While smart devices bring new functions and possibilities to users, they also open up a Pandora's Box of potential problems, from eavesdropping attacks to erasing all data on a storage device. Security research on smart devices mostly focuses on malware installation and activation, privilege escalation, remote control, financial charges, personal information stealing, and permission use. Less attention has been paid to the deceptive mechanisms, which are critical for the success of malware on smart devices. Generally, malware first gets installed and then continues operating on the device without attracting suspicion from users. To do so, smart device malware uses various techniques to conceal itself, e.g., hiding activity, muting the phone, and deleting call logs.

We define the ***malware-hiding (MH) behavior*** as an action by which smart device malware can hide itself or its trail from being viewed (or heard!) by users. According to our preliminary work [1], smart device malware samples exhibited ***1.5 MH behaviors per sample*** on average, while benign samples exhibited only ***0.2 MH behaviors***. This indicates that apps with MH behaviors are most likely malware rather than benign apps.

Existing studies on MH focus on static analysis and are ad hoc [1]. In this paper, we develop a dynamic analysis approach, i.e., MH Testbot, to comprehensively find MH behaviors in smart device malware apps. The approach exploits multimedia to automatically test apps in order to capture MH behaviors in runtime.

To more effectively and thoroughly detect MH behaviors, the MH Testbot checks multiple mediums, including vision, sound, vibration, phone calls, messages, and system logs. We have chosen to explore multiple mediums for a number of reasons. First, a malware may hide itself from being seen, heard, or sensed using different means, and some MH behaviors cannot be detected by analyzing only a single medium. For example, detecting "mute phone" in the malware *FunkyBot* requires monitoring both vision and audio; detecting "delete system log" in the Trojan malware *SMSblocker* requires monitoring both log and vision. Second, monitoring multiple mediums can reduce false positives. For example, concurrently monitoring phone calls, vision, and sound can reduce false positives when detecting malware *fakeAV*, which blocks incoming calls. Third, monitoring multiple mediums helps recognize the screen. For example, monitoring both vision and audio can improve the accuracy when recognizing alert or advertisement windows. Fourth, in addition to MH behaviors, monitoring multiple mediums helps locate bugs that require information other than vision. For example, to detect the recent Face Time bug [2], it is necessary to recognize the audio of the remote person.

II. Related Work

Smart device malware-hiding mechanisms are studied much less than desktop/server malware. All studies on MH focus on static analysis, but existing MH static analysis schemes are not generic. Moreover, as far as we know, there is no applicable dynamic analysis technique for MH.

Existing static analysis approaches are ad hoc and cannot detect unknown MH behaviors. Wei et al.

[3] mentioned four approaches used by malware to hide evidence of malicious activity, but the approaches were not studied. StateDroid [4] proposes a two-layer, attack-action-driven finite state machine model to reason about stealthy attacks in Android applications. StateDroid uses a set of attacks and actions to detect specified malicious behaviors in Android apps, but this set may not be sufficiently comprehensive to detect unknown attacks. AsDroid utilizes the contradiction between the implemented app actions and user's expected behaviors to detect stealthy behaviors. AsDroid relies on API-based detection of six actions, such as starting a phone call, sending a SMS message, and inserting data into a sensitive database. Therefore, it might not be able to detect behavior that does not use these six actions. HSOMINER [5] is a program analysis technique to detect a hidden sensitive operation (HSO). An HSO is different from MH in that apps deliberately hide their sensitive behaviors behind events triggered only in their target situations. For example, a potentially harmful app only pops up advertisements and collects a user's contacts when it runs on a physical device (not an emulator) and interacts with a human. But MH means hiding the app itself, regardless of the runtime environments. VAHunt [9] conducts data flow analysis to determine stealthy plugin loading behaviors. We develop a self-hiding behavior (SHB) detector [1] to identify a number of malicious behaviors that conceal app activities, such as removing traces of suspicious actions and hiding the presence of the app. But the detection approach is still ad hoc.

Existing dynamic app analysis systems are not applicable for testing MH behaviors. There are three categories of existing dynamic app analysis systems[10][11][12][13][14][15][16][17]. First, vision-based systems (e.g., Sikuli [12]) cannot scale to a large number of app types required by MH testing because they use pre-prepared images to identify screen elements of an app. Second, instrumentation-based dynamic analysis systems (e.g., A^{3}E [10], PUMA [13], Dynodroid [11]) cannot work on pre-installed apps or native codes because they rely on instrumenting OS libraries or apps. But testing MH behavior must refer to system apps like Settings app. Therefore, instrumentation-based systems are not feasible for MH testing. Third, other dynamic analysis systems (e.g., Monkey [14], DroidFuzzer [15], IntentFuzzer [16]) use random exploration strategies [17] and thus are not suitable for testing MH behaviors.

III. MH TESTBOT

The development of our MH Testbot faces **three challenges**. First, triggering most MH behaviors contained in an app is not easy, because the behaviors are often scattered over the codes and activities of the app. Some MH behaviors even need very ad hoc conditions to be triggered. Second, MH behaviors are varied, ranging from hiding activities or dialogs to blocking calls or messages. This requires the detection algorithm to be versatile. Third, in addition to log analysis, MH detection requires recognizing multiple mediums, including image, audio, and vibration. We will present our solutions to these problems.

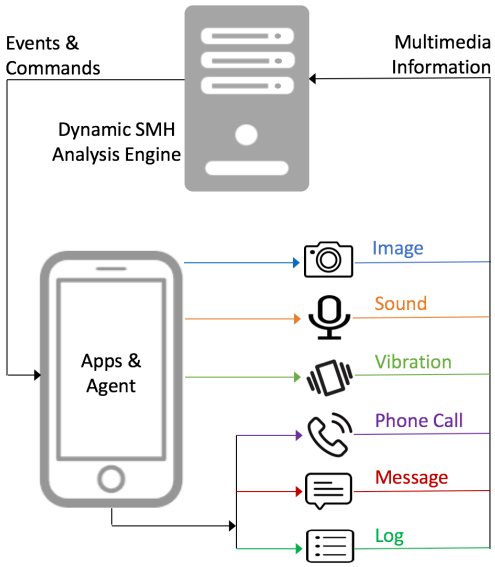

Figure 1: MH Testbot

As shown in Figure 1, our MH Testbot contains a dynamic MH analysis engine that runs on a server connected with a smart device. The engine imitates the user operations to install, invoke, restart, and navigate apps and activities by sending events to the smart device. The engine detects MH behaviors by analyzing multimedia information collected from the smart device. An agent running on the smart device collects and sends information to the engine. To implement the engine, we need to address the following challenges:

Triggering MH Behaviors. Because there are various types of MH behaviors that can hide at many different sites in the Android OS, it is a challenge to trigger them for detection. To invoke as many MH behaviors as possible, we propose a systematic scheme involving four levels, each triggered by commands or events from the engine:

- Level 1—Application: According to our observations, most MH behaviors can be triggered by application-related operations, which include app installation, invocation, and restart. The engine automatically interacts with the smart

device to perform these operations.

- Level 2—Activity: The engine sends events to drive an app to navigate all reachable activities with the guidance of an activity transition graph [10].
- Level 3—Callback Function: This is done by sending messages and phone calls to the smart device, monitoring outgoing messages and phone calls, or clicking buttons or menu items on the screen of the device. Most MH behaviors triggered at this level are related to blocking calls or erasing messages.
- Level 4—System: The operations that can invoke MH behaviors include restarting the system, sending system alerts, and turning on alert/sound/vibration/notification.

Detecting MH Behaviors. MH behaviors can appear in various multimedia sites with different formats. For example, an app can conceal itself in the installed-apps list or remove its messages from the sent-message list. Multimedia sites (such as these two lists) that can hide malware are defined as MH **sites**.

In order to comprehensively detect as many MH behaviors as possible, we use two procedures. First, we build a multimedia-based normal behavior matrix M of dimension $m \times n$ to indicate whether an app should occur at an MH site when triggering an event. A row corresponds to an event triggered, and a column represents an MH site. The matrix is binary, with an entry m_{ij} value set to 1 to indicate that a normal app should appear at the MH site j when the event i is triggered, and 0 otherwise. Second, based on the matrix, our detection algorithm works as follows: After triggering an event i for app a, the engine reads the row M_i. For each $m_{ij} \in M_i$, if $m_{ij} = 1$, the engine checks whether app a appears at the MH site j, and an MH behavior is detected if app a does not appear.

We find MH sites that can hide apps, such as lists for running apps, installed apps, device administrators, messages and notifications, home screens, call logs, system alert windows, and system logs. These sites can be categorized into three styles: app-list, message-list, and pop-window. To be complete, we also consider audio and vibration as MH sites, since muting phone and turning off vibration are also MH behaviors. Each time we send events to trigger MH behaviors, the engine checks the MH sites to detect hiding apps with our multimedia algorithms for image recognition, audio recognition, and vibration recognition.

To determine whether a given app is hiding in an app-list, the MH recognition algorithm first extracts the icon and name of the app from the apk file, and then capture the icons and names of all apps on the screen through the image-recognition algorithm. Next, it calculates the similarity values among icons of the given app a and those on the app-list, as well as similarity values among names. If all similarity values are smaller than the threshold values λ_{icon} and λ_{name}, as shown below, then the app a is not on the list but is hiding.

$$\underset{1 \le j \le n}{MAX}\ Similarity(Icon_a, Icon_j) < \lambda_{icon}\ \&\ \underset{1 \le k \le n}{MAX}\ Similarity(Name_a, Name_k) < \lambda_{name}$$

For MH sites that use message-list or pop-window, the MH recognition algorithm analyzes not only screen images but also audio and vibration. For example, if the image recognition algorithm detects a system alert window but the audio recognition algorithm does not receive an alert sound, then this means that the smart device is muted. If an app sends out a message, then the message is considered by the engine to be background if the audio recognition algorithm does not detect a message sound and the vibration recognition algorithm does not sense the message vibration.

Recognizing Image. Recognizing each type of GUI element on the screen of a smart device can help in navigating activities, handling unexpected alert windows, recognizing hidden apps, and entering text. Typical types of GUI elements include button, link, label, text box, check box, menu, drop-down list, and icon. Existing vision-based dynamic analysis systems search within a screen image for a pre-stored GUI component image [12], in order to recognize a component. Because the pre-stored images are pre-extracted from known apps, these analysis systems are not suitable for testing a large number of apps, which mostly are unknown. Therefore, we design a novel algorithm to recognize GUI elements from a screen image.

- First, rather than going through a conventional segmentation algorithm, such as the Watershed algorithm [18], we leverage some distinct characteristics of the Android app GUI to simply divide a screen image into multiple regions at low overhead. A good example is that many Android app screen images are separated by horizontal rigid lines.
- Second, we use Belongie's algorithm [19] to recognize shapes, and an optical character recognition algorithm [20] to recognize text in each region. We also recognize colors and coordinates of texts and shapes.
- Third, we collect a large number of screen images from apps, based on those in which we can

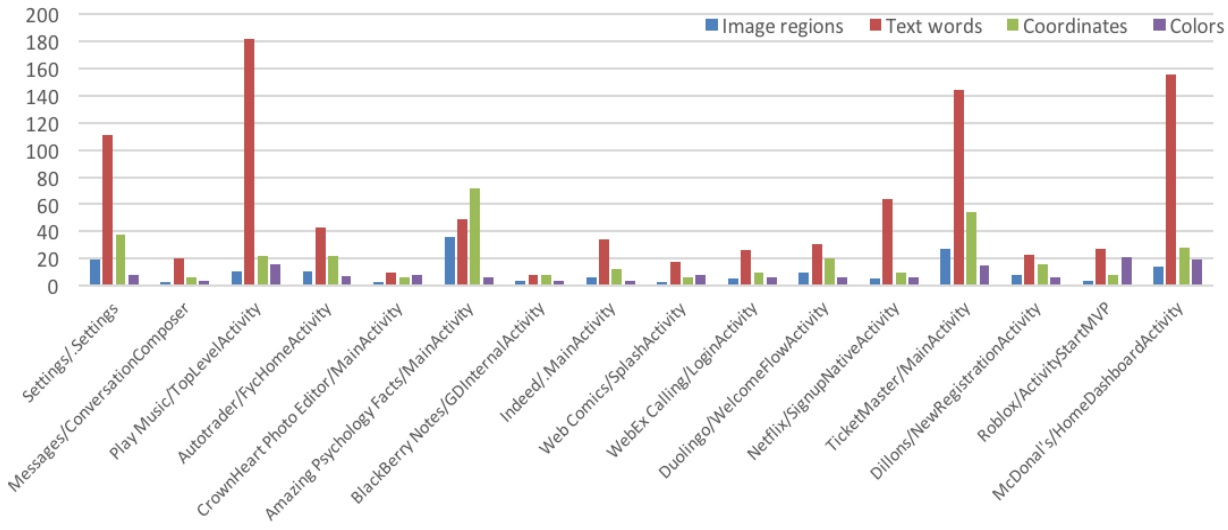

Figure 2: Results of Recognizing Image

identify the types of GUI elements on each screen image via static analysis of app binary files.

- Last, we develop a GUI element-recognition algorithm based on machine learning with decision trees [21] built from the list of possible attributes and the set of training cases. The trees are used to classify subsequent sets of test cases. Classification attributes are obtained from the second step, and training data sets are obtained from the third step.

Recognizing Audio. As a part of the procedure to detect malware, our MH Testbot recognizes the type of sound playing from a smart device to remain aware of the arrivals of phone calls, messages, system alerts, screen locks, etc. Typical sound types to detect include ringtones, notifications, alerts, media, touch sounds, screen lock sounds, emergency tones, dialing keypad tones, and keyboard sounds. Our audio recognition scheme requires three steps: recording a short segment of audio from the microphone of the smart device, finding audio files and their types from the smart device, and searching for the audio segment within the audio files to determine the audio type.

An audio segment generally cannot exactly match any section of the audio files, due to the existence of noise from different methods of digitization, environment, and thermal sources. Instead of using an exact match, we use similarity as the metric. We slide the audio segment across the audio files to find the part that is similar. Whenever similarity rises above a threshold value, it indicates that the reference audio file is the one containing this segment, and the file name and directory provide the sound type. The similarity between the audio segment A_s and audio file A_f can be computed using histogram intersection [22]:

$$S(A_s, A_f) = \sum_{i=1}^{B} \min\,(A_f^i, A_s^i)$$

where B is the number of histogram bins, and $min(A_f^i, A_s^i)$ represents the minimum histogram intersection match value between bins A_f^i and A_s^i.

Recognizing Vibration. To detect whether the smart device is vibrating, we install a vibration sensor near the device. The sensor sends signals to the engine when the device is vibrating. Then the engine can detect the vibration.

IV. Evaluation

The MH Testbot prototype was implemented upon Appium [24], which is an open source automation tool for running scripts and testing mobile applications. We evaluated the MH Testbot on its effectiveness and performance overhead. We analyzed 16 malware samples from well-known sources, such as AndroZoo [6], Virustotal [7], and AAGM [8].

Evaluating Effectiveness. As there is no existing oracle to determine MH, we manually verified each dynamic analysis results. The results are shown in Table 1. Our tool has reported 29 MH behaviors in total; of these 25 were true MH behaviors, while 4 MH behaviors were over-reported (false positives) and 3 were false negatives, i.e., our tool missed those MH behaviors. This yields an F-measure of 87.7%, indicating that our tool is quite effective.

Our tool uses a computer vision library, OpenCV [23]. It can take a snapshot of a smart device screen, divide the image into multiple regions, recognize text and its color, and compute the coordinates of each region.

TABLE 1. EFFECTIVENESS RESULTS

True MH behaviors	Over-reported MH behaviors (FP)	Under-reported MH behaviors (FN)
25	4	3
Precision: 25 / (25+4) = 86.2%		
Recall: 25 / (25+3) = 89.3%		
F-measure: 2 * (86.2*89.3) / (86.2+89.3) = 87.7%		

The testing results for 16 activities from 16 apps are shown in Figure 2. Each activity shows the number of image regions, text words, coordinates, and colors. Most of these apps are popular in the Google Play Store. We have manually verified the results, and they are correct.

Analyzing Performance Overhead. Table 2 shows the performance overhead of our prototypes for recognizing image, audio, and vibration, respectively. Our MH Testbot consisted of the following: an Intel Xeon server (36 logical cores@4.3 GHz, 4 doublewidth GPUs, 128 GB RAM, Ubuntu 18.04) where we ran the multimedia recognition algorithms, and an Android phone (4 ARM cores@1.40 GHz, 2 GB RAM, 16 GB storage, OS version 6.0). We tested 16 activity images, 16 audio fragments, and 16 vibration signal segments (each has 1 second in length). From Table 2 it can be seen that the average times for image, audio, and vibration recognition are 75, 3.9, and 0.18 milliseconds, respectively. Therefore, the performance overhead of multimedia recognition is acceptable.

TABLE 2: PERFORMANCE EVALUATION

Algorithm	Min	Max	Average
Image recognition (msec per activity)	32	193	75
Audio recognition (msec per second length of audio)	2.2	5.7	3.9
Vibration recognition (msec per second length of vibration)	0.11	0.3	0.18

V. SUMMARY

In this paper, we built a novel automated dynamic analysis bot to detect MH. This bot exploits multiple mediums (e.g., vision, audio, vibration) to mimic a human user, in order to automatically drive the test and recognize MH behaviors. Our testbot is different from existing approaches because it relies on vision/audio rather than specific app implementation (e.g., code snippets and data flows). Anti-malware companies and app markets can use it to analyze suspicious apps.

The MH Testbot can potentially be extended and applied to detect other malware behaviors, app errors, or normal functionalities of Android apps. This includes, for example, advertisement pop-up windows, ransomware, undeletable apps, deathless administrators, resume and restart errors, crashing errors, account deletion buttons, and in-app purchases.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under Grant No. CNS-2154483. 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 U.S. Government.

REFERENCES

[1] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-hiding behavior in Android apps: detection and characterization. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 728-739.

[2] Ben Dickson. Is iOS Secure After FaceTime Bug Fix? No One Knows, https://www.pcmag.com/commentary/366632/is-ios-secure-after-facetime-bug-fix-no-one-knows. PC Magazine, 2019.

[3] Wei F., Li Y., Roy S., Ou X., Zhou W. (2017) Deep Ground Truth Analysis of Current Android Malware. In: Polychronakis M., Meier M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2017. Lecture Notes in Computer Science, vol 10327. Springer, Cham.

[4] Mohsin Junaid, Jiang Ming, and David Kung. 2018. StateDroid: Stateful Detection of Stealthy Attacks in Android Apps via Horn-Clause Verification. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18). Association for Computing Machinery, New York, NY, USA, 198–209.

[5] Pan, Xiaorui & Wang, Xueqiang & Duan, Yue & Wang, XiaoFeng & Yin, Heng. (2017). Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. 10.14722/ndss.2017.23265.

[6] AndroZoo, https://androzoo.uni.lu/, 2016.

[7] Virustotal. https://www.virustotal.com. 2020

[8] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, ``Towards a network based framework for Android malware detection and characterization," in Proc. 15th Annu. Conf. Privacy, Secur. Trust (PST), Aug. 2017.

[9] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. 2020. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20). Association for Computing Machinery, New York, NY, USA, 535–549.

[10] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA '13). ACM, New York, NY, USA, 641-660.

[11] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of

Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 224-234.

[12] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI testing using computer vision. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1535-1544

[13] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys '14). ACM, New York, NY, USA, 204-217.

[14] The Monkey UI android testing tool. http://developer.android.com/tools/help/monkey.html.

[15] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In Proceedings of International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[16] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. In Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.

[17] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, USA, 429-440.

[18] Luc Vincent and Pierre Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. In IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, Num. 6 (1991), pages 583–598.

[19] S. Belongie, J. Malik and J. Puzicha, "Shape Matching and Object Recognition Using Shape Contexts," in IEEE Transactions on Pattern Analysis & Machine Intelligence, vol. 24, no. 4, pp. 509-522, 2002.

[20] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. 1999. Optical Character Recognition (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

[21] Information on See5/C5.0 – RuleQuest Research Data Mining Tools, 2011. [Online]. Available: http://www.rulequest.com/see5-info.html.

[22] MJ Swain DH Ballard. Color indexing. In International Journal of Computer Vision, Vol. 7, pp. 11-32, November 1991.

[23] G. Bradski and A. Kaehler, Learning OpenCV: Comput. Vision with the OpenCV Library. Sebastopol, CA, USA: O'Reilly Media, 2008.

[24] http://appium.io/docs/en/latest/#, 2024