An Approach to Reveal Unknown Malware Hiding Techniques

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

Harsha Vardhan Reddy Channu
School of Computing
Wichita State University, USA
hxchannu@shockers.wichita.edu

***Abstract*—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. In this work, we developed an approach to semi-automatically reveal unknown malware hiding techniques. First, it extracts SMH behaviors from malware descriptions by using natural language processing techniques. Second, it maps SMH behaviors to SMH-related APIs based on the analysis of API documents. Third, it performs static analysis on the malware apps that contain unknown SMH behaviors to extract the code segments related to the SMH API calls. For those verified SMH code segments, we describe the techniques used for unknown SMH behaviors based on the code segments. Our experiment tested 119 malware apps with hiding behaviors. The F-measure is 85.58%, indicating that our approach is quite effective.**

Keywords—Android, Hiding Behavior, Malware

I. Motivation

Without the awareness of users, millions of smart devices are infected by malware [2][3]. One core reason is that malware can conceal its behavior and trail in a smart device. This problem is significant because hiding techniques are commonly used by millions of malware programs to perform privileged malicious actions without being noticed. We define the smart device malware-hiding (**SMH**) mechanism as a means by which smart device malware can hide itself or its actions from being viewed (or heard!) by users. In this paper, we exclude the hiding behaviors on desktop/server OSes (e.g., Windows, Linux) and the behaviors taken to evade security mechanisms (e.g., anti-malware tools or access control), because they have been studied thoroughly and are outside the scope of this paper.

Here are three examples of SMH behaviors: (a) benign apps typically show up in the running app list, while malware may attempt to run in the background, e.g., Spyware *Candy_corn*; (b) to hide an activity, malware can make the activity transparent or destroy it before it becomes visible, e.g., Malware *DroidKungFu3*; and (c) to cover its presence, malware may mute the phone or disable the vibrate function, in order to prevent the user from hearing the sound of alarms, notifications, phone calls, or incoming short messages, e.g., Trojan *iBanking*.

According to our previous work [11], smart device malware samples exhibited 1.5 SMH behaviors per sample on average, while benign samples exhibited only 0.2 SMH behaviors. This indicates that apps with SMH behaviors are most likely malware rather than benign apps. Figure 1 shows experimental results from our previous work [11]: the number of SMHs in sample sets of 1,000 malicious apps and 1,000 benign apps. The 1,000 benign apps are a random sample extracted from 6,233 benign apps, while the 1,000 malicious apps are a random sample extracted from 3,219 malicious apps. This indicates that most SMH behaviors can be used for detecting malicious apps.

Uncovering the unknown malware hiding behavior is of great benefit to malware detection and prevention. The major challenge is the difficulty in automatically retrieving code

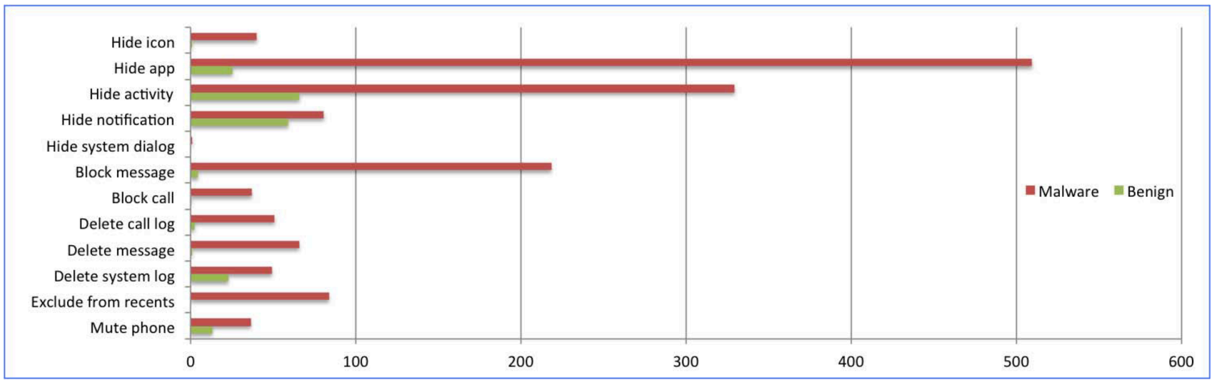

Figure 1: Number of SMH behaviors in two sample sets of 1,000 malware apps and 1,000 benign apps.

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

sections of unknown SMH mechanisms from apps, which are the basis for writing descriptions of SMH techniques or designing malware detection algorithms. In this paper, we address this challenge in order to systematically reveal unknown SMH techniques. As far as we know, *there are no existing studies on how to systematically reveal unknown SMH techniques.*

II. Proposed Approach

To search for all possible ways of malware hiding, we begin our work by analyzing malware descriptions. A large volume of these has been created by professional malware analysts from well-known anti-virus companies, such as Symantec and Norton, based on analysis of malware samples. To increase reliability, we use malware descriptions from multiple companies for each type of malware. DescribeMe [8] can automatically generate the descriptions of malicious behavior but not the SMH behavior of malware. Moreover, the majority of our work is to find code segments of SMH behavior based on malware descriptions, while DescribeMe generates descriptions from codes.

Our approach to reveal unknown SMH techniques is a novel scheme, as illustrated in Figure 2. First, we design natural language processing (NLP) algorithms to extract SMH behaviors from a large volume of malware descriptions. Second, we map SMH behaviors to SMH-related APIs based on the analysis of API documents. Third, we perform static analysis on the malware apps that contain unknown SMH behaviors to extract the code segments related to the SMH API calls. We verify the SMH code segments by reproducing and logging SMH behaviors. Eventually, for those verified SMH code segments, we describe the techniques used for unknown SMH behaviors based on the code segments.

To implement the scheme, we need to address three challenges: How to extract SMH behaviors from malware descriptions. How to map SMH behaviors to APIs. How to extract code segments of SMH behaviors from apps. We outline the corresponding solutions as follows:

Extracting SMH Behaviors. We design an algorithm that employs NLP techniques to retrieve comprehensive SMH behaviors from malware descriptions. The steps are as follows:

- Preprocessing descriptions. This removes stop words (common words such as "the," "is," "at") and apply stemming to all descriptions, in order to identify the word's root. It also removes non-text items such as numerals, HTML tags, links, and email addresses.
- Splitting descriptions into sentences for subsequent sentence structure analysis [16] [15]. Characters such as "." and ":" are treated as sentence separators.
- Analyzing each sentence for grammatical structure. A series of grammars are designed based on NLTK [20] and Stanford Parser [14] to analyze the grammatical structure of sentences. These grammars check whether a given sentence contains a string that means SMH behavior. To construct the grammars, first, we find comprehensive SMH operations (e.g., hide, delete, close, cancel, remove, conceal, block, clean up, mute, disable, deactivate, disappear, exclude) and SMH objects (e.g., app, icon, activity, message, phone call, audio, vibration, system log, notification, alert, call log), which can form SMH behavior strings. Second, we find synonyms of the operations and objects. Third, we construct grammars based on the SMH operations, SMH objects, and synonyms, which consider various grammatical structures of SMH behavior strings. Fourth, we test and improve the grammars. Last, we run the grammars on all data sets to obtain comprehensive SMH behaviors. Following is an example of a grammar to identify the SMH strings led by the verb phrase "hide":

SMHstring ::= Verbphrase Nounphrase
Verbphrase ::= "hide"
Nounphrase ::= Det Noun | Ppr Noun | Noun
Det ::= "the" | "this"
Noun ::= "app" | "icon" | "activity" | "message" | "phone call" | "notification" | "alert"
Ppr ::= "its" | "their" | "app's"

Another example grammar checks whether a string means "delete message":

ADstring ::= Verbphrase Nounphrase
Verbphrase ::= "delete" | "destroy" | "remove" | …
Nounphrase ::= Det Noun | Ppr Noun | Noun
Det ::= "the" | "this"
Noun ::= "message" | "text message" | "multimedia message"
Ppr ::= "user's" | "his" | "her" | "its"

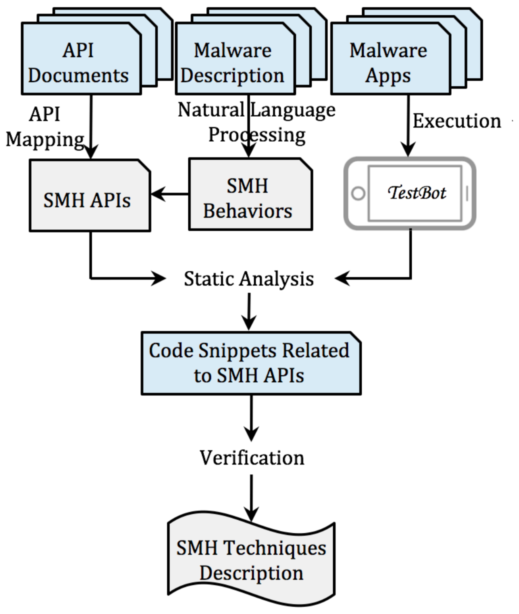

Figure 2: Uncovering SMH Techniques

Mapping API. We design an algorithm to analyze all Android API documents and map SMH behaviors to API functions. The algorithm extract classes and API names from Android API documents, split an API name into a pair that consists of a verb and a noun, and then map SMH behaviors to API names and corresponding classes. For example, the

SMH behavior "*Delete Message*" can be mapped to the API function "*delete()*" of class "*android.content.ContentResolver*"; the SMH behavior "*Mute Phone*" can be mapped to the API function "*setStreamMute()*" of class "*android.media.AudioManager.*" To reduce potential false positives, we consider not only APIs but also parameters and related APIs when determining whether a code segment contributes to an SMH behavior.

Extracting SMH Related Code Segments. The static analysis identifies SMH-related codes to help create descriptions of the SMH techniques, following five steps. First, it identifies the top functions of the SMH-related codes, such as callback functions. Second, it identifies the API functions that actually perform the SMH behaviors. Third, it finds the code that initiates the parameters of the SMH API function; considering both API functions and parameters can reduce potential false positives. Fourth, it finds the related API functions; for example, a behavior to erase a trail usually has a related API to generate the trail, and the related API functions can be used to further reduce potential false positives when identifying SMH behaviors. Last, the tool extracts the app properties, intent categories, and actions.

Static analysis is sound but also prone to false positives. Therefore, it is necessary to verify the SMH code segments obtained by reproducing SMH behaviors. We inject statements into the apps with SMH code segments for logging SMH API calls, related API calls, and parameter values. Then we run the apps on our SMH Testbot to reproduce SMH behaviors and check the logs. An SMH code segment is verified if the API calls and parameter values are correctly logged when SMH behaviors are exhibited.

III. Evaluation

In this section, we present an evaluation of our approach along two dimensions: effectiveness and efficiency. For a given malware and its description, a tool is considered effective if it can *correctly* extract code segments of the SMH behaviors mentioned in the malware description. We measure efficiency using the metrics average, median, maximum, and minimum handling time for malware samples. We begin by describing the datasets and platform used in our evaluation.

Datasets. We analyzed 119 malware samples from well-known sources, such as AndroZoo [17], Virustotal [5], and AAGM [6].

Platform. Our tools 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.

A. Effectiveness

As there is no existing oracle to determine SMH, we manually verified each static analysis results. Specifically, we reverse-engineered each app – decompiled the app (to source code) via the JADX decompiler [4]. 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. The results are shown in Table 1. Our tool has reported 209 SMH code segments in total; of these 178 were true SMH code segments, while 31 SMH code segments were over-reported (false positives) and 29 were false negatives, i.e., our tool missed those SMH code segments. This yields an F-measure of 85.58%, indicating that our tool is quite effective.

TABLE 1. EFFECTIVENESS RESULTS

True SMH code segments	**Over-reported SMH code segments (FP)**	**Under-reported SMH code segments (FN)**
178	31	29
Precision: 178 / (178+31) = 85.17%		
Recall: 178 / (178+29) = 85.99%		
F-measure: 2 * (85.17*85.99) / (85.17+85.99) = 85.58%		

Examples. Here we list three example SMH code segments found by our approach:

- Hiding icon. After installation, benign apps add their icons to the home screen. To hide itself, a malicious app removes the icon so the user does not notice the app's presence. This can be done by calling an Android API method *setComponentEnabledSetting()* to disable the icon at runtime. For example, malware *Facebook-otp* (full package name: *jgywwv.jvyjsd.sordvd*) masquerades itself as the *Facebook* app but disables its icon immediately after installation. We show the segment of the code reverse-engineered from this malware:

```
1 PackageManager pm = getPackageManager();
2 ComponentName cn = new
ComponentName("jgywwv.jvyjsd.sordvd",
"jgywwv.jvyjsd.sordvd.Activity1");
3 pm.setComponentEnabledSetting(cn,
PackageManager.COMPONENT_ENABLED_STATE
_DISABLED,
 PackageManager.DONT_KILL_APP);
```

- Hiding in the running app list. When benign apps are running, they typically show up in the running app list. In contrast, a malicious app runs as a service in the background without showing up in the list. In order to automatically start the malware as a service without the user needing to click the icon, a malicious app creates a *BroadcastReceiver* class and registers itself to receive events such as *SMS_RECEIVED* and *BOOT_COMPLETED.* After receiving one of the registered events, the malware's *BroadcastReceiver* launches the malware as a service in the background. As a result, the user cannot see the malicious app in the running app list. For example, the spyware *Candy-corn* automatically records Google voice calls in the background. As shown in the following code segment, *Candy-corn* monitors seven kinds of events and starts itself as a service (if the service is not running already).

```
public void onReceive(Context context, Intent
  intent){
String act = intent.getAction();
if ( Intent
.ACTION_BOOT_COMPLETED.equals(act)|
 Intent .SMS_RECEIVED.equals(act) |
 Intent .NEW_OUTGOING_CALL.equals(act) |
 Intent .SCREEN_OFF.equals(act) |
 Intent .PACKAGE_INSTALL.equals(act) |
 Intent .PACKAGE_ADDED.equals(act) |
 Intent .SIG_STR.equals(act)){
 if(isServiceRunning())
   return;
 Intent serviceIntent = new Intent(context,
 com.google.progress.AndroidClientService.
 class);
 serviceIntent.setAction("com.google.
 ACTION_START_CALL_RECORD");
 context.startService(serviceIntent);
 }
}
```

- <u>Excluded from the recent app list</u>. After an app has run, the system puts its activities into the recent app list. To prevent this, malware can set the flag *excludeFromRecents* in the manifest file, or by calling *ActivityManager.setExcludeFromRecents()*. An example is Trojan *Malapp*.

```
<activity
android:name="com.yangccaa.chengaa.WEYY"
    android:label="@string/notification_name"
    android:taskAffinity=".NotificationActivity"
    android:excludeFromRecents="true">
        <intent-filter>
            <action
android:name="android.intent.action.MAIN" />
            <category
android:name="android.intent.category.LAUNCHER"
/>
        </intent-filter>
</activity>
```

B Efficiency

We show the detailed efficiency results in Table 2. 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 24 MB. The "Time" grouped columns show running time statistics for the apps. The mean analysis time was 46 seconds while the median was 22 seconds, which shows that our analysis is practical. Finally, we believe that even the maximum analysis time of 3,427 seconds (i.e., 57 minutes) is acceptable for a static analysis. To conclude, with a median analysis time of 22 seconds on a median app size of 1.9MB we believe that our approach is efficient at SMH analysis.

TABLE 2. EFFICIENCY RESSULTS

Byte code size (KB)				**Time (Seconds)**			
Min	***Max***	***Average***	***Median***	***Min***	***Max***	***Average***	***Median***
32	15,021	5,321	1,995	10	3,427	46	22

IV. Related Work

There are a few SMH-related research works, but *all of them discuss how to use known SMH to construct detection tools* rather than how to reveal unknown SMH. StateDroid [10] uses features of a set of attacks and actions to detect specified stealthy behaviors in Android apps. AsDroid [18] 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 SMS, and inserting data into a sensitive database. VAHunt [9] conducts data flow analysis to determine stealthy plugin loading behaviors. We developed an SHB detector [11] to identify a number of malicious behaviors of concealing app activities, such as removing traces of suspicious actions and hiding the presence of an app.

No existing malware analysis techniques can automatically extract code sections of a given malware behavior from apps. The code sections of a malware behavior are critical when designing detection algorithms for the malware behavior. Existing malware analysis techniques can be categorized as dynamic analysis, static analysis, hybrid analysis and memory-based analysis [21][22][23][19]. However, all of these four categories of analysis techniques aim to either detect malware or produce malware behavior reports. None of them aim to automatically extract code sections of malware behaviors. Extracting code sections needs to be manually done by experts [11][18].

Existing efforts related to characterizing malware behaviors do not discuss SMH behaviors. FeatureSmith [7] automates the manual procedure of engineering features of malware behaviors that are known to the community, while we aim to uncover techniques of SMH behaviors that are mostly unknown. Zhou and Jiang [1] characterized existing Android malware from different aspects, including installation methods, activation mechanisms, and the nature of carried malicious payloads. CopperDroid [25] characterizes a malware behavior based on how it is initiated, either through Java, JNI, or native code execution. ANDRUBIS [12] discusses the trend of malware behaviors, based on the app observation dating back as far as 2010. Yuan and Lu [13] proposed to associate features from the static analysis with features from the dynamic analysis of Android apps and characterize malware using deep learning techniques. SmartDroid [24] uses a combination of static and dynamic analysis to expose the behavior of Android malware in UI-based triggers.

V. Summary

In this paper, we proposed a semi-automatic approach to extract the code segments of malware hiding behaviors from malicious apps' binary files. Our experiments shown that the approach can effective and efficiently find and report those code segments. The approach can systematically improve the understanding of how smart device malware hides its behaviors, and anti-malware researchers and developers can create new malware detection techniques based on the unveiled SMH techniques. The applicability of this research is twofold. First, the code segments of unknown SMH obtained by this research can potentially be used to develop new techniques for malware detection. Second, the approach to reveal unknown SMH techniques can potentially be applied to study other categories of malware behavior.

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] 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.

[2] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: scalable and accurate zero-day android malware detection. In Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys '12). ACM, New York, NY, USA, 281-294.

[3] 260,000 Android users infected with malware. http://www.infosecurity-magazine.com/view/16526/260000-android-users-infected-with-malware/.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flow- Droid: 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.

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

[6] 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.

[7] Ziyun Zhu and Tudor Dumitraş. 2016. FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 767–778.

[8] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). ACM, New York, USA, 518-529.

[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] 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.

[11] 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.

[12] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS -- 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS '14). IEEE Computer Society, Washington, DC, USA, 3-17.

[13] Zhenlong Yuan, Yongqiang Lu, Yibo Xue. Droiddetector: android malware characterization and detection using deep learning. TSINGHUA SCIENCE AND TECHNOLOGY, Volume 21, Number 1, February 2016, pp114-123.

[14] R. Socher, J. Bauer, C. D. Manning, and A. Y. Ng. Parsing with compositional vector grammars. In Proceedings of the ACL, 2013.

[15] J. C. Reynar and A. Ratnaparkhi. A maximum entropy approach to identifying sentence boundaries. In Proceedings of the fifth conference on Applied natural language processing, 1997.

[16] T. Kiss and J. Strunk. Unsupervised multilingual sentence boundary detection. Computational Linguistics, 32(4):485–525, 2006.

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

[18] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). Association for Computing Machinery, New York, NY, USA, 1036–1046.

[19] A. Vasudevan. 2008. MalTRAK: Tracking and eliminating unknown malware. In Proceedings of the Annual Computer Secur. Appl. Conference (ACSAC'08). 311–321.

[20] Analyzing sentence structure, Oct 2019. https://www.nltk.org/book/ch08.html.

[21] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. ACM Comput. Surv. 52, 5, Article 88 (October 2019).

[22] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44, 2, Article 6 (February 2012)

[23] Sihwail R, Omar K, Ariffin KZ. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. International Journal on Advanced Science, Engineering and Information Technology. 2018;8(4-2):1662.

[24] 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 (SPSM '12). ACM, New York, NY, USA, 93-104.

[25] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in Proc. of the Symposium on Network and Distributed System Security (NDSS), 2015.