Umang Saraf
A13595221

Malware detection

Our goal for this project was to classify apps into Malware or Benign by analysing the relationships between the APIs and app. We represent apps and its API's as a structured heterogeneous network. Using the meta-paths between them, we create an adjacency matrix which is used to aggregate similarity measures over the apps. We then create multi-kernels and fit in a machine learning model which assigns the weight through learning and makes the prediction. We create our own dataset of Benign apps from the APKpure website and use the AMD malware dataset for the malware apps.

Data Collection

The benign app dataset is collected from Apkpure's website. The sitemap of apkure, contains gz files, which when compressed return an XML file containing 1000 links to different app's home page. These .gz files on the sitemap are sorted based on categories, with over 40 categories and 7000 .gz files. The code first obtains all links to the .gz file and randomly selects .gz files. It then creates the xml file into a soup object and randomly selects links embedded in the file.
Once we obtain the link to the homepage of the App, we send a request to it, extract its HTML and then convert it to a soup object. We then find the link to the download page of the APK of that app embedded in the html of the app's home page. We again request the new extracted link, create a soup object and then get the download link to the APK. We send another request to the download link and write its content onto an .apk file. Once we get the .apk file we decompile it into smali code using apktool kit and save it in a directory given in the config file. The dataset in total comprises 1000 apps. All the APKs are stored inside the APK folder and all the decompile smali code is stored inside the smali folder.

Steps for collecting APK's

- Scrape all links from the sitemap of APKpure. These links are gzip files which are sorted on the basis of category of apps and each of these files contains like to app within that category
- Randomly sample the gzip link and download it
- Randomly sample 20 app links from the downloaded gzip file and then download the APK of the app to the apk folder in data

Keep continuing this process till we've reached the number of apps needed to collect

Once we have all the APK's for all the apps downloaded in the APK folder, we one by once decompli all of them in smali code and then save it to the smali folder in the data directory.

Shortcomings

The data we may collect may not well represent all the apps on the android platform. The website APKpure contains APK's for 7 million apps and for over 40 different categories. The data we collect is much smaller than the actual number of apps. We may not have a good representation of different kinds of apps based on categories, popularity and number of votes. Also, The dataset may be really imbalanced since we have a large amount of benign apps but a really small amount of malware apps. A trivial implementation to classify apps as malware or not will not work since the api calls between a benign app and malware app may be really similar. Also, each app contains a lot of irrelevant smali files that are contained in each app. This may take a lot of extra space making it harder to take it to a larger scale.

Legality

On the robots.txt file of the apkpure it allows us to scrape all the content on apkpure. There were certain apps we encountered that required another level of verification to download their APK. At the moment we skipped all the links that required another level of verification.

Data Cleaning

To create graph structures from the resulting smali folders, we first need to clean all smali files and extract the API's and the relationships between them. We parse through all the .smali files in the apps and extract all the API's that are present in a code block. An example of an API -
Ljava/lang/Runtime; →getRuntime() Ljava/lang/Runtime

The data cleaning returns three different structures once it has completed running and each of them is used to build a different adjacency matrix. They're structured in a manner that would ensure the matrices are created in the fastest time.

App_to_api

This structure is created to create the app matrix. It contains the app name as the key and a list of all API's that are present in the app.

App_to_api = {"app_name_1": [api1, ap12,......],

,

“app_name_N”: [api1, ap12,......] }

Code_block

Our second structure is again a dictionary which contains API as the key and all the API’s that have occurred in the same code block with that key. This structure is used to create the B matrix

```
code_block = {“api_1”: (api2, api3,...... ),
                                                    ,
              “api_n”: [api1, ap2,...... )
                                                    }
```

Library_dic

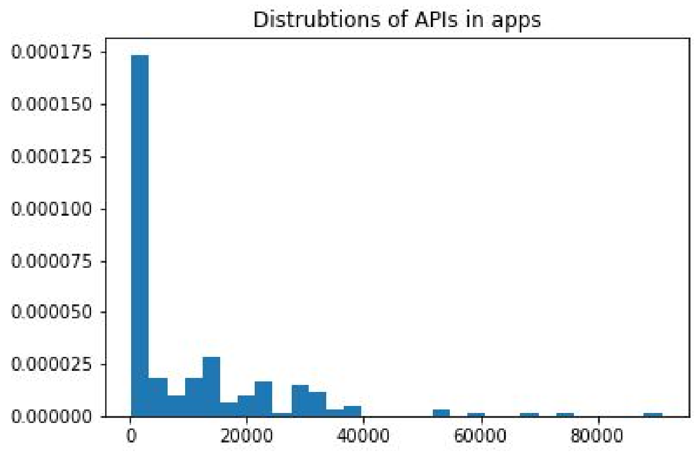
Our third and final structure is a dictionary as library name and the values and a set a of API’s that have that library. This matrix is used to create the P matrix.

```
Library_dic = {“lib_1”: (api2, api3,...... ),
              …,
              “lib_n”: [api1, ap2,...... )
                                                    }
```

These structures are all stored in the folder processed inside data. They are all save as JSON files

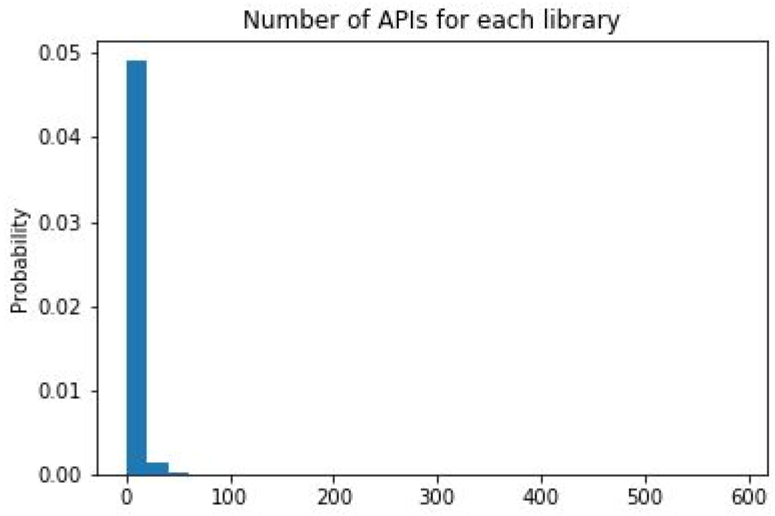
EDA

Mean number of API per app - 10696
Mean number of API per Benign app - 19121
Mean number of API per Malware app - 1920



Top 5 most common library -

- *Lcom/google/android/gms/games/internal/IGamesService$Stub*
- *Lcom/google/android/gms/games/internal/GamesClientImpl*
- *Landroid/support/v7/widget/RecyclerView*
- *Lcom/google/android/gms/games/internal/IGamesService*
- *Landroid/view/View*



Top 5 most common API's occuring in code block

- *Ljava/lang/StringBuilder;->append()*
- *Ljava/lang/StringBuilder;->toString(),*
- *Ljava/lang/StringBuilder;-><init>()*
- *Ljava/util/Iterator;->next()*
- Ljava/util/Iterator;->hasNext()

Feature Extraction

After the data is cleaned and created into the structure required, it is used to create the fesatures. Our features our graph structures that define different relations between apps and apps, apps and APIs and APIs and APIs. There are three different matrices created, A, B and P for each of these relations (Definitions of each of these matrices are defined in the section Hindroid approach).

All three matrices were first initialized as lil_matrix which is a sparse type matrix used for fast indexing. The lil_matrix is then populated with the edges of the relationship we're defining.

Once we’ve populated the lil_matrix, we transform it into a crs_sparse matrix which is a compressed sparse row format. Sparse.crs type matrix is used since its much faster for matrix multiplications.

Baseline model

For the baseline model we classified benign apps based on 2 categories, 'Books and reference' and 'Education'.

I create a model to check what category a given app falls into. I hand craft features to feed in a model. The features that I use are

- **unique_api** - Count of unique_api in app
- **number_unique_libary** - Count of number_unique_libary in app
- **most_common_lib** - Name of most common Library in an APP
- **total_api** - Count of total number of API's in an APP
- **most_Common_api** - Name of most common API in an APP
- **app_name** - Name of the APP

I feed all these features to a logistic regression model. All the categorical features are one hot encoded while the numerical categories are one hot encoded. Running the model on a training set of 50 apps, I get an accuracy of 93%.

The F1 score I found was 86%. We use the F1 score since the model may be unbalanced and the F1 score gives us a better idea about the model.

Hindroid Approach and Graphs

The Hindroid approach represents the Android applications (apps), related APIs, and their rich relationships as a structured heterogeneous information network (HIN).
HIN is a graph structure that provides network structure of the data and is an high level abstraction to categorical associations. To develop HIN, the paper definned 4 different types of relationships between the 2 entity types, apps and API’s.

- Relationship **A** is defined between apps and all the API’s in the app.
- Relationship **B** is defined between API’s and API’s in the same code block
- Relationship **P** is defined between APIs and API’s that share the same library
- Relationship **I** is defined between APIs and APIs that have the same invoke type.

Metapath's can be then derived from these relationships. A simple example of a meta-path can be APP ---->API ----> APP. Meta paths are used to define semantics of higher order between the entities. This meta-path means we connect two apps through the same API they share. Just like this several different meta paths can be created using the relationships defined.

The paper defines a multi-kernel learning approach that combines different meta path to compute a commuting matrix which is regarded as the kernel. The multi-kernel is created to incorporate different similarities of the meta path and creating the kernel gives wightes to these meta paths. Some of the multi-kernel defined by the paper are - AA^T , AIA^T and $ABPB^TA^{T.}$ The kernel created is then fit into a SVM model as features and with labels malware and benign.

API calls in android are used to access the operating system functionality and system resources. Therefore these apps can be used as representations of the apps behavior. The hindoid paper hence makes use of the API to create networks of how different categories of apps, malware and benign, relate to each other. An example in the paper is mentioned on how in the malware app two API calls were seen in the same code block that were common APIs in the benign app but never seen in the same code block. This means that the apps was malware and was trying to load ransomware. The Hindoid paper used these APIto create a heterogeneous structured network that when fed to a machine learning model as features will be able to identify the difference in how APIs react in maware and benign apps.

Model

Once we create multiple the matrix to obtain the multi-kernel, we get an array from the sparse matrix of the kernel and convert it to a pandas Dataframe. The app names are appended and type of apps is added as columns to the dataframe.

The Data Frame is then fit into a Linear SVC model with the dense array as the features and the label as the output. We repeat this step for all the kernels and then calculate different performance metrics

Results

Our test set consisted of a total of 800 apps out of which 400 were benign and 400 were malware. The test set consisted of 357 apps, out of which 189 malware and 168 benign.

Table 1: Model performance evaluations

Method	F1	Accuracy	TP	FP	TN	FN
AA^T	.984	98.31%	186	3	165	3
ABA^T	.9466	94.1%	186	18	150	3
APA^T	.9632	96.07%	183	8	160	6
$APBPB^TA^T$	.84	84.1%	170	50	124	6

Conclusions

Our model performance was on par with Hindroids' perfromae and some kernels having a higher accuracy rate. Findings on model prediction on different kernels -

- Kernel AA^T performed the best overall performance in terms of accuracy and the highest number of benign apps correctly classified.
- Both kernels AA^T and ABA^T classified the highest number of malware apps but ABA^T had a much higher FP.
- Even though ABA^T classified more malware apps than APA^T, APA^T had a higher overall accuracy.
- $APBPB^TA^T$ has the lowest accuracy among all the kernels.

For the task of identifying malware, it is important that we have a low amount of false negatives i.e reduce the number of malware apps being labeled as Benign. The model trained on Kernel AA^T returns the least number of False negatives and the best accuracy score.

Improvements

All kernels were able to run on a 32g dram except the $ABPBA^T$ kernel. The kernel required 48gb DRAM as it was too big to exist on the memory. Our current implementation wouldn't be able to be scaled to a much larger dataset because of the large size of the matrices. Our current implementation for 1100 apps returned 2 million unique API which creates a sparse matrix of two million rows and columns. In future more work will be needed to reduce the number of API's being extracted from the smali files or reduce reading the number of smali files.