HNUE JOURNAL OF SCIENCE DOI: 10.18173/2354-1059.2020-0047
Natural Sciences 2020, Volume 65, Issue 10, pp. 49-60
This paper is available online at http://stdb.hnue.edu.vn

A FEATURE REPRESENTATION METHOD BASED ON HETEROGENEOUS INFORMATION NETWORK FOR ANDROID MALWARE DETECTION

Thai Thi Thanh Van, Nguyen Van Phac, Truong Quoc Quan and Le Van Hung
Faculty of Information Technology, Academy of Cryptography Techniques

Abstract. The rapid growth in number, sophistication, and diversity of Android malware poses a great difficulty in extracting and analyzing features and behaviors. The traditional approach, which using only API calls and permissions to extract features, has no longer yielded meaningful results. In this research, we propose a method that utilizes both information about API function calls and the relationships between API functions. First, we represent the relationship between API functions using a heterogeneous information network (HIN). Then, we use the concept of meta-path to extract information features from HIN. Finally, a machine learning algorithm is used to build classification models. Experimental results on a practical dataset of Android applications show that the proposed method gives more reliable results than the existing ones.

Keywords: malware, android, heterogeneous, classification, machine learning.

1. Introduction

Nowadays, the Android operating system has become a popular platform for many smart devices because of its open-source nature and easy-to-use interface. Statistics showed that Android is still the dominating operating system of the global mobile phone market (87.7% - according to IDC 2018). This trend is still to remain until 2021. However, because of this popularity, Android devices have become attractive targets for malware. Hackers exploit Android application features to evade the security and privacy of the device, posing an imminent threat of personal data leaks. These leaks range from user location, contact information, accounts, photos, and furthermore. The severity of damage to smart devices makes it essential to solve the Android malware detection problem [1, 2].

In the past, researchers mainly relied on traditional pattern recognition techniques to solve the malware detection problem. However, machine learning techniques and artificial intelligence have undergone a rapid transformation. Thus, methods regarding the development of systems that automatically detect malware using data mining and machine learning algorithms are gaining the interest of researchers in the field of

Received October 5, 2020. Revised October 22, 2020. Accepted October 29, 2020.
Contact Thai Thi Thanh Van, e-mail address: thanhvan0110@gmail.com

information security. However, one of the biggest problems is how these methods techniques analyze, represent, and extract features. There are two approaches to solve this problem, either by using behavioral analysis techniques or signature analysis techniques [3].

Many behavioral analysis techniques rely on analyzing API calls, permissions, system calls, or other specific markers to extract features of Android applications (apps in short) and to put into training sets for training machine learning models. Specifically, Wu *et al.* used 13 basic features from apps as data to implement the Support Vector Machines (SVMs) algorithm to build prediction models [4]. While other groups extracted permissions and API calls to train and test their malware prediction models [5-7]. Otherwise, Bai *et al.* [8] focused on specific API calls and expressed them with CAG graphs. CAG graphs are then closely inspected against control flow graphs (CFG) to identify malicious behaviors and detect them in suspicious executable files.

In general, the efficiency of these approaches ranges from 85 to 90 percent. With a system using machine learning algorithms, these results are not good enough. That is because the data from practical systems, especially, data about malware are diverse and contain underlying meanings. So, if we only use API calls and permissions to extract features for classification and prediction models, the efficiency of detecting malware will be low. Therefore, in these past few years, researchers spent much attention to find more effective ways to extract features for malware detection. One of the most commonly used models today is the Heterogeneous Information Network (HIN).

HINs are comprised of many different types of objects and edges may contain different meanings [9]. Therefore, mining HINs will help us to look for more insights into network structures. HIN has been used in many different fields, including text data mining, biological data mining, and recently in mining information security data [10]. For example, Yanfang *et al.* [13] used HINs to represent Android malware data. Their networks consist of five types of vertices (application, API, IMEI, manufacturer, signatures) and five types of edges (Apps - APIs, Apps - IMEIs, IMEIs - Manufacturers, Apps - Manufacturer, Apps - Signature). Combined with deep learning neural networks, their system for malicious code detection has achieved an accuracy of 96%.

In this paper, we use behavioral analysis based on API call analysis to predict malware for Android. Similar to the previous studies, we decompile Android applications, then propose a method to extract and represent API calls from the output of the decompilation process. Unlike previous studies, we analyze not only the API calls but also the relationships between API calls, such as the ones that are in the same block or the same package. We hope the relationship between API calls will provide additional information that is useful for malware detection. There will be more than one type of relations between API calls and more than one type of object. Therefore, HINs are perfect to describe the relationship between API calls and the overall applications. From HINs, we will construct meta-paths to represent the relationship between applications through the vertices and edges. Based on these meta-paths, we will build feature vectors to represent Android applications. The problem of malware detection becomes a binary classification. We then use several general machine learning models to train and test the accuracy of the proposed method.

2. Content

2.1. Basic concepts related to the heterogeneous information network

In this section, we will go through some of the basics about HINs, meta-paths, and similarity measures on HINs.

Definition 1. *Information network: An information network is defined as a directed graph* $G = (V, E)$ *with an object type mapping function,* $\varphi: V \to A$*, and a link type mapping function* $\psi: E \to R$*. Each object* $v \in V$ *belongs to one particular object type in the object type set* $A: \varphi(v) \in A$ *and each link* $e \in E$ *belongs to a particular relation type in the relation type set R:* $\psi(e) \in R$*. An information network is heterogeneous when the number of elements in set* $A > 1$ *and in set* $R > 1$ *[14].*

Definition 2. *Network schema: A network schema is presented as a* $TG = (A, R)$ *is a directed graph with each vertex is a type of entity in A and each edge is a type of relation in R [15]*

Each type of relation R between object S and T is written as $S \xrightarrow{R} T$, in which S is called a source and T is called a target. They can be notated as R_oS or R_oT, respectively. The inverse relation R^{-1} is defined as $T \xrightarrow{R^{-1}} S$.

Definition 3. *Meta-path: A meta-path P is a path between nodes in a network schema graph* $TG = (A, R)$*, and is notated like a chain:* $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_l} A_{l+1}$*, defining relation* $R = R_1 \circ R_2 \circ \dots \circ R_l$ *between a set of objects* $A_1, A_2, \dots, A_{l+1}$*. (°) is defined as an operator that signifies this particular relation and l is defined as the length of path P [16].*

Definition 4. *Similarity measure AvgSim: Let P be a meta-path* $P = (R_1 \circ R_2 \circ \dots \circ R_k)$*, the AvgSim between source element s and target element t is:*

$$AvgSim(s, t|P) = \frac{1}{2}[RW(s, t|P) + RW(t, s|P^{-1})]$$

while:

$$RW(s, t|R_1 \circ R_2 \circ \dots \circ R_k) = \frac{1}{|O(s|R_1)|} \sum_{i=1}^{|O(s|R_1)|} RW(O_i(s|R_1, t|R_2 \circ \dots \circ R_k)$$

Here, $O(s|R1)$ *is an out-neighbor of s based on relation* R_1 *[17].*

Definition 5. *Let* $G = (V, E)$ *be a network and* TG *be a network schema.* M_p *is called a similarity measure matrix according to meta-path* $P = (A_1, A_2, \dots, A_{l+1})$*, and* $M_P[i, j] = AvgSim(A_i, A_j|P)$*, with* A_i *being the source element, and* A_j *being the target element.*

2.2. The proposed method

First, we decompile *.apk* files to obtain smali codes. From smali codes, we will extract API calls and the relationships between API calls to build a HIN. On this heterogeneous network, we calculate the similarity measures from meta-paths to extract features for each Android application (app in short). Finally, we use a machine learning

algorithm for training and testing prediction models. An overview of our proposed method is described through the following steps (Figure 1).

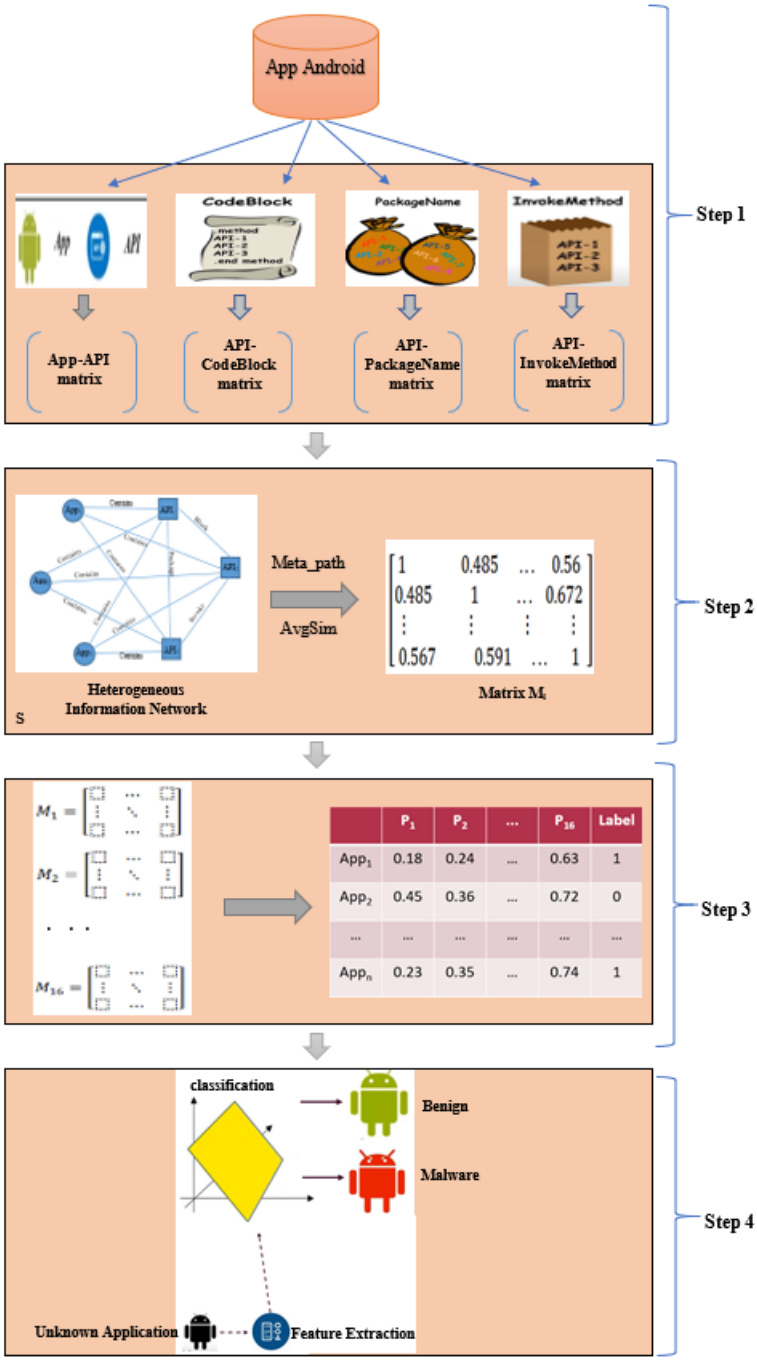

Figure 1. Overview of the proposed method

Step 1. After unpacking and decompiling the collected Android applications, we extract API calls and the relationships between them. Specifically, whether the APIs

belong to the same block, call to the same package, or are called by the same invoke method or not.

Step 2. With the data acquired from Step 1, we build a HIN with two types of vertices (Apps and API) and four types of edges (between Apps and APIs, between APIs in the same block, between APIs in the same package, between APIs in the same invoke). We use the AvgSim method according to meta-paths, to measure the similarity between any two apps.

Step 3. Based on the information obtained in Step 2, we extract a feature vector for each Android application.

Step 4. From the data about the Apps with their features, we train, test, and evaluate with some common machine learning models, e.g Support Vector Machines, Decision Tree, and Naïve Bayes.

2.2.1. Analyzing API calls

After an Android application is compiled, it will be packaged into a file of **.dex* archive. We use APKTool, a popular tool to decompile samples into smali code. Figure 2 shows an example of smali code after decomplication. Based on this code snippet, we can see that when an API is called, it will be called through the invoking statement.

```
1  .method protected
2  loadLibs(Landroid/content/Context;)V
3  .locals 4
4  :try_start_0
5  new-instance v0, Ljava/io/BufferedReader;
6  new-instance v1, Ljava/io/InputStreamReader;
7  invoke-static {}, Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;
8  move-result-object v2
9  const-string v3, "getprop ro.product.cpu.abi"
10 invoke-virtual {v2, v3}, Ljava/lang/Runtime;->exec(Ljava/lang/String;)
       Ljava/lang/Process;
11 move-result-object v2
12 invoke-virtual {v2}, Ljava/lang/Process;->getInputStream()Ljava/io/InputStream;
13 ......
14 .end method
```

Figure 2. An example of smali code

We use a matrix A_{nxm} to store information of the relationship between apps and APIs. n is the number of apps and m is the number of API. If API_j is present in an App_i, then $a_{ij} = 1$ else $a_{ij} = 0$. For example, a matrix that represents the relationship between APIs and the apps:

$$A=\begin{bmatrix} 1 & 0 & \cdots & 1 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}$$

Also according to Hou [18], in the process of analyzing the behavior of Android applications through API calls, we also found relationships between API calls with one another. For example, API calls in the same block either appear in the same package or are called together by the invoke method, always express similar intentions.

For example, the API calls in package "Lorg/apache/HttpRequest" always partake in actions relating to the Internet. The two API calls (Lines 10 and 12 in Figure 2) that are called by the same invoke method (invoke - virtual) behave similarly. Extracting this information is very useful in predicting malware.

The matrix representing the relationship between APIs of the same block: To describe the relationship, we use a two-dimensional array B_{mxm}. A block is a set of commands located between the keywords "method" and "endmethod" in the smali code. Matrix B is defined as follows:

$$b_{ij} = \begin{cases} 1\ if API_i\ and\ API_j are\ in\ the\ same block \\ 0\ if\ API_i\ and\ API_j\ are\ in\ different\ blocks \end{cases}$$

Matrix representing the relationship of APIs in the same package: to describe that relationship, a matrix P_{mxm} is used, with the following definition:

$$p_{ij} = \begin{cases} 1\ if\ API_i\ and\ API_j\ are\ in\ the\ same\ package \\ 0\ if\ API_i\ and\ API_j\ are\ in\ different\ packages \end{cases}$$

Take a look at the example shown in Figure 2, both API "*Ljava/lang/Runtime;* ⟶ *getRuntime() Ljava / lang / Runtime*" and API "*Ljava / lang / Runtime;*⟶ *exec (Ljava / lang String;) Ljava/lang/Process*" are from the same package, thus the element representing the relation of these two API in the matrix will be set 1.

The matrix representing the relationship between APIs with the same invoke method: In a typical smali code, there are different methods to execute API call: invoke-static, invoke-virtual, invoke-direct, invoke-super, invoke-interface. If two APIs are called with the same invoke method, there may be many other implicit relationships between them. Therefore, to represent this relationship, we use a matrix I_{mxn}, where each I_{ij} element indicates whether API_i and API_j are called by the same invoke method or not. Specifically

$$I_{ij} = \begin{cases} 1\ if\ API_i\ and\ \ API_j\ are\ called\ by\ the\ same\ invoke \\ 0\ if\ API_i\ and\ API_j\ \ are\ called\ by\ different\ invokes. \end{cases}$$

Table 1 describes the different relationships between the elements in the extracted matrices from APIs calls.

Table 1. A summary of matrices A, B, P, and I

Matrix	Element	Description
A	a_{ij}	If App_i contains API_j, then $a_{ij} = 1$, else $a_{ij} = 0$
B	b_{ij}	If API_i and API_j are in the same block then $b_{ij} = 1$, else $b_{ij} = 0$
P	p_{ij}	If API_i and API_j are in the same package then $p_{ij} = 1$, else $p_{ij} = 0$
I	i_{ij}	If API_i and API_j share invoke then $i_{ij} = 1$, else $i_{ij} = 0$

2.2.2. Heterogeneous Information Network construction

From the four matrices obtained above, we construct a heterogeneous information network. This network consists of two types of vertices and four types of edges. The different types of edges represent different relationships between APIs. The vertices are apps and API calls. The first type of edge describes whether an App_i contains API_j. The second type represents the relationship between APIs of the same block, the third

describes the relationship between APIs of the same package, and the fourth describes the relationship between APIs with the same invoke method.

To measure the similarity between the two Android applications, we use the AvgSim measure. AvgSim is a measure of symmetry and has the property of generality to evaluate the likeness of two objects of the same or different types. AvgSim value of two nodes is the average of the probability that one node can reach the other according to the given path and its inverse path. Because of that trait, AvgSim is a measure of symmetry. AvgSim calculation does not consider path length and therefore does not need to decompose any meta-path with odd length, making AvgSim simpler and more efficient than HeteSim [19].

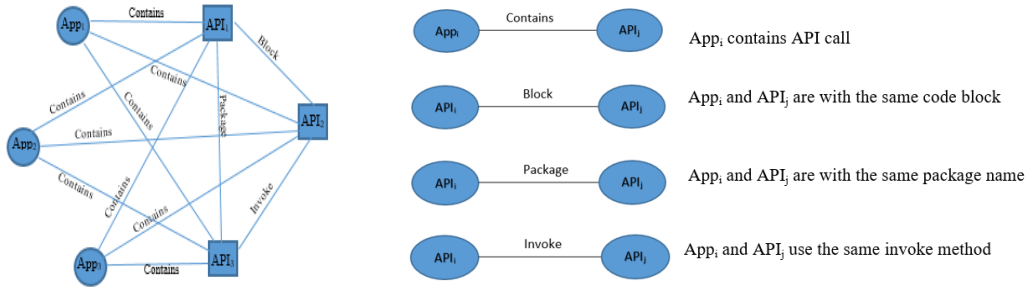

Figure 3. HIN network illustration

The AvgSim measure is calculated based on meta-paths, a concept covered in Definition 3. A meta-paths from vertex *a* to vertex *b* represents the relationship between *a* and *b*. With the HIN above, we build meta-paths from any App_i to any App_j. A typical meta-path between two Android applications is $App \xrightarrow{Contains} API \xrightarrow{Contains^{-1}} App$, aka. AAT. For example, the HIN shown in Figure 2.3 contains these specific paths: App_1 API_1 App_2, App_1 API_1 App_3, App_2 API_1 App_3, etc. Another meta-path between two Android applications that signify the similarity between two apps via API call is $App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Contains^{-1}} App$, or APAT (for example, $App_1API_1API_3$ App_3, $App_1API_1API_3$ App_3).

During the data processing step, we found that calculating the number of meta-paths between two apps in the HIN is time-consuming because this is a conditional graph traversal problem with a significant number of vertices. Also, some studies show that in the HIN, long meta-paths often do not bring much information to distinguish the relationship between the vertices in the network. So in this paper, we choose symmetrical paths, the maximum length of a meta-path is seven, and the number of meta-paths is sixteen.

From the above sixteen meta-paths, we extract sixteen M_k matrices (Definition 5). M_k is a square matrix of n Android applications, each M_k value (i, j) is the value calculated by the AvgSim measure, showing the similarity between App_i and App_j based on the P_k meta-paths.

2.2.3. Extracting feature vectors

From the sixteen M_k matrices calculated above, we extract information to form a feature vector for the classification problem. The similarity between App_i and all the other App_j is shown by the sum of the values per row i in the matrix M_k. Therefore, we construct matrix V, size n by m, where n is the number of Apps, m is the number of M_k matrices, which is equivalent to the number of the meta-path P_k.

$$V_{ij} = \sum_{k=1}^{n} M_j[i,k], for\ i\ \in [1,n]\ and\ j\ \in [1,16].$$

Meaning that for every App_i in the dataset, there will be a specific feature vector for that app, v_i = (v1, v2, …, v16). All inputs after being collected are preprocessed to exhibit within the range of [0, 1] using scaling techniques.

2.3. Experiment and results

2.3.1. Datasets

Currently, many projects provide malicious and benign samples to malware analysts, such as Drebin AndroZoo or Android Malware Dataset (AMD), etc. In this research, we use datasets which were collected from Drebin [20], with 4022 Android applications, of which 2510 are malicious, and 1512 are benign. All are checked with Bitdefender antivirus software.

After unpacking and decompiling the Android apps into smali code, we extract the API calls and the relationship between them. API is a method of connecting to other libraries and applications. API calls are used in Android applications to access functions of the operating system and system resources. There are many approaches to extract API calls; we can filter instructions that call to some standard Android functions. For example, it is possible to store all calls to any method in class *Landroid/telephone/SmsMessage*. However, the list of calls can be excessive, and more importantly, it can vary from Android versions to Android versions, we focus only on the calls that belong to the Android and Java namespaces (i.e., classes starting with *Landroid /LJava*).

On the other hand, since the number of API calls in an App is often large, the entire API call extraction used for training is entirely unfeasible and makes no sense in machine learning training. Therefore, we choose to extract the APIs that often appear in malware in 3010 data samples collected from Drebin. We limit the number of APIs we need to extract to around 200.

We use the AvgSim to measure the similarity between any two applications according to the meta-paths. Sixteen meta-paths in Table 2 are used. These paths start from an App and end in another App. The maximum length is seven.

We proceed with the steps described in the proposed method. Finally, we obtained two sets representing the Apps as two-dimensional matrices (with dimensions of 4022 by 17 and 1834 by 17); where each row represents an App, each of the first sixteen columns represents a meta-path, the last column is the label of the corresponding App_i (1 - App_i is malicious; 0 - App_i is benign).

2.3.2. Evaluation

To evaluate the effectiveness of the classification models, we use the 10-folds cross-validation method. It is a familiar technique in evaluating machine learning and data mining field. The dataset is divided into ten non-intersecting parts; nine parts are taken as data to train the models; the rest is used for testing. For each time we train and test a classification model, we compute the following metrics:

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

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

$$F_1 = \frac{2*Precision*Recall}{(Precision+Recall)}.$$

In which, TP, FP, FN is the number of malware correctly identified, the number of benign software mistakenly diagnosed, the number of malware incorrectly labeled, respectively. After ten sessions of training and testing, the Precision, Recall, and F_1 are the mean output of the ten training sessions.

Table 2. Sixteen meta-paths and their descriptions

STT	Siêu đường	Mô tả
P_1	AA^T	$App \xrightarrow{Contains} API \xrightarrow{Contains^{-1}} App$
P_2	ABA^T	$App \xrightarrow{Contains} API \xrightarrow{Block} API \xrightarrow{Contains^{-1}} App$
P_3	APA^T	$App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Contains^{-1}} App$
P_4	AIA^T	$App \xrightarrow{Contains} API \xrightarrow{Invoke} API \xrightarrow{Contains^{-1}} App$
P_5	$ABPB^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Block} API \xrightarrow{Package} API \xrightarrow{Block^{-1}} API \xrightarrow{Contains^{-1}} App$
P_6	$APBP^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Block} API \xrightarrow{Package^{-1}} API \xrightarrow{Contains^{-1}} App$
P_7	$ABIB^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Block} API \xrightarrow{Invoke} API \xrightarrow{Block^{-1}} API \xrightarrow{Contains^{-1}} App$
P_8	$AIBI^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Invoke} API \xrightarrow{Block} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Contains^{-1}} App$
P_9	$APIP^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Invoke} API \xrightarrow{Package^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{10}	$AIPI^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Invoke} API \xrightarrow{Package} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{11}	$ABPIP^TB^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Block} API \xrightarrow{Package} API \xrightarrow{Invoke} API \xrightarrow{Package^{-1}} API \xrightarrow{Block^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{12}	$APBIB^TP^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Block} API \xrightarrow{Invoke} API \xrightarrow{Block^{-1}} API \xrightarrow{Package^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{13}	$ABIPI^TB^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Block} API \xrightarrow{Invoke} API \xrightarrow{Package} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Block^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{14}	$AIBPB^TI^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Invoke} API \xrightarrow{Block} API \xrightarrow{Package} API \xrightarrow{Block^{-1}} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{15}	$AIPBP^TI^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Invoke} API \xrightarrow{Package} API \xrightarrow{Block} API \xrightarrow{Package^{-1}} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Contains^{-1}} App$
P_{16}	$APIBI^TP^TA^T$	$App \xrightarrow{Contains} API \xrightarrow{Package} API \xrightarrow{Invoke} API \xrightarrow{Block} API \xrightarrow{Invoke^{-1}} API \xrightarrow{Package^{-1}} API \xrightarrow{Contains^{-1}} App$

2.3.3. Evaluation

We conducted experiments to verify the ability of the data representation model we proposed. This experiment uses a dataset of 4022 Android applications and uses three popular machine learning algorithms: Support Vector Machines (SVMs), Decision Tree (DT), and Naïve Bayes (NB). For each algorithm, we performed several different runs with various parameters, finally choosing the right set of parameters for each model on the prepared dataset. The averaged results after ten iterations are shown in Table 3.

Table 3. Results of different models on the given dataset

Method	Precision (%)	Recall (%)	F1 (%)
SVMs	91.09	99.86	95.27
DT	95.61	96.95	96.27
NB	95.17	88.49	91.70

The results show that with our method, the F1 score is relatively high, especially the results of SVMs (95.27%) and DT (96.27%). For the binary classification problem, this accuracy is considered to be fair and can be used to predict unlabeled data. The results also show that the proposed method has been vastly improved compared to that of our research [21] because the proposed method extracts additional information from the APIs with their invoke and uses the AvgSim measure - a measure that is recommended for its efficiency and simplicity on the HIN.

To compare results of malware prediction using our method of representing and extracting features, we conduct two experiments on DT model as follows. In Experiment #1, we extract the API calls which present in each Android application, then build the D_{nxm} information matrix, where n is the number of applications, m is the number of APIs. Each element $D_{ij} = 1$ if App_i contains API_j, otherwise $D_{ij} = 0$. This only extracts information about the API calls in the applications, regardless of the relationship between them. After that, we conduct training and testing in the same way as we did in the experiment outlined above.

In Experiment #2, we extracted the permissions in each application. Like how the dataset for the API extraction method is constructed, the empirical matrix T_{nxm} is introduced, where n is the number of applications, m is the number of permissions. Each element $T_{ij} = 1$ if App_i contains permission j, otherwise $T_{ij} = 0$. According to studies [5], 6] we combine the D matrix and T matrix to the DT matrix.

Table 4 shows that our method has better results than Experiment #1 and #2 on the same dataset. The reason is that the difference in information extraction methods from Android applications. In previous studies, most either only extracted the API calls or the permissions contained in each application to build training data. Meanwhile, our approach not only extracts the information of the API calls present in each application but also extracts the relationship between the API calls within an application. We believe this approach can be extended so that other information of API calls in an Android application can be integrated.

Table 4. Comparison of results between ours and other methods

Methods	Precision (%)	Recall (%)	F1 (%)
Our method	95.61	96.95	96.27
Experiment #1	90.70	92.34	91.51
Experiment #2	94.28	95.25	94.76

3. Conclusions

Android applications are continually being developed and widely used, leading to more diverse penetration and malware distribution. Malware is increasingly capable of sophisticated concealment, making it more challenging to analyze and extract features and behaviors. Finding ways to represent and extract data for Android malware detection is getting more and more attention from the research community in the field of information security. In this paper, we propose the representation and extraction of features utilizing HINs built upon API call analysis. Unlike previous studies, we create training datasets based on representing relationships between Apps and APIs and relationships between APIs into a heterogeneous network. We then use meta-paths on this network to extract the information describing the relationship between the Apps.

The experimental results on the Drebin malware dataset show that our proposed method is relatively accurate. F1-score in the best scenario is 96.27%. Compared to the previous methods, we found that the proposed method's predicted results were higher, meaning that the representation of Android Apps by a HIN can be a viable approach for the problem of classification and malware prediction.

However, with this approach, computational costs will be higher, since a HIN has to be built and meta-paths must be found on those networks. On the other hand, the extraction of information on the heterogeneous network using meta-paths was not as effective as expected. In the near future, we will seek to optimize heterogeneous network building algorithms and extract information combined with deep learning models to increase efficiency and reduce computation costs.

Acknowledgements. This research was supported by the Academy of Cryptography Techniques, project 08.DT20.C2.

REFERENCES

[1] J. Walls and K. K. R. Choo, 2015. A review of free cloud-based anti-malware apps for android. *in Proc. 14th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun*, Vol. 1, pp. 1053-1058.

[2] A. Souri and R. Hoseini, 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. *Comput. Inf. Sci*, Vol. 8.3.

[3] M. Nilay and P. Nitin, 2013. Review of Behavior Malware Analysis for Android. *International Journal of Engineering and Innovative Technology*, Vol. 2, pp. 230-235.

[4] W. Wu and S. Hung, 2014. DroidDolphin: A dynamic Android malware detection framework using big data and machine learning. *RACS '14*, pp. 247-252.

[5] N. Peiravian and X. Zhu, 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. IEEE 25th International Conference on Tools with Artificial Intelligence, pp. 300-305.

[6] A. Sharma, S.K. Dash, 2014. Mining API Calls and Permissions for Android Malware Detection. Proceedings Cryptology and Network Security, Vol. 8813, pp. 191- 205.

[7] L. Matthew, K. Miclain and A. Travis, 2017. A Comparison of Features for Android Malware Detection. Proceedings of ACM SE '17, pp. 63-68.

[8] L. Bai, J. Pang, Y. Zhang, W. Fu, and J. Zhu, 2009. Detecting malicious behavior using critical api-calling graph matching. *ICISE* , pp. 1716-1719G.

[9] C. Shi, Y. Li, J. Zhang, Y. Y. Sun and J. Han, 2012. Mining Heterogeneous Information Networks: Principles and Methodologies. *SIGKDD*, vol 14, pp. 20-28.

[10] Y. Sun, B. Norick, J. Han, Xifeng Y, Philip S. Yu, and Xiao Yu, 2012. Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. *In KDD'12*, pp.1348-1356.

[11] D.H. Nguyen, T.T. Truong, D.H. Tran, 2015. Prediction of microRNA-disease associations using RWRs. *Journal of Science of HNUE*, Vol. 60, No. 7A, 10-20 (in Vietnamese).

[12] T.H. Le, T.V. Thai, D.H. Tran, 2015. Prediction of disease-causing genes using unlabeled data. *Journal of Science of HNUE*, Vol. 60, No. 7A, 61-69 (in Vietnamese).

[13] Y. Yanfang, H. Shifu, C. Lingwei, L. Jingwei, 2018. AiDroid: When Heterogeneous Information Network Marries Deep Neural Network for Real-time Android Malware Detection. *CoRR*, Vol. 2, pp. 232- 340.

[14] C. Shi, P. S. Yu, 2017. Heterogeneous information network analysis and applications. *In Data Analytics, Springer*, pp. 2-5.

[15] C. Shi, Y. Li, J. Zhang, Y. Sun, P. S. Yu, 2017. A survey of heterogeneous information network analysis. *IEEE Trans. Knowl. Data Eng*, Vol. 29, No. 1, pp.17-37.

[16] Y. Sun, J. Han, X. Yan, Philip S. Yu, and T. Wu, 2011. PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. *VLDB*, Vol. 11, pp. 992-1003.

[17] D. Xiao, X. Meng, Y. Li, and C.Shi, 2016. AVGSIM: Relevance measurement on massive data in heterogeneous information networks. *JATIT*, Vol. 84, pp. 101-110.

[18] H. Shifu Hou, Yangqiu Song, Melih Abdulhayoglu, 2017. Hindroid: An Intelligent Android Malwave Detection System Based on Structured Heterogeneous Information Network. *KDD*, pp. 1507-1515.

[19] C. Shi, X. Kong, Y. Huang, P. S. Yu and B. Wu, 2014. HeteSim: A General Framework for Relevance Measure in Heterogeneous Networks. IEEE Transactions on Knowledge and Data Engineering, Vol. 26, No. 10, pp. 2479-2492.

[20] https://www.sec.cs.tu-bs.de/~danarp/drebin/download.html.

[21] Thanh Van Thai, The Dung Luong, 2019. A method for detecting Android mailware based on heterogeneous information network. *Jounal of Military Science and Technology*, pp. 79-89 (in Vietnamese).