Detecting Malware for Android Platform: An SVM-based Approach

Wenjia Li*, Jigang Ge, Guqian Dai
Department of Computer Science
New York Institute of Technology
New York, NY 10023, USA
{wli20, jge02, gdai01}@nyit.edu

***Abstract*—In recent years, Android has become one of the most popular mobile operating systems because of numerous mobile applications (apps) it provides. However, the malicious Android applications (malware) downloaded from third-party markets have significantly threatened users' security and privacy, and most of them remain undetected due to the lack of efficient and accurate malware detection techniques. In this paper, we study a malware detection scheme for Android platform using an SVM-based approach, which integrates both risky permission combinations and vulnerable API calls and use them as features in the SVM algorithm. To validate the performance of the proposed approach, extensive experiments have been conducted, which show that the proposed malware detection scheme is able to identify malicious Android applications effectively and efficiently.**

***Keywords*— Android, malware, Support Vector Machine (SVM), TF-IDF**

I. INTRODUCTION

The rapid growth of mobile devices and remarkable advances in 4G/5G mobile networking technologies have both inspired and facilitated a plethora of mobile applications (apps). In support of these mobile devices, various mobile operating systems have been developed, such as Android, iOS, and BlackBerry 10, etc. Among these mobile operating systems, Android has become the most popular one because of numerous mobile apps it provides. Unfortunately, smartphones running Android have been increasingly targeted by attackers and infected with malicious apps: according to the mobile threat report released by F-Secure in 2014 Q1 [1], over 95% of malicious apps were distributed on the Android platform. Moreover, as the major sources of Android apps, third-party markets contain a lot of cracked or tampered apps, and there is no sign to indicate whether the apps have been checked for security risks or not in these third-party markets. The lack of security inspection on Android apps intensifies the spreading of malicious apps (malware) on the Android platform. Zhou et al. [2] performed some Android malware analysis by popular security software tools, and the detection rate was only between 20.2% and 79.6%, which clearly indicates an urgent and rapidly growing demand on malware detection for Android applications.

To address the increasingly wide-spread security risks, the Android platform itself provides several security solutions that harden the installation of malware, such as the Android permission system and Google "Bouncer". To perform certain tasks on Android devices, such as sending a SMS message, each Android app has to explicitly request the corresponding permission from the user during the installation process. However, many users tend to arbitrarily grant permissions to unknown Android apps without even looking at what types of permissions they are requesting and thereby significantly weaken the protection provided by the Android permission system. As a result, it is very difficult to limit the propagation of malicious apps by the Android permission system in practice. On the other hand, Google "Bouncer" is a service added to Google Play, the official Android market, in 2012, which aims to automatically scans apps (both new and previously uploaded ones) and developer accounts in Google Play with its reputation engine and cloud infrastructure. Even if Bouncer adds another line of defense for Android security, it still has many limitations. First, Bouncer can only scan Android apps for limited time, which means a malicious app can easily bypass it by not doing anything malicious during the scan phase. Second, no malicious code needs to be included in the initial installer when it gets scanned by Bouncer. In this case, the malicious app can have an even higher probability to evade Bouncer's detection. Once the application passes Bouncer's security scan and gets installed on a real user's Android device, then the malicious app can either download additional malicious code to run or connect to its remote control and command (C&C) server to upload stolen data or receive further commands.

Recently, there have been some research efforts on detecting Android malware by using various machine learning algorithms. Drebin [3] extracts permissions, APIs and IP address as features and uses the Support Vector Machine (SVM) algorithm to learn a classifier from the existing ground truth datasets, which can then be used to detect unknown malware. DroidMat [4] alternatively applies KNN (K-Nearest Neighbor) algorithm to permissions and intents to identify malware. In addition, DoridAPIMiner [5] focuses on providing several lightweight classifiers based on the API level features. However, the exiting malware detection approaches have some limitations that have to be addressed. For instance, the recall value of DroidMat is significantly lower than its precision value, which indicates that some malware cannot be correctly detected. In addition, the high accuracy of DroidAPIMiner owes to the fact that its testing dataset contains far more benign

* Wenjia Li is the corresponding author, email: wli20@nyit.edu.

apps than malware. Moreover, it generally takes a large amount of time for Drebin to build the classifier because of the high-dimensional feature vector, which contains over 545,000 features. Thus, it is critical to develop an effective and efficient approach to detect malware for Android platform.

In this paper, we propose a malware detection scheme for Android platform using the SVM-based approach. In this scheme, we first calculate the similarity scores between malware and benign applications in terms of suspicious API calls, and use these similarity scores as features in the feature vector. Then, we also use risky permission requests as additional features when we train the SVM classifier. Experimental results on real benign and malicious application dataset show that the proposed scheme can accurately identify Android malware.

The rest of this paper is organized as follows. Section II reviews the prior research efforts that have been made to identify malware for Android platform. In Section III, the proposed malware detection scheme is described in details. We present the performance evaluation results in Section IV. Finally, we conclude the work in Section V.

II. Related Work

Many research studies on android malware detection have been performed in recent years, in which static analysis and dynamic analysis are the two major directions among these research works. Enck et al. [6] gives an overview of Android malware regarding its dangerous behaviors and vulnerabilities.

A. Static Analysis BasedMalware Dection

The first approach for detecting Android malware is inspired by concepts from static program analysis. Several methods have been proposed that statically inspect mobile apps and disassemble the code. Decompiling and data flow tracking are two main techniques in most of the static analysis methods.

For instance, Kirin [7] checks the permission of Android apps for indications of malicious activity. Similarly, Stowaway [8] analyzes API calls to detect over-privileged apps, and RiskRanker [9] statically identifies Android apps with different security risks. There are also some common open-source tools for static analysis such as Smali [10] and Androguard [11], which enable dissecting the content of apps with little effort.

In addition, AndroidLeaks [12] focuses on finding privacy or sensitive data leaks for Android apps using decompiled code. FlowDroid [13] aims to identify malware by building precise model of Android's lifecycle. LeakMiner [14] is a tool that detects leakage of sensitive information on Android with static taint analysis.

B. Dynamic Analysis Based Malware Detection

Dynamic analysis focuses on detecting Android malware when Android apps are being executed, and most of them monitor the behaviors of apps in terms of accessing private data or using restricted API calls. For instance, both TaintDroid [15] and DroidScope [16] dynamically monitor Android apps when they are actually executed, where the former focuses on taint analysis and the latter aims to introspection at different layers of Android system.

In general, dynamic analysis cannot be performed on the smartphones themselves to directly identify malware because they usually incur a large amount of computational overhead. Thus, they are mainly used to detect Android malware offline via scanning and analyzing a lot of Android applications. For example, malware detection schemes such as AppsPlayground [17] and DroidRanger [18] have been proposed to dynamically analyze Android apps and detect the possible malicious behaviors they may perform.

C. Malware Detection Using Machine Learning Algorithms

It is generally difficult to manually specify and update detection patterns for Android malware in static analysis process. Therefore, some recent research efforts have been focusing on using various machine learning algorithms to automatically extract app features efficiently distinguish malicious apps from benign ones by models without manual operation. Drebin [3], DroidMat [4], and DroidAPIMiner [5] build models with different machine learning algorithms on features, including permissions, API calls, and so on.

III. SVM-based Malware Detection

In this section, we first introduce the overall architecture of proposed malware detection scheme, and then describe each of the functions in details to demonstrate how the proposed scheme works for malware detection. Figure 1 depicts the overall structure of the malware detection scheme.

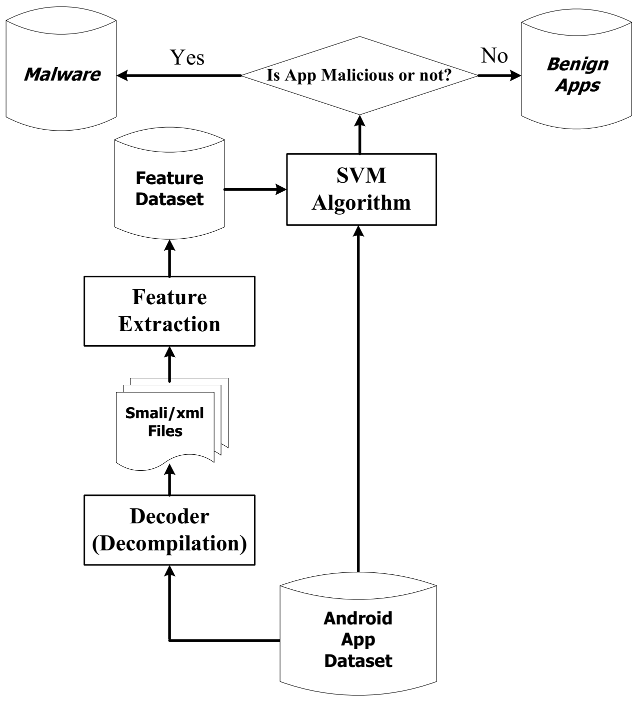
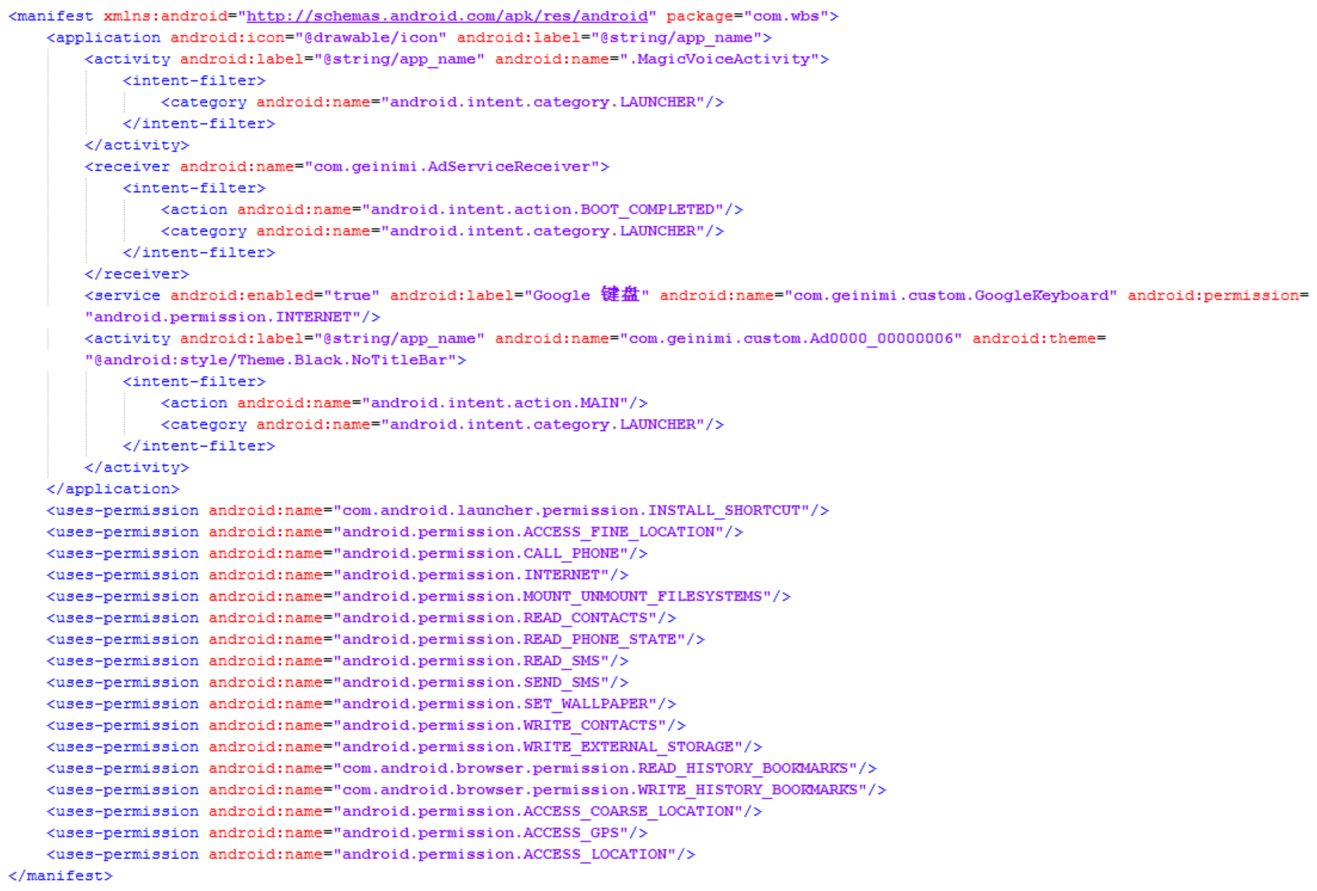


Fig. 1. The schematic diagram of SVM-based malware detection



```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.wbs">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:label="@string/app_name" android:name=".MagicVoiceActivity">
            <intent-filter>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <receiver android:name="com.geinimi.AdServiceReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </receiver>
        <service android:enabled="true" android:label="Google 键盘" android:name="com.geinimi.custom.GoogleKeyboard" android:permission=
        "android.permission.INTERNET"/>
        <activity android:label="@string/app_name" android:name="com.geinimi.custom.Ad0000_00000006" android:theme=
        "@android:style/Theme.Black.NoTitleBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.SET_WALLPAPER"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS"/>
    <uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_GPS"/>
    <uses-permission android:name="android.permission.ACCESS_LOCATION"/>
</manifest>
```

Fig 2. An example of xml file extracted from Android apk package

A. *Schematic Overview*

As is shown in Fig. 1, there are three major components in the malware detection scheme, namely decoder (decompilation), feature extraction, and classifier. In the decompilation component, each Android app is unpacked and decoded into a readable smali file. Some key features, such as risky permissions, suspicious API calls, and URLs are then extracted in feature extraction components according to several important and widely accepted measures, such as TF-IDF and cosine similarity. Finally, we use machine learning algorithm to build a classification model and evaluate them on the Android app dataset by classifying them into malware or benign apps.

B. *Android App Decompilation*

Android apps are packed into *apk* format, and the features we are interested in are encrypted in the *apk* file, such as permissions, APIs, actions, IP and URLs. To extract these features, we implement a decoder based on an open source recompilation tool [10], which unpacks apps to readable xml or smali files. Figure 2 demonstrates a sample xml file that is extracted from an apk file.

From Figure 2 we can clearly find that this Android app requests for several permissions from the Android OS, such as *android.permission.CALL_PHONE*, etc. The following are some examples of dangerous Android permissions that have been widely used by Android malware.

1. *android.permission.CALL_PHONE*: this permission allows an Android app to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed. It is of high danger to grant this permission to an app because a malicious Android app can easily call a premium-rate number (such as a 1-900 number in USA and Canada), and this whole dialing process is automatic and hidden to the mobile phone user.

2. *android.permission.SEND_SMS:* this permission lets an app send an SMS on behalf of the mobile user, and similar to the phone call permission, it could cost you money by sending SMS to for-pay numbers. Certain SMS numbers work much like 1-900 numbers and automatically charge your phone company money when you send them an SMS. Therefore, this permission is also very dangerous.

3. *android.permission.WRITE_EXTERNAL_STORAGE*: when granted this permission, an app can read, write, and delete anything stored on your phone's SD card. Note that this permission is also frequently requested by a lot of legitimate apps, such as camera apps, audio/video apps, and also document processing apps.

Thus, it is very important to use additional evidences (such as the combination of other dangerous permissions or API calls) to further distinguish malicious apps from benign ones because both of them can request for this permission.

4. *android.permission.ACCESS_LOCATION*: this makes the current location information accessible to the app, which may introduce potential privacy leakage when the location information is disclosed to unwanted third-party without the mobile user's consent. However, many benign apps may also ask for the permission to access your current location, such as social network apps, service (such as restaurant) recommendation apps, etc.
5. *android.permission.READ_CONTACTS*: this let an app read the mobile user's contact data. This may also be dangerous because of the privacy concerns when a mobile user's contact data (including name, phone number, email address, etc.) are accessed and even shared without the user's awareness and approval. Sometimes, benign apps, such as social networking apps and contact management apps may also request for this permission.

From the description of various dangerous Android permissions, we clearly find that many of those permissions can be used by both malicious apps and benign ones. Therefore, it is very important to combine multiple dangerous permission requests as well as other evidences such as API calls to identify malware in an accurate and efficient manner.

C. Weight Calculation Using TF-IDF

It is well known that most apps on Android platform need to call various APIs to fulfill their functionality. Once we decompile Android apps into smali files, we then further analyze the smali files, each of which contains a list of dangerous API calls (such as *SmsManager.sendTextMessage* or *WifiManager.setWifiEnabled*) that each app uses. Then, we analyze the smali files and calculate the weight of every dangerous API in the feature vector using the TF-IDF (Term Frequency-Inverse Document Frequency) algorithm, which is described as follows.

Let $N_{i,j}$ be the number of times that an Android app D_j calls a specific dangerous API$_i$. M_j is the total number of times that D_j calls all different dangerous APIs. Then TF$_{i,j}$ is defined in the following equation (1).

$$TF_{i,j} = \frac{N_{i,j}}{M_j} \qquad (1)$$

Also note that D is the total number of malware in the training dataset. D_i is the number of times that a certain dangerous API is called. Then IDF$_i$ is defined as in (2).

$$IDF_i = \log \frac{D}{D_i + 1} \qquad (2)$$

Finally the weight equation is defined in (3).

$$W_i = TF_{i,j} \times IDF_i \qquad (3)$$

D. Determining Similarlity Among Android Apps

In an Android app D with n feature items, the weight of every feature item is equally important. Suppose $D = D(W_1, W_2, \ldots, W_n)$, in which W_i is the weight of i-th dangerous API call by the Android app D. Each weight value of feature items is calculated based on the TF-IDF algorithm that we discussed in Section III.C.

Now let us assume that there are two Android apps A and B that can be represented as follows: $A = A(A_1,A_2,\ldots.A_n)$ and $B = B(B_1,B_2,\ldots,B_n)$. Here, A_i and B_j are the weighted i-th and j-th dangerous API calls for Android app A or B, respectively. Then, the similarity between app A and app B is calculated as in equation (4).

$$Sim(A,B) = \cos\theta = \frac{\sum_{i=1}^{n} A_i \times B_i}{\left(\sqrt{\sum_{i=1}^{n}(A_i)^2}\right) \times \left(\sqrt{\sum_{i=1}^{n}(B_i)^2}\right)} \qquad (4)$$

The closer the value of $Sim(A, B)$ is to 1, the more similar these two apps A and B are. By this means, we can identify whether or not an unknown Android app is malicious by checking to see if it is more similar to the existing dataset of malware or to benign apps.

For instance, let us assume that there are three Android apps, namely P, S, and K, that we want to analyze and find out the similarity among each of them. We first should calculate the number of times each Android app uses the corresponding API$_1$ through API$_7$, which is shown in Table 1 below.

Table 1. Similarity Analysis for Android app P, S, and K

		API$_1$	API$_2$	API$_3$	API$_4$	API$_5$	API$_6$	API$_7$
P	N$_p$	1	2	11	1	10	15	23
	TF$_p$	0.016	0.032	0.175	0.016	0.159	0.238	0.365
	IDF$_p$	0.959	0.796	0.509	0.959	0.678	0.509	0.387
	W$_p$	**0.015**	**0.025**	**0.089**	**0.015**	**0.108**	**0.121**	**0.141**
S	N$_s$	1	2	10	2	10	13	24
	TF$_s$	0.016	0.032	0.161	0.032	0.161	0.210	0.387
	IDF$_s$	0.959	0.796	0.509	0.959	0.678	0.509	0.387
	W$_s$	**0.015**	**0.025**	**0.082**	**0.031**	**0.109**	**0.107**	**0.150**
K	N$_k$	0	0	0	0	1	0	0
	TF$_k$	0	0	0	0	1	0	0
	IDF$_k$	0.959	0.796	0.509	0.959	0.678	0.509	0.387
	W$_k$	**0**	**0**	**0**	**0**	**0.678**	**0**	**0**

Then, we compute the corresponding TF and IDF values based on the number of API calls, and finally the features for each Android app can be represented as follows:

$$P = P(0.015, 0.025, 0.089, 0.015, 0.108, 0.121, 0.141)$$
$$S = S(0.015, 0.025, 0.082, 0.031, 0.109, 0.107, 0.150)$$
$$K = K(0, 0, 0, 0, 0.678, 0, 0)$$

Finally, we can derive the similarity among each of *P*, *S*, and *K* according to formula (4).

$$Sim(P, S) = 0.995$$
$$Sim(P, K) = 0.455$$
$$Sim(S, K) = 0.456$$

From the above similarity scores, we can easily find that the app *P* and *S* are very similar because the similarity score between them is very close to 1, whereas the app *P* and *K* and the app *S* and *K* are not that similar.

E. *Identifying Android Malware Using SVM Algorithm*

In this component, we use Support Vector Machine (SVM) algorithm to classify Android apps, and identify malicious ones (malware). SVM has been widely used in many research areas, such as text classification and image processing. Here, we model the typical features as a raw feature vector. The feature vector is defined as follows:

$$FV = \{f_1, f_2, \cdots, f_n\} \tag{5}$$

$$FV^{app} = \{f_1^{app}, f_2^{app}, \cdots, f_m^{app}\} \tag{6}$$

In equation (5), *FV* represents possible features from all different Android apps. On the other hand, FV^{app} stands for the features for a given Android *app*. In addition, *n* is the size of all possible features, whereas *m* is the size of the feature space for a given Android *app*.

Based on equation (5) and (6), we define the feature vector of an Android app as in equation (7).

$$V = \{v_1, v_2, \cdots, v_i\}, v_i = \begin{cases} 1 \text{ if } \mathrm{f_i} \in FV \text{ and } \mathrm{f_i^{app}} \in FV^{app} \\ 0 \quad\quad\quad\quad\quad\quad \text{otherwise} \end{cases} \tag{7}$$

Thus a feature vector can be translated into V ={1, 0, 0, 1, 1, 1, …}, in which 1 indicates that the feature is contained in this app, whereas 0 indicates not. We can then formulate the feature vector V for each Android app, and take all these features as training dataset to train an SVM classifier, which can be then used to identify unknown Android malware from a different testing dataset of Android apps.

IV. PERFORMANCE EVALUATION

To validate the malware detection scheme, we conducted some experiments, and the results show that the scheme is able to accurately identify malware in Android platform.

A. *Dataset*

In our experiments, we utilized the Drebin dataset [3] as the source of Android malware. Moreover, we also download benign apps from Google play for comparison purpose. In the experiments, we calculated the similarity score in terms of the 31 dangerous API calls that we identified between 400 Android apps (in which 200 are benign, and 200 are malware) for the training stage of SVM model.

In addition, based on the scanning results of the same 400 Android apps, we found that in these 400 applications, 180 of them have some particular sets of risky permissions. Thus, we chose to use the risky permission sets as the additional features in the feature vector for SVM model..

B. *Experimental Results*

In our experiments, we first used the ground truth set of 400 Android apps, including 200 benign apps from official Android market and 200 malware from Drebin dataset, to train a SVM model on Matlab. Then, we used the trained SVM model to test a new unlabeled set of 300 Android apps, which contains 150 benign apps and 150 malware. Figure 3 shows the SVM classification result, and the accuracy of the SVM classifier is listed in Table 2.

Table 2. Accuracy of SVM-based Malware Detection

Features Used	**Accuracy of SVM Classifier**
Dangerous API Calls ONLY	81%
Dangerous API Calls AND Risky Permission Combination	86%

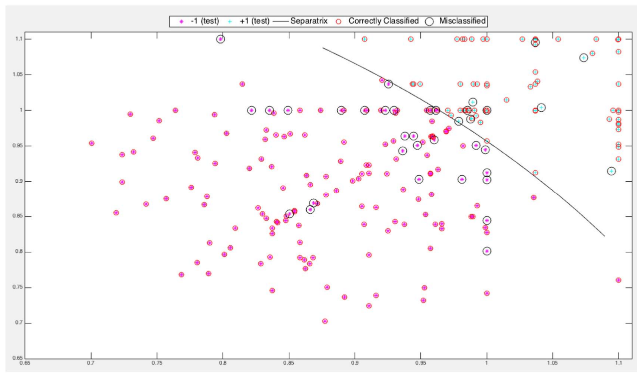


Fig. 3. SVM Classification Result with Testing Dataset

Note: Big circle represents wrong results, and small represents correct results

V. CONCLUSION

In this work, we propose a SVM-based malware detection scheme for Android platform, and use both dangerous API calls and risky permission combinations as features to build an SVM classifier, which can automatically distinguish malicious Android apps (malware) from legitimate ones. Experiment results show that the proposed scheme is able to identify malware in an accurate manner.

References

[1] Mobile threat report, F-Secure, https://www.fsecure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf, visited on August 15, 2015.

[2] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (IEEE S&P 2012)*, pages 95–109. IEEE, 2012.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of *2014 Network and Distributed System Security Symposium* (*NDSS 2014*), February 2014.

[4] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In Proceedings of *Seventh Asia Joint Conference on Information Security* (*Asia JCIS 2012*), pages 62–69. IEEE, 2012..

[5] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android, in Proceedings of *9th International ICST Conference on Security and Privacy in Communication Networks* (*SecureComm 2013*), pages 86–103. Sydney, Australia, September 2013..

[6] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, volume 2, USENIX Association, Berkeley, CA, USA, 21-21, 2011.

[7] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security* (ACM CCS '09)., Chicago, IL, USA, 235-245.

[8] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (ACM CCS '11)., Chicago, IL, USA, 627-638.

[9] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (ACM MobiSys '12)., Low Wood Bay, Lake District, United Kingdom, 281-294.

[10] J. Freke. An assembler/disassembler for android's dex format. Google Code, http://code.google.com/p/smali/, visited August 2015.

[11] A. Desnos and G. Gueguen, et al. https://github.com/androguard/androguard, visited August 2015.

[12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer Berlin Heidelberg, 2012.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." In *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259-269. ACM, 2014.

[14] Zhemin Yang, and Min Yang. "Leakminer: Detect information leakage on android with static taint analysis." In *Software Engineering (WCSE), 2012 Third World Congress on*, pp. 101-104. IEEE, 2012.

[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems (TOCS)* 32, no. 2 (2014): 5.

[16] Lok-Kwong Yan, and Heng Yin. "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis." In *Proceedings of USENIX security symposium*, pp. 569-584. 2012.

[17] Vaibhav Rastogi, Yan Chen, and William Enck. "AppsPlayground: automatic security analysis of smartphone applications." In *Proceedings of the third ACM conference on Data and application security and privacy*, pp. 209-220. ACM, 2013.

[18] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In Proceedings of *2012 Network and Distributed System Security Symposium (NDSS 2012)*, February 2012.