Notebook

February 7, 2021

1 m2vDroid: Perturbation-resilient metapath-based Android Malware Detection

1.1 INTRODUCTION

Over the past decade, malware has established itself as a constant issue for the Android operating system. In 2018, Symantec reported that they blocked more than 10 thousand malicious Android apps per day, while nearly 3 quarters of Android devices remained on older versions of Android. With billions active Android devices, millions are only a swipe away from becoming victims. Naturally, automated machine learning-based detection systems have become commonplace solutions as they can drastically speed up the labeling process. However, it has been shown that many of these models are vulnerable to adversarial attacks, notably attacks that add redundant code to malware to consfuse detectors.

First, we introduce a new model that extends the Hindroid detection system by employing node embeddings using metapath2vec. We believe that the introduction of node embeddings will improve the performance of the model beyond the capabilities of HinDroid. Second, we intend to break these two models using a method similar to that proposed in the Android HIV paper. That is we train an adversarial model that perturbs malware such that a detector mislabels it as a benign app. We then measure the performance of each model after recursively feeding adversarial examples back into them. We believe that by doing so, our model will be able outperform the Hindroid implementation in its ability to label malware even after adversarial examples have been added.

1.2 METHODOLOGY

First, we will describe the details of our proposed model, m2vDroid, and then we will describe the method we used to attack each model in the Adversarial Attack section.

1.3 m2vDroid

m2vDroid is another malware detection model that we implemented that is largely based off HinDroid. However it uses node embeddings for the final vector representations of apps instead of the bag-of-APIs/commuting matrix solution that HinDroid applied.

1.3.1 Preliminaries

There are a few concepts that we should introduce before we get into details:

- *Definition 1)* A **Heterogeneous Information Network (HIN)** is a graph in which its nodes and edges have diferent types.

- *Definition 2)* A **Metapath** is a path within a HIN that follows certain node types. For example, let us define a HIN with a set of node types T and a path $P = n_1 \longrightarrow n_2 \longrightarrow ... \longrightarrow n_N$ of length N. P follows metapath $M_P = t_1 \longrightarrow t_2 \longrightarrow ... \longrightarrow t_N$ if $type(n_i) = t_i$ for all $i \in [1, 2, ..., N]$.

1.3.2 Feature extraction

Our ETL pipeline begins with Android apps in the form of APK files. These APKs are unpacked using Apktool to reveal the contents of the app, but we are primarily concerned with `classes.dex`, the app's bytecode. We decompile the bytecode using Smali into readable `.smali` text files. From here we extract each API call, along with the app and method it appears in, and the package it is from. This is done for every API in an app and for every app in a dataset, forming a table with the the information needed for the next step.

1.3.3 HIN Construction

Using the data extracted previously, we construct a heterogeneous information network using the Stellargraph library. Our HIN contains 4 types of nodes which we define as: - *Apps*: Android apps determined by name or md5, i.e. `com.microsoft.excel` or `09d347c6f4d7ec11b32083c0268cc570`. - *APIs*: APIs determined by their smali representation, i.e. `Lpackage/Class;->method();V` - *Packages*: the package an API originates from, i.e. `Lpackage`. - *Methods*: Methods (or "functions") that API calls appear in, i.e. `LclassPackage/class/method();V`.

The distinct nodes for each type correspond to the distinct values of their column in the API data table described earlier. *Apps* and *APIs* share an edge if an *API* is used within an *App*. Likewise with *APIs* and *Methods*, they share an edge if a *Method* contains an *API*. *Packages* and *APIs* share an edge if an *API* orginates from a *Package*.

1.3.4 Metapath2vec

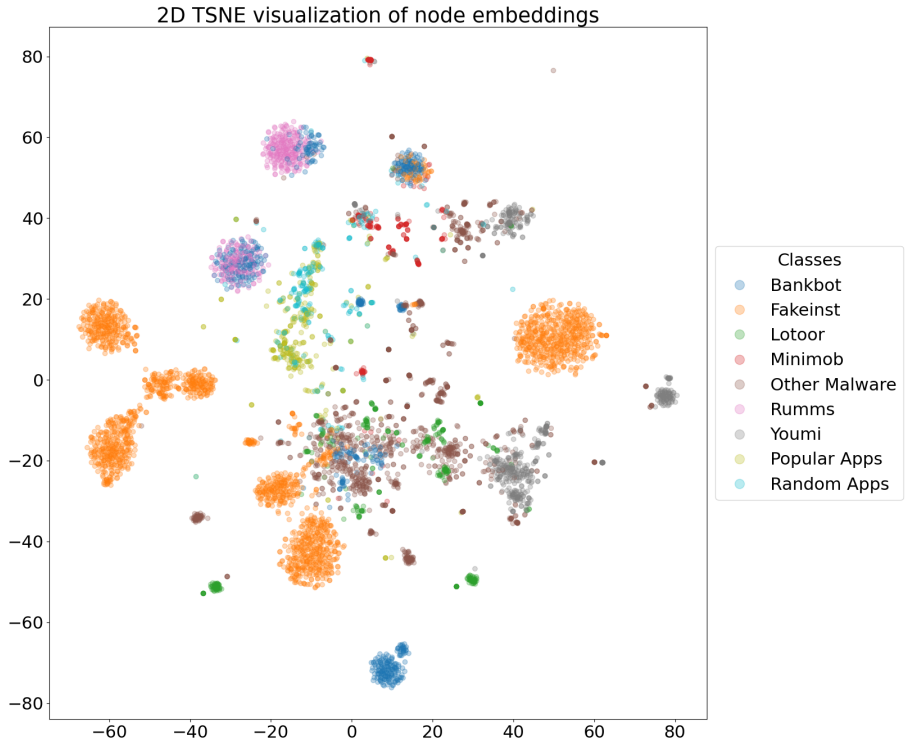
To generate our features, we apply the metapath2vec algorithm on the *App* nodes of our HIN. That is we 1) perform a random-walk starting from each app following designated metapaths to generate a corpus consisting of nodes in our HIN, then we 2) pass this corpus into the gensim implmentation of the word2vec model to transform each *App* into a vector. For the metapath walk, we leveraged Stellargraph's `MetaPathRandomWalk` algorithm and specified a walk length of 60, walking on each of the following metapaths 3 times per *App* node: - $App \rightarrow Api \rightarrow App$ - $App \rightarrow Api \rightarrow Method \rightarrow Api \rightarrow App$ - $App \rightarrow Api \rightarrow Package \rightarrow Api \rightarrow App$ - $App \rightarrow Api \rightarrow Package \rightarrow Api \rightarrow Method \rightarrow Api \rightarrow App$ - $App \rightarrow Api \rightarrow Method \rightarrow Api \rightarrow Package \rightarrow Api \rightarrow App$

We chose these metapaths as they formed some of the top performing kernels in the HinDroid paper that we were able to compare.

For word2vec, we used a skip-gram model trained over 5 epochs. We used a window size of 7 so that 2 connected apps could appear in a window even in the longest metapaths and `min_count=0` so that all nodes in the metapath walk were incorporated. We were sure to include negative sampling as part of the process, as negative samples would help further distinguish nodes the are not associated with each other. For this we specified `negative=5` for a final output of vector of length 128.

After running this ETL on our data, we observed clear clustering after plotting a TSNE transformation. For the most, part it seems that this method is able to distinguish between not only malware and non-malware, but can also distinguish between different classes of malware to a reasonable

extent. Notably, we have not tested the node2vec or metapath2vec++ algorithms for generating our random walk.



1.4 ADVERSARIAL ATTACK

Our adversarial attack follows many of the techniques applied by Chen, et al. 2018 to attack the MaMaDroid and Drebin models. To perform our adversarial attack, we simulated *Scenerio FB* with the following conditions: we have blackbox access to a malware classifier so we may query it to predict examples as they are generated and we also have access to the feature set with which to input into the classifier. The classifier may vary between m2vDroid and different kernels of HinDroid for initial benchmarks, but we will primarily use the m2vDroid classifier once we begin recursive training. The feature set will be the set of APIS derived from our training app set. The input vector will be the one-hot-encoded set of APIs for the example app. Our goal is to have the adversarial model add APIs to the example until it is misclassified or we reach a maximum. We want the app to retain its original function, so no APIs can be removed. Therefore, we modified the C&W attack from Android HIV to fit our application.

To be expanded in the future...

1.5 EXPERIMENT

To test our models, we used a dataset of 5,840 apps. 5,516 of these apps have been deemed malicious through other methods. We will use this set the malware set. 324 apps were selected at random from the Popular category (We are looking into increasing this number). We believe the apps from this category would be unlikely to contain malware, so we treated them as a benign app set. Between these apps, there were 5,160,691 distinct API calls from 551,624 packages, contained in 5,517,770 distinct method declarations.

We will compare 5 of Hindroid's best performing single-kernel models (AA^T, ABA^T, APA^T, $ABPB^TA^T$, $APBP^TA^T$) against m2vDroid at 3 steps: 1) Before adversarial training, 2) After 1 round of adversarial training, 3) After being retrained on the adversarial examples generated from step 2.

In step 1, we split the apps into train and test sets, training each model on the training set and evaluating their performance on the test set. At step 2, we select a 10% of all malware from both the training set and the test set to generate adversarial examples using the trained m2vDroid model as the classifier. Breaking m2vDroid should in theory break the HinDroid models as they rely the same graph structure. We also believe that this should put our model at a disadvantage and further prove its efficacy. At Step 3, we retrain each model on the training set and evaluate on the test set, but with adversarial examples replacing their original apps.

1.6 RESULTS

1.6.1 Baseline Model performance

The performance of each model is outlined below. - Initial performance of models on normal data - Performance after Android HIV trained on data - Performance of models

Intial performance (Note: Hindroid was trained/tested on the full dataset due to time limitations):

Model	acc	recall	f1	TP	TN	FP	FN
m2vDroid	0.923	0.994	0.959	1829	21	144	10
AA^T	0.983	0.994	0.990	5488	484	72	28
ABA^T	0.876	0.944	0.932	5208	113	443	308
APA^T	0.982	0.994	0.990	5488	479	77	28

1.7 REFERENCES

1.8 APPENDIX

1.8.1 Original Proposal

The data that will be used to build our model is benign and malignant android applications, which are then broken down into smali files. Our proposal is building the malware detection system that was created from the HinDroid paper, which was a malware detection system that used the metapaths generated from apps' code to connect apps through the API calls they contain. Our proposal would differ from simply recreating the model that was created in the HinDroid paper, as we would instead use the metapath2vec to generate a final representation of our network that would

then be used to train our model on. The benefits of using metapath2vec instead of simply just using HinDroid, which would allow us to use more sophisticated models, instead of the multi-kernel SVMs that were used in HinDroid. From this point, we can then continue to improve the robustness of our model by setting it against the Android HIV that perturbs the source code of malware that makes it more difficult to detect. As a result, our new model would be able to detect the malware even when we take Android HIV into consideration, which would make it harder for malware to evade detection. The output of our project would be a model that would be able to determine whether there is malware or not within an app based on the app, api calls, and codeblocks.

To set a baseline, we explored a sample of the apps we were given access to. We explored 150 apps, 75 known malicious apps and 75 popular apps we can presume benign. These apps tended to be smaller and contained less API calls overall. Due to this disparity, we normalized the data by the number of APIs each app contained.

malware	apps	num_apis	num_unique_apis
False	75	158266.933333	34913.093333
True	75	15060.320000	3590.706667