SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:5639–5650
Published online 22 February 2017 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1723

RESEARCH ARTICLE

A collaborative approach on host and network level android malware detection

Chanwoo Bae[1]* and Seungwon Shin[2]

[1]Graduate School of Information Security, School of Computing KAIST, Daejeon, South Korea
[2]School of Electronic Engineering, KAIST, Daejeon, South Korea

ABSTRACT

We suggest a collaborative approach for revealing malicious behaviors on Android smartphones by which monitoring four observable parts: (i) network usages (ii) network connections, (iii) APIs and (iv) permissions. Therefore, we have designed a detection system which consists of four engines: network behavior analysis engine, host domain reputation analysis engine, critical API call pattern analysis engine, and Android permissions use analysis engine. Each of them monitors its specific part from Android apps and independently detects malicious behavior and, given the information from four engines, the correlator determines a final decision. Finally, to show efficiency, we have evaluated our system with real world 1,621 apps. Copyright © 2017 John Wiley & Sons, Ltd.

KEYWORDS
Android; Malware; Machine-Learning; Dynamic Analysis; Malicious Behavior

***Correspondence**
Chanwoo Bae, 291 Daehak-ro Yuseong-gu, Daejeon, South Korea.
E-mail: cwbae@kaist.ac.kr

1. INTRODUCTION

The popularity of smart devices has grown rapidly in recent years, and now they are necessary elements connecting us to the Internet (or other network services) everywhere. As the number of smartphone users has explosively increased, malware authors are moving their main targets from legacy computers to smart devices. This trend can be also observed in many real world cases. For example, Android malware can steal private information from a smartphone, charge the price with short message service (SMS), utilize compromised devices as zombie workers. This trend can be also observed in recent survey studies [1,2] which systemically categorize discovered malware examples running on Android platforms.

Therefore, we are facing new types of threats (unlike attacks against legacy systems), and many research proposals (and even commercial products) have been suggested so far to detect or prevent those threats. Surveying these proposals, we have discovered that they can be classified into two main categories: (i) static analysis, which investigates the source code of malware to detect malicious behavior [3–15]. and (ii) dynamic analysis, which monitors the runtime behavior of malware to detect its forbidden operations [16–22]. Each method has clear advantages and disadvantages. While the static method does not add much overhead to the device, it can be evaded by some advanced attack methods (e.g., obfuscation). The dynamic analysis method provides better chances of detection even if the malware employs some advanced evasion ways, but it commonly adds more overhead to the device.

Observing that dynamic analysis method can increase the chance of malware detection, we have investigated if it is possible to employ a dynamic analysis method with less cost to smartphone. Then, we have found that correlating several different features that do not add much overhead can present the fair detection results.

In our approach, we minimize the use of high overhead functions (e.g., control-flow tracking) and replace them to lightweight features (e.g., method call monitoring). Here is the approach how we have leveraged those features instead of using high overhead operations. First, we capture all network packets from/to Android apps and identify the malicious behaviors upon network traffic. We claim that network usages of malware apps are distinct from the benign purposes, they show abnormal network patterns. Second, we evaluate the reputation of remote hosts that Android malware apps connect to steal private information or relay commands/responses from/to malware authors. It is likely that a malicious application is trying to connect

Copyright © 2017 John Wiley & Sons, Ltd.

to some suspicious hosts that show relatively poor reputations. By watching whom, an application connects to, we can infer its malicious behavior. Third, all Android applications run on application program interface (API) provided by Android platform. Hence, malicious behavior of an Android application should be monitored by capturing the invocation of some sensitive Android APIs. Lastly, to access smartphone system resources, an application must request permissions. For example, an Android application which sends a SMS message must indicate that it uses the permission to send the message. By monitoring the permission usage, we can catch the application's access to system resources.

In summary, we have devised three orthogonal points to monitor: (i) the network patterns (ii) whom an application connects to, (iii) the invocation of sensitive APIs, and (iv) the use of permissions. Based on those, we devise three engines that monitor their scope and make own decision, and then their decisions are combined into a final decision. Each engine in our system leverages machine learning algorithms, such as Support Vector Machine (SVM) and k-means, to increase the chance of detection.

To demonstrate the efficiency and the effectiveness of our proposed method, we have implemented a prototype system on an Android device and tested it with real world Android malware/benign samples. We prove that our system can effectively detect malicious applications with low rate of error and be efficiently deployed with low amount of performance overhead. We also provide the APIs list and permissions that our system has employed during evaluation to help readers understand what are the sensitive APIs and permissions.

Our contributions can be summarized as followings:

• We suggest a dynamic malware detection system for the Android platform, and it turns out that its overhead is less than existing systems and its accuracy is comparable with existing systems (90.26% of precision, average 6.52% runtime overhead).
• We implement a prototype system and evaluate it with real Android malware/benign applications.
• We open the set of critical APIs and critical permissions. These APIs and permissions must be watched carefully to detect malicious behavior.

2. SYSTEM ARCHITECTURE

We design our system to consist of two layer: network-level monitoring layer and host-level monitoring layer. The malicious behavior from Android devices could be captured at the network usage patterns (e.g., network packets) and also, host level (e.g., Android APIs, permissions). We leverage both aspects to maximize the efficiency and adopt the network-level layer which composed of **network behavior analysis engine**, **host domain reputation analysis engine** and host-level layer that consists of **critical API call pattern analysis engine**, **Android permission use analysis engine**.

Overall, our system consists of four engines (two from network-layer and two from host-layer) and each engine makes its own decision about whether monitored app is suspicious (or malicious), then, the correlator takes all decisions and combines them into a final decision. Employing multiple engines is good in reducing the chance of missing malicious applications, because it is tough for malware to evade all engines, we can effectively compensate for error commit by a single decision maker. Followings are list of engines in our system.

[N-1]The **Network Behavior Analysis Engine** which monitors network patterns from the packets and captures malicious behaviors at the network level.
[N-2]The **Host Domain Reputation Analysis Engine** which monitors the URL of hosts that the application connects to and investigates their reputations.
[H-1]The **Critical API Call Pattern Analysis Engine** which monitors the API invocation pattern to detect those that are related to with malicious behavior.
[H-2]The **Android Permissions Use Analysis Engine** which monitors the patterns of Android permissions use, and raises an alarm if an application is suspicious.

2.1. Network-level analysis engines

2.1.1. Network behavior analysis engine.

First, we focus on network patterns that Android malware is showing. Malware apps mostly use network function (or Internet) for achieving their malicious purposes (e.g., information leak and connection to command server). In this sense, we try to leverage network traits to single out malware apps. We have find that the purposes of network usages between malware apps and normal apps are different, that is, they show distinct network behavior patterns. Using this fact, we design the Network Behavior Analysis Engine which monitors network packets from/to Android apps and discovers malicious behaviors (or apps) upon them. To evaluate a network behavior, we define the set of features to extract the behavioral aspects from the network packets. Those are as below.

- FN_1: Bytes per second (bps)
- FN_2: Packets per second (pps)
- FN_3: (Outgoing bps)/(Incoming bps)
- FN_4: (Outgoing pps)/(Incoming pps)
- FN_5: The number of distinct IP address(es) which app connects to
- FN_6: Average time of intervals between network packets
- FN_7: The variance of time intervals
- FN_8: Average size of packets

First, malicious apps show relatively large amount of traffic volume (We admit, in some cases, it does not) (FN_1, FN_2). Another notable fact is that malicious apps send out the large amount of messages than those they receive (e.g.,

c&c workers) (FN_3, FN_4). And we found they connect more distinct IPes then benign apps do (e.g., scan attacks) (FN_5) Network packets that are automatically driven by malicious code show relatively short time intervals (FN_6, FN_7). For the last, we found each packet from malicious apps shows small size of packet segments (FN_8).

To build this engine, we use the SVM, one of the most popular machine learning classifiers. We collect sample benign and malicious apps and train the classifier with the value of features (FN_1–FN_8) from the sample, finally, the model is built. After that, the Network Behavior Analysis Engine will flag out apps which show abnormal (or malicious) network usage patterns.

2.1.2. Host domain reputation analysis engine.

Malicious Android apps commonly connect to a remote server (e.g., C&C server) for several reasons. For example, they leak private information from an Android device to a remote server operated by malware authors. To connect to the remote server, malware depends on the Domain Name Server (DNS) system (They hardly connect to the remote server with IP addresses). And malicious apps are likely to connect to low reputation domains while benign apps connect to sound domains. Motivated by this, we have designed the host domain reputation analysis engine which monitors to whom the monitored app connects to. Whenever the monitored app is trying to connect to a remote server, the engine grades reputation of domain. To grade reputation, we try to leverage existing knowledge, and we select features employed by the work of EXPOSURE [23], which is known as a decent malicious domain detection system. The features are as following.

- FD_1: Validity of the domain
- FD_2: Days elapsed since registration
- FD_3: Time between a domain registration and an expiration date
- FD_4: The number of IP addresses linked with the domain
- FD_5: Average Time-To-Live (TTL) value for the domain
- FD_6: The number of numerical characters found in the domain name
- FD_7: Length of the domain name

First, malicious domains are generally registered for a short period because they are moving in the near future (FD_2, FD_3). High price of domain registration is another reason that malware domains have a short registration period. Second, malicious domains have a relatively long TTL value (FD_5). Each domain has a TTL value that denotes how long it is stored in the DNS server cache. Malware authors tend to give a long TTL to their domains value so that DNS queries can be answered without being blocked. Another aspect is shown by the domain name string (FD_6, FD_7). The domain name of a popular company is generally short and easy to remember (e.g., google.com). However, malicious or suspicious domain names are long and often contain numerical characters because domain names are randomly generated by a program. We also check the validity of domain because malicious domains are often disconnected (FD_1), the number of IP addresses because large scale companies operate multiple servers to support high Quality of Service (QoS) (FD_4).

We again use the SVM classifier, and for training the model, we collect sample malicious/benigns domains (we have collected them from the local DNS server on campus). Those are further used for training the classifier, then the SVM classifier generates a model. Whenever an application connects to the remote server, malicious domains are alarmed by the SVM classifier and the pre-built model.

2.2. Host-level analysis engines

2.2.1. Critical API call pattern analysis engine.

Like general Android apps, malwares also call Android APIs. We have found that there is a set of APIs frequently invoked by them. Based on the fact, we have designed the second engine that monitors the invocation of some critical Android APIs from the monitored app. We have followed three steps to build our engine: (i) compiling a list of APIs to monitor, (ii) training the model based on the patterns from malicious/benign apps, and (iii) detecting unknown apps with the trained model. For the detail, see figure 1.

Critical APIs Extraction Phase. A list of APIs to monitor (let us say critical APIs) is the prerequisite for building the engine. In this phase, we extract critical APIs from benign/malware samples. Extracting critical APIs is described as three steps. First, we read the source code of sample apps and count how many times they invoke each API. Then we pick out APIs frequently invoked by malwares and by benign apps (i.e., two sets of APIs). Secondly, we run benign/malware apps to count how frequently those APIs are on use in runtime. We sort them by the gap of call ratio by malwares and that of benign apps. From the sorted list, we pick out major APIs, finally get critical APIs.

Training Phase In this step, we build the engine using the critical APIs that we have extracted in the previous step. At first, we extract runtime call ratio of critical APIs from malware/benign apps A call ratio can be expressed as a sequence of numbers whose elements are between 0 and 1 and whose length is equal to the number of critical APIs. Each element denotes ratio of how many times that API has been called among all critical APIs (i.e., the sum of all elements in a sequence must be equal to one). After extracting call ratio from sample apps, we build our engine using k-means cluster algorithm. K-means is a well-known algorithm that groups given elements (in this case, call ratio of app) into clusters so that elements in the same cluster have similar trait (call ratios in the same cluster have short distance pairwise). The k-means algorithm groups extracted call ratios into clusters which guarantees that the

(a)Training phase with clustering (b)Predicting phase

Figure 1. Overview of critical API call pattern analysis engine.

applications whose call ratios are in the same cluster shows similar patterns of invoking critical APIs. After clustering, each cluster is tagged as "malware" or "benign". With a threshold value τ, cluster C is tagged as "malware" if $\frac{m_C}{m_S} - \frac{b_C}{b_S} \geq \tau$ where m_C is the number of malware samples in cluster C, m_S is that of malware samples in all clusters. Likewise, b_C and b_S represent benign samples. The cluster C is tagged as "benign" if $\frac{b_C}{b_S} - \frac{m_C}{m_S} \geq \tau$. Otherwise, the cluster C is not tagged. The tag will be used later in the next phase (predicting phase).

Application Prediction Phase In this phase, the engine predicts whether a unknown app is malicious or benign. From an app to monitor, the engine extracts call ratio of critical APIs and match the application to the closest cluster. If the matched cluster is tagged as "malware", the behavior of application is predicted as malware, if tagged as "benign", it is predicted as benign. If that cluster is not tagged, testing phase fails to predict. In this case, the final decision depends on predictions from other engines.

The choice of a threshold value in the clustering phase is a trade-off between the precision of the engine and answering rate. If threshold value is set to zero, our engine always predicts without fail, but, in terms of false positive and false negative rate, some error rate will be shown. In contrast, with the high threshold value, precision of the system will become better, but more untagged clusters occur, our engine fails to predict for these cases.

2.2.2. Android permission use analysis engine.

Android apps must request permissions to utilize major system resource (e.g., SMS, Internet, and external storage) before being installed. Therefore, by observing which permissions are using, we could infer which resources in the device have been accessed. Upon this, we have devised another insight for way of detecting suspicious behavior (most malicious behavior of Android malware requires accessing the device's resources). By monitoring which permissions the monitored apps are used in run-time, we could infer malicious behavior from malwares.

To build the engine, we have followed a couple of steps. For the first step, we have extracted critical permissions to monitor from the stack of sample apps. For this, we have extracted the list of permissions that have been frequently used by the malicious applications and selected some major permissions. Let the list of permissions be critical permissions. Secondly, we have trained the model. Given the training samples of benign/malicious applications, we have extracted sequences which denote whether critical APIs have been used. These sequences are given to the SVM classifier so that it can train the model. By doing these steps, the engine is finally built.

After finishing all the steps, the Android permission use analysis engine can distinguish malicious apps by monitoring use of critical permissions. When app is running, it observes runtime usage of critical APIs. Then, those are sent to the SVM classifier, malicious behavior of the monitored app would be catched.

2.3. Correlation engine

Thus far, we have described four engines: (i) the network behavior analysis engine, (ii) the host domain reputation analysis engine, (iii) the critical API call pattern analysis engine, and (iv) the Android permission use analysis engine. The correlation engine has a role for (i) taking all answers from the lower layer (where our four engines are located), (ii) correlating them, and (iii) making a final decision. To build a correlation engine, we again use SVM-classifier. We let the sample applications be tested by four engines, collected their responses, then train our model for the correlation engine. When predicting an unknown application, the correlation engine makes a final decision with information from four engines and pre-trained model.

2.4. Overall system design

Figure 2 shows the overall design of our system. We separate our system into two schemes: runtime monitor installed in android device and remote analysis system. The runtime monitor installed in android device collects the information needed for analysis and sends it to the remote analysis system. The remote analysis system involves two layers: predicting layer which includes the four prediction engines and correlation layer which has the correlation engine. All works related to analysis scheme (training model, predicting) are handled by the remote server. It is to prevent our system from burdening the device resources

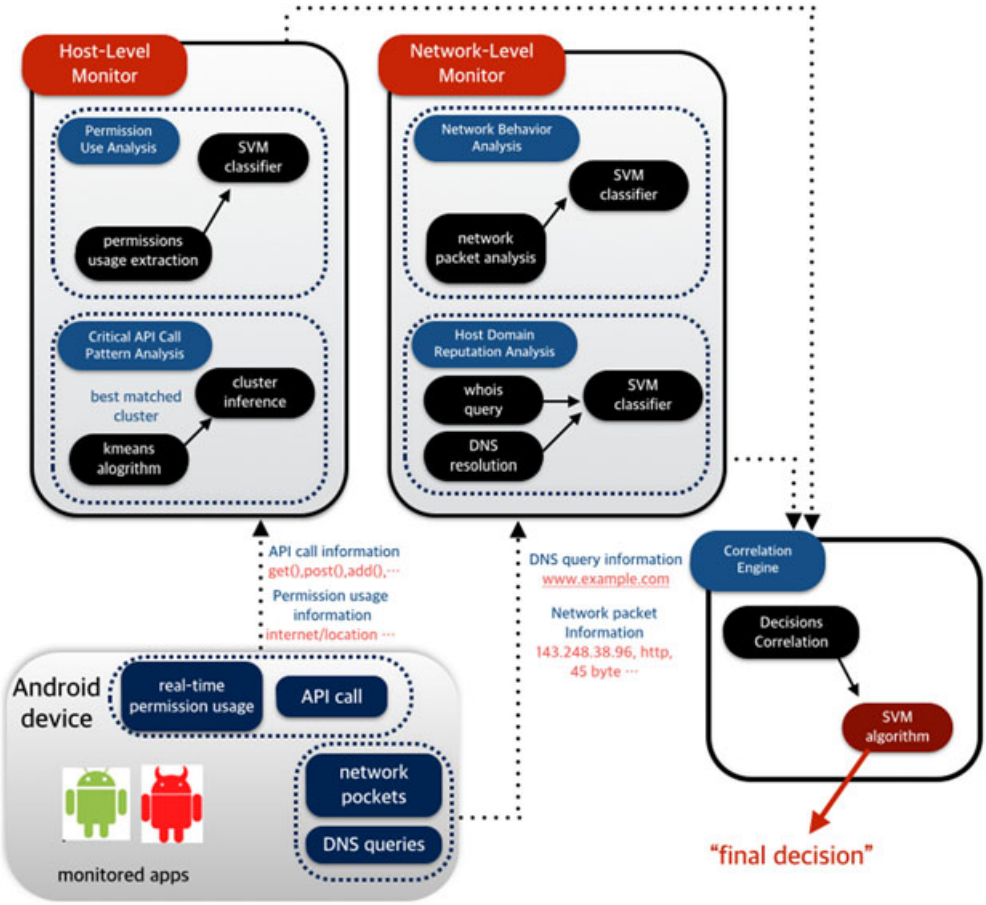

Figure 2. System design of a malware detection system.

and slowing down the phone. In addition, placing the analysis engine to the same place as the applications is not a good security decision.

3. IMPLEMENTATION

To implement the runtime monitor, we used the dynamic dalvik hooking tool by Mullier [24]. By hooking the APIs, we can monitor which APIs are invoked. The dalvik virtual machine contains an address table of methods. By overwriting address in the table, we can intercept control-flow when the method is invoked. The code of monitor is run and it sends method call information to the remote analysis server. We implemented monitoring code with 500 lines of C code. To monitor which permissions are used by the app, we hook all APIs that require those permissions. Kathy *et al.* [25] completed the permission map which specifies the relation between permissions and the APIs by noting that which API requires which permission. We check whether the permission is used by monitoring whether matched APIs are invoked.

It is hardly possible for a human to directly launch thousand of apps and generate events by clicking buttons or touching the screen. Instead, we used the Android monkey tool [26]. The monkey tool automatically launches an application, and generates random events, such as starting a new activity, touching events, clicking button or clickable view and etc. We set our monkey tool to generate 1000 events after launching the app.

The Android device for the experiment is Nexus 5 which equips with Android version 4.4.4 with Linux kernel 3.4.0. To enable our monitor to overwrite address table of method, we gave our device root privilege.

For the remote analysis server, a machine equipped with intel i5 (3.20 Ghz) CPU, 8 GB RAM is used. We have written analysis system, including all engines in Python with 2000 lines of code. We run the SVM classifier with libsvm tool [27]. For the host domain reputation analysis engine, we need whois lookup to acquire the domain registration and expiration date, DNS queries to get the TTL value and the number of IP addresses. For the whois lookup, we use Pywhois library [28]. All DNS queries are sent to Google DNS server (whose IP address is 8.8.8.8).

For the Network Behavior Analysis, we partly leveraged the source code from Android Network Log Monitor to capture network raw packets [29] from the Android device.

The number of critical APIs is 43, that of critical permissions is nine. For critical API call pattern analysis engine, the number of clusters created by k-means is set to 500, threshold value for tagging cluster is set to 0, 0.2, 0.4 (Every cluster is tagged with 3-tuple and this engine sends three answers to the correlation engine).

4. EVALUATION

4.1. Collection of malware/benign apps

To set up our evaluation, we have collected more than thousand apps to demonstrate the efficiency of our system. From drebin dataset [8], we have downloaded 795 malicious applications in May 2014. Also we have collected 826 benign applications from Google play store (the official app market operated by Google) with a market crawler [30]. The crawler automatically searches through the store, traveling all of the categories in the market. These collected apps are afterward divided into half for the train and the other half for test (i.e., to show efficiency). The apps for training are used for building four engines and the correlator. Last of them (apps for test) are used to show how efficiently our system can detect malicious apps.

4.2. Collection of domains for reputation train

Unlike other engines, to train host domain reputation analysis engine, we need sample domains rather than malware/benign apps. For this, we have collected domains from the local DNS server on our campus. The duration of collection was from July to August in 2013. We classify them into two categories (malicious or benign) using well known reputation grading services: McAfree SiteAdvisor [31], Google Safe Browsing [32]. Google Safe Browsing is a free service for detecting web servers that are used for malicious purposes (e.g., code download). SiteAdvisor is another good service for identifying the malicious domains. When we query SiteAdvisor with domain to investigate, it finds threat factors such as file downloading, phishing, etc. Both are reliable sources of criteria for dividing them into malicious group and benign group. Among domains collected from campus, we treated them as malicious domain if both services said "malware", as benign domain if both said "benign". Domains with different answers are discarded. By following these steps, we have collected 1033 benign domains and 1223 malicious domains. These domains become seed for building the host domain reputation analysis engine.

4.3. Efficiency

To show efficiency of our system, we have run our system with test apps (413 benign apps and 398 malicious apps). In this section, we cover the precision (or detection rate) from the experiment. We will provide evaluation results from stand-alone of each engine in our system (to show how precisely each of engine predicts), then we present the precision of the final decision which is the correlated answer (overall precision).

4.3.1. Network behavior analysis engine.

We found that 360 malware apps out of 398 used network, 403 benign apps out of 413 did so. Among 360 malware apps, 328 apps (91.11%) were correctly flagged by their abnormal network patterns, and out of 403 benign apps, 51 apps were wrongly flagged by this engine (87.34% of precision).

4.3.2. Host domain reputation analysis engine.

Among 826 benign apps, 415 apps (50.2%) required DNS queries and among 795 malware apps, 320 apps did so (40.2%). We have found that among the 415 benign applications that connected to a remote host, 87 applications were flagged (20.96%) by host domain reputation analysis engine. Among 320 malicious apps, 173 applications were flagged (54.06%).

4.3.3. Critical API call pattern analysis engine.

As it is explained in section 2.2 (see clustering phase), critical API call analysis engine gives out a decision only if $|\frac{b_C}{b_S} - \frac{m_C}{m_S}| > \tau$. As threshold value increases, there might be some loss in prediction rate but more precise it should be (trade-off between "how exactly it can predict?" and "how many apps are classified?").

The precision of benign case moves from 72% to 86% while precision of malware case does from 87% to 92% Table I. The prediction rate shows somewhat decreasing which amounts near 80% when threshold is 0.4. That the prediction rate is 80% means two apps out of 10 will not be predicted by this engine. In this case, the final decision is come from other engines.

4.3.4. Android permission use analysis engine.

We have run permission using analysis engine to evaluate its efficiency. Among 413 benign applications, 394 apps (95.3%) were rightly unflagged by our engine. Among 398 malicious apps, 283 apps (71.3%) were correctly flagged.

4.3.5. Final decision.

In this section, we discuss the overall precision (i.e., precision of final decision) of our system. Each of the engine in our system makes its own decision. By combining them, the correlation engine judges the target app. If remind, to correlate decisions from the lower layer (each engine's decision) into one final decision, we use the SVM classifier. In this section, we present the precision of final decision which represents the ultimate efficiency of our system.

DOI: 10.1002/sec

Table I. Precision rate of critical API call analysis engine
• bengin precision = 1 - (# Detected Apps)/(# Benign Apps)
• malware precision = (# Detected Apps)/(# Malware Apps).

Threshold	Bengin precision	Malware precision	Prediction rate
0	72.02%	87.88%	100%
0.2	86.80%	91.39%	82.99%
0.4	86.65%	92.77%	81.92%

Table II. Precision rate of final decision (the SVM classifier).

	Predicted as benign	Predicted as malware	Precision rate
Benign	359	54	86.92%
Malware	25	373	93.72%
TN & TP	93.49%	87.35%	90.26%

Table II shows precision rate of the final decision. Among 413 benign apps, 359 apps are rightly answered by the correlation engine while among 398 malicious apps, 373 apps are correctly alarmed. Therefore, the precision rates of benign case and malware case are 86.92% and 93.72% separately. Additionally, true negative rate is 93.49% and true positive rate is 87.35% as shown in the last law of Table II.

4.3.6. Comparison by other classification algorithms.

Other than the SVM classifier, we have set other classifier algorithms: Naives Bayes, Decision Tree to the correlation engine. Table III and IV show results from the comparison evaluation.

4.4. Performance

In this section, we discuss the performance overhead caused by our system. We have tested by three measures: single API call overhead, runtime overhead, and time consumption by some asynchronous works (model training and DNS resolutions).

4.4.1. Single API call overhead.

Table V shows additional time consumption when a target app calls monitored API. When monitored APIs are invoked, runtime monitor requires additional work to send out information (by intercepting flow from the app), elapsed time for single API call gets longer than without our system. To measure how much cost does it have, we have arbitrarily chosen five APIs and measured time consumption for method call (from "invoke" to "return") with and without our system. Every elapsed time in Table V is average value from 10 000 times repetition.

4.4.2. Runtime overhead.

To measure the runtime overhead of our system (i.e., how much performance decline is cost to the monitored app), we have run two widely used Android benchmark programs: Vellamo Benchmark [33] and GFX Bench [34]. Vellamo Benchmark supports browser benchmark which automatically runs Android browser and measure time consumption for each task in the browser instance. GFX Bench is another good benchmark program which performs 3D image rendering work in the Android mobile and shows the result. The result of those benchmarks is described in Table VI.

4.4.3. Time consumption by SVM train and DNS resolution.

We also measured the time required by tasks in the remote analysis server. Figures 3 and 4 show time consumed while training our model using the SVM classifier and k-means. We verified required time for training model with randomly generated thousand cases. Time consumption for training them is no more than 60 ms for the SVM classifier, 200 ms for the k-means classifier. Considering that we build models not frequently, it does not burden to our system much. We have also measured time for domain resolutions. For a DNS query, it costs 8.6 ms on average of 10 requests. For the whois query, it spends 0.495 s as a mean value for 10 times.

5. LIMITATION

As we all know, there is no prefect detection system without committing any error. Our detection system is not the exception. For instance, the malicious app could use well known domains such as Google, Amazon as a remote server to evade our host domain reputation analysis engine. In addition, Malwares could intentionally invoke some critical APIs to confuse the critical API call pattern analysis engine. We admit that malwares have room for not being detected by a single detector in our system. However, because we have designed our system that multiple engines correlate together, they could create the synergy effect, demonstrate the high accuracy for Android malware prevention.

Table III. Precision rate (by Naive Bayes)

	Predicted as benign	Predicted as malware	Precision rate
Benign	382	31	92.49%
Malware	94	304	76.38%
TN & TP	80.25%	90.75%	84.59%

Table IV. Precision rate (by Decision Tree).

	Predicted as benign	Predicted as malware	Precision rate
Benign	367	46	88.86%
Malware	29	369	92.71%
TN & TP	92.68%	88.92%	90.75%

Table V. Single API call overhead (time in ms).

Android API method name	w/o	w	Time gap	Overhead
getDeviceID	0.3522	0.5188	0.1666	47.28%
sendTextMessage	2.5226	2.6162	0.0936	3.71%
vibrate	0.2242	0.2794	0.0552	24.61%
isProviderEnabled	0.0988	0.1418	0.0430	42.90%
openConnection	0.1738	0.2297	0.0559	32.16%

Table VI. Benchmark runtime overhead.

Benchmark tool	Benchmark task	Without monitoring	With monitoring	Overhead
Vellamo Browser Benchmark	Image Re-focus	5.62 s	5.83 s	3.74%
	WebGL Jellyfish	16.41 s	16.65 s	1.46%
	Sun Spider	10.10 s	12.39 s	22.67%
	Octane v1	44.09 s	48.85 s	10.80%
GFX Bench 3.1	ALU2	618.9 frame/s	617.9 frames/s	0.16%
	1080p ALU2	1,161 frames/s	1,159 frames/s	0.17%
	Driver Overhead	134.9 frames/s	135.4 frames/s	-0.37%

To make our system more robust, we will further improve our system to reduce false positive rate (now is 13.91%). There are millions of widely used Android apps in the market and it is never possible to train entire categories of benign apps all of the world (this causes abnormal detection to have false positive). The larger size of benign apps sample (for train), the lower rate of false positive would be, but that cannot be zero. To compensate for it, we plan to conduct more large scale analysis on Android apps. By doing so, we believe that our detection system would become even more efficient and effective.

6. DISCUSSION

In this section, we discuss issues that we have encountered during our work.

6.1. Critical APIs and critical permissions

In this section, we present top critical permissions and critical. We believe these set of APIs and permissions would give a huge insight into the malware analysis. Table VII shows the top six critical permissions. The critical permissions are those which are frequently used by malware apps (see the Section 2.3).

Additionally, we present top 10 critical APIs (see critical APIs extraction phase in Section 2.2). The Table VIII shows APIs that are frequently called by malwares but seldom called by benign applications and Table IX shows the reverse. The ratio gap is differential between average call ratio of malwares and that of benign apps. The bigger ratio gap of the API shows, the more useful it is for catching out malwares.

6.2. Lack of domain reputation service for malware

While designing the host domain reputation analysis engine, we felt the lack of reputation services for domains used as a malware remote server. It is hardly able to directly use siteadvisor [31] or Google Safe Browsing [32] to

DOI: 10.1002/sec

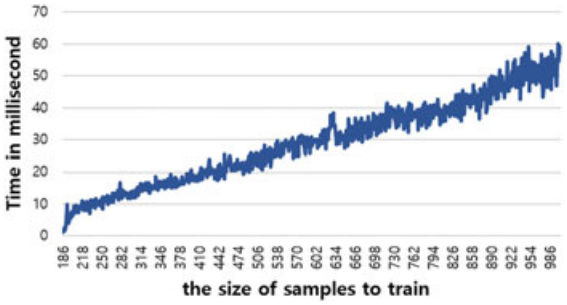

Figure 3. Time required for model training (svm classifier).

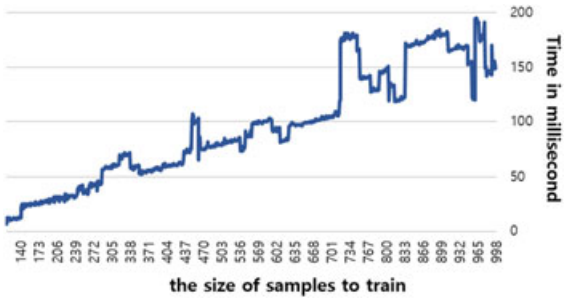

Figure 4. Time required for model training (k-means).

Table VII. Top 6 permissions frequently used by malwares.

Permission	Function
INTERNET	To be allowed to access Internet
ACCESS_COARSE_LOCATION	To be allowed to get location
ACCESS_NETWORK_STATE	To check if Internet is able
READ_PHONE_STATE	To access state phone
SEND_SMS	To send SMS
CHANGE_WIFI_STATE	To check if WIFI is able

Table VIII. Top 5 APIs frequently used by malware apps but seldom by benign apps.

Android API method	Ratio gap
InputStream;read	37.60%
View;getLayoutParams	20.91%
Context;getStringExtra	16.85%
View;getTop	4.38%
SQLiteDatabase;execSQL	4.23%

Table IX. Top 5 APIs frequently used by benign apps but seldom by malware apps.

Android API method	Ratio gap
AtomicInteger;getAndIncrement	38.23%
Resources;getDisplayMetrics	37.67%
Character;getDirectionality	2.49%
CopyOnWriteArrayList;isEmpty	2.43%
os/Handler;init	1.70%

catch out domains from malicious apps because we found no domains from malware apps that were alarmed. We have implemented our domain reputation analysis engine with machine learning approaching based on features from EXPOSURE [23], however the detection precision rate should be improved. If a blacklist/whitelist service for domains for malware is launched, we believe that we can detect malicious behavior connecting to a remote C&C server more accurately.

7. RELATED WORK

7.1. Android malware

Because early version of Android was released, there have been plenty of studies which surveyed and well organized security in Android. Among them, we summarize major works in our view that have strong contribution for anatomy of Android security.

Zhou *et al*. [1] is the first study which systematically characterized Android malwares in our knowledge. They collected more than 1200 malware samples, arranged how they are installed, which malicious behavior they commit. By them, activation of Android malwrare are classified into privilege escalation, remote control, financial charge, and information collection. They also opened their dataset. Download is free for academic purpose in [35].

Arp *et al*. [8] collected big dataset of Android malware. They collected 5560 malicious applications (up to Feb. 2015). They collected malware applications from Google Play Store and other alternative markets in China and Russia. To determine malicious applications from the collected applications, they are sent to VirusTotal service and other 10 common anti-virus scanners (AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos). They also opened their dataset to who have research purpose. The link for download page is [36].

One of the large size analysis on Android malware is ANDRUBIS [2]. They made a lot of effort to collect large size of malicious applications which amount about a million. They provided big analysis driven from their thorough survey.

Other notable studies are [25,37–40]. They well probed the internal of Android security, that assists descendant studies in various fields.

7.2. Static analysis

There have been large scale of anti-malware research with static analysis. Recently, by their effort, static analysis shows high precision in detecting malicious applications.

Mu Zhang *et al*. [3] designed a system detects anomaly and signature by storing behavior graph. They suggested way of representing behavior of application by graph. They stored all of behaviors from large-scale sample of applications. Abnormal behavior (suspicious to be malicious) is not found in dataset, therefore is detected.

Flowdroid [41] is information leak detector with taint analysis. Information leak by malicious application is detected by tracking the flow from source to sink in the source code.

Arp *et al*. [8] used SVM to detect malicious from the application. They withdrew various of information (permissions, invocation of APIs and etc) from the Android application (APK file). With this information, SVM classifier detects malicious application. The contribution is that they organized extractable features from Android applications.

There have been also various researches with static analysis as [5–7,9–15,42,43] as well. By them, a state-of-the-art static analysis has become very precise and effective on detecting Android malware.

7.3. Dynamic analysis

Several works have been proposed in dynamic methods. Surveying these, we realized the narrow scope of conducted studies on dynamic analysis.

Taintdroid [16] is real-time information flow tracking system. By tracking the flow of information with taint analysis, Taintdroid can detect leaking of them to outside of device (e.g., network and SMS). The taint analysis is deployed on every memory (stack and heap on dalvik machine) and registers. In our knowledge, this is the first approach that tracking information flow by dynamic analysis.

Yuan Zhang *et al*. [44] approached to perform permission use analysis to vet malicious behavior. Their framework extract behavior from the application by monitoring usage of permissions.

Crowdroid [17] is a framework of distinguishing repackaged application from the normal applications. They discovered that call ratio of system from repackaged application is distinguishable from normal applications. Crowdroid collects system call ratios from huge number of users, analyzes them, detects abnormal application which is repackaged.

Other researches with dynamic methods are [18–22,45,46]. Our approach is different from them, in that they handled specific case of malware (repackaging and information leak) while we organized features of malicious application, built detection system of general malwares.

8. CONCLUSION

In this paper, we have studied behaviors of Android malicious applications and features that enable to detect Android malware: (i) Network patterns, (ii) Domain which application is connecting to, (iii) which APIs are invoked by application, and (iv) which permissions are used by application. We have built the detection system based on these features. It is the first trial of organizing features of Android malwares and correlative approach to detect them. Evaluation with real-world Android applications had proved that a detection with this method can be achieved with very low rate of error without much overhead.

REFERENCES

1. Zhou Y, Jiang X. Dissecting Android Malware: Characterization and Evolution. *In IEEE Symposium on Security and Privacy*, San Francisco, 2012; 95–109.
2. Martina L, Matthias N, *et al.* ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, Wroclaw, 2014; 3–17.
3. Mu Z, Yue D, *et al.*Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. *ACM Conference on Computer and Communications Security*, Arizona, 2014; 1105–1116.

4. Kevin ZC, Noah J, *et al.* Contextual Policy Enforcement in Android Applications with Permission Event Graphs. *Network & Distribution System Security Symposium*, San Diego, 2013.
5. Yajin Z, Zhi W, *et al.* Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. *Network & Distribution System Security Symposium*, San Diego, 2012.
6. Chao Y, Zhaoyan X, *et al.* DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. *European Symposium on Research in Computer Security*, Wroclaw, 2014; 163–182.
7. Micheal G, Deokhwan K, *et al.* Information Flow Analysis of Android Applications in DroidSafe. *Network & Distribution System Security Symposium*, San Diego, 2015.
8. Daniel A, Micheal S, *et al.* Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. *Network & Distribution System Security Symposium*, San Diego, 2014.
9. Yinzhi C, Yanick F, *et al.* EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. *Network & Distribution System Security Symposium*, San Diego, 2015.
10. Zhemin Y, Min Y, *et al.* AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. *ACM Conference on Computer and Communications Security*, Berlin, 2013; 1043–1054.
11. Fengguo W, Sankardas O, *et al.* Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Conference on Computer and Communications Security*, Arizona, 2014; 1329–1341.
12. Sanae R, Zhiyun Q, *et al.* AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users. *ACM conference on Data and application security and privacy*, San Antonio, 2013; 221–232.
13. Yu F, Saswat A, *et al.* Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis. *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Hong Kong, 2014; 576–587.
14. Long L, Zhichun L, *et al.* CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. *ACM Conference on Computer and Communications Security*, Raleigh, 2012; 229–240.
15. Mu Z, Heng Y, *et al.* Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding. *ACM Symposium onInformation, Computer and Communications Security*, Kyoto, 2014; 259–270.
16. William E, Peter G, *et al.* TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 2014; **32**(2): 5.
17. Iker B, Urko Z, Simin N. Crowdroid: behavior-based malware detection system for Android. *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, 2011; 15–26.
18. Wei Y, Hanlin Z, *et al.* On Behavior-based Detection of Malware on Android Platform. *IEEE Global Communications Conference*, Atlanta, 2013; 814–819.
19. Victor VDV. Dynamic Analysis of Android Malware. *Thesis of M.D.*, VU University Amsterdam, 2013.
20. Droidbox, (Available from: https://code.google.com/p/droidbox/ [Accessed: December 16, 2016].
21. Lok KY, Heng Y. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. *USENIX Conference on Security*, Bellevue, 2012; 569–584.
22. Mads D, Gurvan LG, *et al.* TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies. *ACM Conference on Computer and Communications Security*, Raleigh, 2012; 894–905.
23. Leyla B, Engine K, *et al.* EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. *Network & Distribution System Security Symposium*, San Diego, 2011.
24. Dynamic Dalvik Instrumentation Toolkit, (Available from: https://github.com/crmulliner/ddi) [Accessed: December 16, 2016].
25. Kathy W, Yi F, *et al.* PScout: Analyzing the Android Permission Specification. *ACM Conference on Computer and Communications Security*, Raleigh, 2012; 217–228.
26. Monkey Test Tool, (Available from: http://developer.android.com/tools/help/monkey.html) [Accessed: December 16, 2016].
27. LIBSVM, (Available from:http://www.csie.ntu.edu.tw/cjlin/libsvm)[Accessed: December 16, 2016].
28. Pywhois, (Available from: https://code.google.com/p/pywhois/)[Accessed: December 16, 2016].
29. Android Network Log Monitor, (Available from: https://github.com/pragma-/networklog)[Accessed: December 16, 2016].
30. Scrape Google Play, (Available from: https://github.com/anuvrat/scrape-google-play)[Accessed: December 16, 2016].
31. McAfee SiteAdvisor, (Available from: https://www.siteadvisor.com/)[Accessed: December 16, 2016].
32. Safe Browsing API, (Available from: https://developers.google.com/safe-browsing/)[Accessed: December 16, 2016].

33. Vellamo Mobile Benchmark, (Available from: https://play.google.com/store/apps/details?id=com.quicinc.vellamo)[Accessed: December 16, 2016].
34. GFXBench GL Benchmark 3.1, (Available from: https://play.google.com/store/apps/details?id=net.kishonti.gfxbench.gl)[Accessed: December 16, 2016].
35. Android Malware Genome Project, (Available from: http://www.malgenomeproject.org/)[Accessed: December 16, 2016].
36. The Drebin Dataset, (Available from: http://user.cs.uni-goettingen.de/darp/drebin/)[Accessed: December 16, 2016].
37. Siegfried R, Steven A, *et al.* A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. *Network & Distribution System Security Symposium*, San Diego, 2014.
38. Lei W, Micheal G, *et al.* The Impact of Vendor Customizations on Android Security. *ACM Conference on Computer and Communications Security*, Berlin, 2013; 623–634.
39. Jin H, Su MK, *et al.* Comparing Mobile Privacy Protection through Cross-Platform Applications. *Network & Distribution System Security Symposium*, San Diego, 2013.
40. William E, Damien O, *et al.* A study of android application security. *In USENIX security symposium*, San Fransisco, 2011; 2. Vol. 2.
41. Steven A, Siegfried R, *et al.* FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh, 2013; 259–269.
42. Gordon MI, *et al.* Information Flow Analysis of Android Applications in DroidSafe. *Network & Distribution System Security Symposium*, San Diego, 2015.
43. Zhang M, *et al.* Towards Automatic Generation of Security-Centric Descriptions for Android Apps. *In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, 2015; 518–529.
44. Yuan Z, Min Y, *et al.* Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. *In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, Berlin, 2013; 611–622.
45. Tam K, *et al.* CopperDroid: Automatic Reconstruction of Android Malware Behaviors. *Network & Distribution System Security Symposium*, San Diego, 2015.
46. Wong MY, Lie D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. *Network & Distribution System Security Symposium*, San Diego, 2016.