Three-Phase Detection and Classification for Android Malware Based on Common Behaviors

Ying-Dar Lin, Chun-Ying Huang, Yu-Ni Chang, and Yuan-Cheng Lai

Original scientific paper

***Abstract*—Android is one of the most popular operating systems used in mobile devices. Its popularity also renders it a common target for attackers. We propose an efficient and accurate three-phase behavior-based approach for detecting and classifying malicious Android applications. In the proposed approach, the first two phases detect a malicious application and the final phase classifies the detected malware. The first phase quickly filters out benign applications based on requested permissions and the remaining samples are passed to the slower second phase, which detects malicious applications based on system call sequences. The final phase classifies malware into known or unknown types based on behavioral or permission similarities. Our contributions are three-fold: First, we propose a self-contained approach for Android malware identification and classification. Second, we show that permission requests from an Application are beneficial to benign application filtering. Third, we show that system call sequences generated from an application running inside a virtual machine can be used for malware detection. The experiment results indicate that the multi-phase approach is more accurate than the single-phase approach. The proposed approach registered true positive and false positive rates of 97% and 3%, respectively. In addition, more than 98% of the samples were correctly classified into known or unknown types of malware based on permission similarities. We believe that our findings shed some lights on future development of malware detection and classification.**

***Index Terms*—Android, behavioral analysis, permissions, malware, system call sequences**

I. Introduction

In the past, mobile devices were used solely for making phone calls and sending and receiving short messages. However, the rapid development of computing technology and wireless bandwidth has turned mobile devices into universal devices in digital life. Activities such as watching videos, playing games, checking e-mails, and online shopping can now be performed anytime and anywhere with an Internet-connected mobile device. Therefore, many users have migrated from PCs to mobile devices and the number of mobile devices has thus grown exponentially.

Because of its openness, Android is one of the most popular operating systems (OSs) adopted by modern mobile devices [1]. Statistics collected in 2014 indicated that there are more than one billion devices that run the Android OS. The widespread deployment of Android also renders it an attractive target for attackers; therefore, problems associated with mobile security are becoming more critical [2]. In addition to the behavior of PC-based malware, mobile malware also attempts to steal sensitive data and conducts financially motivated attacks. Mobile malware can read the location of a user by using built-in GPS receivers, intercept short messages, or steal contact lists. Furthermore, such malware can send short messages, make phone calls, or relay phone calls to gain economic benefits. The widespread deployment of the Android OS renders it an attractive OS to attackers.

Approaches for detecting malicious applications are commonly classified into two classes: static- and dynamic-based approaches. In general, a static-based approach is faster than a dynamic-based approach; however, a dynamic-based approach can obtain more detailed information and thus creates the possibility of conducting a further in-depth analysis of an application that is being inspected. In our previous study [3], we reported that a dynamic-based approach can generate detecting patterns from a group of known malicious Android applications. However, the approach was also marred by slow performance because of the characteristics of dynamic analysis. We attempted to speed up the training and detection process of our previous study, however, this process is a trade-off between detection speed and detection risk. This implies that the detection could be evaded in some rare cases if a malicious application can decompose the malicious software implementation into undetectable fragments. Therefore, we considered other possible approaches to eliminate the possibility of being evaded and improve the overall detection speed.

In this paper, we propose a hybrid approach that detects malicious Android applications based on both static features (the requested permissions) and dynamic features (the system call sequences). Combining these two features enabled the proposed approach to detect unknown malware efficiently. The proposed detector operates in two phases. In the faster first phase, permissions are investigated to filter out benign applications quickly. In the slower second phase, a malicious application is detected from the remaining applications based on system call sequences. Furthermore, to determine whether an identified malware is a known or an unknown malware type, behavioral vectors are established from trained malware samples and determine unknown malware types based on the similarity between an inspected malware and behavior vectors. Inspired by our previous work focusing on native Windows binaries [4], we attempt to perform automated malicious software classification by using a multi-phase approach. Compared

Manuscript received November 30, 2015; revised September 30, 2016.
This work was supported in part by Minister of Science and Technology.
Ying-Dar Lin, Chun-Ying Huang, and Yu-Ni Chang are with the Department of Computer Science, National Chiao Tung University. Yuan-Cheng Lai is with the Department of Information Management, National Taiwan University of Science and Technology.

to handling native Windows binaries, there are two major challenges for analyzing Android binaries. First, there is not a good classifier that is able to perform initial detection of malicious behavior for Android applications in a reasonable short time. Second, Android binaries are launched in a virtual machine based runtime environment, and the system calls could be triggered by the virtual machine itself or by the application. Due to the aforementioned challenges, we have to carefully design and implement our proposed approach to adapt the differences on the Android platform.

The rest of this paper is organized as follows. In Section II, a brief survey of related studies is presented. Section III presents the precise problem statement and the details of the proposed mechanism, including the processing of permissions and system call sequences. The experimental results are presented in Section IV. Finally, the concluding remarks are presented in Section V.

II. Related Work

Numerous approaches are available for analyzing malicious malware on Android. In addition to antivirus software and app inspection services such as Google Bouncer [5], we classified the approaches proposed in previous studies into sandboxes, static-based, and dynamic-based approaches. A sandbox monitors the activities of Android applications by running an application inside a constrained environment. Additional events can be sent to a running application for triggering more application behavior. Anubis is an online dynamic analysis tool originally designed for inspecting malware running on personal computers. The core component was developed by Bayer et al. [6]. In 2012, Anubis included a sandbox environment for inspecting Android applications (codename: Andrubis). In addition to dynamic analysis in sandboxes, Andrubis performs static analysis, yielding information such as an application's activities, services, required external libraries, and actually required permission. A detailed introduction to the design of Andrubis can be found in [7]. Huang et al. [8] proposed android behavior monitor (ABM), which integrates open source components and is built upon standard Android emulator. In addition to its open design, ABM adopts several strategies to improve code coverage including emulation of random user inputs, sending short messages, and making phone calls. Yan and Yin [9] proposed DroidScope to analyze Android application behavior. DroidScope is built on Quick Emulator (QEMU) and can reconstruct the OS- and Java-level semantic views completely from the outside. In addition, numerous tools, including an API tracer, native instruction tracer, Dalvik instruction tracer, and taint tracker, have been developed to conduct further analysis. Tam et al. [10] proposed CopperDroid, another dynamic analysis tool built on QEMU that has designs and implementations similar to DroidScope. CopperDroid monitors low-level system calls and can thus monitor malware behavior, regardless of whether such behavior is initiated from Java, Java native interface (JNI), or native code execution.

A static-based approach detects a malicious application by inspecting only the information stored in an application installation package file such as binary signatures and requested permissions. By contrast, a dynamic-based approach detects a malicious application by using additional run-time information such as accessed system resources and invoked system calls. Kirin [11] uses permission security rules to mitigate malware by using voice, location, or short messages; a set of security rules is used to determine whether an application requests specific combinations of permissions. PUMA [12] adopts machine-learning approaches, including simple logistic, naïve Bayes, J48, and random tree approaches to classify applications into benign or malicious applications based on permissions. These two approaches are simple and efficient because they analyze only the manifest file of an application. However, a malicious application can easily evade the detection. Numerous static-based approaches analyze the use of Android permissions. Statistics provided by the Stowaway project [13] indicated that one-third out of 940 applications were provided with over-privileged permissions. Johnson et al. [14] also reported that most developers over-requested permissions that could cause security threats. Zhou et al. [15] obtained the permissions and behaviors by manually analyzing 10 malware families. They used the permissions to filter out benign applications quickly and detected the remaining applications through behavioral footprint matching. However, their approach is not scalable because the approach cannot be automated.

A number of dynamic-based approaches are also available. AAsandbox [16] observes suspicious applications by using system call counts. Crowdroid [17] monitors system calls invoked by an application and used a clustering algorithm to determine whether the application is benign or malicious. However, this approach must collect several user experiences for the same application, otherwise it could return several false positives. The approach detects only anomalous behaviors of analyzed applications. Isohara et al. [18] defined three categories of threats and there is information leakage, jailbreaking, and destructive application detection. They generated signatures by applying a set of regular expression rules to the name of system calls or file paths. A malicious activity in these three categories was then detected by matching the signatures. However, their system cannot detect malicious activities except the three threat categories. Lin et al. [3] extracted longest common substrings (LCS) of system calls for similar malicious applications and used probabilities derived from the Bayes model to discriminate malicious behaviors from regular behaviors. They then detected repackaged malware with the obtained LCS. Although the proposed layered multithread comparison approach demonstrated a favorable efficiency, it could be evaded if malware attempts to split system calls into distinct threads. System call sequences can be combined from distinct threads; however, with the approach, the false positive rates could also be increased.

III. Three-phase Behavioral Detection and Classification

We propose three-phase behavioral detection and classification for handling Android malware. Unlike our previous work [4], which heavily depends on an external malware

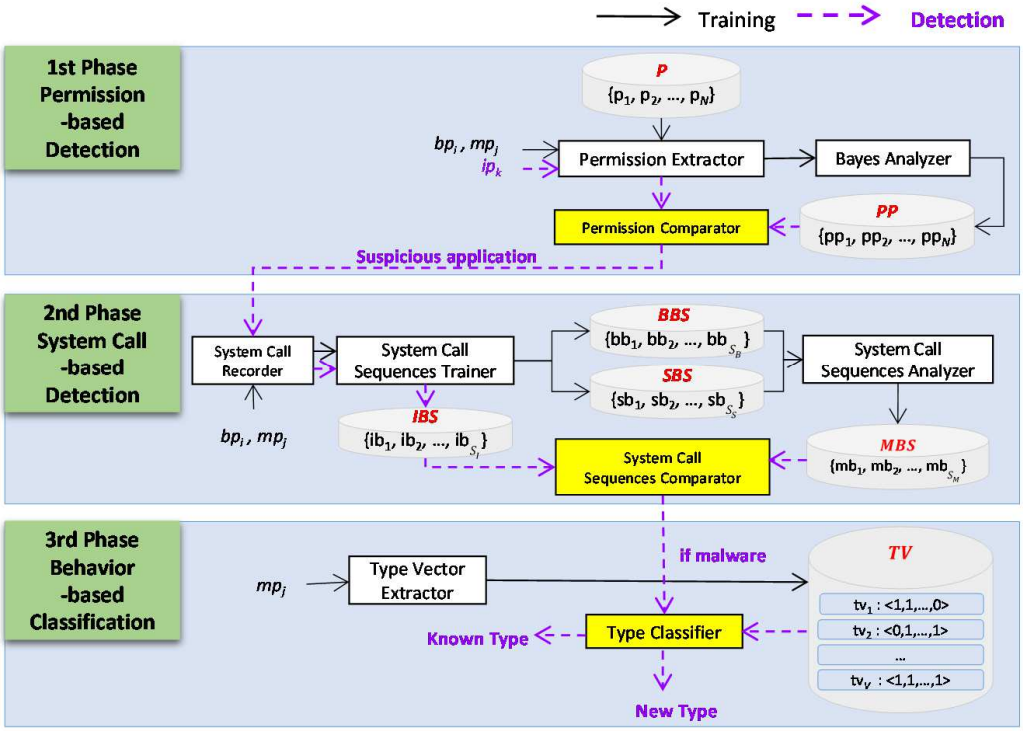

Fig. 1. Overview of the proposed approach.

behavior extractor, the proposed approach is self-contained and does not depend on other classifiers. Given a set of benign programs (BP), a set of malicious programs (MP), and a set of programs that must be inspected (IP), the proposed approach attempts to detect malicious programs from the IP and also classifies detected malware into either a known or an unknown malware type. Figure 1 shows an overview of the three phases: the permission-based detection (PBD) phase, system-call-based detection (SBD) phase, and behavior-based classification (BBC) phase. Each phase comprises a training process and detection process. The system is first trained with BP and MP and then used to detect malware from IP. The details of the proposed two detection phases and one classification phase are introduced in Sections III-A, III-B, and III-C, respectively. Implementation issues are discussed in Section III-D. Table I shows the notations used in this paper.

A. Permission-Based Detection Phase

The PBD phase comprises three components: the permission extractor, Bayes analyzer, and permission comparator. In the training phase, the permission extractor retrieves built-in permissions from each inspected application from the BP and MP. For all the trained programs and their requested permissions, the Bayes analyzer was used to compute the probability of a program being malicious for each permission. The probabilities were evaluated as follows:

$$P(M|p_l) = \frac{P(p_l|M) \cdot P(M)}{P(p_l|M) \cdot P(M) + P(p_l|B) \cdot P(B)}, \quad (1)$$

where p_l represents one of the 139 built-in permissions that must be evaluated, $P(B)$ denotes the ratio of BP, and $P(M)$ denotes the ratio of MP. The terms $P(p_l|B)$ and $P(p_l|M)$ represent the probability that p_l is requested by BP and MP, respectively. The probability $P(M|p_l)$, which indicates the probability of an inspected application being malicious based on the condition the application requested permission p_l is finally obtained. The permission probabilities for all of the 139 built-in permissions were obtained using Equation 1 and stored in a permission probability (PP) vector for future use.

Given an inspected program ip_k from IP in the detection phase, the permission extractor retrieves requested built-in permissions from the program. The permission comparator computes the product of permission probabilities by using the PP vector obtained in the training phase and then filters out the program if the product is lower than a predefined threshold T_{perm}. Otherwise, the application is considered a suspicious program and is passed to the next phase for further inspection.

B. System-Call-Based Detection Phase

The SBD phase comprises four components: system call recorder, system call sequence tokenizer, system call sequence analyzer, and system call sequence comparator (Figure 2). In the training phase, the system call recorder retrieves the system calls issued from programs in BP and MP. All system calls are collected by running a specific program in an Android emulator. In addition to the program launch, several system events including rebooting, receiving short messages, and receiving phone calls are sent to the program. System calls issued from the program are collected for a period of time. The traces of system calls for programs in BP and MP are then passed to the system call sequence tokenizer.

TABLE I
NOTATIONS USED IN THIS ARTICLE.

Notation	Description
BP	Set of benign programs bp_i, $i = 1 \ldots \lvert BP \rvert$
MP	Set of malicious programs mp_j, $j = 1 \ldots \lvert MP \rvert$
IP	Set of inspected programs ip_k, $k = 1 \ldots \lvert IP \rvert$
BBS	Set of benign behavior sequences bb_e, $e = 1 \ldots \lvert BBS \rvert$
SBS	Set of suspicious behavior sequences sb_f, $f = 1 \ldots \lvert SBS \rvert$
MBS	Set of malicious behavior sequences mb_g, $g = 1 \ldots \lvert MBS \rvert$
IBS	Set of inspected behavior sequences ib_h, $h = 1 \ldots \lvert IBS \rvert$
P	Set of permissions p_l, $l = 1 \ldots \lvert P \rvert$
PP	Set of permission probabilities pp_m, $m = 1 \ldots \lvert PP \rvert$
TV	Set of type vectors tv_v, $v = 1 \ldots \lvert TV \rvert$

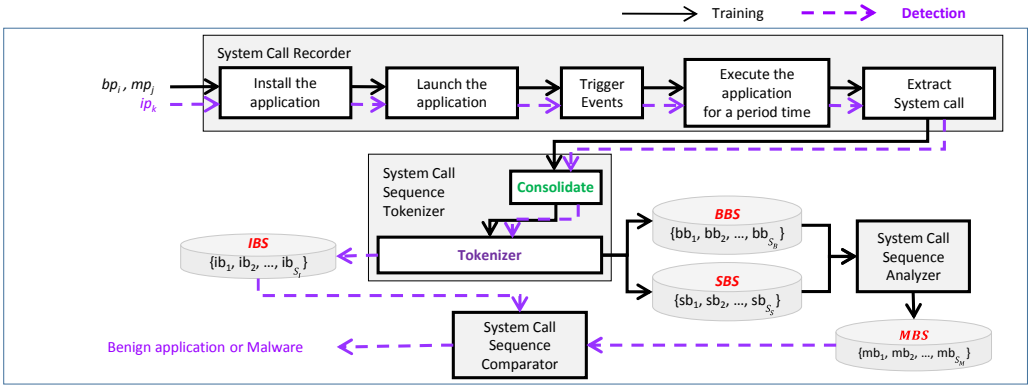

Fig. 2. The working flow of the system call-based detection phase.

The system call sequence tokenizer first consolidates successive system calls into a single call. The process is performed because a system call could be issued in loops. For example, a system call sequence of "`open`, `read`, `read`, `read`, `close`" would become "`open`, `read`, `close`." The consolidated system call sequences are then inserted into either a benign behavior sequence set (BBS) or a suspicious behavior sequence set (SBS) depending on whether the system call sequences are collected from BP or MP, respectively. The BBS and SBS are then used as inputs in the system call sequence analyzer.

The system call sequence analyzer generates a malicious behavior sequence set (MBS) based on the input BBS and SBS. This study used two types of MBS; the first is based on the N-gram algorithm and the second is based on the longest common subsequence (LCS) algorithm. The MBS for these two types are generated based on two assumptions. For the N-gram based MBS, we assumed that sequences retrieved from a malicious program would also contain benign behaviors. Therefore, the sequence of malicious behaviors can be obtained by removing sequences of benign behaviors. We transformed system call sequences into N-grams and then obtained the MBS by removing the BBS from the SBS. For the LCS-based MBS, we assumed that malicious programs demonstrated similar behaviors. Therefore, the same malicious sequences can be observed from different malicious programs. The MBS can be obtained by deriving the LCS from two MPs. The resulting MBS was then used for detection in the second phase.

In the detection phase, given an IP, the system call recorder operates as usual and the system call sequence tokenizer also outputs a processed behavior sequence called the inspected behavior sequence (IBS). The IBS is fed to the system call sequence comparator and then compared with the MBS obtained in the training phase. With the N-gram-based MBS, the IP is identified as malicious if an equivalent N-gram is discovered in the IBS. Similarly, with LCS-based MBS, the inspected program is identified as malicious if a subsequence is equivalent to one sequence in the MBS.

C. Behavior-Based Classification Phase

In the BBC phase, if a malicious application is detected in the previous phase, the malware is further classified as a known type or an unknown malware type. The BBC phase comprises a training process and a classification process. In the training process, a bit vector is used to denote the behavior of a malicious application. Assume a total of k different MBSs are observed in all training samples, each trained malicious sample would have a bit vector of k bits, and the bits for the corresponding malicious sequences observed in the sample are labeled as one. The bit vectors for all the training samples are then used to determine whether an IP is a known or an unknown type of malware. In this study, the bit vector was denoted as a type vector (tv). The number of tvs is equal to or lower than the number of trained malicious samples less if there are equivalent tvs.

In the classification process, the tv is retrieved from a detected malicious application. The retrieved tv is then compared with all the trained tvs by using a cosine similarity

measure [19]. The cosine similarity for two tvs tv_1 and tv_2 can be calculated as follows:

$$\frac{tv_1 \cdot tv_2}{||tv_1|| \times ||tv_2||}. \quad (2)$$

The detected malicious application is then classified into a similar class as the tv that has a higher cosine similarity than a predefined threshold T_{sim}. If there is no tv with a cosine similarity greater than T_{sim}, the detected malicious application is classified as an unknown type of malware.

D. Additional Implementation Note

We took advantage of several existing tools to simplify implementing the proposed approach. The tools were used to retrieve permissions and system call sequences of Android applications automatically.

1) Permission Analyzer: Because an application package (APK) file is basically a ZIP archive file with the `apk` file extension, we decompressed an application to retrieve its permissions by using the apktool [20]. The apktool provides assets, resources, source codes of an application (in assembly language), and the manifest file. We retrieved permissions by parsing only the manifest file because a developer must declare requested permissions in this file.

2) System Call Recorder: To capture the system call sequences of an application, the system image file `ramdisk.img` was modified and `strace` was installed in the emulator. First, we decompressed the default `ramdisk.img`, installed the `strace` tool into the image, and modified the `init.rc` file to launch the `strace` tool. The `strace` tool is located in the `/data` directory. The exact command we inserted into the `init.rc` file is "`/data/strace -F -ff -tt -o /data/tracefile/zygote`". With the presented modifications, `strace` was launched to record system calls immediately after booting up the emulator. The output of the `strace` tool was placed in `/data/tracefile/zygote` file.

IV. Evaluation

To evaluate the effectiveness of the proposed approach, we conducted experiments with various types of repackaged applications. The environment of the experiment and number of trained and inspected applications are described in Section IV-A. Various aspects of the performance of the proposed approaches are discussed in the remaining subsections.

A. Evaluation Environment

In this section, the training and detection processes of the experimental environment are discussed; furthermore, we also introduce the samples used in the experiments. Figure 3 shows the detailed procedures for the experimental environment. The requested permissions were parsed from the *AndroidManifest.xml* file contained in each APK file. All of the components and the emulator were operated on an Intel Core i3 3.1 GHz machine running the Ubuntu Linux OS. The system call recorder currently launches each application for 3 min [3]. In addition to the components, databases are used to store permission probabilities, system call sequences, malicious behavior sets, and tvs.

We prepared 1198 sample applications, comprising 933 benign applications and 265 malware applications, to conduct the experiments. The sample applications were divided into two sets (i.e., a training set and a detection set). We used 863 applications (700 benign and 163 malicious) for training and 335 applications (233 benign and 102 malicious) for detection. The benign applications were obtained from third-party markets and malware were collected by Zhou et al.[21]. We also used several antivirus tools to scan all the benign applications to ensure that they were virus-free.

B. Permission-Based Detector

First, we evaluated the PBD. We calculated the malicious probabilities for the 139 built-in permissions by using Equation 1. The permission probabilities of an application were multiplied together and the product is then compared against a predefined threshold T_{perm}. Figure 4 shows the accuracy of various thresholds. A higher threshold filters out more benign and malicious applications than a lower threshold does. Because the objective of this phase was to filter out benign application and obtain as many malicious applications as possible, we used a low threshold value to avoid filtering out excessive malicious applications. We used a threshold T_{perm} of 0.1 for the permission-based detector in the remaining experiments. Although the PBD has a relatively higher false positive rate[1], it can filter out more than 75% of the benign applications, thus reducing time costs considerably in the subsequent phase. The PBD registered a false negative rate of 2% and false positive rate of 24%. The average time required for inspecting an application was 2.57 s, including the unpack and permission-retrieval time.

C. System Call Based Detector

We then evaluated SBD, which was operated by the N-gram and LCS algorithm. When using the N-gram-based SBD, the value of N must be selected appropriately. The value of N in the N-gram means the unit length of system call sequences retrieved from full system call traces and this value affects the overall detection performance. A low value of N filters out substantial system call sequences, thus increasing false negatives. By contrast, a high value of N confuses benign sequences with malicious sequences, thus increasing false positives. We use various values of N that ranged from 2 to 150 and used an N value of 15 for the rest of our experiments. The performance of the SBD is summarized as follows. The N-gram-based SBD registered a false negative rate of 0% and false positive rate of 35%. In contrast to the N-gram-based SBD, the LCS-based SBD registered a false negative rate of 3% and false positive rate of 14%. The average time required to inspect an application was 600 s for both detectors. Although each application was launched for only 3 mins, several pre-processing operations including creating a

[1]A false positive means that a benign application is detected as a malicious application.

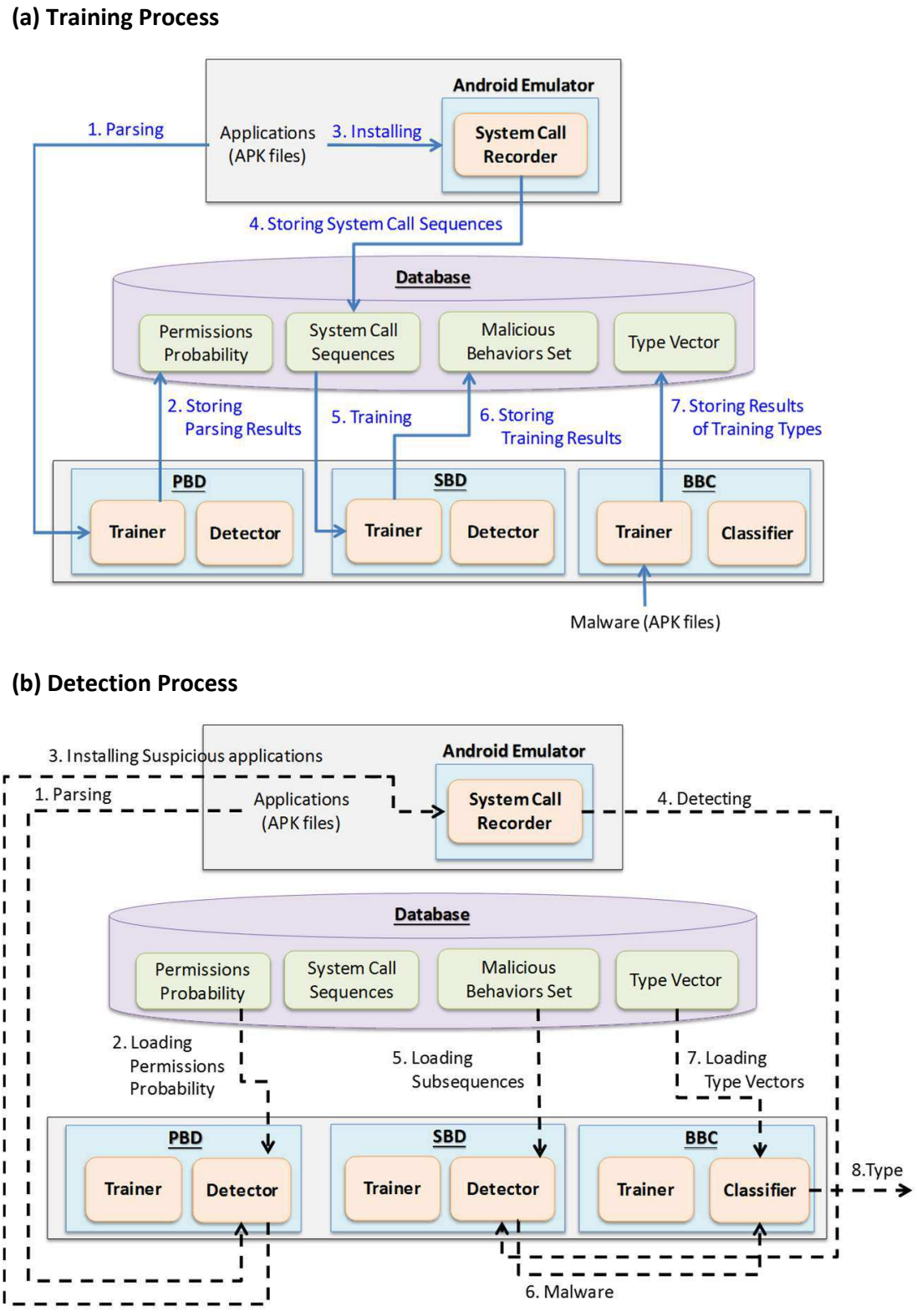

Fig. 3. The detailed procedures to train and detect malicious applications.

Fig. 4. Performance of permission-based detector using various threshold.

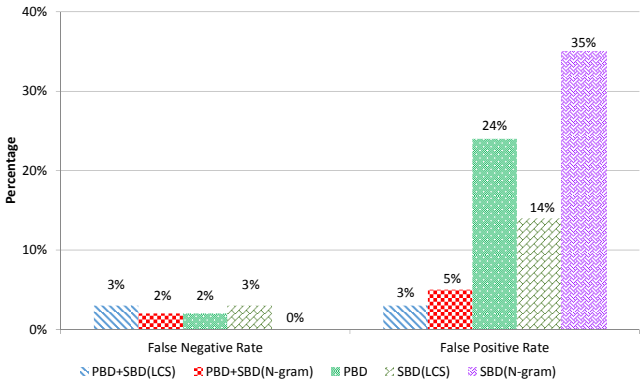

Fig. 5. Comparison of detection accuracies for one-phase and two-phase detectors.

clean evaluation environment, installing the application, and rebooting also consumed considerable time.

D. Effectiveness of the Two-Phase Detector

The experimental results indicated that the two-phase detectors demonstrated equivalent performance in the overall detection accuracies, regardless of the phase position of the PBD or SBD. Therefore, we placed the PBD and SBD in the first phase and second phase, respectively, because of their processing efficiency. In addition to shorter processing time, the PBD quickly filtered out more than 75% of benign applications and registered a relatively low false negative rate (approximate 2%).

This section further presented the evaluated performance of the combined two-phase detector. Figure 5 shows the performance of one-phase detectors and the combined two-phase detectors. For the one-phase detectors, the PBD and SBD produced poor performance in detecting malicious applications. We examined the cause of the false negatives and false positives and we discovered that for the PBD, some malicious applications request only a few noncritical permissions. Therefore, those applications cannot be detected by the PBD. The false negatives reported for the PBD were safe because a malicious application that does not possess appropriate permissions cannot cause damage. Regarding the false positives, the difficulty in detecting malicious applications based on permissions increased because it is challenging for Android developers to *declare a minimum set of required permissions* [13]. For the SBD, we analyzed the undetected malicious applications and discovered that the false negatives were caused by *untouched malicious parts*. Because the system events sent by the SBD are limited, if malicious parts are not triggered, the corresponding malicious sequences cannot be captured. The false positives were caused by two major factors. First, because most Android malware applications are repackaged applications, benign and malicious sequences are always mixed. Second, Android applications are launched *in its own virtual machine*. Therefore, it is impossible to distinguish system call sequences generated by the virtual machine or inspected application. Nevertheless, when combined, the PBD and SBD complement each other and obtain a more favorable performance compared with the single-phase detectors.

E. Behavior Based Classifier

This section presents the evaluation of the performance of the BBC. Based on the type of classification [21], we used 22 types of malware and divided them into two sets. One set comprised known types of malware and the other set comprised unknown malware types. The behavior was represented as a tv constructed from various sources, including LCS-based system call sequences, permissions, or mixed. We generated tvs from one half of the malicious samples belonging to the set that comprises known types and then classified the remaining malicious samples into either known or unknown malware types. The tvs used to classify malicious applications were constructed from the LCS-based system call sequences, permissions, or mixed. The BBC works only for applications that have been detected by the two-phase detector.

We first demonstrated that tvs can efficiently classify malware types. A detected malicious application was classified into the appropriate class by evaluating cosine similarities. A malicious application was classified into the type that demonstrated maximum cosine similarities. We used two strategies to classify malware types: the greedy strategy and regular strategy. In the greedy strategy, no threshold was used to filter out a low value of cosine similarity. A malicious application was always classified into one type if the value of its cosine similarity value was not zero. By contrast, in regular strategy, low values of cosine similarity were filtered out by using a threshold. If a similarity value was less than a predefined threshold, the malicious application was classified as an unknown malware type.

For the greedy strategy, the correctly classified rates for the tvs constructed from LCS-based system call sequences, permissions, and mixed were 93%, 99%, and 96%, respectively. The correctly classified rate is applicable only for the set of known malware types because malware from the set of unknown types were always classified incorrectly. The regular strategy must be used instead of the greedy strategy to classify unknown malware types. We used different cosine similarity thresholds for tvs constructed from different sources. The optimal thresholds we obtained from the system-call-sequence-based, permission-based, and mixed tv were 0.5, 0.8, and 0.65, respectively. Table II shows the classification results. Based on the results of the greedy strategy and optimized threshold, we

TABLE II
CLASSIFICATION RESULTS FOR KNOWN AND UNKNOWN MALWARE TYPES.

Vector	Malware Type	Classified as ...	
System call sequence	Known	Correct Type:	83%
		Incorrect Type or Unknown:	17%
	Unknown	Unknown:	81%
		Incorrect Type:	19%
Permission	Known	Correct Type:	98%
		Incorrect Type or Unknown:	2%
	Unknown	Unknown:	98%
		Incorrect Type:	2%
Mixed	Known	Correct Type:	93%
		Incorrect Type or Unknown:	7%
	Unknown	Unknown:	99%
		Incorrect Type:	1%

concluded that the permission-based tv demonstrated optimal performance in classifying malware types.

Finally, it is impossible to predict the type of unknown malware in real applications when detecting unknown malware types. Therefore, for the correctly classified results of unknown malware types, the 11 unknown types of malware were treated as one large group without detecting the number of types in this group.

V. CONCLUSION

We propose a three-phase behavior-based approach for detecting and classifying Android malware. The proposed approach achieved a high detection performance and accuracy. In the proposed approach, the first two phases detect malicious applications and the final phase classifies a detected malware. We detected and classified malicious applications from two aspects (i.e., permissions and system call sequences). The experimental results indicated that the proposed approach achieved optimal performance with a true positive rate of more than 97% and false positive rate of less than 3%. For classifying the malware type, the proposed approach correctly classified more than 98% of the detected applications into known and unknown malware types. Although permission or system call sequences alone are not efficient detectors, the results indicated that the two features complement each other. We also concluded that permission vectors can efficiently classify detected malicious applications into the appropriate class of malware types. The difficulty in detecting malicious applications is increasing. We believe that an appropriate solution for designing and implementing effective approaches may involve hybrid features and multiphase designs.

REFERENCES

[1] M. Butler, "Android: Changing the mobile landscape," *IEEE Pervasive Computing*, vol. 10, no. 1, pp. 4–7, 2011.

[2] D. Dagon, T. Martin, and T. Starner, "Mobile phones as computing devices: the viruses are coming!" *IEEE Pervasive Computing*, vol. 3, no. 4, pp. 11–15, 2004.

[3] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Computers & Security*, vol. 39, pp. 340–350, November 2013.

[4] Y.-D. Lin, Y.-C. Lai, C.-N. Lu, P.-K. Hsu, and C.-Y. Lee, "Three-phase behavior-based detection and classification of known and unknown malware," *Security and Communicatoin Networks*, vol. 8, no. 11, pp. 2004–2015, July 2015.

[5] H. Lockheimer, "Android and security," February 2012. [Online]. Available: http://googlemobile.blogspot.tw/2012/02/android-and-security.html

[6] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," in *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference*, ser. EICAR, 2006.

[7] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[8] C.-Y. Huang, S.-P. Ma, M.-L. Chang, C.-H. Chiu, and T.-C. Huang, "An open and automated android behavior monitor in cloud," *Journal of Internet Technology*, vol. 18, no. 2, pp. 297–305, Mar 2014.

[9] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29.

[10] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2015.

[11] 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*, 2009.

[12] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. lvarez, "Puma: Permission usage to detect malware in android," in *Proceedings of International Joint Conference CISIS12-ICEUTE12-SOCO12*, 2012.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.

[14] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of android applications' permissions," in *Proceedings of IEEE 6th International Conference on Software Security and Reliability Companion (SERE-C)*, 2012.

[15] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in ofcial and alternative android markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[16] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, 2010.

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[18] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the 7th International Conference on Computational Intelligence and Security*, 2011.

[19] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008. [Online]. Available: http://nlp.stanford.edu/IR-book/

[20] R. Winiewski and C. Tumbleson, "android-apktool: A tool for reverse engineering android apk files," February 2013. [Online]. Available: https://code.google.com/p/android-apktool/

[21] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

Ying-Dar Lin is professor of computer science, and founder and director of the Network Benchmarking Lab, and founder of the Embedded Benchmarking Lab at National Chiao Tung University. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms; quality of service; network security; deep-packet inspection; P2P networking; and embedded hardware/software codesign. He is an IEEE fellow and on the editorial boards of IEEE Transactions on Computers, Computer, IEEE Network, IEEE Communications Magazine Network Testing Series, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, Computer Networks, and IEICE Transactions on Information and Systems. Contact him at ydlin@cs.nctu.edu.tw.

Chun-Ying Huang received the Ph.D. degree in electrical engineering from National Taiwan University in 2007. He joined the faculty of the Department of Computer Science and Engineering at National Taiwan Ocean University in 2008 and has been an associate professor since 2013. His research interests include multimedia networking, system security, and embedded systems. Dr. Huang is a member of IEEE, ACM, IICM, and CCISA. He can be reached at chuang@ntou.edu.tw.

Yu-Ni Chang received the B.S. degree in Computer Science and Engineering from the National Taipei University of Technology, Taiwan, in 2011, and the M.S. degree in Computer Science from the National Chiao Tung University in 2013. Her researches focus on mobile security, malicious application analysis, and wireless networking. She is now an engineer in MediaTek. Contact her at yunichang.cs00g@g2.nctu.edu.tw.

Yuan-Cheng Lai received the Ph.D. degree in computer science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been a professor since 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking. He can be reached at laiyc@cs.ntust.edu.tw.