Malware Detection

Yikai Hao
University of California, San Diego
La Jolla, California
yih307@ucsd.edu

Yu-Chieh Chen
University of California, San Diego
La Jolla, California
yuc399@ucsd.edu

Ruoyu Liu
University of California, San Diego
La Jolla, California
rul141@ucsd.edu

ABSTRACT

As the technology grows fast in recent years, more and more people cannot live without cell phones. It is important to protect users' data for cell phone companies and operating system providers. Therefore, detecting malwares based on the code they have can avoid publishing of malwares and prohibiting them from the source. This report aims at finding a model which can detect malwares accurately and with a small computational cost. It uses different matrices and graphs to search the relationships between applications and detecting malwares based on the similarity. As a result, the best model can achieve a test accuracy around 99%.

1 INTRODUCTION

As the internet techniques are growing at a fast speed nowadays, people are starting to worry about their data safety. Since many of us will store our important information on our cellphones, we need to find an appropriate way to secure our cell phone away from malwares. As the operating system which takes over 80% of the cellphone market, Android operating system is always a large target for malwares. Since the Android system uses an open market where everyone has the ability to upload application packages, malwares can easily be uploaded and spread among the internet. In addition, malwares can easily evade detection by repackaging or using code obfuscation. As a severe problem faced by Android, it is important to detect the malwares to ensure the safety of user's data. Lots of scholars are starting to participate in the malware detection research area. Many new technologies are being applied to the malware detection area, including Machine Learning strategies and Natural Language Processing (NLP) methodologies.

In order to develop a more powerful tool for malware detection, we do not only focus on Application Programming Interfaces(API). Some other features, like the same return type or same package name, are also considered. Those features can help us find the inner relationships among applications. By using the idea of Heterogeneous Information Network(HIN), we use matrices to represent each kind of relationship. Aiming at developing a faster model for malware detection, we also use TF-IDF(term frequency - inverse document frequency) to select out part of APIs and use them for model construction. Besides matices, graphs are also considered as features for the implementation of classifiers. Using the logic of Network Representation Learning and NLP, we are able to change matices to different graphs where edges represent different kinds of relationships. We compare all those models and find some useful models for malware detection within a quick speed. Related Work

2 RELATED WORK

In recent years, lots of studies are focused in the area of Intelligent Android malware detection systems. They use machine learning and data mining strategies to contribute their model for detecting malwares. And our project is based on previous research on such kind of malware detection, especially HinDroid[3]. HinDroid focuses on utilizing API features in code and customizing kernels to identify malwares. It uses multi-kernel with different assigned probabilities as its final model for malware detection. HinDroid is based on the static method which focuses on the internal component of an application. There is also some previous research related to Network Representation Learning. Word2Vec[4] designs a new simple way to represent words. It learns the word vector using the Skip-gram model which predicts the previous and future words based on current words. Node2Vec[2] is an algorithm used for learning the feature representation for nodes by random walks. Metapath2Vec[1] is the scalable representation algorithm which formalizes the random walks by metapath.

This report implements those algorithms to generate graphs for classifiers and use them to detect malwares.

3 DATA OVERVIEW

3.1 Data Source

The data source is called Android Malware Dataset (AMD). The dataset is published in 2017 by the Argus Lab from the University of South Florida. This data source is used by many other malware detection papers and widely used in the research domain.

3.2 Data Description

The original source is the APK(Android Application Package), which can be decompiled by Apktool. After decompiling, we select smali files, which are a type of files containing a proxy of the original code, specifically for detecting malwares. We select 905 malwares and 905 benigns from the dataset. Benigns are separated in two categories - popular applications and random applications. Popular applications are those on the application chart for top downloads. And random applications are random applications selected from apkpure.

3.3 Smali File

In order to analyze the smali files, we should understand the structure of it. Therefore, figure 1 shows the description of the smali files and the features contained in the smali files.

```
.class final Lbolts/a;
.super Ljava/lang/Object;
.source "SourceFile"

# static fields
.field static final a:I
.field static final b:I

# direct methods
.method static constructor <clinit>()V
        ...
    .line 58
    invoke-static {}, Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;
        ...
    return-void
.end method
```

Figure 1: Smali Structure

The basic components of a Smali file are the following:

-Class information: In this example, Lbolts/a is the class name for this file.
-Static fields: It contains the shared variables among whole class
-Method direct virtual: It contains the methods in the original java file. This is the part we will use to extract our main feature - API and other related features.

3.4 API Calls

In order to understand which part of the smali files do the malicious action, we put our attention on API (Application Programming Interface) calls. There are four main components in an API call.

```
invoke-static {}, Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;
```

Figure 2: API Call Example

The basic components of an API call are the following:
-Invoke method: there are five methods to invoke in API calls, including invoke-static, invoke-virtual, invoke-direct, invoke-super, and invoke-interface.
-API package
-Method name
-Return type of the method

3.5 Data Extraction

After understanding the data structure of the data source and the exact data section we want to focus on, we start the data extraction process. We decide to use api name, class type, package type, code block number, method type, return type, and invoke type to be our features. Those features almost include every kind of information we can get from the smali file. In addition, we also do some EDA(Exploratory Data Analysis) to make sure every feature we get is distributed differently among different types of applications. Therefore, the features can present the original data and they are useful hints to detect malwares.

3.6 Database

We design a special database to store the data we get. Since the main feature is the api, there are over 2 millions unique apis and more than 50 millions apis appeared in different smali files. Separated csv files and unique ids are used to store the specific unique string

values and represent the string value. Then, in the main csv files, we store the unique ids from different references. This will reduce the space and time dramatically. For example, our storage originally will take up to 90G spaces and now only takes 1.7G. The structure of the database is shown in figure 3.

3.7 Data Statistics

After picking out the features we want, we do some simple analysis based on the data we have. As the table shows, the size difference between malwares and benigns are huge. In addition, the unique api calls in benigns are about 10 times larger than malwares. Therefore, the difference between malwares and benigns do exists and we are able to find some way to detect malwares. The statistics is shown in table 1.

4 MODEL

4.1 Feature distribution

In order to check whether the features we generate are useful for checking malwares, some Exploratory Data Analysis (EDA) has done on features. We check the difference between unique values among features considered the type of applications. The result shows that our features can clearly identify the different types of applications since the value between different types of applications are large (table 2).

We also plot out some distributions of features considering different types. Taking the number of unique return types for example (figure 4), malwares are often having a relatively small number of unique values. However, popular applications are always having a large number of unique return types. Random applications are more spread over the x-axis, but still more condensed at a large number of unique return types. Therefore, the number of unique return types will be a useful feature to detect malwares.

4.2 HinDroid

HinDroid [3] is the baseline model we use for our report. It contains different types of kernels and the meaning behind each kernel is different. It uses the features we extract from the data generating process to build some matrices. Each matrix shows a type of relationship between apis or applications. Each matrix is an adjacent matrix for a graph with a specialized relationship. By

choosing different types of matrices we want to combine together, we get the kernel we have. Then we will be able to use the custom kernels and put in different classifiers like SVM or Random Forest for malware detection.

The four types of matrices are: A, B, P, and I matrices.

- A matrix shows the connection among apis and applications. The value within the A matrix will be one if the application contains the api.
- B matrix shows the connection between apis. The value within the B matrix shows whether two apis are contained in the same code block.
- P matrix also shows the connection between apis. The value within the P matrix shows whether two apis use the same package.
- I matrix shows the connection within the apis. The value within the I matrix shows whether two apis use the same invoke type.

Currently, due to the large size of the unique apis we get, we are not able to calculate out the I matrix yet. Therefore, the kernel we have now for HinDroid is $AA^\top$, $ABA^\top$, $APA^\top$, and $APBP^\top A^\top$.

4.3 New Model

The HinDroid model runs pretty slow since there are a large number of APIs. However, lots of APIs only appear once among all applications and they are meaningless for detecting malwares. In addition, there are also some APIs which appeared in almost every application. Those APIs are also not meaningful enough to help us pick out the malwares. Therefore, new models are being considered and built. Based on the logic of HinDroid, we try to develop some new matrices to replace the original matrices which will have a faster speed and similar accuracy.

4.3.1 Reduce API / Pack. The inspiration of this method comes from the MAMADROID [5]. Instead of using the whole API call, API name and API library have been selected separately. The number of unique API calls for around 2000 applications are originally over 1,000,000. We design two new matrices based on the separation of values in an API call, which is shown in Figure 2.

- Reduce API: This matrix only contains the API Name, which is the red part in the example. The new matrix size is around 130,000 x 130,000, which is way smaller than the original A matrix.
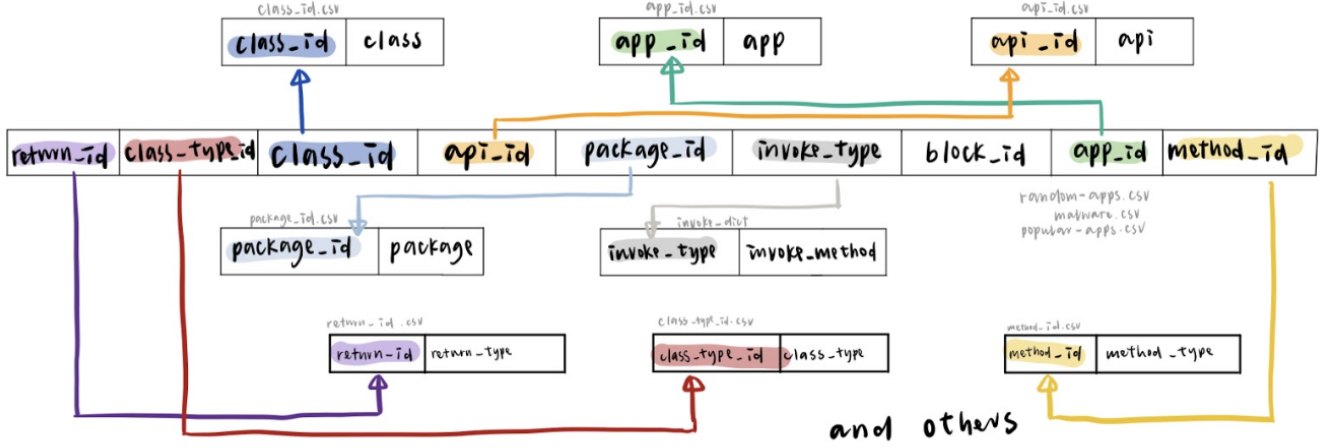
Figure 3: Database

Type	API called once (sum/app)	Number of API	Number of Class	Number of APP
Malware	29.05	792.08	284.32	905
Popular	689.09	8214.12	3930.60	324
Random	340.34	6387.03	2893.34	581

Table 1: Data statistics

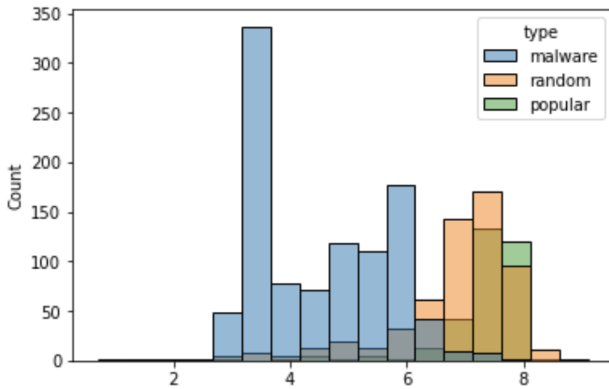|	#UniqueAPI	#Unique API Lib	#Unique API Name	#Unique API Return Type
Malware	792.08	277.34	359.55	172.18
Benign	7041.15	2551.21	2571.26	1485.79

Table 2: API Statistics

Figure 4: Log Scaled of Unique API Return Type

- Reduce Pack: This matrix only contains the API Library, which is the blue part in the example. The

new matrix size is around 350,000 x 350,000. The size is about 2/3 smaller than the original A matrix

4.3.2 TF-IDF. Besides Reduce API and Reduce Pack, we are also considering can we select out some APIs which are considered "important" for detecting malwares. The method we choose is TF-IDF(term frequency - inverse document frequency). It is a useful method to check the importance of a word for a document. We generate a corpus list which each element in the list is representing a corpus for an application. In each corpus, it contains all API calls. We then use the TF-IDF to get a token score for each API call. After calculating the mean score over all corpus, part of the API calls are selected out according to their rank. The numbers we select out are Top1000, Top2000, Top5000, and Top10000.

Top3 API Example Rank by TF-IDF
Ljava/lang/StringBuilder;->append()
Ljava/lang/StringBuilder;->toString()
Ljava/lang/Integer;->valueOf()

Table 3: TF-IDF Top 3 API Example

4.3.3 New Features. New features are also being considered to build new matrices. We use the return type as our new feature and build a matrix called R. The element in the R matrix represents whether two applications are using the same return type. R matrix can replace the original A matrix and its size is only around 170,000 x 170,000. As the feature description part shown, the return type is also a useful feature to detect malwares. Additionally, in order to build a new kernel for the R matrix, the new B_R matrix represents whether two return types are in the same code block. Therefore, we have two different kernels - RR and RB_RR What's more, we also built a new I matrix after finishing the API reduction. This also provides more kernel options while putting the features into classifiers.

We design lots of new matrices which can replace the original A matrix using new features or reduced number of APIs. Those matrices will be used when we are building kernels.

4.4 Word2Vec

Word2Vec is the new vector embedding we generate. This model is a powerful NLP model to help us find not only the direct relationship between apps but also the cluster connection between apps using the graph, which is a different approach to solve the malware detection problem with HinDroid.

Our Word2Vec takes $AA^\top$ as an input and builds a graph based on the $AA^\top$. Therefore, the graph contains two components - applications and apis. We then generate sentences as our input for the Word2Vec model. Firstly, we randomly pick an app out, then we follow the path in the graph to search for the next api and app. We will end our path with an app. The length of the path will be a number randomly chosen within the range of maximum length.

For example, with a maximum length of 5000 and a metapath $AA^\top$, the possible text generated will be like:

```
APP1 -> API234 -> APP34 -> API12 ->
```

After finishing the sentence generating process, we will implement the genism's Word2Vec model to get our vector embeddings for every application and api. The final vector embeddings will be easily used in different machine learning models.

We use data visualization to check if our model makes sense. Our plot shows the vector embeddings after the dimension reduction using a method called t-SNE(t-distributed stochastic neighbor embedding). This method can project a high dimensional vector into a two dimensional space. t-SNE uses Barnes-Huts approximations to reduce the dimensions. As the graph shows (figure 5), the distribution of malwares and benigns are separated. Benigns are condensed at the left side with small x and y values. However, malwares are distributed at the right side, with a large x value and widespread y value. From the information on the graph, the model can detect malwares well. Although a few points are mixed in the graph, they might be separable in higher dimensions.
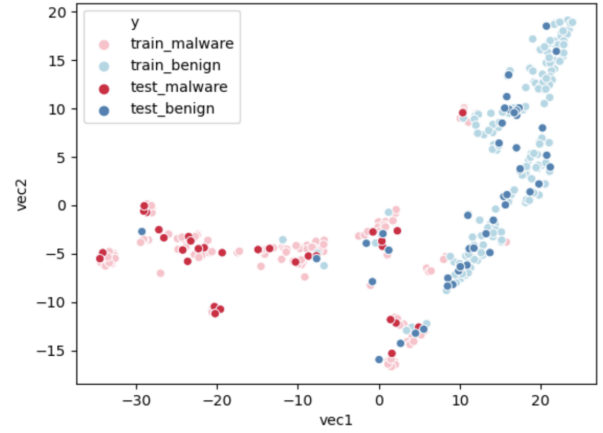
Figure 5: Word2Vec Embedding Visualization

4.5 Node2Vec

The only difference between Node2Vec and Word2Vec is the random walk procedure. This change improves the inability of Word2Vec and tracks the path with no specific rules about where to go.

We use all A, B, and P matrices to build our Node2Vec. Since the B and P matrices both represent the relationships between apis, we combine the two matrices into

one larger matrix to replace the B and P matrices. The values within the large matrix represent whether two apis have some relationships, no matter whether they are within the same code block or use the same package.

For the probability of random walks, there are three types of probability. For example, we have a path from t -> v shown in figure 6. When choosing the next step for v, we have three different probabilities. If we get from v -> t, we have a probability of 1/p. In addition, if the next node from v has a connection with t, then the probability of the node will be 1. Other nodes will have a probability with 1/q. We then implement sentences into the genism's Node2Vec model. The p value we select in our Node2Vec is 1 and the q value we select is 1/2. We choose a larger p value since we do not want our path going back to its previous node.
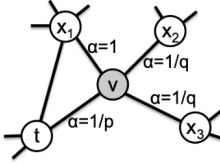
$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

Figure 6: Node2Vec Formula

Similar to Word2Vec, we also plot out the vector embeddings after finishing the dimension reduction (figure 7).
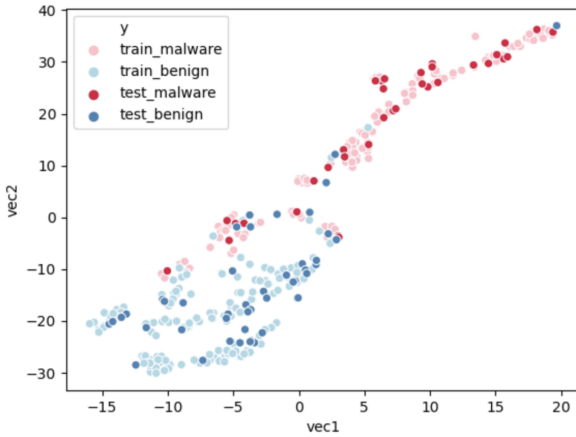

Figure 7: Node2Vec Embedding Visualization

4.6 Metapath2Vec

Methpath2Vec is an extension of Node2Vec on heterogeneous graphs. The difference between Metapath2Vec

and Node2Vec is that the Metapath2Vec assigns a path for the random walk and decides where the next node to go. The Metapath2Vec model uses all A, B, and P matrices. The sampling method of Metapath2Vec is based on the equation (1), which means the next node will be accessed if the edge exists and the node belongs to the correct type. For example, if the path given is $\mathbf{ABA}^{\top}$, we will generate a sentence from an app to an api first. Then we will check the next node is an api which is in the same code block with the previous api. Finally, our path will go to another app. We repeat this loop until we reach the maximum length we set or have no next node.

We then implement sentences into the genism's Word2Vec model. After the dimension reduction process is done, the embedding plot is shown in figure 8.

$$p(v^{i+1}|v_t^i, \mathcal{P}) = \begin{cases} \frac{1}{|N_{t+1}(v_t^i)|} & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) = t+1 \\ 0 & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) \neq t+1 \\ 0 & (v^{i+1}, v_t^i) \notin E \end{cases}$$
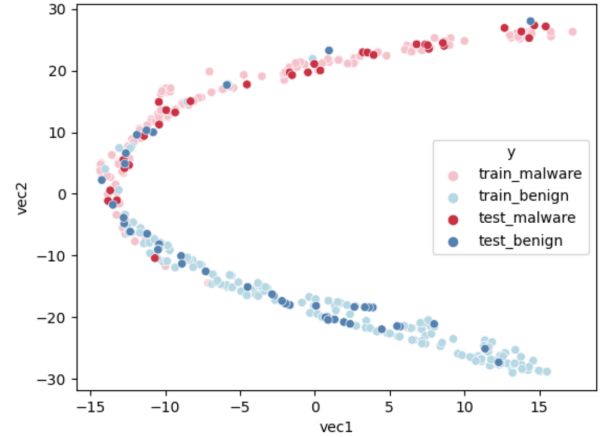
(1)

Figure 8: Metapath2Vec Embedding Visualization

5 RESULT

Below are the results of different models.

5.1 Classifiers

After different models are built, SVM(Support-Vector Machines), Random Forest, and Gradient Boosting are selected as classifiers while doing the final malware

detection. SVM is the baseline classifier we choose. It uses different matrices as custom kernels to classify the type of applications. Random Forest and Gradient Boosting both use decision trees as their base. Decision tree is a tree model in which each node represents a decision rule that separates the dataset. Random forest uses the idea of "Bagging". It builds lots of decision trees at the same time using a subset from the dataset. Then, Random Forest will combine the result with weight and produce the final prediction. Gradient Boosting uses another idea called "Boosting". It will also build lots of decision trees. And Gradient Boosting will update the newest model by making improvement on the last model.

The classifier with highest accuracy will be chosen as the classifier of a specific model. As the result table shows, most classifiers will be SVM. However, the Node2vec model shows a preference on Gradient Boosting.

5.2 Result Tables

As the table 4, train accuracy, test accuracy, and F1 score are the values to evaluate the performance of the model. We also include False Positive and True Negative count to check which kind of error will the model make.

As the result table 4 shows, the best performance is the original HinDroid model with $\mathbf{AA}^\top$ kernel and SVM classifier. It can achieve a test accuracy around 99% with only three benigns misrecognized as malwares. And the reduced API with top 2000 APIs selected by TF-IDF also has a similar accuracy. It can achieve a 99% test accuracy by $\mathbf{AA}^\top$ kernel and SVM classifier.

The table 4 also shows that most of the models perform best under more baseline kernel and classifier. With a kernel $\mathbf{AA}^\top$ and classifier SVM, most models reach their highest test accuracy. However, using different kernels like $\mathbf{ABA}^\top$ or $\mathbf{APBP}^\top\mathbf{A}^\top$, the classifier will switch to Gradient Boosting or Random Forest with a slightly lower test accuracy.

In addition, when comparing the results of reduced API kernels we build for new models, we find out that the accuracy is high enough to do prediction. Most of them have a test accuracy around 99%, which is higher than models based on graphs. Graph models, including Word2Vec, Node2Vec, and Metapath2Vec, are performing the poorest among all models, shown in table 5. With a much less time complexity, reduced API kernels

can be considered as a powerful tool to replace basic HinDroid with similar test accuracy.

Most models perform better on detecting malwares. However, most of the mistakes in identifying the type of applications make on benigns. As the table indicated, most False Positive values are much higher than False Negative values. As a malware detection model, our model should be more focused on detecting every malware out. Therefore, having some misclassified benigns within an acceptable rate is allowed. Multi-kernel might not be a useful improvement since the original Hin-Droid model already has the best performance on False Positive.

5.3 Research on Misclassified App

After seeing the result, we do some research on the mis-recognized applications. As the table 6 shown, the original HinDroid model with metapath $\mathbf{AA}^\top$ and classifier SVM only missed 3 applications. Those three applications are considered to be False Positive, which means that they should be benigns but identified as malwares. We select those 3 applications out and find that they are all in the category Random application. By checking the features used for malware detection and comparing it with the 25% - 75% range for both malwares and benigns, those applications are at the boundary of malwares and benigns. Therefore, it is reasonable for the classifier to misrecognizing those applications. In addition, as the Data Description section mentioned, random applications are selected randomly out of apkpure. There is a small possibility that those three applications are actually malwares. There is a small possibility that those three applications are actually malwares.

6 CONCLUSION

In this report, we implement different methods for malware detection. Based on the weakness we find in using HinDroid, we also design some new matrices and kernels in order to save space and time. As the result section shows, the outcome is positive. With a much smaller matrix and time complexity, the new model can perform as well as the original HinDroid model. Although graph-based models do not perform as well as kernel based models, they are achieving a high accuracy around 95%. Graph is still a useful strategy to

---

[1]GB: Gradient Boosting

[2]RF: Random Forest

Model	Kernel	Classifier	Train Acc	Test Acc	F1	FP	FN
HinDroid	AA	SVM	1.0	0.9917	0.9919	3	0
	ABA	GB[1]	0.9917	0.9419	0.9440	13	8
	APA	SVM	1.0	0.9779	0.9788	8	0
	APBPA	RF[2]	1.0	0.9337	0.9358	14	10
Reduce API Name	AA	SVM	1.0	0.9834	0.9839	5	1
	ABA	RF	1.0	0.9419	0.9442	14	7
Reduce API Pack	AA	SVM	1.0	0.9889	0.9893	4	0
	ABA	GB	0.9965	0.9419	0.9415	17	4
TF-IDF 1000	AA	SVM	1.0	0.9861	0.9865	2	3
	AIA	RF	1.0	0.9143	0.916	14	17
TF-IDF 2000	AA	SVM	1.0	0.9917	0.9919	2	1
	ABA	RF	1.0	0.9475	0.9493	12	7
	APA	SVM	1.0	0.9834	0.9839	5	1
	APBPA	DT	1.0	0.9309	0.9326	13	12
	ABPBA	SVM	1.0	0.9806	0.9812	5	2
	AIA	RF	1.0	0.9198	0.923	18	11
	ABPIPBA	GB	0.9261	0.9088	0.9133	22	11
TF-IDF 5000	AA	SVM	1.0	0.9889	0.9892	3	1
	AIA	GB	0.9488	0.9198	0.9238	20	9
TF-IDF 10000	AA	SVM	1.0	0.989	0.9892	3	1
	AIA	GB	0.9537	0.9171	0.9215	21	9
API Return Type	RR	SVM	1.0	0.9862	0.9867	5	0
	RBR	GB	0.9896	0.9282	0.9319	19	7

Table 4: Model Statistics

Model	Metapath	Classifier	Train Acc	Test Acc	F1	FP	FN
Word2Vec	AA	GB	0.9993	0.9475	0.9501	15	4
Node2Vec	AA	RF	1.0	0.9420	0.9440	13	8
	All	GB	0.9965	0.9475	0.9501	15	4
Metpath2Vec	AA	GB	0.9717	0.9448	0.9465	12	8
	ABA	GB	0.9869	0.9337	0.9371	18	6
	APA	GB	0.9931	0.9448	0.9474	15	5
	APBPA	RF	0.9848	0.9088	0.9147	177	152
8	25						

Table 5: Graph Statistics

consider while detecting malwares since it can catch the cluster relationship among applications.

There is lots of future exploration that can be done based on current results. For example, we can implement the multi-kernel idea to combine high accuracy models together in order to improve our overall accuracy. The reducing API by using TF-IDF can also be applied to other research areas while we want to save time complexity. New matrix with return types as its element also performs well on test accuracy. Thus, some further studies can be done on this feature. We can also extract out most common applications which are misclassified and understand the reason behind it.

	# of Uni Lib + Name	# of Uni API Name	# of Uni API Pack	# of Uni Return Types
Missed APP1	4064	2001	1281	728
Missed APP2	2346	1068	729	429
Missed APP3	2270	1035	683	413
Benign (Q1-Q3)	4062-9851	1395-3602	1446-3667	884-2069
Malware (Q1-Q3)	96-1224	71-174	38-433	34-276

Table 6: Misclassified Analysis

REFERENCES

[1] Yuxiao Dong, Nitesh V. Chawla, and Ananthram Swami. "Meta-path2vec: Scalable Representation Learning for Heterogeneous Networks". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 135–144. ISBN: 9781450348874. DOI: 10.1145/3097983.3098036. URL: https://doi.org/10.1145/3097983.3098036.

[2] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. 2016. arXiv: 1607.00653 [cs.SI].

[3] Shifu Hou et al. "HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 1507–1515. ISBN: 9781450348874. DOI: 10.1145/3097983.3098026. URL: https://doi.org/10.1145/3097983.3098026.

[4] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[5] Lucky Onwuzurike et al. "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version)". In: *ACM Trans. Priv. Secur.* 22.2 (Apr. 2019). ISSN: 2471-2566. DOI: 10.1145/3313391. URL: https://doi.org/10.1145/3313391.