Improving and Explaining HinDroid

Umang Saraf
Liam McCarthy
Daniel Alemu

Abstract

Recent work introduced a model using a Heterogeneous Information Network (HIN) representation of Android applications utilizing a meta-path approach to link applications through the API calls contained within them. It was found with multi-kernel learning, the model was able to identify malicious applications with high accuracy. This recent work was the first approach of this kind to be published; therefore, a replication process would allow for deeper understanding of this approach. In this paper, we introduce a framework for improving upon the model through scalability and testable measures with the purpose of maintaining or increasing accuracy while creating an easily executable pipeline. In particular, we employ dimensionality reduction and stochastic techniques to achieve reasonably replicable results. Additionally, we attempt to understand, through model explainability practices, the inner mechanisms of the complex model to better understand possible inaccuracies which may arise in creating a scaled version of a HIN approach.

Introduction

An estimated 1,500 apps are downloaded per day and with the potential for users to unknowingly download a malicious application there needs to be a way to find and remove such applications. The HinDroid paper attempts to address this problem by creating a model that can identify the difference between a malicious and benign application using the decompiled code of that application. The work being done in this field helps protect the information, data, and wellbeing of Android users. It’s important to continue this work and build on it because hackers and malware developers are always evolving their methods so the security industry must do the same.The first part of our project explores different methods that focus on scalability of the project particularly pertaining to methods involving reducing the number of unique API calls required in the classification task. We test several different methods and techniques that help us reduce the number of API’s and scale up the project while maintaining the model performance. We use different multi kernel learning algorithms that involve using all different meta paths created by learning weights of each metapath and combining them into one kernel.

Our second part of the project investigates how thorough the classification of malware is in the Hindroid paper and also improves upon the Hindorid model by exploring different methods not mentioned in the paper. We look at what Hindroid classifies well and what it doesn't classify well. These incorrectly detected malware apps are the most detrimental because an incorrectly detected malware app means that a person could be downloading an app that could steal their bank information, hold their phone for ransom, or mangle the software in any number of ways. We build the same linear SVM discussed in HinDroid but thoroughly investigate the explainability of such a model utilizing the very easily interpretable model coefficients. We use these coefficients of the linear SVM model to explain the decisions of the classifier based on weights that are either extremely positive or extremely negative and look into why certain features have this effect on the classification. This helps us understand where false positives and false negatives are coming from, which packages or code blocks are they associated with, and so forth.

Data Generation Process

Our dataset is compiled from two different sources depending upon the type of app. The *maware* apps are provided from AMD and present on the DSMLP server. The server contains a total of 4500 apps, out of which 1200 are randomly sampled for training and 280 apps are samples for our test set. The dataset contains malware apps from various different families such as Bankbot, Minimob, SimpleLocker etc. The dataset obtained from AMD website is in smalu files.

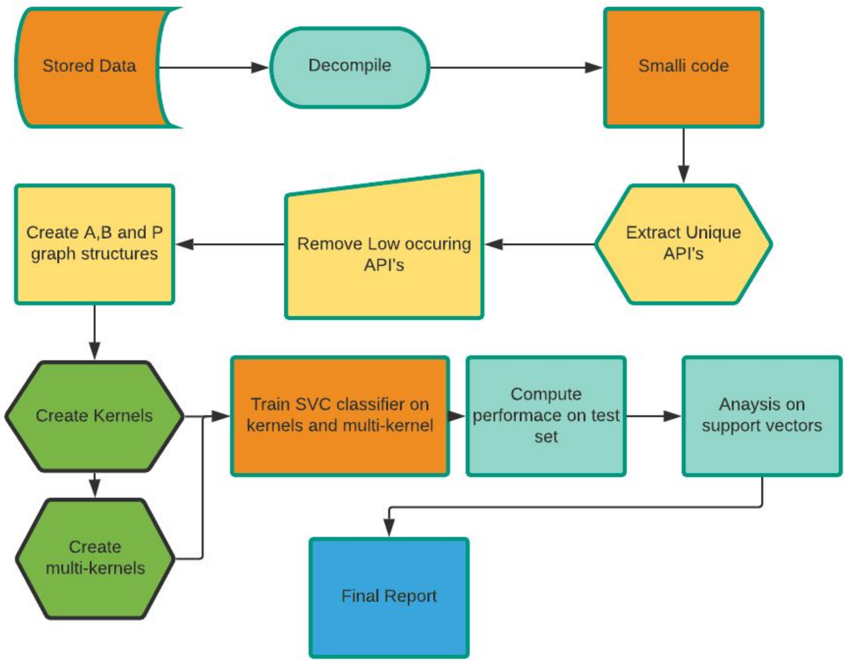
For the *benign* applications we create our own dataset by collecting apps from the Apkpure (https://apkpure.com) website
Scraping Process - The sitemap of the Apkpure website contains links to existing apps in its database. The sitemap of apkure link contained gz files, which when compressed returned an XML file containing 1000 links to the apkpure app's home page. The .gz file on the sitemap is sorted by categories with over 40 categories and 7000 .gz files. We first obtain all links to the .gz file and randomly select .gz files based on categories and total number of links needed to be collected. We then create the xml file into a soup object and randomly select 20 links embedded in the file. We follow such practices to make sure our benign dataset is not biased towards a certain category of benign apps. The APK file for the app is downloaded from the website. We collect a total of 1500 apps, out of which 1226 are randomly sampled for the training set and the remaining 274 are used in the test set.

System overview

A downloaded APK file for the benign apps usually contains *AndroidManifest.xml, classes.dex,* and *resources.arsc file*; as well as a *META-INF* and *res* folder. The .dex file is an unreadable file and needs to be converted into a readable one. Using the APK tool kit we extract smali files for each app that is human readable and can be analysed to find out if an app is malware or benign.

Decompile - We first decompile the .dex files into smali files using the APK tool kit [1].

Cleaning and extractions - Once we have the smali files we extract the unique API's from each app, remove certain API's that aren't detrimental in the classification tasks and store them in three different dictionary structures, each appropraire for the task in hand



Stored Data
Decompile
Smalli code
Extract Unique API's
Remove Low occuring API's
Create A,B and P graph structures
Create Kernels
Create multi-kernels
Train SVC classifier on kernels and multi-kernel
Compute performace on test set
Anaysis on support vectors
Final Report

Graph structures - Once we have all the extracted API's in the three different dictionary structures, we used each of them to create three different graph structures that are used to create the kernels.

Kernels - The three different graph structures are used to create the kernels by taking a dot product between the graph structures. The resulting product is defined as a kernel which is then used to train the SVM model on.

Multi-kernel - A multi kernel is created, which is a combination of all the previous kernels created. The weights for each of the kernels are learned using a multi kernel learning algorithm and after the weights are learned, they are multiplied with each of the kernels to create the multi kernel. The Linear SVM model is then trained on the multi kernel

Kernel on test set - Similarly as for training set, kernels and multi kernels and created for the test set and the performance for each of the kernels is computed on the test set. The kernels for the test set only encode the API's that are seen in the training set. *For eg*. if we receive an app that has API's that are never seen in our training set, the Kernel for that app will have 0's all across the border.

Support vector analysis - Once the model has been trained and tested, the Support Vectors and the coefficients assigned to them are investigated. The coefficients are ranked and the most extreme positive and negative coefficients and their vectors are analyzed for any trends or anomalies.

Methods

Our first step 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*

Data cleaning

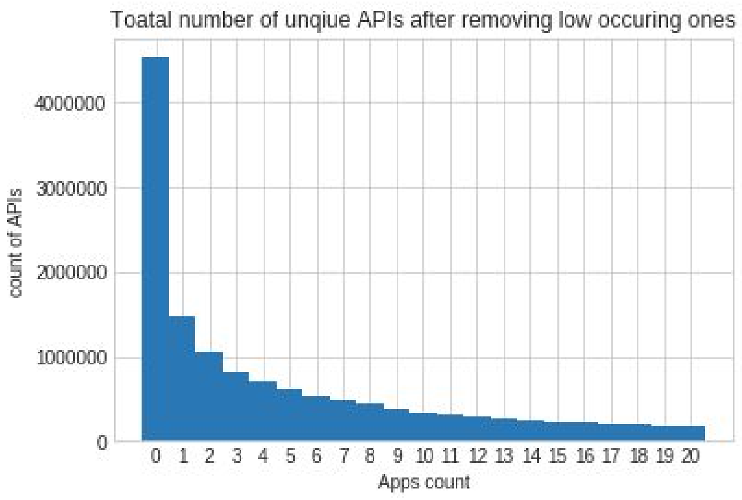
For this project our analysis consisted of around 2400 apps, with the same amount of benign and malware and 360 apps for testing. Making use of last quarters code, we cleaned the dataset to extract all the data structures required for the creation of the matrices and got a list of unique API's. For 2400 apps, we received a total of 4.8 million unique API. This was an issue as the matrices were too massive and the matrix calculation was not practically feasible. So our first step was to reduce the number of API's we were collecting. Here are the steps taken to do so.

1. We noticed that several smali file names had a $ sign in its name. We found out that it was a java naming convention where inner classes were donated with a $ sign. The name of the inner class accompanied the $ sign and if a number accompanied the $ sign it means it was an anonymous inner class. For eg.
    a. *Testouter.smali*
    b. *Testouter$inner.smali*
    c. *Testouter$1.smali*

An inner class is usually a private or protected class meaning the API's call made within these classes will be mostly unique and won't be seen commonly among other API's. Hence the decision was made to ignore all such files with a $ sign name in it

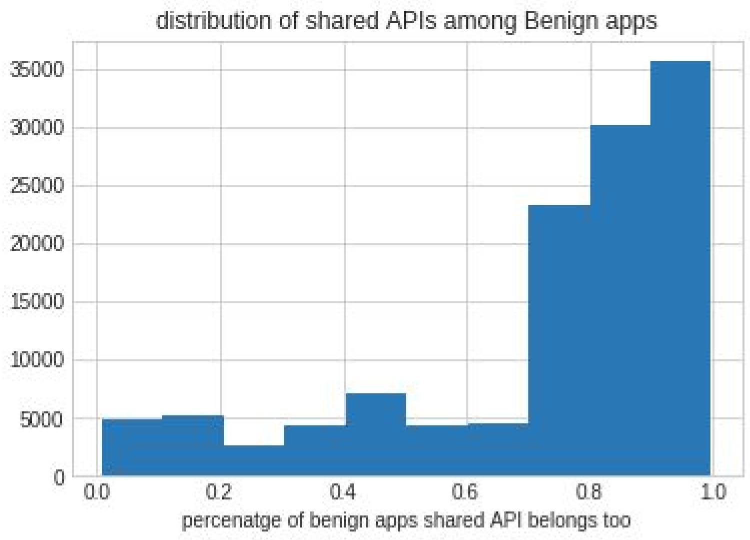
2. We also noticed that there were two types of method calls, private and public. Applying the same logic as above, we ignore all API calls that occurred between a private code block

Applying the above 2 techniques on our dataset, we saw the number of unique API's went down from 4.8 million to 4.3 million.



3. Out of the 4.3 million unique API's we noticed that a majority of them only occured in one or a small subset of the total apps. Out of the 4.3 million unique API's, 2 million of the API's were just seen in one app. We then decided to remove all API's that occured in less than 5 apps, which reduced our number of unique API's just to 650K. We also created training sets with API's removed occurring in less than 10 and 20 apps.

4. Lastly, we explored another method involved removing all the API's that commonly occur in both malware and benign. We did so as we believed that these API's that were common in both malware and benign won't be important in making the classification decision. Out of the 650k remaining API's we saw that 110k of those APIs were shared among both benign and malware. We then defined an index to find the percentage of these common API's that were found in benign. We removed all such common API's that were heavily present in both malware and benign. All the shared API's that were present between the split between 10% to 90% of the benign apps were removed. This left us with only API's that



were either only present in malware or benign or if even shared, they were mostly present in one type.

After the unique API's are extracted and some are removed, we create three different structures 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. They layout of all the three structure looks like -

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 which has keys as library names and the values as a set 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,......)

}

Graph structures and Kernels

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 a 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

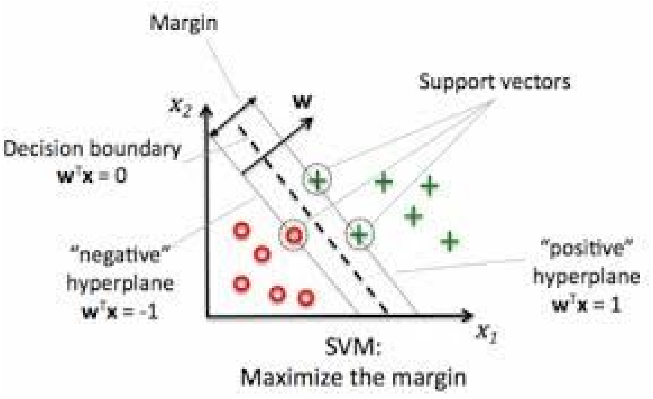
Metapath's can be derived from these relationships. A simple example of a meta-path can be APP ---->API ----> APP. Meta paths are used to define semantics of higher order between the entities. This meta-path means we connect two apps through the same API they share. Just like this several different meta paths can be created using the relationships defined. We create 4 such different kernels which include - AA^T ,ABA^T, APA^T and APBPA^T, . The kernel created is then fit into a SVM model as features and with labels malware and benign. Some of the kernels defined by the paper are 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 API to 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.

Multi-Kernel

We also create something known as a multi-kernel which is a combiannations of all the previous created kernels using the mklaren algorithm which is entirely based on geometrical concepts. The algorithm does not require access to full kernel matrices yet it accounts for the correlations between all kernels [2]. Each of the kernels are first transformed into polynomial kernels which turns it into a sparse matrix making it easier for the model to learn the corresponding weights. Using the Alignf model, which learns the weight by considering the correlation between the kernels when maximizing the alignment, we find the corresponding weights for each of these kernels, then multiply the kernels with the weights and add all the resulting kernels to create a multi kernel. The resulting multi-kernel is then similarly used to train the Linear SVM model

Support Vector Analysis

The Hindroid paper specifies using a Linear Support Vector Machine (SVM) model because the utilization of kernels of the HIN "use more expressive representation for the data, and build the connection between the higher-level semantics of the data and the final results" [3]. This means the calculated kernels which represent higher dimensional relationships are better suited than traditional feature engineering. Thus, we are able to leverage this fact to better understand the decisions the model is making by looking into the components of the SVM.



Depicted on the left is the basic idea for a Linear SVM model. A decision boundary is learned based on the support vectors. The positive hyperplane in this case is benign apps and the negative hyperplane refers to malware apps. The weights of the support vectors which is denoted by the W vector. The weights that are extremely positive or extremely negative correlate to vectors that represent benign or malware apps respectively. This premise is the basis for our analysis because these extreme coefficients (either positive or negative) could lead to the model making mistakes on different or larger test sets. For example, if a vector is assigned an extremely positive coefficient but actually corresponds to a malware app, this could lead to malware apps similar to it being wrongly classified as benign. So, in analyzing these coefficients and the vectors they correspond to, we can begin to understand the way the model works in hopes of identifying possible errors or improvements.

In order to actually conduct the analysis on these weights and vectors, the model needs to be trained on the kernels to create the coefficients. Thankfully, with the help of the Linear SVM provided through Sci-Kit Learn the model coefficients are easy to obtain. After the coefficients are gathered and assigned to the app vector they correspond to, they are ranked to find the most extremely positive and the most extremely negative coefficients as specified above. We only looked at the top ten positive and negative coefficients for our analysis in order to get the best picture of the vectors that are weighted the most. These apps are then validated to see if the model is assigning the correct coefficient to the correct category of application. That is, we check to make sure that positive coefficient weights correspond to benign apps while negative coefficient weights correspond to malware apps. If there are any inconsistencies, such as a benign app being given an extreme negative weight, then this would be cause for concern as this could lead to errors of the model classifying benign apps as malware and, even more detrimental, malware apps as benign. These app vectors that may be inconsistent will then be investigated as to why they may be getting assigned a weight contradictory to their true classification. Additionally, since classifying a malware app as benign is a severe threat to the security of users, we use the test set to obtain predictions of whether or not the apps or malware or benign using the trained Linear SVM and find which Malware apps were falsely predicted to be malware. The vectors of these false positive malware apps are investigated in a way similar to the apps that are given contradictory coefficients. In both the analysis, we work to understand what the distribution of the vector values are in comparison to each other and we further contextualize them to the kernel that is used to create the Linear SVM model. This analysis consists of gathering statistics and figures that represent the space of malware and benign apps within the kernel matrix. Once these statistics are gathered in an Exploratory Data Analysis (EDA) of the kernel, they are then compared to the inconsistent classifications and contradictory coefficients in order to fully understand if there are any trends or anomalies occurring within the data. This will lead us to discovering how and why the model is making mistakes and what, if anything, can be done to improve upon the model.

Results

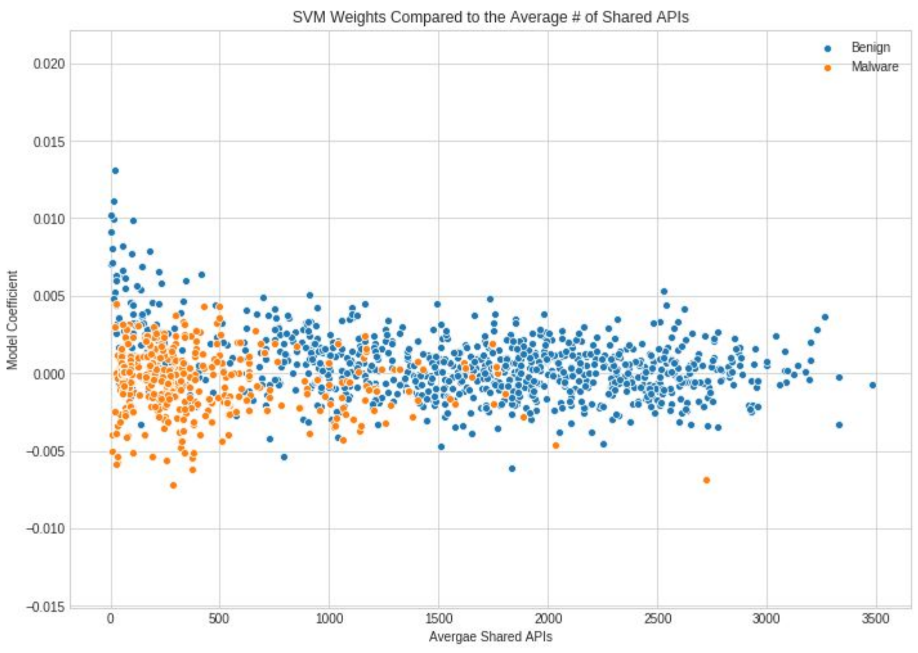
Kernels with API's removed occurring in 5 apps or less						
Kernel	Accuracy	TP	FN	FP	TN	F1-score
A.A^T	95.66%	276	4	20	254	95.48%
A.B^T	87.1%	269	11	60	214	85.7%
A.P.A^T	91.5%	270	10	37	237	90.9%
A.P.B.P.A^T	77.9%	269	11	111	163	72.7%
Multi kernel	**96.2%**	**276**	**4**	**17**	**17**	**96.07%**

Kernels with API's removed occurring in 10 apps or less						
Kernel	Accuracy	TP	FN	FP	TN	F1-score
A.A^T	94.7%	275	5	24	250	94.5%
A.B^T	86.1%	268	12	65	209	84%
A.P.A^T	88.4%	269	11	53	221	87.3%

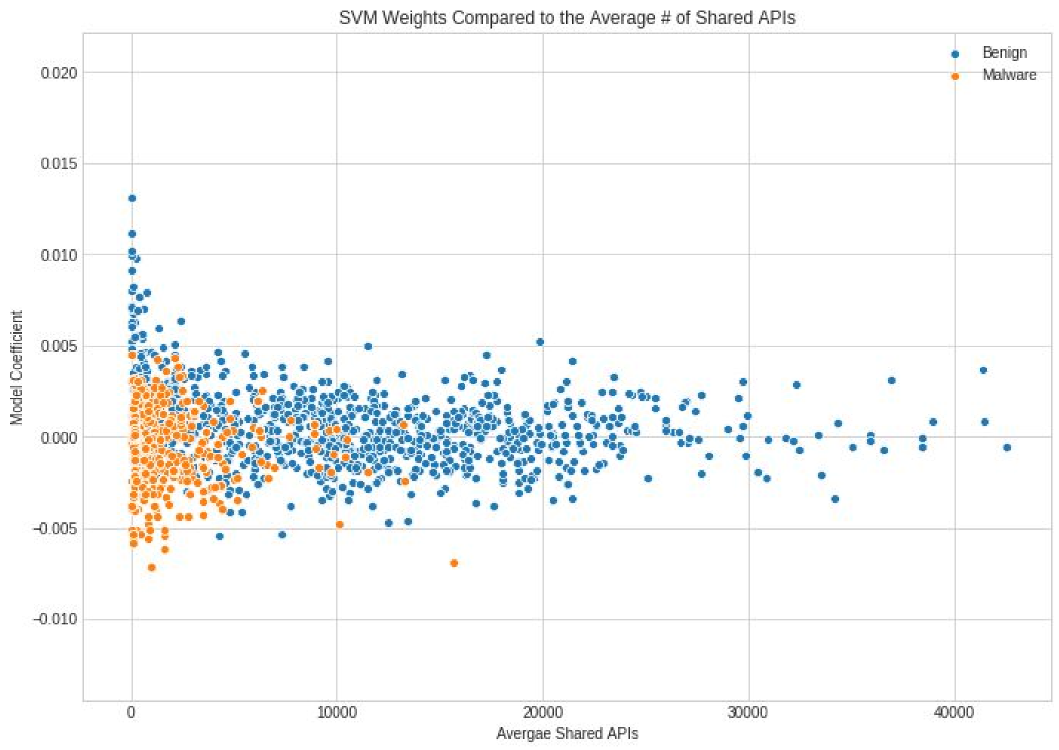
A.P.B.P.A^T	75.9%	242	38	95	179	72%
Multi kernel	**94.7%**	**275**	**5**	**24**	**250**	**94.51%**

Kernels with API’s removed occurring in 20 apps or less						
Kernel	Accuracy	TP	FN	FP	TN	F1-score
A.A^T	**95.8%**	**273**	**7**	**16**	**258**	**95.7%**
A.B^T	88%	258	22	44	230	87.4%
A.P.A^T	86.4%	252	28	47	227	85.8%
A.P.B.P.A^T	66.6%	185	95	88	186	67%
Multi kernel	94.7%	272	8	21	253	94.5%

Kernels with API’s removed in 5 apps or less and common API’s removed						
Kernel	Accuracy	TP	FN	FP	TN	F1-score
A.A^T	**94.2%**	**276**	**4**	**28**	**246**	**93.8%**
A.B^T	91.3%	277	3	45	229	90.5%
A.P.A^T	92.2%	273	7	36	238	91.1%
A.P.B.P.A^T	89.5%	268	12	46	228	88.7%
Multi kernel	93.8%	276	4	30	244	93.4%



This plot is showing support vectors for the AA^T kernel and their model coefficients. The Y axis is the model coefficient for a given support vector and the X axis is the average value of a given support vector. The Blue represents benign apps and orange represents malware.



This plot is similar to the previous in that it shows results for the AA^T kernel and the model coefficients. The Y axis is the model coefficient for a given support vector and the X axis is the average value of a given support vector. The Blue represents benign apps and orange represents malware. However, this plot shows the metric of maximum vector value instead of average vector value.

Discussion

Improving and Scaling

For the improvements and scaling of the Hindroid paper, we ran multiple different pipelines using the same set of data but changing the composition of the matrices specified by the Hindroid paper. These matrices are created using different API removal thresholds which can be found on the top row of each of the tables. This serves to reduce the dimensionality of the A matrix and therefore the B and P matrices and still maintain performance. The best performance of any model was the Multi-Kernel Learning model utilizing only APIs that occur in more than 5 apps which can be seen from the first table on the last row. However, on all other thresholds, the

Multi-Kernel learning performs at or below the level of the AA^T kernel. And, it only has the lowest false positive rate for a threshold of an API appearing in more than 5 or 10 apps. We found that the performance of multi-kernel decreases when more APIs are removed which is intuitive based on what we know from the Hindroid paper. The paper states that the APIs capture the relationships between apps and therefore are necessary to accurately classify malware. But, we saw when we removed the common APIs *and* the APIs occurring in 5 or less apps, the APA^T and ABA^T kernels performance improved as compared to the experiment where APIs were removed if they occurred in 5 or less apps. This finding is interesting and contradictory to the previously stated information from Hindroid. However, through these experiments we were able to show that the dimensionality of the A, B, and P matrices can be reduced without sacrificing performance.

Coefficient Analysis

In terms of the analysis of the coefficients, after utilizing the methodology laid out previously we were able to find some trends in the extreme coefficients as well as the false positives. We found that AA had the highest variation in its coefficients so it's differences were easier to analyze and easier to visualize. The coefficients for the Support vectors were between 0.00005 and -0.00005 while the coefficients for AA^T were between 0.013 and -0.00716 and the coefficients for APA^T were between 0.00186 and -0.001397 respectively. The first plot is showing support vectors for the AA^T kernel, their coefficients, and statistics about them. The Y axis of this plot depicts the model coefficient for a given support vector and the X axis depicts the average value of the given support vector. Blue in the plot represents Benign apps and orange represents malware. We found through our research that the distribution and statistics of the support vector are the indicators of model coefficients. Meaning, if the distribution of the vector is typical to what the model knows to be a malware app, it learns a negative coefficient for that support vector. This can be seen in the first plot where on the left side, the malware apps are on the lower end and all clustered towards a negative coefficient. The benign apps increase in average vector value and have both positive and negative coefficients but, they are more likely to have a positive coefficient. This can lead to mistakes because occasionally support vectors have similar statistics and distributions meaning they are assigned the wrong coefficient. As can be seen from the plot there are some benign apps assigned a negative weight because of its lower average vector value and vice versa for malware apps.

The second plot is similar to the previous in that it shows the AA^T kernel and model coefficients of support vectors, but it shows the metric of maximum vector value. The similar results can be seen as the lower values are malware apps and given negative coefficients and the higher values are benign apps and given positive coefficients. However, we encounter the same issue of the benign support vectors having relatively low maximum values and given large negative coefficients. These errors are what can lead to the False Positives, malware apps classified as benign, which could be dangerous is scaling to large sets of data and attempting to ship a commercial product. Since each kernel had this issue of giving incorrect weight to different support vectors, the multi-kernel learning algorithm may help in mitigating these issues because if too many mistakes are made by a certain kernel, it can be given a lower weight.

Through our research, we found that it is possible to utilize different metrics to scale the expensive computations of Hindroid without sacrificing performance. In addition, we found that any misclassification and reductions in accuracy may be caused by support vectors of the kernels being too similar in composition. This leads to the ultimate conclusion of the work we've done this quarter which is that the multi-kernel learning of a heterogeneous information network is the key in reducing the impact of these similar support vectors. By leveraging the techniques we've laid out, we can effectively understand and improve upon the work done in the Hindroid paper to make downloading apps a little bit safer for everyone.

Limitations

In attempting to replicate the paper itself we ran into many issues with the resources we had and this required us to innovate and rethink the problem in different ways that impact how other people may implement this framework of classification. The idea of utilizing a different dataset to validate the approach of any scientific work is challenging but necessary especially in the field of security of information. Therefore, the work being done is tantamount to adapting and evolving the proposed solution. Additionally, the attempts to explain the results of the model in order to understand how and why decisions are being made by the model lead to a more solidified concept and allow for more growth and development in the field. With this being said, there are some limitations with our experiment and the work we are currently doing. The dataset we are using is considerably small in comparison to the one used in the Hindroid paper and how they go about cleaning and extracting their smali files is interpreted by the reader and executed to the best of our ability. However, hindrances such as creating a representative sample using the apps from APKPure as well as the extraction of the useful smali code from those apk files may lead to an inability to extract the best possible replication and results for the project. Yet, the work we have conducted and the results we have obtained are able to open the door to further research on the subject of explaining Linear SVM models using kernels as well as improving on a model built using the principles of embedding graphs representing a Heterogeneous Information Network (HIN). It's always important to validate studies to raise new questions and push the envelope in terms of what can be understood with the goal of improving the model and the foundation with which the model is built. Additionally, finding a way to scale a model to be able to take in more samples and still run in a fast and reasonable manner while not sacrificing accuracy is an important frontier in terms of creating a product that can be used commercially by cybersecurity experts.

Future Work

In regards to future work on this project, there are many avenues to take to expand on the results and conclusions brought about by our work. First, we could try to implement a pipeline to tune the parameter of the threshold to optimize the performance while also decreasing the number of APIs needed to run the model. This would mean the model could be continually optimized as more samples are added and new APIs are found within those samples. Additionally, we would like to hypothesis test our results to validate that they are accurate and not anomalies in and of themselves. This would involve utilizing random shuffles of the data to see if these results hold for different training and test samples of different sizes, utilizing only certain apps or APIs, etc. Lastly, we would like to explore the possibility of attempting to explain the multi-kernel learning algorithm using the same methodology as for the single kernel. Because the multi-kernel was the best performing model from the Hindroid paper, it would be imperative that we investigate this model as well as an attempt to validate our results further as well as procure any additional hypotheses that could be tested to ensure thorough results and understanding of how to improve the Hindroid model.

References

[1] APKTool. http://ibotpeaches.github.io/Apktool/.
[2] Strazar M, Curk T. Learning the kernel matrix via predictive low-rank approximation. arXiv 2016. arXiv:1601.04366.
[3] Yanfang Ye, Shifu Hou, Yangqiu Song. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. 2017. In KDD.