Malicious Application Dynamic Detection in Real-Time API Analysis

Shiting Xu, Yuandong Liu, Yanhui Guo
BUPT National Engineering Lab for Mobile Network Technologies
Beijing University of Posts and Telecommunications
xvshiting@live.com, cyrus.cl@outlook.com
yhguo@bupt.edu.cn

***Abstract*—There are various malicious applications (app) in mobile platform, especially for Android devices, it is difficult to develop a model directly for malwares, due to the limitation of application testing samples. In this paper we propose a novel malicious application detection model RT-MAD for Android devices: Real-Time Malicious Application Detection. This model can generate a malicious app space through normal application modeling by (i) first we develop an Android Real-time API monitor tool to collect API data for each app running on the devices, and cleaning them into time series data, (ii) then we modify Hidden Markov Model (HMM) to train the majority genres of normal apps, obtaining the normal apps space, (iii) and finally we use Randomized Real-Valued Negative Selection (RRNS) to generate a set of likelihood vectors based on the normal app space, covering all possible malicious applications, thus we get the malicious app space for malwares detection. We conduct experiments on HMM training and RRNS malicious apps space generation, the result shows that we can get precision of 91% for normal genres of apps in HMM model. However, in some situation, the malicious apps space generated in RRNS would cover the normal apps, for the safety of devices, it is acceptable since our RT-MAD can achieve precision of 96% in malwares detection.**

***Keywords*—Android Malware Dynamic Detection, API Monitor, HMM, RRNS**

I. Introduction

In recent years, there has been a gradual improvement in smartphone adoption. According to IDC [1], Android owned 82.8% of the global smartphone market in 2015 Q2. It also dominated the smartphone market with 84.8% in 2014 Q2. At the same time the number of malware is also increasing, it can cause adverse effect on user's daily life. Although there are a number of ways to ~~help~~ distinguish between normal and abnormal applications, but how to detect malicious applications accurately and efficiently is still an open question [3].

Android malware detection methods are mainly divided into static analysis and dynamic detections. Machine learning methods are widely used in both of them. [2], the difference between static analysis and dynamic detections is the different information collected, which are used as identifying features in detection models. For static analysis， the approaches are usually focused on permission requests called by apps [4], and there are some other methods use both permission and API calling as features [5]. Semantics-based detection method [7] also widely used in static analysis. The advantage of static detection is high efficiency but when the application's developers adopt the technology of obscured or anti-unpack this method would be invalid. With respect to dynamic detection, in our previous work, we developed a method to detect malicious apps by collecting behaviors data of applications running on devices [8]. Generally, the dynamic data refers to API invoking, mobile data connecting and memory consuming etc. In recent literatures, machine Learning algorithms have been used in malware detection [6] include: SVM [10] (Support Vector Machine, SVM), NBM (Naive Bayesian Model, NBM), GBDT (Gradsaient Boost Decision Tree, GBDT), Decision Tree or ensemble learning method [11] etc.

There has been many research on the Real-time dynamic detection. Iker Burguera and Urko Zurutuza present a Behavior-Based Malware Detection System for Android named Cro-wdroid [12]. In this system they use a tool available in Linux called Strace to collect the system calls and then use a simple 2-means clustering algorithm to distinguish between normal application and abnormal application. The most important contribution of this work [12] is the mechanism they propose for obtaining real traces of application behavior. Luoxu Min proposes a runtime-based behavior dynamic analysis detection method [13]. In this method Android application run on the emulator to generate the run-time log file. Then they use the sematic analysis and regular expression technology to analyze the filtered log file. Gerardo Canfora's approach take account all the system calls and they also consider sequence of system call [14]. Xiao Xi presents an approach for detecting Android malware with system call sequences based on Markov chains and Back-propagation neural network [15]. Dong Hang's method [16] is the first time to adopt HMM (Hidden Markov Model, HMM) in dynamic detection. In his approach real-time network's and memory's information has been used to build feature. Y Wei capture the behavior of software, then use machine learning method to learn the dynamic behavior of malwares[17]. However, all dynamic detection methods from above literatures were try to construct detection model based on malwares, it is difficult to cover all malicious space due to the inadequate of malicious application.

In this paper, we propose a novel dynamic detection model named RT-MAD (Real-Time Malicious Application Detection, RT-MAD), which can use real-time API data of an application to detect Android malware efficiently and accurately. Instead of study the malicious applications directly, RT-MAD modeling the normal applications that frequently used by people, and generates the abnormal space for malicious application detection. Fig 1 illustrates the framework of our model.

RT-MAD model consists of five modules, in real-time API data collection module, we develop a tool to collect run-time API data of applications, and clean them in data processing module, generating time sequence data. HMM features space training module has two components, the one is to build HMM modules with time sequence data of typical applications, and the other component is, by using HMM models we built respectively, to compute likelihood vectors between applications and the typical applications. Following, we adopt RRNS [18] (Randomized Real-Valued Negative Selection, RRNS) in auto-produce malicious vector module to construct a malicious vector database. Finally, in detection module, we calculate the minimum Euclidean distance between test application's likelihood vector and the malicious vector database, identifying the current application is a malware or not.

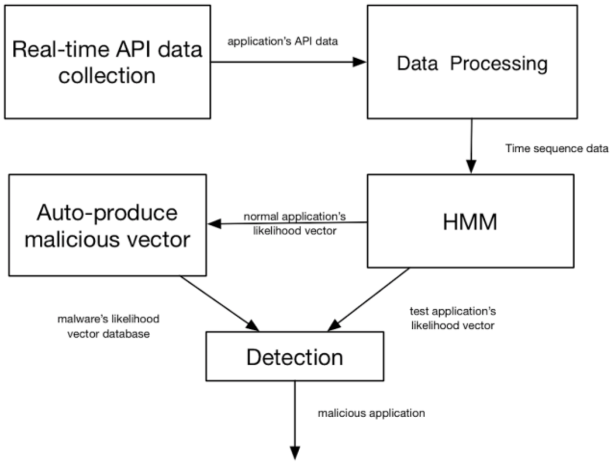

Real-time API data collection
application's API data
Data Processing
Time sequence data
HMM
normal application's likelihood vector
Auto-produce malicious vector
malware's likelihood vector database
test application's likelihood vector
Detection
malicious application

Fig. 1.   Framework of RT-MAD model

The remainder of this paper is structured as follows. Section II introduces our model in details. In Section III, we present experiments and evaluation of our model. Finally, related works are discussed in Section IV and we conclude our study in Section V.

II. RT-MAD Modeling

Our RT- MAD model consists of three parts: Data Pre-processing, HMM Features Space Training and Malicious Application Detection. For the data pre-processing, we firstly develop a tool to dynamically collect all the API data invoked by applications running in the device, and then we write python scripts to cleaning all the data into a specific form. In HMM Features Space Training component, we build HMM models based on the data obtained in the first part, and compute the likelihood vectors between applications and the typical applications. In the Malicious Application Detection part, we demonstrate how to use RRNS algorithm to generate malicious applications vectors and malwares detection according to its vector space.

A. Data Pre-processing

Our model is trained by API data of mobile applications, in order to study the application behaviors, especially for the normal applications, API invoking is a good way to observe details of the application actions when it is running. However, those API data are usually invisible in the operation system，we develop a tool to record the API invoking and its parameter value, and then sparse log file by time tags.

1) Real-time API data collection

We already have some research experiences [22] on tracing real-time API calling data on Android device with an open source project Xposed framework[1]. Based on this project, we develop an Android API monitor[2], detecting and recording real-time API data. The structure of this module is manifested in Fig 2.

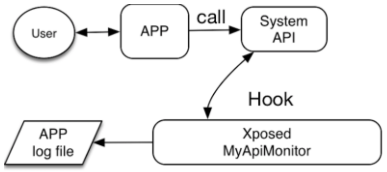


Fig. 2. Real-time API data collection component

When user interact with an application, the app will invoke some system API. Our API monitor can hook system API, insert our code and get the inserted code executed each time when the API invoked by the application. After we hooked an API, each time an application invokes it we can run our own code to modify and record the parameters, and then output the data into a log file.

The length of time to collect data denoted as $T_{collect}$, counting in seconds. We denote the set of monitored API as S_{API} and the number of APIs in S_{API} as n. We analyze all the APIs and choose 38 most representative APIs, adding them into S_{API}. Android API monitor can record all the calling of APIs in S_{API} for $T_{collect}$ seconds in log files. Each record in log files consists of time stamp, package name and calling API information etc., log files are sent to our local server through http protocol.

2) Time series data production

The structure of this part is illustrated in Fig 3. The input of this part is log files of applications. Time series data production consists log files slicing by time and API frequency counting. The output of this component is An m× n Matric M_t.

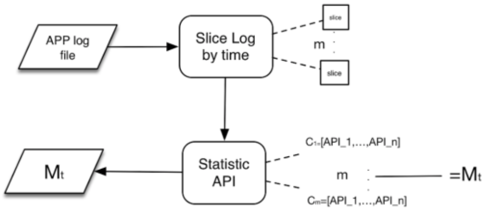


Fig. 3. Structure of Data Processing

In log files slicing process we split log file into m slices by time interval denoted as $\Delta t. m = T_{collect}/\Delta t$. Time stamp of each record in log file makes the slice processing much easier. In API frequency counting process, we compute the frequency of each API in S_{API} respectively. The $i-th$ slice will generate a vector $C_i, i = 1, \dots, m$. The output M_t is

[1] https://github.com/rovo89/Xposed
[2] The code can be found here: https://github.com/donggobler/Sensitive_API_Monitor

composed of all C_i. The pseudo-code of generating M_t is given in Table I.

$$map_{API}(N) = \begin{cases} 0 & N = 0 \\ 1 & N \in (0, \delta_{API}] \\ 2 & N \in (\delta_{API}, +\infty] \end{cases} \quad (1)$$

For each API in S_{API} has a unique mapping function map_{API} . The value of $map_{API}(N) = 0$ means this API does not occur in the period we monitored, 1 means this API appear but in a normal frequency, and we denote the line of the normal number as δ_{API}. Excessive invoked of an API can cause the value of $map_{API}(N) = 2$. δ_{API} in (1) is different for each mapping function.

TABLE I
DATA PROCESSING ALGORITHM

```
Input(logfile)
 Init Mt=[]
 Split logfile into m slices
 For each slice in slices:
     C=[]
     For each API in SAPI:
         Num=0
         For each record in slice:
             If record contains API:
                 Num=Num+1
             End If
         Num=mapAPI(Num)    -------Eq(1)
         C= C.append(Num)
     Mt.append(C)
Return Mt
```

B. HMM Features Space Training

According to API invoking data, we describe an application in four states: normal, slight malicious, malicious, serious malicious. In this situation, an application can be a combination of four states in specific time. For example, with time goes by, the state of an application would transform from normal to slight malicious and then, in next period, back to normal again. Applications from same typical class have similar state transformation, however, we can't observe four states of transformation directly. In a hidden Markov model, the sequence of tokens generated by an HMM gives some information about the sequence of states. Therefore, by using HMM, we can deduce application's hidden states from its API invoking data, and we can also compute similarity between an application and a typical class of applications.

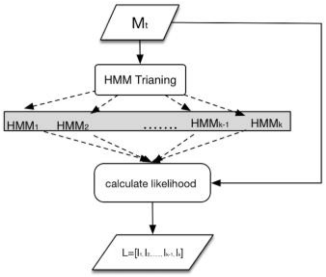


Fig. 4. Structure of HMM Features Space Training

HMM Features Space Training model consists of HMM training and likelihood vectors computing units. The HMM training unit uses time series data $M_t$and Baum-Welch[20] algorithm to build HMM[19]. The likelihood vectors computing unit uses forward algorithm [19] to calculate likelihood vector. The structure of HMM Features Space Training module is given in Fig 4.

We use the compact notation $\lambda_t = \langle S, V, A_t, B_t, \Pi_t \rangle$ to indicate the complete parameter set of the HMM_t,t ∈ [1, k]. The meaning of each notation is given in Table II. All HMM have the same S and V.

TABLE II
NOTATION OF HMM

$\lambda_t = \langle S, V, A_t, B_t, \Pi_t \rangle$	
S:	a set of hidden states of HMM
V:	possible observed result set $V = \{0,1,2\}^n$
A_t:	State transition probability matrix HMM_t
B_t:	Observation probability matrix HMM_t
Π_t:	Initial probability matrix of HMM_t

One application from an app store has always been labeled with one or more classes. We have surveyed abundant app stores and choose K classes as fundamental classes (FC). Every fundamental class has J applications as typical applications (TA).All TA are normal applications.

HMM Features Space Training module can produce K HMM for K FC in HMM Training unit. FC's HMM is built with M_t of all J TA belong to it. Every HMM can be described as a kind of application's running behaviors. To build an HMM of a FC, we regard each vector in M_t as observable outputs, which can be characterized as signal produced by application. According to the formula (1), $V = \{0,1,2\}^n$. We set $S = 4$ and use Baum-Welch algorithm to optimize A_t, B_t.

Likelihood vectors computing unit uses application's M_t and K HMM to produce likelihood vector $L = [l_1, l_2, \dots, l_k]$ with forward algorithm. Each element in L represents the degree of similarity between the application and a T. In consideration of likelihood value could be ranged in $(-\infty, +\infty)$, each element in L should be normalized (2). The higher absolute value of likelihood can represent the better similarity between application and TC. In (2) when l_i is close to $\pm\infty$, after normalization, its value close to 1. If $l_i = 0$, after normalization, its value is still zero. After normalization $l_i \in [0,1), i = 1,2, \dots, k$.

$$L = L' = \left\{1 - e^{-|l_i|} \middle| l_i \in L\right\} \quad (2)$$

C. Malicious Application Detection

The malicious application detection procedure consists of auto-produce malicious vector process and application vector testing process. The first process produced malicious likelihood vector using normal application's likelihood vector with RRNS (randomized real-valued negative selection, RRNS). The application vector testing process detecting malicious application.

1) Auto-produce malicious vector

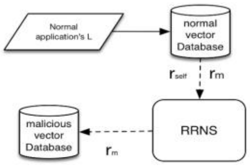


Fig. 5. Auto-produce malicious vector module

The structure of this module is given in Fig . 5. This module saves all normal application's likelihood vector into a database and generate malicious likelihood vector. The algorithm used in auto-produce malicious vector is RRNS.

To construct normal vector database, we need abundant of normal application's likelihood vectors. We assume the number of normal application is num_{normal}. So the size of normal vector database is $num_{normal} \times K$.The data type can be txt、excel or database system.

RRNS algorithm need normal set、r_{self} 、r_m and other parameters [18]. r_{self} means normal application's likelihood vector's radius. r_m means malicious application's likelihood vector's radius. RRNS uses normal vector to estimate volume of malicious space and auto generate a set of malicious sample that cover the malicious space. The effective volume covered by a malicious sample with a radius r_m is define as[18]:

$$V_d = \left(\frac{2r_m}{\sqrt{K}}\right)^k \quad (3)$$

If let $V_{malicious}$ be the volume of malicious space. A rough approximation of the number of malicious vector can be given by [18]:

$$num_{malicious} = \frac{V_{malicious}}{V_d} \quad (4)$$

In (4), we can estimate the number of vector in malicious database and the size of malicious database is $num_{malicious} \times$ K. The flow of RRNS is given in fig 6.

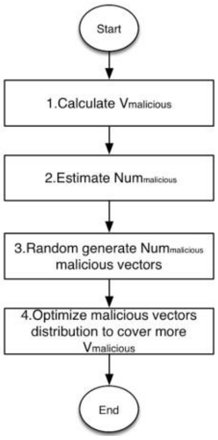

Start
1.Calculate V_malicious
2.Estimate Num_malicious
3.Random generate Num_malicious malicious vectors
4.Optimize malicious vectors distribution to cover more V_malicious
End

Fig. 6. The flow of RRNS

An optimize function in formula (5) is adopted in the step 4 of figure 6:

$$\mathrm{C(D)} = \mathrm{Overlapping(D)} + \beta \cdot \mathrm{SelfCovering(D)} \quad (5)$$

$$\mathrm{Overlapping}(d_i, d_j) = e^{\frac{-\|d_i - d_j\|^2}{r_m^2}} \quad (6)$$

$$\mathrm{Overlapping(D)} = \sum_{i \neq j} e^{\frac{-\|d_i - d_j\|^2}{r_m^2}}, i, j = 1 \ldots num_{malicious} (7)$$

$$\mathrm{NormCovering(D)} = \sum_{s \in S'} \sum_{d \in D} e^{\frac{-\|d-s\|^2}{\left(\frac{r_{self}+r_m}{2}\right)^2}} \quad (8)$$

In(5,6,7,8), D represents malicious vector database, d_i and d_j are vectors in it. S' in our model is normal vector database and s is vector in it. Our goal in step 4 is to get a minimum $\mathrm{C(D)}$. $\mathrm{Overlapping(D)}$ approximate measure of overlapping between different two detectors in D . Minimize $\mathrm{Overlapping(D)}$ can larger the distance between two vectors in D. As the same, minimize $\mathrm{NormCovering(D)}$ can also larger distance between malicious vector and normal vector. Parameter β can adjust or balance this two function's weight.

2) Application Vector Testing

In this process, minimum Euclidean distance between testing application and malicious database is calculated and we compare the minimum Euclidean distance and radius of malicious database to detect malicious applications. The detection algorithm is presented in Table III, and the application vector testing module is given in Fig 7.

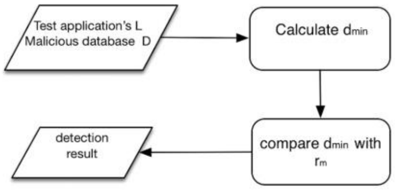


Fig. 7. The structure of Application Vector Testing

By using likelihood vector L and malicious vector database, we can judge the software if it is a malware or not. We denote d_{min} as minimum Euclidean distance between L and D . Compare the value of d_{min} and r_m to get an output. If the output is 1，it represents the test application is malware and 0 means the application is normal.

TABLE III
DETECTION ALGORITHM

```
Input(L,D), $r_{ab}$):
    $d_{min} = +\infty$
    For each  d in D:
        If  $d_{min}$> Euclidean(L,d):
            $d_{min}$= Euclidean(L,d)
        End If
    If $d_{min}$>$r_{ab}$:
        Output(0)
    Else
        Output(1)
    End If
End
```

III. EMPIRICAL EVALUATION

We use 3600 applications in this experiment. We download 3000 normal application from Google Play[3] and other app store. We collect 500 malicious applications in our usual work, and choose 20 typical classes (TABLE IV), each typical class includes download 5 typical applications.

TABLE IV
20 TYPICAL CLASSES OF APPLICATION

Music	Photo	News	Games
Education	Books	Health	Video
productivity	Shopping	Social	travel
weather	business	Finance	Kids
Food	Sports	Entertainment	Utilities

We selected 38 important system API of android to be S_{API} in our Xposed module. APIs we studied here cover behaviors of network, file, database, camera, contact, message, call and

[3] https://play.google.com

media etc. In our experiment, we set $T_{collect} = 600$, $\Delta t = 10$, and then we implement our own RRNS algorithm in Python[4].

A. RT-MDA Data Preprocessing

We use 5 android devices run our application simultaneously. Each time we run single application on one devices. To collect the data comprehensively, we have 5 users interact with those our testing mobile phones respectively.

For typical applications, each user run them once, as for other 3500 applications, we execute one time by a specific user from 5 users. The details are showed in fig 8.

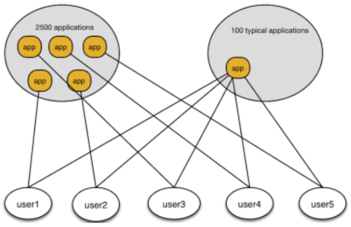


Fig. 8. Difference between typical and other application

3600 applications are divided into three parts according to their function. The partition is given in Fig 9.

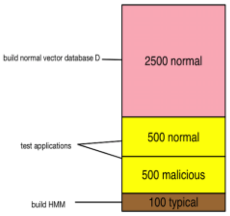
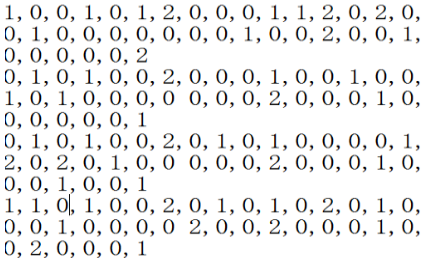


Fig. 9 The partition of 2600 application

2500 of 3000 normal applications are used to build normal vector database D . 500 normal and 500 malicious applications are used to be our test application. 100 typical applications are used to build our HMM. We save all log files into our local server.

We process all log files into time series data and save them into txt file. The time series data's form is given in Fig 10. Every application's log file will be ended up in this form. Each row in Fig 10 represents a Statistical results of an application behaviors in 10 seconds. Every application has 60 rows in their time series data file. Adjacent rows represent adjacent time periods. Each element of one row reflect one API appear frequency (show up times after mapping function).

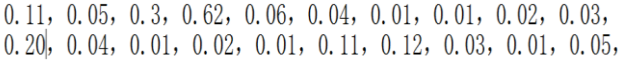


```
1, 0, 0, 1, 0, 1, 2, 0, 0, 0, 1, 1, 2, 0, 2, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 1,
0, 0, 0, 0, 0, 2
0, 1, 0, 1, 0, 0, 2, 0, 0, 0, 1, 0, 0, 1, 0, 0,
1, 0, 1, 0, 0, 0, 0  0, 0, 0, 2, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 1
0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 1, 0, 0, 0, 0, 1,
2, 0, 2, 0, 1, 0, 0  0, 0, 0, 2, 0, 0, 0, 1, 0,
0, 0, 1, 0, 0, 1
1, 1, 0, 1, 0, 0, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0,
0, 0, 1, 0, 0, 0, 0  2, 0, 0, 2, 0, 0, 0, 1, 0,
0, 2, 0, 0, 0, 1
```

Fig. 10 Time series data of one application

B. HMM Evaluation

We use all typical application's time series data to build 20 HMMs. Each HMM represent a kind of application's pattern. Hmmlearn [5]package of python is used in this process. This package can build HMM and can also save and load HMM. We set hidden state of our all HMM is 4. We also use hmmlearn package and our HMM to generate original likelihood vector of 2500 applications. The likelihood vector of an application L is given in Fig 11.



```
0.11，0.05，0.3，0.62，0.06，0.04，0.01，0.01，0.02，0.03，
0.20，0.04，0.01，0.02，0.01，0.11，0.12，0.03，0.01，0.05，
```

Fig. 11 . likelihood vector of application

To evaluate our HMM, we use our 20 HMMs to construct a classifier and evaluate its ability of classification. The classification method is that given an application, we calculate its likelihood vector with our HMM, and denote the index of max element in L as its class (9).

$$\text{class} = \{\text{index} | l_{index} \geq l_i , \forall l_i \in L\} \quad (9)$$

We download another 400 applications from app store, each typical class has 20 of them. We labeled them and use our HMM classifier to make a predict. The probability of correct classification of those applications can evaluate classification ability of HMM. Number of correctly classified of each typical class is given in Fig 12.

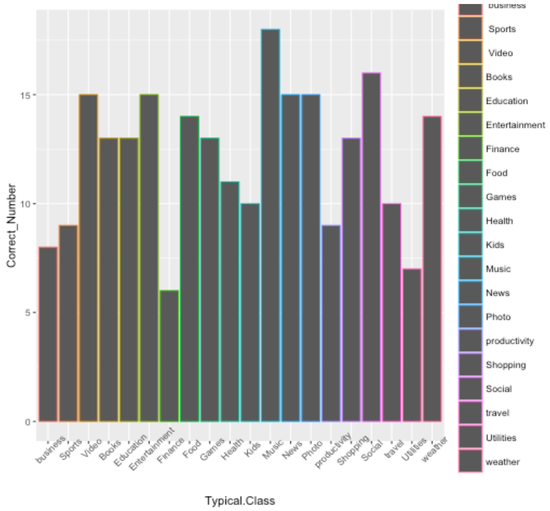


Fig. 12 Correctly classified number of 20 classes

The classification precise rate of our HMM classifier is 64%. For some classes it can up to 85% such as music and games. The results show that our HMM is effective and can be used to generate normal likelihood vector.

C. Evaluation of Malicious Vector Space Generating

We use RRNS and 1500 normalized likelihood vector of normal applications to produce our malicious likelihood vector space. We choose multi group parameters(r_m, r_{ab}) in RRNS . We generate many different malicious likelihood vector spaces according to different parameters. We use algorithm in table III to classify labeled test application. According to predict result, we use precision and recall to evaluate the classification effect of those Malicious vector space.

[4] https://github.com/xvshiting/RRNS

[5] https://pypi.python.org/pypi/hmmlearn

Precision is also referred to as positive predictive value. In classification task, the precision for a class is the number of true positives divided by the total number of elements labeled as belonging to the positive class. Its formula is given below (10).

$$\text{Precision} = \frac{TP}{TP + FP} \quad (10)$$

Recall is also referred to as the true positive rate or sensitivity. The recall for a class is number of true positives divided by the total number of elements that actually belong to the positive class(true positives and false negatives). The formula is given below (11).

$$\text{Recall} = \frac{TP}{TP + FN} \quad (11)$$

The precision and recall of different r_m and r_{ab} is showing in table V. Our model gets good precision and recall rate 92% and 90% respectively when its $r_m = 0.3, r_{ab} = 0.4$.

TABLE V
PRECISION AND RECALL OF DIFFERENT PARAMETERS

Parameter Group Number	r_m	r_{ab}	PRECISION	RECALL
1	0.3	0.4	92%	90%
2	0.4	0.4	85%	89%
3	0.5	0.3	87%	86%
4	0.5	0.2	90%	84%

In paper [21], the author proposes a method use RNS (Real-Valued Negative Selection, RNS) and static features to generate malicious vector space. Its detection effect is given in Table VI.

TABLE VI
PAPER [21] DETECTION EFFECT

	Error rate	Correct rate
normal	9.5%	90.5%
malicious	9%	91%
total	9.2%	90.8%

Paper [16] uses HMM and SVM to detect malware and its result is given in Table VII.

TABLE VII
PAPER [16] DETECTION EFFECT

	Recall	False positive rate
Total	90%	13%

Compare our result with those two previous works, we find out that our model has a better performance in malicious application detection. Contrast with the approach in paper [16] ，our method do not need malwares, which solve the difficulty of collecting malicious applications.

IV. RELATED WORK

In this section we introduce some existing methods of malicious application detection [16,21] and an approach of monitor application behavior [22], as well as an optimized RNS algorithms [18], analyzing how our model distinguishes the previous work.

Monitor system calling is an important problem in our method. In paper [22], the author has introduced a method using Xposed to monitor application's malicious behavior. In this paper, the author use MonkeyRunner[6] to achieve the goal of auto install, uninstall, click application. They also give us the solution of use Xposed to monitor system API and record those calling information into a log file. They execute an experiment to evaluate their model. In their experiment, they find out almost all sensitive API calling behavior of 1000 applications.

Using HMM in malicious application detection is novel approach. The author of paper [16] proposed an approach of how to combine dynamic behavior of android and HMM to detect malicious application. In tthis paper, the author tries to build a model for each fundamental behavior such as network and memory. They also use HMM to get likelihood vector of an application. Then the author use labeled data to train a SVM. The trained SVM is used to classify normal and malicious application. This HMM-SVM detection model got a 90% recall and 13% false positive rate. This paper gives us a hint on using dynamic behavior. But the disadvantage of this method is that it need abundant malwares. It's difficult for us to get newest malicious application database with the android system update.

To address inadequate malwares problem. In paper [21], they present an approach of using normal application to detect malicious behavior. The main idea of this paper is the RNS algorithm. RNS is widely used in abnormal behavior detection area. They collect static features of normal application and filter out useless features to get normal vectors. Then they use RNS and normal vector to produce malicious vectors. The minimize distance is calculated between vector of test application and all malicious vectors. At last, they compare minimize distance with the radius of malicious vector to make a classification. They also get a better correct rate 90.8% and low error rate 9.2%. Using normal application to detect malware is a novel approach and we also use this method.

We use an optimized RNS algorithm named RRNS. In paper [18], they propose RRNS and introduce the detail of this algorithm. RRNS can make a good estimate of optimal number of detector to cover malicious space. They proposed method is a randomized algorithm based on Monte Carlo methods. They also compare RRNS with RNS and find the front one has better performance.

V. CONCLUSION

In this paper, we introduce a novel malicious application detection model. The model can collect real-time API data from an Android device, since the malicious application behavior is difficult to be modeled, our method gives a way to model the malicious applications space by studying the normal application behaviors, and these behaviors are presented in API data. The experiment we conducted shows that our detection model can achieve a high precision of typical application recognition, and by analyzing the relation between the test application and typical applications, our malicious application generation algorithm can get a reasonable malwares space, which can be used for malicious application detection. In this approach, we do not need the malicious application sample to train the model, distinguished the previous work. Although in some cases, the model would group a normal application into

[6] http://www.android-doc.com/tools/help/monkeyrunner_concepts.html

the malicious one, for all the malwares, they are correctly identified by our model.

In future, our work will focus on analysis the malicious likelihood vector database generated by RRNS. We will focus on improving the classification accuracy of our HMM. We will consider add more dynamic feature of Android devices into our model such as network and memory.

VI. Acknowledgment

We thank our experiment participants for their time and helpful comments. This work is supported by *National High-tech R&D Program (863 Program) (2015AA017202)*.

References

[1] Smartphone OS Market Share 2015 Q2 [Online]. Available: http://www.idc.com/ prodserv/smartphone-os-market-share.jsp

[2] Sahs, Justin, and L. Khan. "A Machine Learning Approach to Android Malware Detection." Intelligence and Security Informatics Conference IEEE, 2012:141-147.

[3] Zhou, Yajin, and X. Jiang. "Dissecting Android Malware: Characterization and Evolution." *IEEE Symposium on Security & Privacy* IEEE, 2012:95-109.

[4] Huang, Chun Ying, Y. T. Tsai, and C. H. Hsu. *Performance Evaluation on Permission-Based Detection for Android Malware*. *Advances in Intelligent Systems and Applications - Volume 2*. Springer Berlin Heidelberg, 2013:111-120.

[5] Chan, Patrick P. K., and W. K. Song. "Static detection of Android malware by using permissions and API calls." 1(2015):82-87.

[6] Firdausi, Ivan, et al. "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection." *International Conference on Advances in Computing* IEEE Computer Society, 2010:201-203.

[7] Feng, Yu, et al. "Apposcopy: semantics-based detection of Android malware through static analysis." *The, ACM Sigsoft International Symposium* 2014:576-587.

[8] Rhee, Junghwan, et al. "Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory." *Recent Advances in Intrusion Detection, International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings* 2010:178-197.

[9] Channakeshava, Karthik, et al. "High Performance Scalable and Expressive Modeling Environment to Study Mobile Malware in Large Dynamic Networks." *IEEE International Parallel & Distributed Processing Symposium* IEEE Computer Society, 2011:770-781.

[10] Zhao, Min, et al. "AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android." *International Conference* 2011:158-166.

[11] Yerima, S. Y., S. Sezer, and I. Muttik. "High accuracy android malware detection using ensemble learning." *Information Security Iet* 9.6(2015):313-320.

[12] Burguera, Iker, U. Zurutuza, and S. Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for Android." *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* 2011:15-26.

[13] Min, Luo Xu, and Q. H. Cao. "Runtime-Based Behavior Dynamic Analysis System for Android Malware Detection." *Advanced Materials Research* 756-759(2013):2220-2225.

[14] Canfora, Gerardo, et al. "Detecting Android malware using sequences of system calls." *International Workshop on Software Development Lifecycle for Mobile Esec/fse* 2015:13-20.

[15] Xiao, Xi, et al. "Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting Android malware with system call sequences." *Iet Information Security* (2016).

[16] Dong, H., et al. "A detection model of malware behaviors on android." *Journal of Beijing University of Posts and Telecommunications*（2014）.

[17] Wei, Yu, et al. "On behavior-based detection of malware on Android platform." *GLOBECOM 2013 - 2013 IEEE Global Communications Conference* 2013:814-819.

[18] Gonzlez, Fabio, et al. "A Randomized Real-Valued Negative Selection." (2004).

[19] Schuster-Böckler, Benjamin, and A. Bateman. "An Introduction to Hidden Markov Models." Appendix 3.Appendix 3(2007):4 - 16.

[20] Welch, and R. Lloyd. "Hidden Markov Models and the Baum-Welch Algorithm." *IEEE Information Theory Society Newsletter* 53.2(2003):194-211.

[21] XIE Li-xia,ZHAO Bin-bin. “Malware detection of Android system based on benign samples “ *Computer Enginerring and Design* 2016, 37(5).

[22] Wang Sai, Guo Yanhui, Wu Qiuxin, Liu Yuandong.” A detection method of Android application malicious behaviors based on Xposed framework.” (2015).