POSTER: A Collaborative Approach on Behavior-Based Android Malware Detection

Chanwoo Bae(✉), Jesung Jung, Jaehyun Nam, and Seungwon Shin

Korea Advanced Institute of Science and Technology (KAIST),
Daejoen, Republic of Korea
{cwbae,taegm01,namjh,claude}@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 everywhere. As the number of smartphone users has explosively increased, malware authors are moving their targets from legacy computers to the smart devices. Therefore, we are facing new types of threats.

Many research proposals have been suggested so far to detect and prevent those threats, and these can be classified into two main categories: (i) static analysis, which investigates the source code of malware to detect malicious behavior [1–3]. and (ii) dynamic analysis, which monitors the runtime behavior of malware to detect its forbidden operations [4,5]. 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, however, 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, but with less cost to the smartphone. And, we have found that correlating several different features that do not add much overhead can present similar detection results compared to existing detection systems based on dynamic analysis.

In our approach, we minimize the use of high overhead functions (e.g., control-flow tracking) and replace them to lightweigt features (e.g., function call monitoring). Here is the approach how we have leveraged those features instead of using high overhead operations. First, we monitor the network connections. It is likely that malicious apps are trying to connect to some suspicious hosts with relatively poor reputations. By watching whom, an app connects to, we can infer its malicious behavior (a good heuristic for malware detection). Second, all Android apps run on application program interface (API) provided by Android platform. Hence, malicious behavior of an Android app can be monitored by capturing the invocation of some sensitive Android APIs. Third, we monitor pattern of permission usage of an app. By monitoring the permission usage, we can verify malicious behaviors which are related to those permissions.

B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 594–597, 2015.
DOI: 10.1007/978-3-319-28865-9_38

2 System Architecture

Our system consists of three engines: (i) host domain reputation analysis engine, (ii) critical API call pattern analysis engine and (iii) Android permission use analysis engine. Each of them makes own decision whether a monitored app is malicious, then, the correlator takes all decisions and combines them into a final decision. To employ multiple engines is a good way in reducing the chance of missing malicious apps by compensating errors with ohters' decisions.

Host Domain Reputation Analysis Engine. It is likely that malicious apps are connecting to the host with bad domain (or low reputation). By leveraging this fact, we have designed the host domain reputation analysis engine which monitors to whom the monitored app connects to. In this design, we try to leverage existing knowledge, and we select features employed by the work of EXPOSURE [6], which is known as a decent malicious domain detection system.

To build this engine, we use the Support Vector Machine (SVM), one of the most popular machine-learning classifiers. To train the model, we have collected sample malicious/benign domains (we have collected them from the local DNS server on campus from July to August in 2013) and built a SVM model.

Critical API Call Pattern Analysis Engine. We have observed that there is a special set of APIs frequently or hardly invoked by malwares (let us call those critical APIs). Based on this, we have designed the second engine that monitors the invocation of critical APIs. We have followed three steps to build our engine;

(i) Critical APIs Extraction: By running malicious/benign apps, we have extracted APIs that are frequently used by malicious apps but seldom by benign apps and vice versa. We further use this list of APIs as critical APIs.

(ii) Training a Model: We group apps into clusters by the pattern of using critical APIs. We extract call ratios of every single app from sample (by running them), make groups (or clusters) by K-means which is a well-known clustering algorithm. Apps whose call ratios of critical APIs are in the same cluster must have the similar pattern of usage of those APIs.

(iii) App Prediction: In this phase, we finally predict whether a unknown app is malicious or benign. From apps, extracting call ratio of critical APIs, we match this to the most close cluster. By figuring out the portion of malicious apps in that cluster, this engine could determine its decision.

Android Permission Use Analysis Engine. Like critical APIs, there also is a set of permissions that well-used by malicious apps (let this be critical permissions). We have extracted critical permissions as same as we did for the critical APIs. By training a SVM model based on usage of critical permissions, the third engine has been finally built.

Correlation Engine. To build a correlation engine, we again use SVM-classifier. We have let the sample apps be tested by three engines, and collected their responses, then finally trained our model for the correlation engine. When predicting an unknown app, the correlation engine makes a final decision with decisions from three engines and pre-trained model.

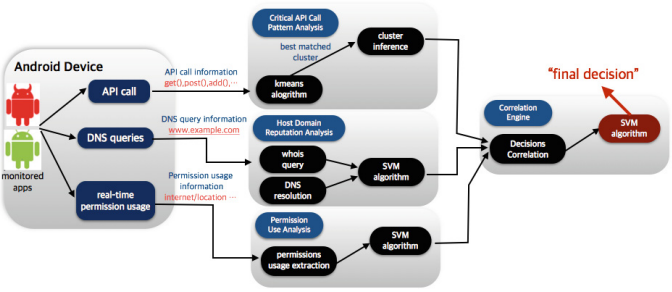

Fig. 1. System design of a malware detection system

3 Evaluation

Collection of Malware/Benign Apps. To train models and test our system, we have downloaded malware sample (795 apps) from the drebin[1]. Also we have collected benign apps (826 apps) by crawling and downloading from Google Playstore which is the official app market of Google. We have divided them into training sample and test sample (half for the training, half for the test).

The Precision of Single Engine. To show the precision of our system, we first measured the efficiency of each engine. The results are shown in table 1.

Table 1. The Precision of Each Engine

engine	result
Host Domain Reputation Analysis Engine	87 apps out of 415 benign apps which have connected to at least one remote host, were alarmed, 173 apps out of 340 malicious apps (with at least one connection) were alarmed.
Critical API Call Pattern Analysis Engine	358 apps out of 413 benign apps have been rightly not flagged (86.80%) and, 363 apps out of 398 malicious apps have been rightly flagged (91.39%).
Android Permission Use Analysis Engine	394 apps out of 413 benign apps have been rightly not flagged (95.40%) and, 283 apps out of 398 malicious apps have been rightly flagged (71.11%).

[1] The link for download is http://user.informatik.uni-goettingen.de/~darp/drebin/

Final Decision. As we mentioned previous section, each engine makes its own decision and then these are correlated into a final decision. Table 2 presents how precise the final decision is. Through the result, we found that, by combining decisions from multiple engines, the precision gets better.

Table 2. The Precision of Final Decision

by SVM	predicted as benign	predicted as malware	precision rate
benign	350	63	84.75%
malware	8	390	97.99%
TN& TP	97.77%	86.09%	91.25%

by Naive Bayes	predicted as benign	predicted as malware	precision rate
benign	358	55	86.68%
malware	26	372	93.47%
TN& TP	93.23%	87.12%	90.01%

Decision Tree	predicted as benign	predicted as malware	precision rate
benign	395	18	95.64%
malware	78	320	80.40%
TN& TP	83.35%	94.67%	88.16%

majority rule	predicted as benign	predicted as malware	precision rate
benign	320	93	77.48%
malware	26	372	93.47%
TN& TP	92.49%	80.00%	85.33%

The tables show results each by SVM (upper left), by Naive Bayes (upper right), by Decision Tree (bottom left) and with the decision by majority (bottom right).

Performance. For the last, we have measured performance of our system (i.e., performance overhead). We have run two widely-used Android benchmark tools: Vellamo Benchmark and GFX Bench.[2] The results show that our system has caused 7.27% and 0.16% (by average of three tasks) performance overhead from each benchmark tool.

References

1. Mu, Z., Heng, Y., Zhiruo, X.: Semantics-aware android malware classification using weighted contextual api dependency graphs. In: ACM CCS, Arizona (2014)
2. Yajin, Z., Zhi, W., et al.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS, San Diego (2012)
3. Kevin, Z.C., Noah, J., et al.: Contextual policy enforcement in android applications with permission event graphs. In: NDSS, San Diego (2013)
4. William, E., Peter, G., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX OSDI, Vancouver (2010)
5. Iker, B., Urko, Z., Simin, N.: Crowdroid: behavior-based malware detection system for android. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Chicago (2011)
6. Leyla, B., Engine, K., et al.: EXPOSURE: finding malicious domains using passive DNS analysis. In: NDSS, San Diego (2011)

[2] The unique app ID of Vellamo is com.quicinc.vellamo, that of GFX Bench is net.kishonti.gfxbench.gl. Both two apps are available in Goolgle Playstore for free.