A STRUCTURE SIMILARITY-BASED APPROACH TO MALICIOUS ANDROID APP DETECTION

Gu-Hsin Lai, Department of Information Management, Chinese Culture University, Taipei, Taiwan, R.O.C., guhsinlai@gmail.com

Yen-Hsien Lee*, Department of Management Information Systems, National Chiayi University, Chiayi, Taiwan, R.O.C., yhlee@mail.ncyu.edu.tw

Tsai-Hsin Chu, Department of E-learning Design and Management, National Chiayi University, Chiayi, Taiwan, R.O.C., thchu@mail.ncyu.edu.tw

Tsang-Hsiang Cheng, Department of Business Administration, Southern Taiwan University of Science and Technology, cts@stust.edu.tw

Abstract

The advance of computational power and storage device equipped the mobile devices to involve more and more peoples' daily works, and store voluminous organization's confidential documents as well as general user's personal data. The extensibility feature of mobile device has attracted many app developers' contributions; while it in turns becomes the attacking target of the computer hackers. The F-Secure has reported that the profit-motivated threats on mobile device have been increasing; that is, an infected mobile device might send out personal or organizations' confidential data or send SMS messages to premium rate numbers without user's consent. Generally, the Android app developer can publish their apps on either official stores (i.e., Google Play) or third-party stores or both. In the Android market, the accumulated number of applications and games has been over one million. However, due to the lack of checking and validating mechanism, attackers can also distribute their malicious apps via the online store platform quickly and easily. As a result, the needs for real-time malware detection and classification become critical for Android users and official market as the number of Android apps increases sharply. In this study, we proposed the structure similarity-based malicious app detection approach to address the need of malicious Android app detection. On the basis of source code analysis, we intend to identify the sensitive features in malicious apps; that is, the API calls and system commands that related to some malicious behaviors, to build their Class-Method-API hierarchies. A new-coming app can be detected as malicious or not by assessing the structure similarity between its hierarchy and that of each malicious app. We have collected 1,259 malwares from Android Malware Genome Project and 1,259 benign apps from Google Play market for the evaluation purpose. We intend to implement a k-fold cross-validation and adopt VirusTotal as our performance benchmark. Overall, the proposed approach is expected to effectively and efficiently detect Android malwares and is appropriate for mobile devices because the maintenance and similarity assessment of partial hierarchies cost less space and computation resources.

Keywords: Mobile Malware Detection, Malicious App Detection, Source Code Analysis, Structure Similarity

1. INTRODUCTION

Recently, the Internet-connected mobile devices like smartphone or tablet have become popular. The advance of computational power and storage device equipped the mobile devices to deal with more and more peoples' daily works, and store voluminous organization's confidential documents as well as general user's personal, private data. The feature of extensible function has attracted programmers to develop and publish their applications (apps) to meet the smartphone user's needs; while the mobile platform in turn becomes the new attacking target of the computer hackers. According to IDC's investigation, Android holds 84.3% of the total market share in mobile phones and tablet devices in Q3 2014 (IDC 2014). Juniper networks study (Juniper 2013) further states that 92% of mobile malware targets on Android platform.

Google Play is the official store where the developers of Android apps can publish their apps. The accumulated number of applications and games has been over one million in the Android Market. Due to the lack of checking and validating mechanism, attackers can also distribute their malicious apps via the online store platform quickly and easily. The investigation of F-Secure showed that the profit-motivated threats have been increasing, which typically make monetary profit by sending premium-rate SMS messages from infected devices without the users consent (F-Secure 2013). Once the user downloads and installs such malicious app, the infected device would try stealthy to send out his or her confidential data or send SMS messages to premium rate numbers. For example, FakeInstaller, a widespread mobile malware family, compromised devices would send SMS messages to premium rate numbers without the user's consent (McAfee Lab 2012). Its variant, FakeInstaller.S, used the function "Android Cloud to Device Messaging" to register the infected devices in the database and send them messages (URLs) from malware authors' Google accounts. (McAfee Lab 2012).

As a result, the needs for real-time malware detection and classification become critical to Android users and official market as the number of Android apps increases sharply. Specifically, an effective and efficient mechanism is required for official market to examine the great volume of growing apps as well as for Android users to check whether the downloading app is malicious or not in real time. Dynamic analysis and static analysis are two popular approaches to malware detection. Dynamic analysis creates a controlled environment (e.g., sandbox) and executes the suspicious app to observe its behaviors. Dynamic analysis could be effective in detecting malware; however, its efficiency is challenged on how to trigger the malicious behaviors. Furthermore, dynamic analysis is resource consuming, and running dynamic analysis on a mobile device seems impractical. On the other hand, static analysis examines the suspicious apps directly without executing them, such as anti-virus software. Anti-virus software generally examines the binary files of programs to check whether they match some pre-defined signature. Nevertheless, most malware developers may develop a series of variants to deceive or to evade the detection of anti-virus software.

Because of the nature of Android apps, their source codes could be easily available with the use of reverse code engineering. Basically, the malicious behaviors, like sending SMS messages without user's consent, shall be associated with some API calls and system commands of Android system; that is, performing API calls or system commands to do some action. With the availability of the source codes, we might be able to identify the sensitive API calls and system commands used in the suspicious app and compare them with those used in the known malwares. In this study, we accordingly proposed a source code-based analysis approach to the detection of malicious Android apps. We first apply reverse code engineering to extract the source codes of the sample apps. Then, a statistical or data mining approach will be applied to analyze the source codes to get a list of sensitive API calls and system commands regarding the mobile malwares. We determine whether a new-coming app is malicious by examining the similarity of the hierarchical structures formed by the sensitive API calls and system commands and the critical user defined methods they exist in. Traditional anti-virus software examines binary code directly to check whether it contains any pre-define string (signature) or not. However, the malware developer will try to evade detection by developing a series of variants. Our proposed approach intends to apply reverse code engineering to recreate the source code of mobile malwares so as to analyze their malicious behaviors directly rather

than meaningless strings (signatures). Overall, the proposed approach would detect malware and their variants by means of comparing their Class-Method-API hierarchical structure.

2. RELATED WORKS

In this section, we briefly review prior works related to our study, including the dynamic analysis and static analysis on detecting malicious apps. Studies that adopted dynamic analysis observed the behaviors of malwares and intended to identify their behavioral patterns. For example, Kim et al. (2008) proposed a power-aware malware detection framework to monitor, detect, and analyze energy-greedy threats. Kim's system first built the power consumption history for the collected samples, and then induced power signatures from the power consumption history. An anomaly can then be detected by the comparison between the power signatures of an app with those in the database. Chekina et al. (2012) proposed a system to detect abnormal network traffic in mobile device for malware detection. Wei et al. (2012) designed a detection mechanism which observed the behaviors of domain name resolution communication of suspicious app and applied independent component analysis (ICA) to determine the intrinsic domain name resolution behavior of Android malware. Li et al. (2014) adopted support vector machine (SVM) to build threat pattern of malicious app based on network flow data. Dynamic analysis is resource consuming; that is, it requires sufficient power and computation capability for analyzing behavior of malware. Though sandbox systems are proposed to address the performance issues of mobile apps, the anti-VM techniques are developed to evade the detection. Moreover, in contrast with traditional attack, mobile malware causes small traffics and resources; e.g., sending out confidential personal data or premium-rate SMS messages, and is not easy to detect. Therefore, dynamic analysis approach seems not suitable for mobile malware detection.

Static analysis approaches perform malware detection by analyzing the permission requests or use of API calls in app. For example, Google creates a permission-based security mechanism for Android to limit the access to specific components, behaviors and data of the mobile device. The app has to declare the corresponding permissions in a specific file as it requires the access permission to the protected features or components[1]. Some studies used permission-based approach to identify apps with abnormal permission requests. Aswini and Vinod (2014) proposed a static analysis approach to identify prominent permissions among files of Android malware. It used bi-normal separation (BNS) and mutual information (MI) feature selection techniques to select suitable features. Nevertheless, the experiment evaluation suggested an unsatisfying result. Cerbo et al. (2010) assumed that apps of the same type may request similar permissions. As a result, they adopted Apriori algorithm to discover the permission subset of the apps of the same type. An app would be determined as malware while it requests a permission differing from the subset. However, the permission information is not sufficient for detecting malicious apps because some permission requests are not defined explicitly in Android platform; for example, the "INTERNET" permission is usually required by most applications.

Apvrille1 and Strazzere (2012) present a heuristics engine which used 39 flags of different natures such as Java API calls, presence of embedded executables, code size, URLs, and etc.. Each flag is assigned a different weight according to the statistics induced from the techniques that mobile malware developers most commonly used. The engine outputs a risk score to highlight the samples that are the most likely to be malicious. However, the study reported a bad detection rate. Grace et al. (2012) developed a system, named RiskRanker, to spot zero-day Android malware by sifting through the large number of untrusty apps in the Android markets. RiskRanker used vulnerabilities in the OS kernel to detect high-risk apps and used some privilege-related API calls, like COST_MONEY permission-group, as features to detect mid-risk apps. Wu et al. (2012) proposed DroidMat, a feature-based mechanism to provide a static analysis paradigm for detecting the malwares in the Android system. Its feature-based mechanism considers the static information, including permissions, deployment of components, Intent messages passing, and API calls for characterizing behaviors of Android apps. DroidMat used 1529 permission protected APIs defined in Felt's study (Felt et al,

[1] http://developer.android.com/guide/topics/manifest/manifest-intro.html

2011). However, to match reverse source code with numerous API calls will cause a heavy loading on detection system.

Overall, this related research inspired us to consider API calls and system commands as features to detect malicious apps. However, there are over 20 thousands API calls within Android SDK, and it will be a bottleneck for malware detection. Furthermore, due to the nature of java language, the source code and the structure of Android apps could be rebuilt using the reserve code engineering technique. Thus, in this paper, we will proposed an approach that measures the structure similarity between malicious and benign apps on the basis of the sensitive API calls and system commands identified from their source codes.

3. MALICIOUS APP DETECTION APPROACH

In this study, we proposed the structure similarity-based malicious App detection approach to address the need of malicious Android app detection. It detects malicious app by measuring the similarity between an unknown app and the known malicious apps on the basis of their structures constructed using the sensitive API calls and system commands in their respective source codes. We discussed our proposed approach in the following.

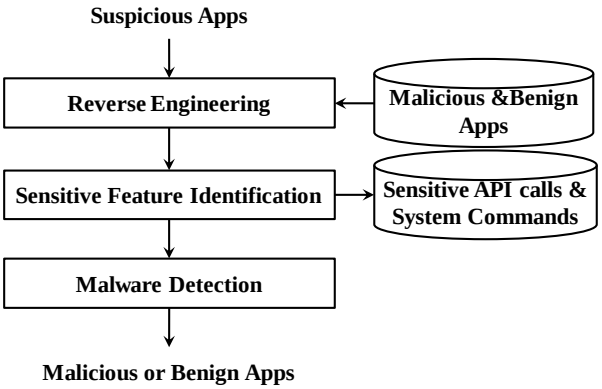

Figure 1. Overall Process of Structure Similarity-Based Malicious App Detection Approach

As shown in Figure 1, the structure similarity-based approach consists of three phases, including reverse engineering, sensitive feature identification, and malware detection. In the reverse engineering phase, all collected known malicious and benign Android apps (i.e., their APK files) will be reversed by a reverse engineering technique to retrieve their source codes. Figure 2 illustrates how the reverse code engineering works. It first unpacked all APK files using APK tool, transferred .dex file into .jar file that comprises a set of .class files using dex2jar tool, and disassemble .jar file using Jad tool to recreate a list of java source files for further analysis.

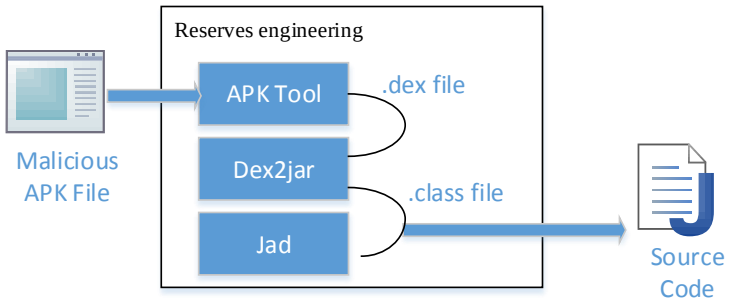

Figure 2. Process of Reverse Engineering

The task of sensitive feature identification phase is to discover the set of particular API calls and system commands used in malicious Android apps. We first develop a crawler to get all available API calls from Android developer website[2] and the native Android executable programs located in system's bin directory as system commands, and based on which to identify the API calls and system commands used in the decompiled source codes of apps. Our preliminary survey suggests a number of 21193 APIs and 258 system commands. We then analyze the difference of API calls and system commands appearing in between malicious and benign apps to determine the sensitive API calls and system commands. Information gain and mutual information are two common approaches to measure the discrimination of features in machine learning (Yang & Pedersen 1997). In this study, we will implement and analyze the performance difference between information gain and mutual information. As suggested by our preliminary analysis based on mutual information, 19 sensitive API calls, such as getSubscriberId(), getSimSerialNumber(), sendTextMessage(), getLine1Number(), and etc., are identified from our collected Android apps.

In addition to the appearance of sensitive API calls, the proposed system further considers the structure of source code rather than whole API calls. The task of the malware detection phase is to determine whether a suspicious app is malicious or not by comparing its Class-Method-API hierarchical structure with those of the known malicious apps. Instead of the complete structure constructed from the whole source codes, we adopted the partial structure that contains the sensitive API calls and system commands for similarity assessment. As illustrated in Figure 3, the source code of the app comprises method A, B, and C. We discard methods A and C and keep method B for further analysis because it calls, snedTextMessage(), a sensitive API.

```
Method A {
   context = context1;
 db = (new OpenHelper(context)).getWritableDatabase();
 insertStmt = db.compileStatement("insert into table1(was) values ('was')");
 }
```

```
Method B {
   TextView textview = new TextView(this);
  textview.setText("\u0418\u0434\u0435\u0442 \u043F\u043E\u043B\u0443\
u0447\u0435\u043D\u0438\u044F \u043F\u0435\u0440\u0441\u043E\u043D\
u0430\u043B\u044C\u043D\u043E\u0433\u043E \u043A\u043B\u044E\u0447\
u0430...");

 setContentView(textview);
SmsManager smsmanager = SmsManager.getDefault();
smsmanager.sendTextMessage("7132", null, "849321", null, null);
smsmanager.sendTextMessage("7132", null, "845784", null, null);
smsmanager.sendTextMessage("7132", null, "846996", null, null);
smsmanager.sendTextMessage("7132", null, "844858", null, null);
 datahelper.was();
 }
```

```
Method C {
     String s = "no";
 Cursor cursor = db.query("table1", new String[] {"was"}, null, null, null, null,
null);
   if(cursor.moveToFirst())   do
   s = cursor.getString(0);
while(cursor.moveToNext());
……..
}
```

Figure 3. An example of Source Code of Android App

The rationale behind the use of partial structure of source code is that most malware developer may repack the same malware on different apps or redesign the interface of apps. That is, the malwares of identical behaviors might have different source codes. As shown in Figure 4, the APK B is a variant of APK A. Although APKs A and B have different hierarchical structure, they behave similarly because they call identical sensitive API; i.e., sendTextMessage() and getLine1Number(). Therefore, when we remove the methods that do not contain sensitive API calls from hierarchical structure, APKs A and B may share similar partial hierarchical structure as shown in Figure 5. Because similar malicious code could be injected into different benign apps, analysis of partial structure of suspicious app could mitigate the misclassification of such malicious apps. The partial hierarchical structure

[2] http://developer.android.com

could also help identify malware families and use to describe their behavior threat patterns for detecting malicious apps with fewer computing resource.

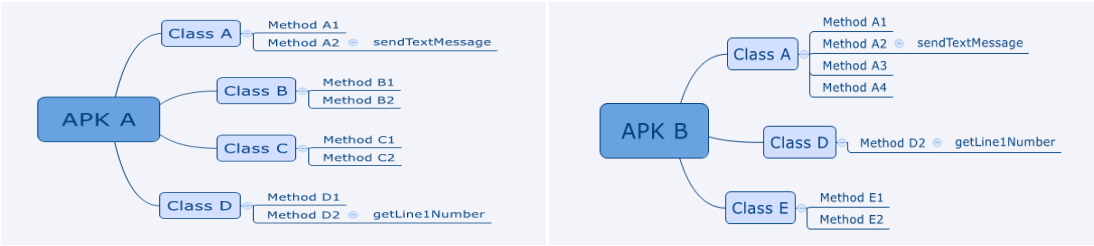

Figure 4. Example of Compelte Hierarchical Structure of Variant of Android App

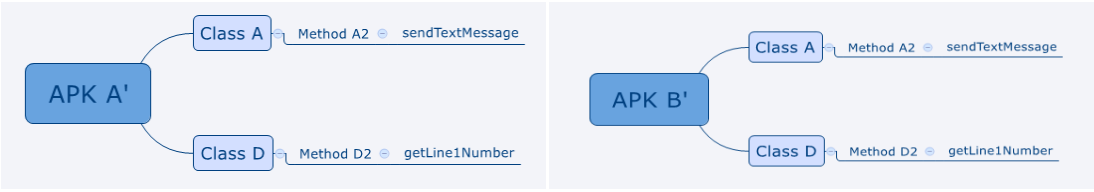

Figure 5. Example of Partial Hierarchical Structure of Android App with Sensitive API Calls

On completing the construction of partial hierarchies for the suspicious apps and for the known malicious apps, we can assess their similarity by applying the hierarchy similarity measure like the structure similarity metric proposed in prior study (Wei et al. 2009). Finally, we determine a suspicious app as malicious if the highest similarity it achieves over a pre-specific threshold.

4. EXPECTED RESULTS

We are conducting an empirical evaluation to assess the effectiveness of our proposed structure similarity-based malicious App detection approach. For the purpose of evaluation, we have collected 1,259 pieces of malwares belonging to 49 mobile malware families from Android Malware Genome Project[3] and 1,259 benign apps that are the top popular free apps in each category and published over three months in Google Play market. We intend to implement a *k*-fold cross-validation to reduce the experimental bias and adopt VirusTotal[4] as our performance benchmark. VirusTotal is a subsidiary of Google providing free online service that analyzes files and URLs enabling the identification of viruses, worms, trojans and other kinds of malicious content detected by antivirus engines and website scanners. Virustotal also provides Android app for users to scan suspicious Android apps with their own Android mobile device. As a result, Virustotal seems to be a good performance benchmark in this study.

Overall, our proposed malicious app detection approach that assesses the similarity between suspicious apps and known malicious apps on the basis of the partial hierarchies formed by the sensitive API calls and system commands in their own source codes. Based on the representative, sensitive API calls and system command discovered from the malicious and benign Android apps, the proposed approach is expected to effectively and efficiently detect Android malwares. Furthermore, our proposed approach could be appropriate for mobile devices because the maintenance and similarity assessment of partial hierarchies cost less space and computation resources.

[3] http://www.malgenomeproject.org
[4] https://www.virustotal.com/

References

Apvrille, A., and Strazzere, T. (2012). Reducing the window of opportunity for Android malware Gotta catch 'em all. Journal in Computer Virology, 8(1-2), 61-71.

Aswini, A.M. and Vinod, P. (2014). Droid permission miner: Mining prominent permissions for Android malware analysis. In Proceedings of the Fifth International Conference on the Applications of Digital Information and Web Technologies, pp. 81-86

Cerbo, F.D., Girardello, A., Michahelles, F., Voronkova, S. (2010). Detection of malicious applications on Android OS. In Proceedings of the 4th International Conference on Computational Forensics, pp. 138-149.

Chekina, L., Mimran, D., Rokach, L., Elovici, Y., Shapira, B. (2012). Detection of deviations in mobile applications network behavior, CoRR abs/1208.0564.

Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D. (2011). Android permissions demystified. In Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627-638.

F-Secure (2013). Mobile Threat Report. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf

Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X. (2012). RiskRanker: scalable and accurate zero-day android malware detection. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281-294.

IDC (2014). Smartphone OS Market Share. Q4 2014, http://www.idc.com/prodserv/smartphone-os-market-share.jsp

Juniper (2013). Juniper Networks Third Annual Mobile Threats Report. http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf

Kim, H., Smith, J., Shin, K. G. (2008). Detecting energy-greedy anomalies and mobile malware variants. In Proceedings of the 6th International Conference on Mobile Systems, pp. 239-252.

Li, J., Zhai, L., Zhang, X., & Quan, D. (2014). Research of android malware detection based on network traffic monitoring. In Proceedings of 2014 IEEE 9th Conference on Industrial Electronics and Applications, pp. 1739-1744.

McAfee Lab (2012). FakeInstaller' Leads the Attack on Android Phones. https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones

Wei, C.P., Hu, P., and Lee, Y.H. (2009). Preserving user preferences in automated document-category management: An evolution-based approach. Journal of Management Information Systems, 25(4), 109–143.

Wei, T.E., Mao, C.H., Jeng, A.B., Lee, H.M., Wang, H.T., and Wu, D.J. (2012). Android malware detection via a latent network behavior analysis. In Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 1251-1258.

Wu, D.J., Mao, C.H., Wei, T.E, Lee, H.M., and Wu, K.P. (2012). DroidMat: Android malware detection through manifest and API calls tracing. In Proceedings of the 7th Asia Joint Conference on Information Security, pp. 62-69.

Yang, Y. and Pederson, J. (1997). A comparative study on feature selection in text categorization. In Proceedings of the Fourteenth International Conference on Machine Learning, pp. 412-420.