Problem Introduction

The problem investigated in this setting is to detect malware (malicious software) from benign software based on applying innovatively new methods studying the smali code of the application. Nowadays, Android is dominating the smart phone market as an open source and customizable operating system, attracting attackers to disseminate malware, posing a serious threat to users and the smartphone market. And historically, major defense against malware attack is mobile security products, such as Norton and Lookout, which relies on signature-based method to recognize threats. Such traditional methods can be easily bypassed by techniques such as code obfuscation and repackaging, rendering them vulnerable in front of more sophisticated malwares. Generally, this is an inherent shortcoming of static analysis which one cannot get around. However, more resilient method can be performing very well and make malwares harder to bypass the detection.

Here, we implemented an innovative method based on the structured heterogeneous information network [1]. This method analyzes different relationships between API calls within the codes and create higher-level semantics for better malware detection. Thus, the data needed is smali code, a human readable representation of binary Dalvik bytecode of each Android App. To make a complete dataset, we would need both benign Android Applications and malware samples. We will collect benign Android App codes from apkpure.com and decompile the apk file to smali code by methods introduced in part 2. Smali codes of malware samples will be provided by class professor. Since the malware samples retrieved is from the professor, there might be limitations of sample size. Also, there might be biases in the malware set such as they are easily accessible and no more hazardous since they are provided to us.

Data and Data Ingestion Process

The data source for the benign Android Apps is from apkpure.com, which allows all user agent to scrape data based upon its robots.txt file. Also, apkpure provides https://apkpure.com/sitemap.xml for us to easily access the addresses of all kinds of Android Apps from the xml file. Apk files scraped will be converted to Smali code by apktool, which can decode resources to nearly original form and rebuild them after making modifications. Since we are using it only for academic purpose and do not plan on making modifications on the original form, we will strictly enforce the terms and conditions to ensure our process is legal. After converting apk files to smali codes, which is human readable in smali language. And the smali codes of each App serves as the raw data for our prediction model. The data of malwares are provided by Prof. Aaron Fraenkel at HDSI. The malwares data are already in smali codes, so no more preprocessing is needed. However, possible drawbacks of this data source are that such malwares are old-fashioned that they are in storage for a long time hold by Prof. Fraenkel. Thus, the model we trained upon these malwares might not have a equal descent performance on most up-to-dated ones. Detailed data injection process will be described below.

First of all, we will retrieve all valid xml files that contains linkages to the download page of the app. And every xml file has linkages for a certain category of Apps. Rather than trying to capture the whole population of Apps on apkpure.com by enforcing an even number of Apps in each category, I choose to ensure that our sample mimics the real-world situation mostly by using random sample to retrieve certain number of xml files, maintaining the natural distribution of App categories. Then, within the xml file, we sample one link out to download the App. One

possible drawback of this sampling method is that malwares might massively concentrated in an underrepresented category. Then, random sampling will make this model poor at distinguishing malwares from benign Apps in this category. Thus, another sampling method that should be tried out in the future is to enforce a certain number of samples from each category. However, this way, the natural distribution of each category will be destroyed.

After getting the apk files, we will save them to the apk_file (decided by configuration file) directory as AppName.apk. And then, we will use apktool to convert apk files to smali code and save them to smali_file (decided by configuration file) directory as AppName folder. Since the only interest of this model is the smali code, all others will be deleted for space saving.

Data ingestion process is wrapped up as pipelines with codes and configurations separated. Thus, this pipeline is highly applicable towards other webpages for apk file downloads with small modifications. Thus, the data source will be more abundant if such websites are found.

In run.py, it takes in smali codes generated by data-injestion.py and works on generating graph representation of each app. And then different features are extracted from theses graph representation as raw data for model training to differentiate between each category of apps.

```
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 1: An example of smali code

Feature Extraction and Graph Definitions

After transformed all apps to smali codes, here we will introduce how each Android apps is represented. In figure one, we can see how a portion of smali code is for an app. API calls are used by the Android apps in order to access operating system functionality and system resources. Thus, they can be used as representations of the behaviors of an Android app. An API call in smali code refers to those followed an invoke method. For example, in Figure 1, "Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime" is an API call invoked by the method "static {}". Furthermore, the relations among different API calls can also imply import information as they specify how the APIs are related in a certain way. To further explore such relationship, two new definitions are introduced. One is code block, defined as the code between a pair of ".method" and ".end method" in the smali file. For example, "Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime" and "Ljava/lang/Runtime;->exec(Ljava/lang/String:)Ljava/lang/Process" in Figure 1 are within the same code block as they are within the same ".method" and ".end method". Another one is

package, which is the first part of an API and it indicate the general intent of an API. For example, “Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime” and “Ljava/lang/Runtime;->exec(Ljava/lang/String:)Ljava/lang/Process” are both from “Ljava/lang/Runtime” package. Based upon these definitions, we will build three different graphs to represent each app.

Graph A

This is graph in which describes the relationship between Apps and APIs. The node in A can be APIs and APPs. If an API is called within an APP, then, there is an edge between this APP and API. In order to extract all API calls used in the app, we combine all smali files of an app, and uses regex to parse API calls following by "invoke" method.

Graph B

This graph is used to indicate whether two APIs is within the same code block, in order to catch groups of APIs coexist within the same method. In graph B, for all APIs within the same code block, there is an edge between all possible pairs. In order to extract all APIs in the same code block, we use regex pattern to find all code blocks in the combined smali file, and extract APIs that are within each code block for edge connections.

Graph P

Except for that whether the API calls co-exist in the same code block, API calls which belong to the same package always show similar intent. To represent the relationship of existing in the

same package among APIs, we generate the graph P whose nodes are APIs and two APIs are connected by an edge if they are in the same package.

EDA And Data Cleaning

In this baseline model, apps are from two different categories. One is malware while another is benign wares. I have extracted 200 malware Apps and 200 benign wares. In the process of feature extraction, some Apps are empty that they contain 0 APIs. Without API calls, then I cannot extract related features of this APP. Thus, I decided to delete the APPs and resample a new one whenever it is empty. This process guarantees that all APPs can be transformed to the proposed features. The target here is to extract different features based on graph representation and train a baseline model to predict its category. Here, I extract the number of edges and number of degrees for each graph of the APP as features. The average number of nodes in strategy game is 13702 while that in board game is 8923. This is agreed with common sense as strategy game are way more complicated than a board game, thus should have more nodes. Beyond this, I also calculate the degree of each node in each graph. Then I calculate the min, max, and mean for all degrees within the same node as features.

Baseline Classification Model

With features extracted above, I tried three different classifiers on categorizing the APPs as either "strategy game" or "board game". The three classifiers are logistic regression, random forest and gradient boosted tree. With a train test split of 2:1, the test accuracies are 0.60, 0.80, 0.73 respectively. Except accuracy, another important metric that I incorporate is True Positive

Rate, which indicates how many malwares have the model detected from all those true malwares. This metric is important as malware predictions care more about capturing all true malwares even by mistakenly classify some benign wares. This is because the harm by a leaked malwares can be huge while shielding a benign ware doesn't cause as much damage. The features that we extracted are: num_api, num_unique_api, num_code_block, num_unique_package for each app. The baseline result is shown below.

	ACC	True Positive Rate
Logistic Regression	0.826	0.810
Random Forest	0.894	0.905
Gradient Boosted Tree	0.894	0.905

HinDroid Approach

The HinDroid approach is basically creating a heterogeneous information network (HIN) from different meta-path joined by the three different graphs (A, B, P). HIN is a graph containing multiple types of node entity. It can not only provide the network structure of the data but also provides a high-level abstraction of the categorical association. Here, we have two entity types, i.e., Android app and API calls. There are three types of relations, e.g., app containing API call (graph A), API calls in the same code block (graph B), and API calls with the same package name (graph P). We first covert graph A, B, P to matrices. E.g., graph A is converted to matrices A such that rows are apps and columns are APIs. If app i connects APP j in the graph A, then matrix $A_{i,j} = 1$. The same idea applies to graph B and P to convert to matrices. Then, different meta-path connecting these relationships can formulate different semantic of higher-order

relationships among entities. For example, AA^T is a meta-path $App \rightarrow API \rightarrow APP$, which is indicating all possible ways that two apps can be connected by a common API. Each feature vector in AA^T can be regarded as using a bag-of-APIs to represent an app. More complicated similarities can be defined more complicated meta-path, e.g. ABA^T, APA^T and $APBP^TA^T$. Finally, different meta-path can be considered as a precomputed kernel matrix in Support Vector Machine for prediction task. This is because, after meta-path, apps are transformed into a different dimension represented by high-level abstraction based upon the APIs that each APP have. In prediction stage, we construct the matrix A from a bunch of testing apps. The features representation of the testing apps is represented by the same meta-path but changes the first training A to the testing A.

Results

With a train-test split 2:1 and training on 200 benign apps and 200 malwares, the results for different meta-paths are summarized in the table below.

	Train Acc	Train TPR	Test Acc	Test TPR
AA	0.985	0.971	0.977	0.968
ABA	0.877	0.869	0.826	0.746
APA	0.985	0.971	0.947	0.984
APBPA	0.836	0.883	0.780	0.714

Conclusion

The result is very interesting that for meta-path including B matrix, the performance is not so well, even worse than baseline. One possible explanation is that we still have trained too few Apps. Then, the code-block may not catch much useful information. However, if we look at AA and APA, they've already been pretty close or even better than the HinDroid paper. One possible limitation that this result have is that it only trains upon 400 apps. With more apps feed in, the meta-path can learn more comprehensive relationship among APPs and further improve the model performance. Also, we've only experimented with four meta-paths. The HinDroid paper have more meta-path experimented. We could also experiment with those meta-paths in the future or invent and trial out new meta-paths.

[1] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. Hin-Droid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In KDD '17. 1507--1515.